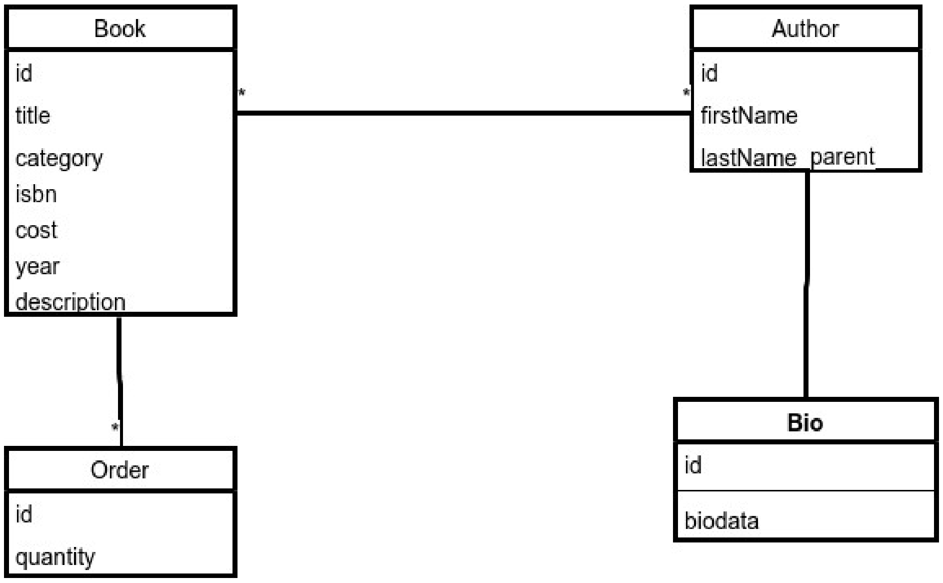
SEG3102 – Lab 7
Data Web Services

The objective of this lab is to introduce to the development of Web Services for Data Access with SpringBoot. We will (1) develop a REST Web Service API for a data model backed by a SQL 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/dataRestService.git. A SpringBoot Server Application can be found in branch *server*, while an Angular Client Application can be found in branch *client*.

1. Springboot Data Access API

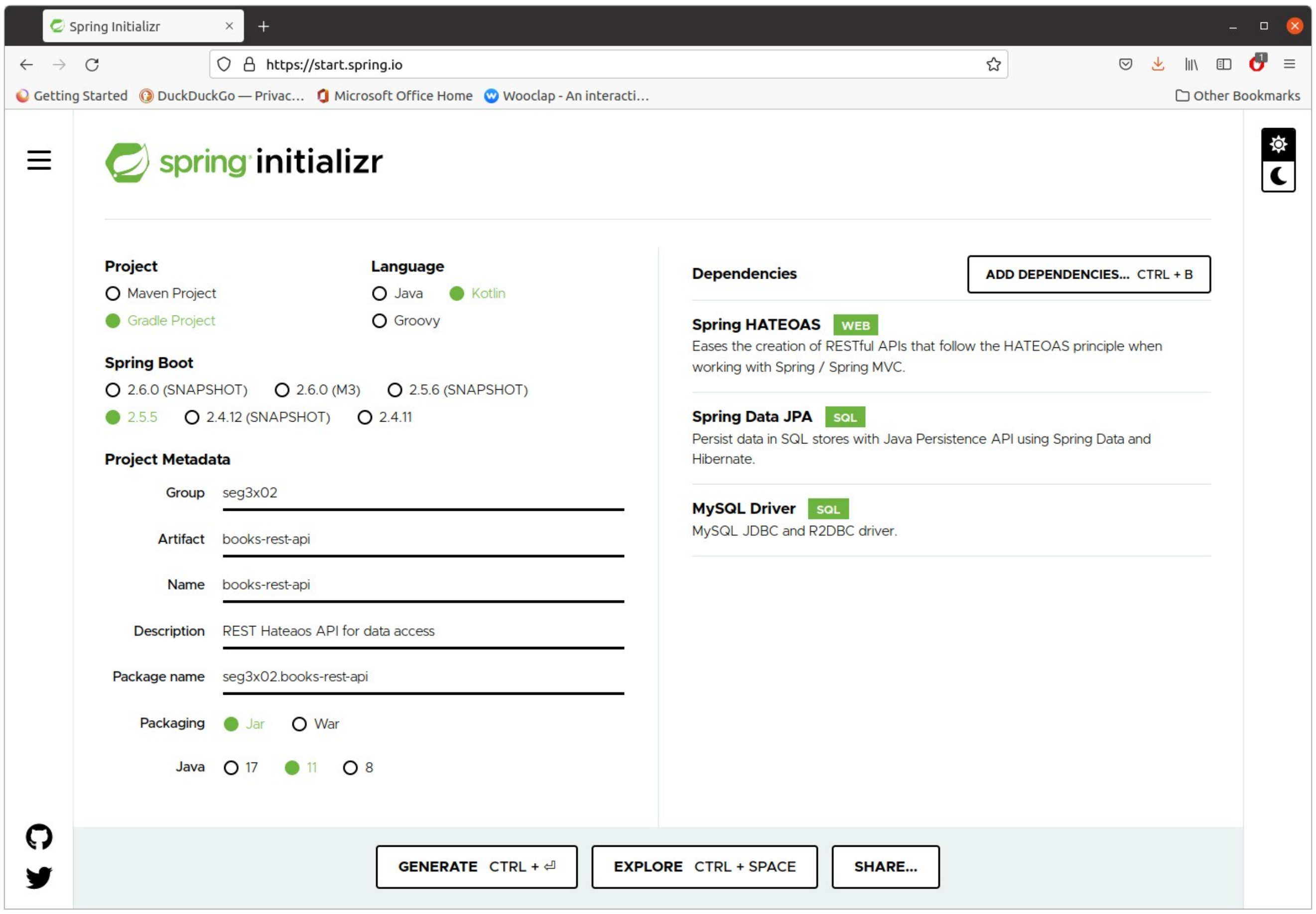
We will develop a SpringBoot backend application to store and retrieve data for the following Domain model for Book Store.



