SEG3102 – Lab 8
GraphQL Web Services

The objective of this lab is to introduce to the development of GraphQL based Web Services for Data Access. We will (1) develop a GraphQL Web Service API for a data model backed by a NoSQL Database and (2) create an Angular Client Application that consumes the API.

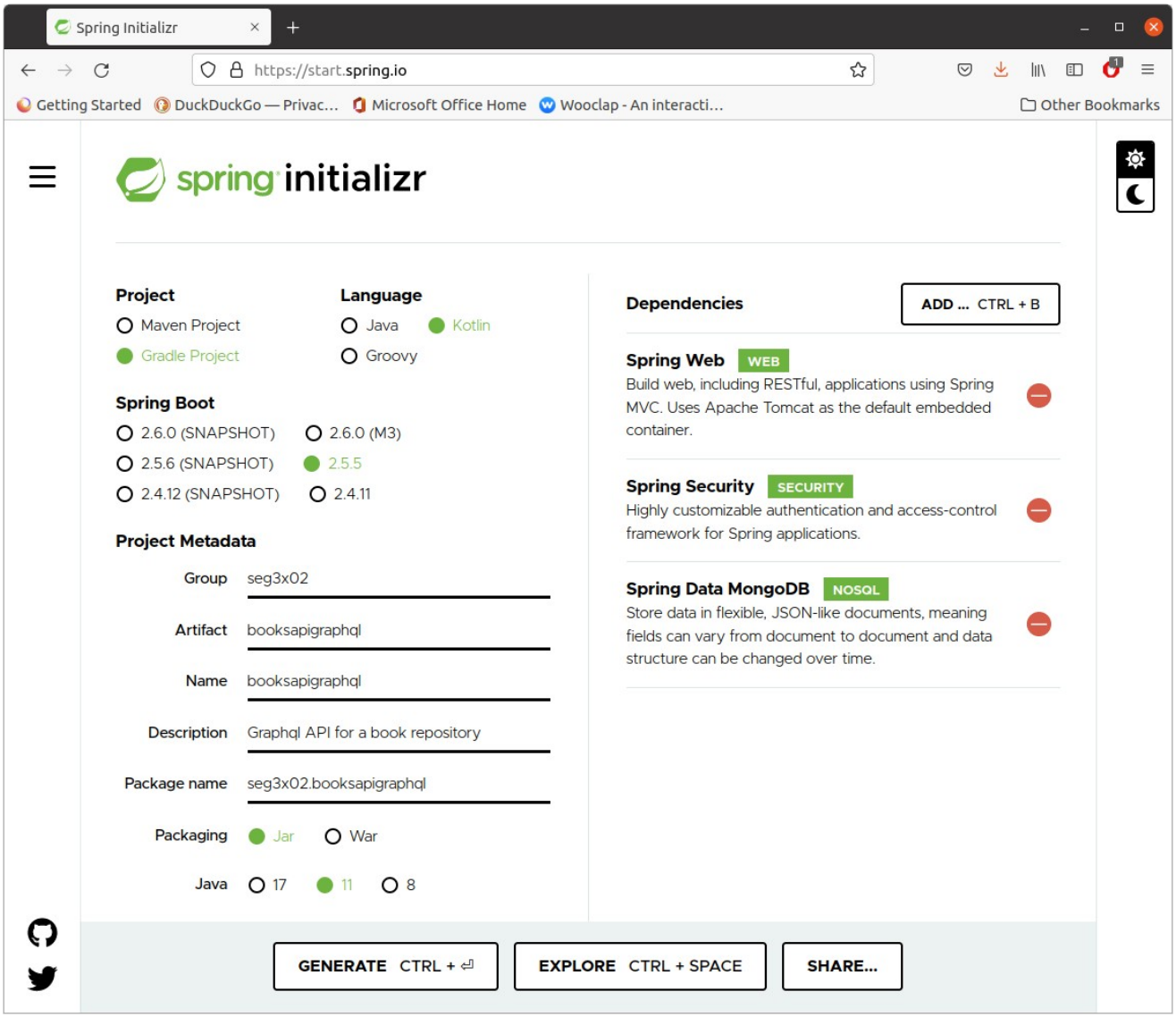
The source code for the lab is available at https://github.com/stephanesome/graphQLService.git. A SpringBoot Server Application can be found in branch *server*, while an Angular Client Application can be found in branch *client*.

GraphQL Server

We will develop a GraphQL Server for the Bookstore developed as part of Lab 3. We will store the data in a MongoDB data store.

Application Setup

Setup a SpringBoot project using Spring Intializr.



Select the following dependencies: Spring Web, Spring Data MongoDB and Spring Security. Spring Web is needed to expose the API over HTTP, Spring Data MongoDB provides the necessary to interact with the MongoDB NoSQL documents-oriented data store and Spring Security is needed to configure the security access to the API.

Add a dependency to com.graphql-java-kickstart:graphql-spring-boot-starter:14.0.0 in addition to the above to `build.gradle.kts` as follow.

```
1.  dependencies {
2.          ...
3.          implementation("com.graphql-java-kickstart:graphql-spring-boot-starter:14.0.0")
4.          ...
5.          testImplementation("com.graphql-java-kickstart:graphql-spring-boot-starter-test:14.0.0")
6.  }
```

MongoDB

We use MongoDB (https://www.mongodb.com/) as Data Store to persists our entities. You can download and install the Community Edition directly on your computer (https://docs.mongodb.com/manual/installation/). However, we will use Docker for a simpler and more convenient installation.