The data is stored in a SQL relational database and the application exposes a REST API for CRUD (Create Read Update Delete) operations. The API is created to satisfy the Hypermedia As The Engine Of Application State (HATEOAS) principle that stipulates that we should “use a gradual unfolding methodology to expose data for clients” by returning minimum data with Hyperlinks that allow users to dynamically navigate to resources.

Application Setup

Setup a SpringBoot project using Spring Intializr.



Project: Maven Project [✓ Gradle Project]

Language: Java [✓ Kotlin] Groovy

Spring Boot: 2.6.0 (SNAPSHOT) 2.6.0 (M3) 2.5.6 (SNAPSHOT) [✓ 2.5.5] 2.4.12 (SNAPSHOT) 2.4.11

Project Metadata

Group	seg3x02
Artifact	books-rest-api
Name	books-rest-api
Description	REST Hateaos API for data access
Package name	seg3x02.books-rest-api
Packaging	[✓ Jar] War
Java	17 [✓ 11] 8

Dependencies

Spring HATEOAS WEB
Eases the creation of RESTful APIs that follow the HATEOAS principle when working with Spring / Spring MVC.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

Select the following dependencies: Spring HATEOAS, Spring Data JPA and MySQL Driver.

Spring HATEOAS provides support to create Hyperlinks in responses, Spring Data JPA supports Relational Databases using the Jakarta Persistence API (JPA) – an Object-Relational mapping specification, and the MySQL Driver is a driver to interact with MySQL Relational SQL database.

Add a dependency to org.springdoc:springdoc-openapi-ui in addition to the above in `build.gradle.kts` (see previous lab). This will generate an OpenAPI Specification for the Web API.

MySQL

We use MySQL (https://www.mysql.com/) as Relational Database System to persists our entities. You can download and install MySQL directly on your computer (https://dev.mysql.com/doc/mysql-installation-excerpt/5.7/en/). However, we will use Docker for a simpler and more convenient installation. Run the following command to download a Docker image and set up the MySQL Database system for this application.

```
docker run -d -p 6033:3306 --name=docker-mysql --
env="MYSQL_ROOT_PASSWORD=root" --env="MYSQL_PASSWORD=root" --
env="MYSQL_DATABASE=booksDb" mysql
```

Edit the application Spring configuration file `src/main/resources/application.properties` as follow.

```
spring.datasource.url=jdbc:mysql://localhost:6033/booksDb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.sql.init.platform=mysql
```

The configuration specifies the database location and access credentials. ***Note that for security reasons, it is strongly recommended in production not to specify the database access credentials in a property file and not to use root access***. This has been done here strictly for the sake of simplicity.

We also set property spring.jpa.hibernate.ddl-auto to specify how Spring JPA generates the schema for the database. With value update, the database schema is automatically generated based on the JPA annotations in the source code and kept up to date when changes are made in the source code. Other possible values are create (for the schema to be re-created for every run), create-drop, validate and none.

Data Entities

Create a package `seg3x02.booksrestapi.entities`. Create a Kotlin class Book in the entities package and edit as follow.

```
package seg3x02.booksrestapi.entities

import javax.persistence.*

@Entity
class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0
     var title: String = ""
     var category: String = ""
     var isbn: String = ""
     var cost: Double = 0.0
     var year: Int = 0
     var description: String = ""

     @ManyToMany(mappedBy = "books", cascade = [CascadeType.PERSIST, CascadeType.MERGE])
     var authors: MutableList<Author> = ArrayList()

     @OneToMany(cascade = [CascadeType.ALL])
     var orders: MutableList<Order> = ArrayList()
}
```

The class is annotated @Entity to denote that it is a JPA Entity that is persisted in a Relational Database.

We specify auto-generation as primary key generation strategy (line 7-8). A unique primary keys for the entity (and corresponding database row) will be provided by the (DBMS).

The Book entity has a many-to-many bidirectional relation with the Author entity and a one-to-many unidirectional relation with the Order entity. This is expressed using the annotations on lines 17 and 20.

Create the other entities: Order, Author, Bio and edit as follow.

```
package seg3x02.booksrestapi.entities

import javax.persistence.*

@Entity
@Table(name = "BookOrder")
class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0
    var quantity: Int = 0
}
```

```
package seg3x02.booksrestapi.entities

import javax.persistence.*

@Entity
class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0
    var firstName: String = ""
    var lastName: String = ""

    @ManyToMany
    var books: MutableList<Book> = ArrayList()

    @OneToOne(fetch = FetchType.LAZY, cascade = [CascadeType.ALL])
    var bio: Bio = Bio()
}
```

```
package seg3x02.booksrestapi.entities

import javax.persistence.Entity
import javax.persistence.GeneratedValue
import javax.persistence.GenerationType
import javax.persistence.Id

@Entity
class Bio {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0
    var biodata: String = ""
}
```

Repositories

Spring Data makes it possible to create a data access repository for an entity by simply specifying an interface. Create a package `seg3x02.booksrestapi.repository`. Create a Kotlin interface BookRepository in the repository package and edit as follow.

```
package seg3x02.booksrestapi.repository

import org.springframework.data.repository.CrudRepository
import seg3x02.booksrestapi.entities.Book

interface BookRepository: CrudRepository<Book, Long>
```

BookRepository extends the CrudRepository interface. Spring Data will generate and register a concrete class with all the interface methods such as findAll, findById, save, deleteById, ...

Create BioRepository and OrderRepository similarly and edit as follow.

```
package seg3x02.booksrestapi.repository

import org.springframework.data.repository.CrudRepository
import seg3x02.booksrestapi.entities.Bio

interface BioRepository: CrudRepository<Bio, Long>
```

```
package seg3x02.booksrestapi.repository

import org.springframework.data.repository.CrudRepository
import seg3x02.booksrestapi.entities.Order

interface OrderRepository: CrudRepository<Order, Long>
```

Create AuthorRepository and edit as follow.

```
package seg3x02.booksrestapi.repository

import org.springframework.data.jpa.repository.Query
import org.springframework.data.repository.CrudRepository
import seg3x02.booksrestapi.entities.Author

interface AuthorRepository: CrudRepository<Author, Long> {
    @Query(value=
      "select aut from Author aut where aut.firstName = :firstName and aut.lastName = :lastName")
    fun findAuthorsByName(firstName: String, lastName: String): List<Author>
}
```

AuthorRepository includes a custom query in addition to the queries specified on the CrudRepository interface. The query is defined by declaring function findAuthorsByName signature (lines 8-11). The annotation @Query specifies a query string in the *JPA Query Language* for the query.

Spring HATEOAS

Spring HATEOAS (https://spring.io/projects/spring-hateoas) provides a set of classes for creating resource representations with hyperlinks. The main classes are RepresentationModel, Link, and WebMvcLinkBuilder.

RepresentationModel is a base class for creating a resource representation. It provides a method add to create hyperlinks.

The Link class is used to create hyperlinks. Each hyperlink identifies a relation to the resource and includes a *href* attribute for the link.

We can create links with Link by specifying all the information. However, WebMvcLinkBuilder simplifies the task by providing methods to dynamically create links based on mappings of REST Controllers.

The following is a JSON representation for an instance of Book.

```
{
  "id": 1,
  "title": "SpringBoot JPA and REST",
  "category": "Tech",
  "isbn": "4444-777-3423",
  "cost": 25.45,
  "year": 2017,
  "description": "This is by far the best resource for SpringBoot, JPA and REST",
  "authors": [
    {
      "firstName": "Koredo",
      "lastName": "Tablanga",
      "_links": {
        "self": {
          "href": "http://localhost:8080/books-api/authors/4"
        }
      }
    }
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8080/books-api/books/1"
    },
    "orders": {
      "href": "http://localhost:8080/books-api/books/1/orders"
    }
  }
}
```

Attribute _links (lines 20-27) specifies the links added to the representation. There are two links here a *self* link providing the location of the resource and a link *orders* to the Book Orders associated to the book resource.