Run the following command to download a Docker image and set up MongoDB for this application.

```
docker run -p 27017:27017 --name mongodb -d mongo
```

Spring Properties

Specify the application properties in `src/main/resources/application.properties` as follow

```
server.port=9000
spring.data.mongodb.database=books-mongo
spring.data.mongodb.port=27017
graphql.graphiql.enabled=true
```

We specified a port for the application, the name of the MongoDB data store and its access port. We also enabled the GraphiQL tool (https://github.com/graphql/graphiql) that provides a Web UI for interacting with GraphQL Servers. This is useful to check that a server is working properly before connecting clients.

Entities

Create a package `seg3x02.booksapigraphql.entity`. Create a Kotlin data class Book in the package and edit as follow.

```
package seg3x02.booksapigraphql.entity

import org.springframework.data.annotation.Id
import org.springframework.data.mongodb.core.mapping.Document

@Document(collection = "book")
data class Book(var bookNumber: Int,
                var category: String,
                var title: String,
                var cost: Float,
                var year: String?,
                var description: String?) {
    @Id
    var bookId: String = ""

```

```
16.     @Transient
17.     var authors: List<Author> = ArrayList()
18. }
```

The class is annotated @Document, a Spring Data annotation that identifies a class as an entity persisted as document to MongoDB.

Similarly, create the data class Author and edit as follow.

```
1.  package seg3x02.booksapigraphql.entity
2.
3.  import org.springframework.data.mongodb.core.mapping.Document
4.
5.  @Document(collection = "author")
6.  data class Author(var bookNumber: Int,
7.                    var firstName: String,
8.                    var lastName: String)
```

Repositories

We create repositories for access to the data store. Create a package `seg3x02.booksapigraphql.repository`. Create a Kotlin interface BookRepository in the repository package and edit as follow.

```
1.  package seg3x02.booksapigraphql.repository
2.
3.  import org.springframework.data.mongodb.repository.MongoRepository
4.  import org.springframework.stereotype.Repository
5.  import seg3x02.booksapigraphql.entity.Book
6.
7.  @Repository
8.  interface BookRepository: MongoRepository<Book, String>
```

Similarly create a Kotlin interface AuthorRepository for Author and edit as follow.

```
1.  package seg3x02.booksapigraphql.repository
2.
3.  import org.springframework.data.mongodb.repository.MongoRepository
4.  import org.springframework.stereotype.Repository
5.  import seg3x02.booksapigraphql.entity.Author
6.
7.  @Repository
8.  interface AuthorRepository: MongoRepository<Author, Int>
```

GraphQL Schema

A GraphQL Schema specifies the API exposed by a GraphQL Server. It describes the data can be queried, the supported queries and modifications allowed to the data.

Create a file named `book.graphqls` in folder `src/main/resources` and edit as follow.

```
1.  type Query {
2.      books: [Book]
3.      bookById(bookId: ID!): Book
4.      bookByNumber(bookNumber: Int!): Book
5.  }
6.
7.  type Book {
8.      bookId: ID!
9.      bookNumber: Int!
10.     category: String!
11.     title: String!
12.     cost: Float!
13.     year: String
14.     description: String
15.     authors: [Author]
16. }
17.
18. type Mutation {
19.     newBook(bookNumber: Int!,
20.         category: String!,
21.         title: String!,
22.         cost: Float!, year:
23.         String,
24.         description: String) : Book!
25.     deleteBook(bookId: ID!) : Boolean
26.     updateBook(bookId:ID!,
27.         bookNumber: Int,
28.         category: String,
29.         title: String,
30.         cost: Float,
31.         year: String,
32.         description: String) : Book!
33. }
```

A GraphQL Schema (https://graphql.org/learn/) consists of type definitions. Each type has one or more fields. A field can take zero or more arguments and return a value from specific type.

Type Query defines the entry point of every GraphQL query. We specify three such queries as fields of the type (lines 1-5). A query to retrieve a list of instances of type Book, a query that takes as parameter an ID and returns an instance of Book and query that takes an Int and returns an instance of Book.

Type Book specifies the Book objects that can be obtained from the service. The fields with '!' appended to their type are *non-nullable*. The service always returns a value when such fields are queried. GraphQL provides a set of predefined types (i.e. Int, Float, String, ...) that are scalar types

corresponding to the leaves of the queries. A field may also be typed with an object type as for field authors (line 15).

A mutation is a special type of query that performs changes to data. Type Mutation defines the entry point of every mutation. The schema defines three mutations: newBook to add a new Book, deleteBook to remove a Book and updateBook to update the information for a Book.

Create a file name `author.graphqls` in folder `src/main/resources` and edit as follow to add a GraphQL schema for Author.

```
1.  extend type Query {
2.      authors(bookNumber: Int!): [Author]
3.  }
4.
5.  type Author { bookNumber: Int!
6.      firstName: String!
7.      lastName: String!
8.  }
9.
10. extend type Mutation {
11.     newAuthor(bookNumber: Int!
12.         firstName: String!,
13.         lastName:String!) : Author!
14. }
```

The schema extends the Query and Mutation types for Author. Note that this could have been defined in the main schema. We have separated for better modularity.

Resolvers

A resolver specifies the application logic for processing a field as part of the execution of queries and mutations. Resolvers are defined as functions.

Create a package `seg3x02.booksapigraphql.resolvers`. Create a Kotlin class BookQueryResolver in the package with the following content.

```
1.  package seg3x02.booksapigraphql.resolvers
2.
3.  import graphql.kickstart.tools.GraphQLQueryResolver
4.  import org.springframework.data.mongodb.core.MongoOperations
5.  import org.springframework.data.mongodb.core.query.Criteria
6.  import org.springframework.data.mongodb.core.query.Query
7.  import org.springframework.stereotype.Component
8.  import seg3x02.booksapigraphql.entity.Author
9.  import seg3x02.booksapigraphql.entity.Book
10. import seg3x02.booksapigraphql.repository.BookRepository
11.
```

```
12. @Component
13. class BookQueryResolver(val bookRepository: BookRepository,
14.                         private val mongoOperations: MongoOperations
15. ) : GraphQLQueryResolver {
16.     fun books(): List<Book> {
17.         val list = bookRepository.findAll()
18.         for (bk in list) {
19.             bk.authors = getAuthors(bk.bookNumber)
20.         }
21.         return list
22.     }
23.
24.     private fun getAuthors(bookNumber: Int): List<Author> {
25.         val query = Query()
26.         query.addCriteria(Criteria.where("bookNumber").`is`(bookNumber))
27.         return mongoOperations.find(query, Author::class.java)
28.     }
29.
30.     fun bookById(bookId: String): Book? {
31.         val book = bookRepository.findById(bookId)
32.         return if (book.isPresent) {
33.             val bk = book.get()
34.             bk.authors = getAuthors(bk.bookNumber)
35.             bk
36.         } else {
37.             null
38.         }
39.     }
40.
41.     fun bookByNumber(bookNumber: Number): Book? {
42.         val query = Query()
43.         query.addCriteria(Criteria.where("bookNumber").`is`(bookNumber))
44.         val result = mongoOperations.find(query, Book::class.java)
45.         return if (result.isNotEmpty()) {
46.             val bk = result[0]
47.             bk.authors = getAuthors(bk.bookNumber)
48.             bk
49.         } else {
50.             null
51.         }
52.     }
53. }
```

Query resolvers are defined as functions in a class that implements the GraphQLQueryResolver interface (line 15). Functions books (16-22), bookById (30-39) and bookByNumber (41-52)

correspond to the fields defined for type Query in the book schema. These functions use the repositories to retrieve the corresponding data from the data store.

A mutation resolver implements the interface GraphQLMutationResolver. Create a Kotlin class BookMutationResolver with the following content.

```
1.  package seg3x02.booksapigraphql.resolvers
2.
3.  import graphql.kickstart.tools.GraphQLMutationResolver
4.  import org.springframework.stereotype.Component
5.  import seg3x02.booksapigraphql.entity.Book
6.  import seg3x02.booksapigraphql.repository.BookRepository
7.  import java.util.*
8.
9.  @Component
10. class BookMutationResolver(private val bookRepository: BookRepository):
11.     GraphQLMutationResolver {
12.     fun newBook(bookNumber: Int,
13.                 category: String,
14.                 title: String,
15.                 cost: Float,
16.                 year: String?,
17.                 description: String?) : Book {
18.         val book = Book(bookNumber, category, title, cost, year, description)
19.         book.bookId = UUID.randomUUID().toString()
20.         bookRepository.save(book)
21.         return book
22.     }
23.
24.     fun deleteBook(id: String) : Boolean {
25.         bookRepository.deleteById(id)
26.         return true
27.     }
28.
29.     fun updateBook(bookId: String,
30.                    bookNumber: Int,
31.                    category: String?,
32.                    title: String?,
33.                    cost: Float?,
34.                    year: String?,
35.                    description: String?) : Book {
36.         val book = bookRepository.findById(bookId)
37.         book.ifPresent {
38.             if (bookNumber != null) {
39.                 it.bookNumber = bookNumber
```

```
            }
            if (category != null) {
                it.category = category
            }
            if (title != null) {
                it.title = title
            }
            if (cost != null) {
                it.cost = cost
            }
            it.year = year
            it.description = description
            bookRepository.save(it)
        }
        return book.get()
    }
}
```

Create a Kotlin class AuthorQueryResolver with the following content.

```
package seg3x02.booksapigraphql.resolvers

import graphql.kickstart.tools.GraphQLQueryResolver
import org.springframework.data.mongodb.core.MongoOperations
import org.springframework.data.mongodb.core.query.Criteria
import org.springframework.data.mongodb.core.query.Query
import org.springframework.stereotype.Component
import seg3x02.booksapigraphql.entity.Author

@Component
class AuthorQueryResolver(val mongoOperations: MongoOperations): GraphQLQueryResolver {
    fun authors(bookNumber: Int): List<Author> {
        val query = Query()
        query.addCriteria(Criteria.where("bookNumber").`is`(bookNumber))
        return mongoOperations.find(query, Author::class.java)
    }
}
```

Create a Kotlin class AuthorMutationResolver with the following content.

```
package seg3x02.booksapigraphql.resolvers

import graphql.kickstart.tools.GraphQLMutationResolver
import org.springframework.stereotype.Component
import seg3x02.booksapigraphql.entity.Author
import seg3x02.booksapigraphql.repository.AuthorRepository

```

```
8.  @Component
9.  class AuthorMutationResolver(private val authorRepository: AuthorRepository):
10.     GraphQLMutationResolver {
11.     fun newAuthor(bookNumber: Int,
12.                   firstName: String,
13.                   lastName:String) : Author {
14.         val author = Author(bookNumber, firstName, lastName)
15.         authorRepository.save(author)
16.         return author
17.     }
18. }
```

Web Configuration

We add security configuration to allow client access to the server.