There is one author associated with the book instance (lines 9-19) here. The representation only includes the first name and last name of the author and provides a link to the resource.

Spring HATEOAS provides a CollectionModel class to represent a collection of elements. The following is an excerpt of JSON representation of a CollectionModel for Books.

```
{
  "_embedded": {
    "books": [...]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/books-api/books"
    }
  }
}
```

Representation Models

Create a package `seg3x02.booksrestapi.representation`. Create class BookRepresentation in package representation. Edit as follow.

```
package seg3x02.booksrestapi.representation

import com.fasterxml.jackson.annotation.JsonInclude
import com.fasterxml.jackson.annotation.JsonRootName
import org.springframework.hateoas.RepresentationModel
import org.springframework.hateoas.server.core.Relation

@Relation(collectionRelation = "books")
@JsonInclude(JsonInclude.Include.NON_NULL)
class BookRepresentation: RepresentationModel<BookRepresentation>() {
    var id: Long = 0
    var title: String = ""
    var category: String = ""
    var isbn: String = ""
    var cost: Double = 0.0
    var year: Int = 0
    var description: String = ""
    var authors: List<AuthorNameRepresentation> = ArrayList<AuthorNameRepresentation>()
}
```

The BookRepresentation class extends the base RepresentationModel class and defines the attributes of a representation for a book. The list authors refers to class AuthorNameRepresentation which concerns only a subset of the attributes of entity Author. Annotation @Relation specifies the relation to be used when referring to a collection of books and annotation @JsonInclude exclude properties having null values from being serialized.

Create the other representation model classes AuthorNameRepresentation, AuthorRepresentation, BioRepresentation, BookTitleRepresentation and OrderRepresentation. Edit according to the source code on GitHub.

Model Assemblers

Spring HATEAOS provides a RepresentationModelAssembler interface with a base implementation class RepresentationModelAssemblerSupport, that provides methods to create representation of domain objects.

Create a package `seg3x02.booksrestapi.assemblers`. Create class BookModelAssembler in the assemblers package. Edit as follow.

```
package seg3x02.booksrestapi.assemblers

import org.springframework.hateoas.CollectionModel
import org.springframework.hateoas.server.mvc.RepresentationModelAssemblerSupport
import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder
import org.springframework.stereotype.Component
import seg3x02.booksrestapi.controller.ApiController
import seg3x02.booksrestapi.entities.Author
import seg3x02.booksrestapi.entities.Book
import seg3x02.booksrestapi.representation.AuthorNameRepresentation
import seg3x02.booksrestapi.representation.BookRepresentation
import java.util.*

@Component
class BookModelAssembler: RepresentationModelAssemblerSupport<Book,
        BookRepresentation>(ApiController::class.java, BookRepresentation::class.java) {
    override fun toModel(entity: Book): BookRepresentation {
        val bookRepresentation = instantiateModel(entity)
        bookRepresentation.add(WebMvcLinkBuilder.linkTo(
                WebMvcLinkBuilder.methodOn(ApiController::class.java)
                        .getBookById(entity.id))
                .withSelfRel())
        bookRepresentation.authors = toAuthorsRepresentation(entity.authors)
        bookRepresentation.add(WebMvcLinkBuilder.linkTo(
                WebMvcLinkBuilder.methodOn(ApiController::class.java)
                        .getBookOrdersById(entity.id))
                .withRel("orders"))
        bookRepresentation.id = entity.id
        bookRepresentation.isbn = entity.isbn
        bookRepresentation.category = entity.category
        bookRepresentation.title = entity.title
        bookRepresentation.cost = entity.cost
        bookRepresentation.year = entity.year
        bookRepresentation.description = entity.description
        return bookRepresentation
    }

    override fun toCollectionModel(entities: Iterable<Book>): CollectionModel<BookRepresentation> {
        val bookRepresentations = super.toCollectionModel(entities)
40.         bookRepresentations.add(WebMvcLinkBuilder.linkTo(
41.                 WebMvcLinkBuilder.methodOn(
42.                         ApiController::class.java).allBooks()).withSelfRel())
43.         return bookRepresentations
44.     }
45.
46.     fun toAuthorsRepresentation(authors: List<Author>): List<AuthorNameRepresentation> {
47.         return if (authors.isEmpty()) Collections.emptyList() else authors
48.                 .map{
49.                     authorRepresentation(it)
50.                 }
51.
52.     }
53.
54.     private fun authorRepresentation(author: Author): AuthorNameRepresentation {
55.         val representation = AuthorNameRepresentation()
56.         representation.firstName = author.firstName
57.         representation.lastName = author.lastName
58.         return representation.add(WebMvcLinkBuilder.linkTo(
59.                 WebMvcLinkBuilder.methodOn(ApiController::class.java)
60.                         .getAuthorById(author.id))
61.                 .withSelfRel())
62.     }
63. }
```