Create a class WebSecurityConfig in package `seg3x02.booksapigraphql.configuration` and edit as follow.

```
1.  package seg3x02.booksapigraphql.configuration
2.
3.  import org.springframework.security.config.annotation.web.builders.HttpSecurity
4.  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
5.  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
6.
7.  @EnableWebSecurity
8.  class WebSecurityConfig  {
9.      @Bean
10.     fun filterChain(http: HttpSecurity): SecurityFilterChain {
11.       http.cors()
12.       http.csrf().disable()
13.       http.authorizeRequests().anyRequest().permitAll()
14.       return http.build()
15.     }
16. }
```

Spring Security allows us to configure an application security in a class annoted @EnableWebSecurity. The configuration is specified as a Bean. Our configuration enables Cross-Origin Resource Sharing (CORS) on line 11, we disable Cross-Site Request Forgery (CSRF) checking on line 12 since the application is a Web API, and authorize all requests with no authentication (line 13).

Create a class WebConfiguration in package `seg3x02.booksapigraphql.configuration` and edit as follow.

```
1.  package seg3x02.booksapigraphql.configuration
2.
3.  import org.springframework.context.annotation.Configuration
```

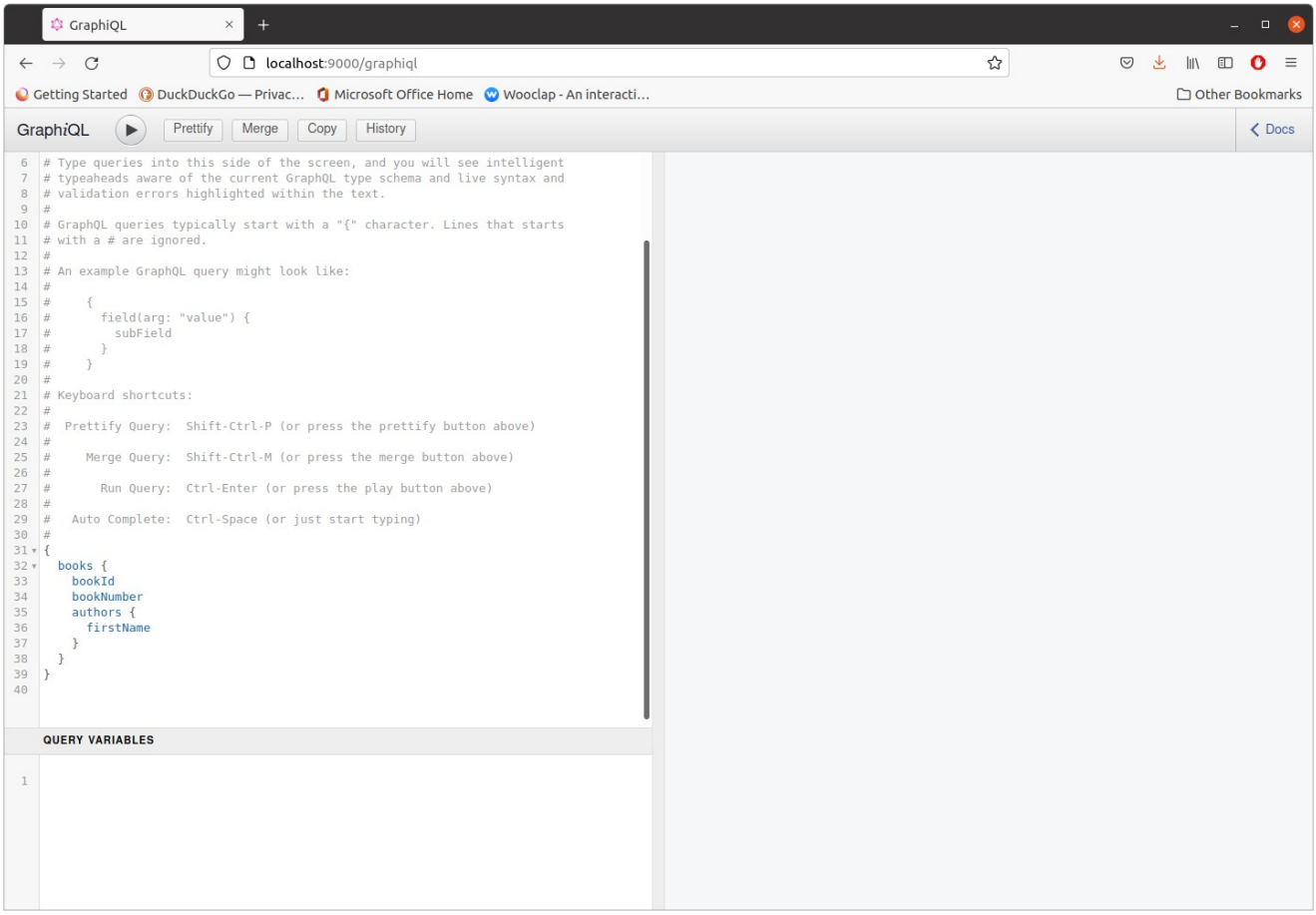
```
4.  import org.springframework.web.servlet.config.annotation.CorsRegistry
5.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurer
6.
7.  @Configuration
8.  class WebConfiguration : WebMvcConfigurer {
9.      override fun addCorsMappings(registry: CorsRegistry) {
10.         registry.addMapping("/**")
11.             .allowedOrigins("http://localhost:4200")
12.             .allowedMethods("*")
13.     }
14. }
```

In this configuration class, we specify settings for CORS by allowing Cross-Origin requests from our Angular client application at http://localhost:4200.

GraphiQL

GraphiQL is a tool that communicate with GraphQL Servers and provides a UI to execute queries and mutations. We can include a web-based version of GraphiQL in our application automatically, by enabling the property graphql.graphiql.enabled as was done earlier.

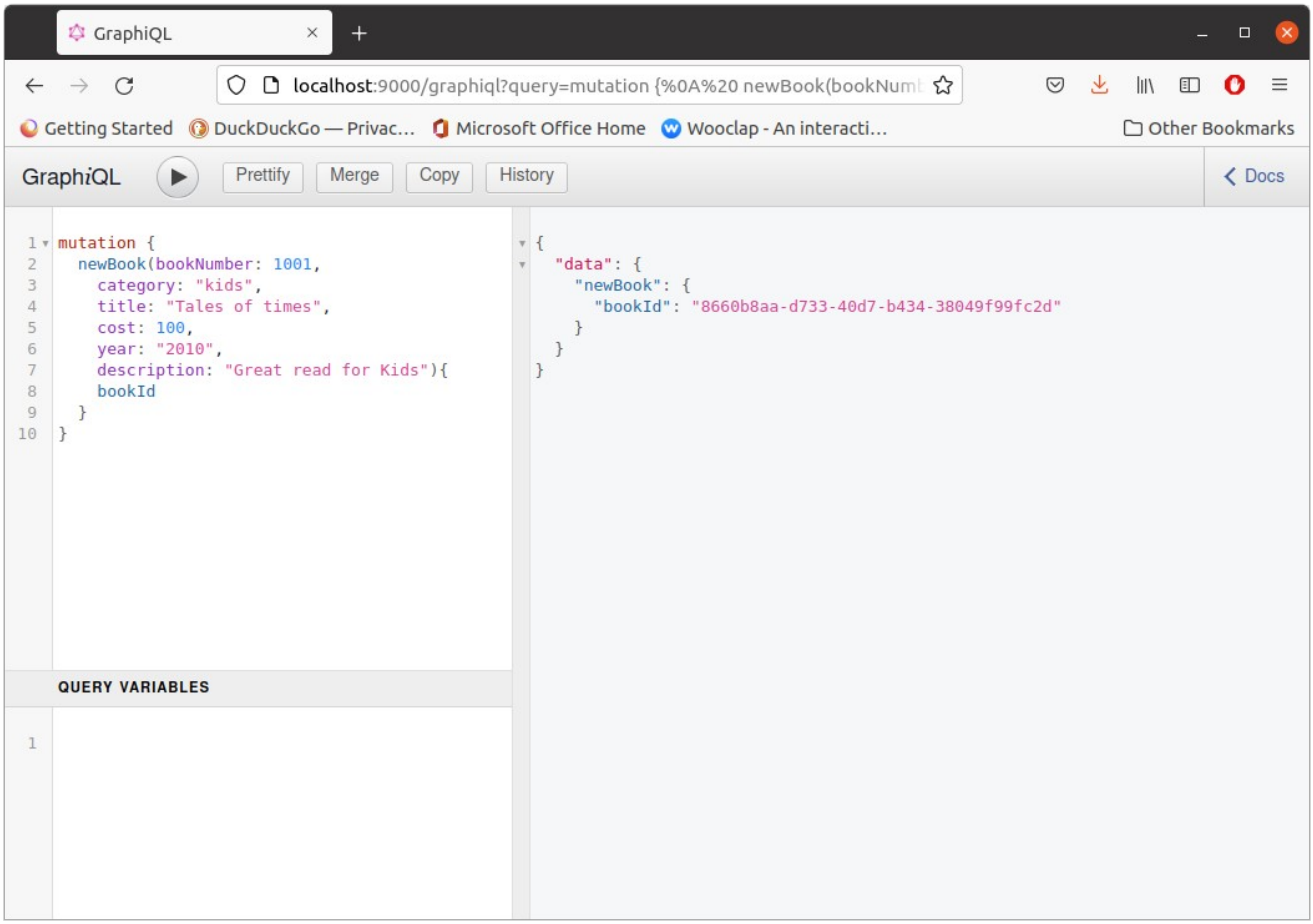
After the application is running, navigate to http://localhost:9000/graphiql to launch the GraphiQL interface for the server.



Execute the following mutation to add a Book.

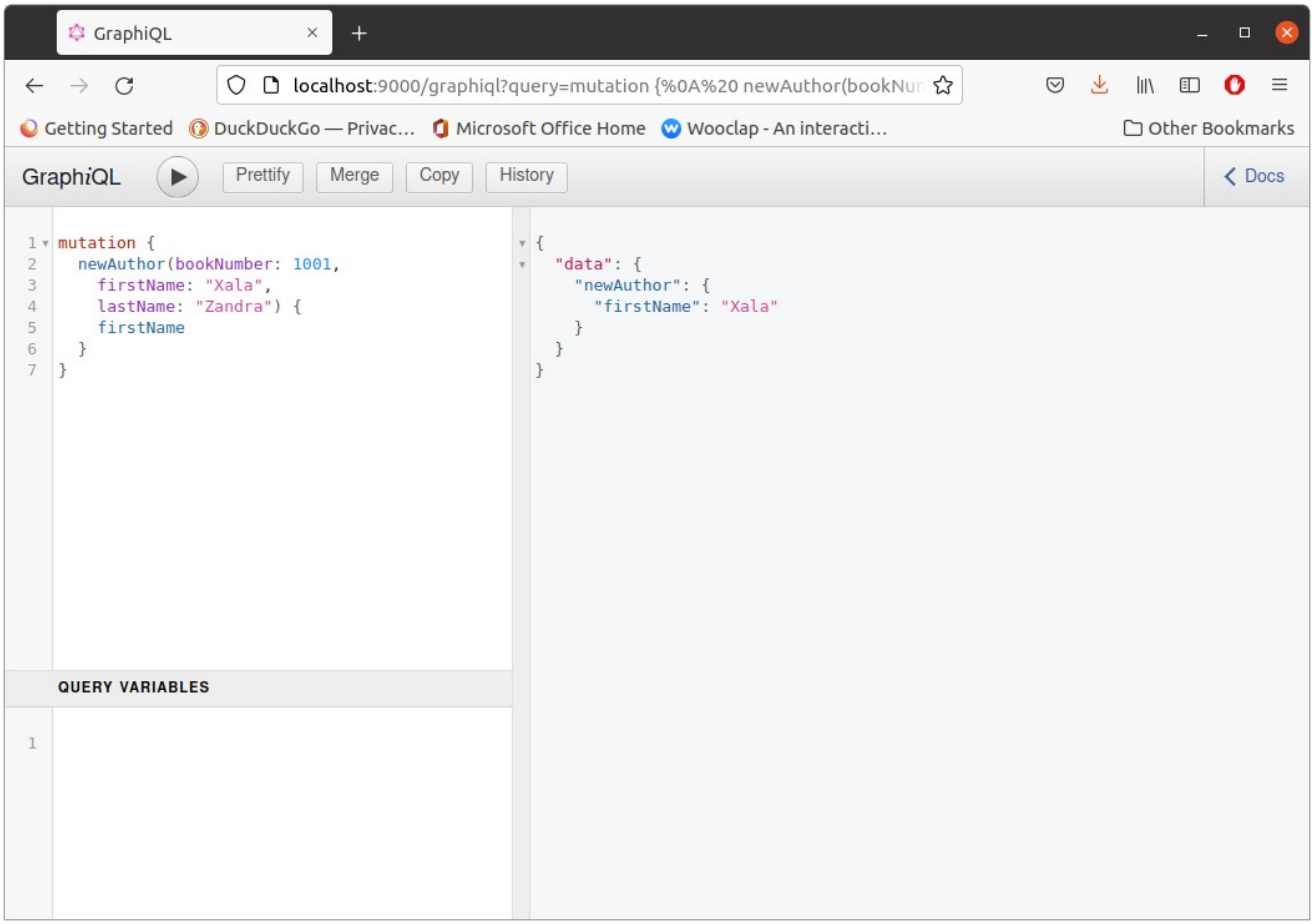
```
mutation {
  newBook(bookNumber: 1001,
    category: "kids",
    title: "Tales of times",
    cost: 100,
    year: "2010",
    description: "Great read for Kids"){
    bookId
  }
}
```

The query specifies that the bookId of the newly created book be returned.



We now execute the following mutation to add an author to the new book.

```
mutation {
  newAuthor(bookNumber: 1001,
    firstName: "Xala",
    lastName: "Zandra") {
    firstName
  }
}
```



Following is a query to return all the books and for each book, only the bookNumber and title as well as the lastnames of authors.

```
{
  books {
    bookNumber
    title
    authors {
      lastName
    }
  }
}
```

Angular Client

We are going to update the example Bookstore application from Lab3 to use the created Server.

Setup

We use Apollo (https://www.apollographql.com/) a library that provides tools to interact with GraphQL Servers. Apollo Angular (https://apollo-angular.com/docs/) provides specific support to Angular.

Add Apollo Angular to the project: `ng add apollo-angular`

Specify http://localhost:9000/graphql as URL to the GraphQL endpoint when prompted. This sets-up the GraphQL server URL in the GraphQLModule in `src/app/graphql.module.ts`.

`Book Model`

Update the Book Model (`src/app/books/model/book.ts`) as follow.

```
export class Book {
  constructor(
    public bookId: number,
    public bookNumber: number,
    public category: string,
    public title: string,
    public cost: number,
    public authors?: Author[],
    public year?: number,
    public description?: string
  ) {}
}

export class Author {
  constructor(
    public bookNumber: number,
    public firstName: string,
    public lastName: string
  ){}
}
```

Books Service

Update Books Service (`src/app/books/service/books.service.ts`) as follow.

```
1.  import { Injectable } from '@angular/core';
2.  import {Author, Book} from '../model/book';
3.  import {Apollo, gql} from "apollo-angular";
4.  import {Observable} from "rxjs";
5.  import {ApolloQueryResult, FetchResult} from "@apollo/client/core";
6.
7.  @Injectable({
8.    providedIn: 'root'
9.  })
10. export class BooksService {
11.   constructor(private apollo: Apollo) {}
12.
13.   public getBook(bookNumber: number): Observable<ApolloQueryResult<any>> {
14.     return this.apollo
15.     .query<any>({
16.       query: gql`
17.         query($bookNumber: Int!) {
18.           bookByNumber(bookNumber: $bookNumber) {
19.             bookId
20.             bookNumber
21.             category
22.             title
23.             cost
24.             description
25.             authors {
26.               firstName
27.               lastName
28.             }
29.           }
30.         }
31.       `,
32.       variables: {
33.         bookNumber
34.       }
35.     });
36.   }
37.
38.   public addBook(b: Book): Observable<FetchResult<unknown>> {
39.     return this.apollo.mutate({
40.         mutation: gql`
41.           mutation newBook($bookNumber: Int!,
42.             $category: String!,
```

```
43.           $title: String!,
44.           $cost: Float!,
45.           $year: String,
46.           $description: String){
47.           newBook(bookNumber: $bookNumber,
48.             category: $category,
49.             title: $title,
50.             cost: $cost,
51.             year: $year,
52.             description: $description) {
53.             bookId
54.           }
55.         }
56.       `,
57.       variables: {
58.         bookNumber: b.bookNumber,
59.         category: b.category,
60.         title: b.title,
61.         cost: b.cost,
62.         year: b.year,
63.         description: b.description
64.       }
65.     }
66.   );
67. }
68.
69. addAuthor(author: Author): Observable<FetchResult<unknown>> {
70.   return this.apollo.mutate({
71.     mutation: gql`
72.       mutation newAuthor($bookNumber: Int!,
73.         $firstName: String!,
74.         $lastName: String!){
75.         newAuthor(bookNumber: $bookNumber,
76.           firstName: $firstName,
77.           lastName: $lastName) {
78.           firstName
79.           lastName
80.         }
81.       }
82.     `,
83.     variables: {
84.       bookNumber: author.bookNumber,
85.       firstName: author.firstName,
86.       lastName: author.lastName
87.     }
```

```
88.       }
89.     );
90.   }
91. }
```

The Books Service is now set to use the Apollo service (injected in line 11) to interact with the GraphQL server. The Apollo service takes GraphQL queries or mutations wrapped in the gql template literal tag (https://github.com/apollographql/graphql-tag). These requests are send to the server and an Observable returned for the result.

Function getBook (lines 13-36) requests the search for a book corresponding to a given book number. The addBook function (lines 38-67) requests the execution of a mutation to add a book, and the addAuthor function (lines 69-91) requests the execution of a mutation to add an author to a book.

Book Component

Update the Book Component Class (`src/app/books/book/book.component.ts`) as follow.

```
1.  import {Component, OnDestroy, OnInit} from '@angular/core';
2.  import {ActivatedRoute} from '@angular/router';
3.  import {Book} from '../model/book';
4.  import {BooksService} from '../service/books.service';
5.  import {Subscription} from "rxjs";
6.
7.  @Component({
8.    selector: 'app-book',
9.    templateUrl: './book.component.html',
10.   styleUrls: ['./book.component.css']
11. })
12. export class BookComponent implements OnInit, OnDestroy {
13.   selectedBook: Book | null = null;
14.   private subscription!: Subscription;
15.
16.   constructor(private route: ActivatedRoute, private booksService: BooksService) {
17.   }
18.
19.   ngOnInit(): void {
20.     this.route.params.subscribe(params => {
21.       const numb = Number(params.id);
22.       this.subscription = this.booksService.getBook(numb).subscribe(
23.         ({ data, loading }) => {
24.           this.selectedBook = data.bookByNumber;
25.         },
26.         (error: any) => {
27.           this.selectedBook = null;
28.         });
29.     });
30.   }
```

```
31.
32.   ngOnDestroy(): void {
33.     this.subscription.unsubscribe();
34.   }
35. }
```

We subscribe to the Observable returned by the Books Service (line 22) and retrieve the returned data when provided.

Update Book Component HTML Template as follow.

```
1.  <div *ngIf="!selectedBook">
2.    <h2>Sorry can't find the requested book...</h2>
3.  </div>
4.  <div *ngIf="selectedBook">
5.    <h2>Here are details of the Book {{selectedBook.bookNumber}}</h2>
6.    <div class="row">
7.      <div class="col-xs-3">Category:</div>
8.      <div class="col-xs-9">{{ selectedBook.category }}</div>
9.    </div>
10.   <div class="row">
11.     <div class="col-xs-3">Title:</div>
12.     <div class="col-xs-9">{{ selectedBook.title }}</div>
13.   </div>
14.   <div class="row">
15.     <div class="col-xs-3">Cost:</div>
16.     <div class="col-xs-9">{{ selectedBook.cost }}</div>
17.   </div>
18.   <div class="row" [hidden]="!selectedBook.authors">
19.     <div class="col-xs-3">Author:</div>
20.     <div class="col-xs-9"><span [innerHTML]="selectedBook.authors | authornames"></span></div>
21.   </div>
22.   <div class="row" [hidden]="!selectedBook.year">
23.     <div class="col-xs-3">Year:</div>
24.     <div class="col-xs-9">{{ selectedBook.year }}</div>
25.   </div>
26.   <div class="row" [hidden]="!selectedBook.description">
27.     <div class="col-xs-3">Description:</div>
28.     <div class="col-xs-9">{{ selectedBook.description }}</div>
29.   </div>
30. </div>
```

Admin Component

Update Admin Component Class as follow.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import {AbstractControl, FormArray, FormBuilder, FormControl, Validators} from '@angular/forms';
3.  import {Author, Book} from '../books/model/book';
```

```
4.  import {BooksService} from '../books/service/books.service';
5.
6.  function categoryValidator(control: FormControl): { [s: string]: boolean } | null {
7.    const validCategories = ['Kids', 'Tech', 'Cook'];
8.    if (!validCategories.includes(control.value)) {
9.      return {invalidCategory: true};
10.   }
11.   return null;
12. }
13.
14. @Component({
15.   selector: 'app-admin',
16.   templateUrl: './admin.component.html',
17.   styleUrls: ['./admin.component.css']
18. })
19. export class AdminComponent implements OnInit {
20.   bookForm = this.builder.group({
21.     bookNumber: ['', [Validators.required, Validators.pattern('[1-9]\\d{3}')]],
22.     category: ['', [Validators.required, categoryValidator]],
23.     title: ['', Validators.required],
24.     cost: ['', [Validators.required, Validators.pattern('\\d+(\\.\\d{1,2})?')] ],
25.     authors: this.builder.array([]),
26.     year: [''],
27.     description: ['']
28.   });
29.
30.   get bookNumber(): AbstractControl {return <AbstractControl>this.bookForm.get('bookNumber'); }
31.   get category(): AbstractControl {return <AbstractControl>this.bookForm.get('category'); }
32.   get title(): AbstractControl {return <AbstractControl>this.bookForm.get('title'); }
33.   get cost(): AbstractControl {return <AbstractControl>this.bookForm.get('cost'); }
34.   get authors(): FormArray {
35.     return this.bookForm.get('authors') as FormArray;
36.   }
37.
38.   constructor(private builder: FormBuilder,
39.               private booksService: BooksService) { }
40.
41.   ngOnInit(): void {
42.   }
43.
44.   onSubmit(): void {
45.     const book =  new Book(0,
46.       Number(this.bookForm.value.bookNumber),
47.       this.bookForm.value.category,
48.       this.bookForm.value.title,
```

```
49.       Number(this.bookForm.value.cost),
50.       [],
51.       Number(this.bookForm.value.year),
52.       this.bookForm.value.description);
53.     const authors = this.bookForm.value.authors;
54.     this.booksService.addBook(book).subscribe(_ => {
55.       authors.forEach((author: Author) => {
56.         this.booksService.addAuthor(Object.assign(author, {bookNumber: book.bookNumber}))
57.           .subscribe(a => {});
58.       });
59.     });
60.     this.bookForm.reset();
61.     this.authors.clear();
62.   }
63.
64.   addAuthor(): void {
65.     this.authors.push(
66.       this.builder.group({
67.         firstName: [''],
68.         lastName: ['']
69.       })
70.     );
71.   }
72.
73.   removeAuthor(i: number): void {
74.     this.authors.removeAt(i);
75.   }
76. }
```

The main change is in function onSubmit where the Books Service is used to add a book (line 54). We then subscribe to the Observable that is returned and once complete, we invoke the Books Service to add all the authors of the new book (lines 56-58).

Update Admin Component HTML Template as follow.

```
1.  <div class="container">
2.    <h1>Book Form</h1>
3.    <button (click)="addAuthor()">Add Author</button>
4.    <form [formGroup]="bookForm" (ngSubmit)="onSubmit()">
5.      <div class="form-group">
6.        <label for="bookNumber">Number:</label>
7.        <input type="text" class="form-control" id="bookNumber"  formControlName="bookNumber"
    required>
8.        <div [hidden]="bookNumber.pristine || bookNumber.valid"
9.             class="alert alert-danger">
10.         Book number must be 4 digits long starting with a number different from 0.
```

```
      </div>
    </div>
    <div class="form-group">
      <label for="category">Category:</label>
      <input type="text" class="form-control" id="category"  formControlName="category">
      <div [hidden]="category.pristine || category.valid"
          class="alert alert-danger">
        Category is required to be <b>Kids</b>, <b>Tech</b> or <b>Cook</b>
      </div>
    </div>
    <div class="form-group">
      <label for="title">Title:</label>
      <input type="text" class="form-control" id="title" required formControlName="title">
      <div [hidden]="title.pristine || title.valid"
          class="alert alert-danger">
        Title is required.
      </div>
    </div>
    <div class="form-group">
      <label for="cost">Cost:</label>
      <input type="text" class="form-control" id="cost" required  pattern="\d+(.\d{1,2})?"
formControlName="cost">
      <div [hidden]="cost.pristine || cost.valid"
          class="alert alert-danger">
        Cost should be a number with two optional decimals
      </div>
    </div>
    <div class="form-group">
      <div formArrayName="authors">
        <div *ngFor="let _ of authors.controls; let i=index">
          <ng-container [formGroupName]="i">
            <label>
              Author First Name:
              <input formControlName="firstName" type="text">
            </label>
            <label>
              Author Last Name:
              <input formControlName="lastName" type="text">
            </label>
            <button class="btn btn-dark" (click)="removeAuthor(i)">X</button>
          </ng-container>
        </div>
      </div>
    </div>
    <div class="form-group">
```

```
55.       <label for="year">Year:</label>
56.       <input type="text" class="form-control" id="year" formControlName="year">
57.     </div>
58.     <div class="form-group">
59.       <label for="description">Description:</label>
60.       <textarea cols="40" class="form-control" id="description"
      formControlName="description"></textarea>
61.     </div>
62.     <button type="submit" class="btn btn-success" [disabled]="bookForm.invalid">Submit</button>
63.   </form>
64. </div>
```

Exercise

The following exercise is the deliverable for the lab. Complete and check-in the code to Github Classroom before the deadline. Only this exercise will be evaluated.

Extend the application developed as part of Lab4 exercise, to:

1. persist user information to a MongoDB database through a GraphQL Server developed with Springboot,
2. allow for the retrieval and display of all the recorded user information in a table.