Annotation @Component specifies BookModelAssembler as a bean. Spring will register and inject an instance when required. The class extends the base implementation of RepresentationModel (class RepresentationModelAssembler) for the assembly of instances of the Book entity.

Function toModel returns a BookRepresentation. The call to instantiateModel (line 18) creates the base representation model. We add a *self* link at lines 19-22. The URI is determined from the controller (ApiController) mapping function getBookById. The embedded representation for the book authors is created in line 23 by a call to function toAuthorsRepresentation defined in lines 46-52. A link named *orders* to the book orders is added on lines 24-27, using again the corresponding controller function. Lines 28-34 copy the attribute values of the entity to the representation.

Create the other assembler classes AuthorModelAssembler, BioModelAssembler, BookModelAssembler and OrderModelAssembler. Edit according to the source code on GitHub.

REST Controller

Create a package `seg3x02.booksrestapi.controller` and class ApiController in the package. Edit according to the source code on GitHub.

The class declaration is as follow.

```
@RestController
@CrossOrigin(origins = ["http://localhost:4200"])
@RequestMapping("books-api", produces = ["application/hal+json"])
class ApiController(val authorRepository: AuthorRepository,
                    val bookRepository: BookRepository,
                    val bioRepository: BioRepository,
                    val orderRepository: OrderRepository,
                    val authorAssembler: AuthorModelAssembler,
                    val bookAssembler: BookModelAssembler,
                      val orderAssembler: OrderModelAssembler,
                      val bioAssembler: BioModelAssembler) {
....
}
```

The @RestController annotation specifies this class as a Controller for HTTP requests.

Spring does not allow Cross-Origin Resource Sharing (CORS) by default. We disable CORS protection to allow requests from our Angular Client application, which runs at http://localhost:4200, using annotation @CrossOrigin. The disabling applies to all the function handlers of the class.

The @RequestMapping specifies the root URI and media type produced by the controller handlers. We specify the *JSON Hypertext Application Language* (*hal+json*) as media type produced. The *hal+json* format supports the addition of hypermedia links into JSON. Various beans for the repositories and assemblers used are injected in the constructor of the class.

The controller handler to retrieve all books is as follow.

```
@Operation(summary = "Get all books")
@GetMapping("/books")
fun allBooks(): ResponseEntity<CollectionModel<BookRepresentation>> {
    val books = bookRepository.findAll()
    return ResponseEntity(
            bookAssembler.toCollectionModel(books),
            HttpStatus.OK)
}
```

We use the SpringDoc annotation @Operation to add documentation to the operation. The annotation @GetMapping specifies that method HTTP GET is used for the operation and the path "*/books*" is appended to the root URI. The function uses the book repository to get all the books from the database and returns a ResponseEntity with the representation of the books (created using the book assembler) with the HTTP Status code OK (200).

The mapping function to retrieve a book based on the id is as follow.

```
1.      @Operation(summary = "Get a book by id")
2.      @GetMapping("/books/{id}")
3.      fun getBookById(@PathVariable("id") id: Long): ResponseEntity<BookRepresentation> {
4.          return bookRepository.findById(id)
5.                  .map { entity: Book -> bookAssembler.toModel(entity) }
6.                  .map { body: BookRepresentation -> ResponseEntity.ok(body) }
7.                  .orElse(ResponseEntity.notFound().build())
8.   }
```

The book id is passed as path variable (line 2) that is mapped to a parameter (line 3). The function queries the database and returns a response entity with a description of the book in the body and HTTP Status Code OK if found. If a book with the specified id is not found, a response with status code NotFound (404) is returned.

A handler for operation to create a new book with the HTTP POST method is as follow.

```
@Operation(summary = "Add a new book")
@PostMapping("/books")
fun addBook(@RequestBody book: Book): ResponseEntity<Any> {
    return try {
```

```
5.             val newBook = this.bookRepository.save(book)
6.             val location: URI = ServletUriComponentsBuilder
7.                   .fromCurrentRequest()
8.                   .path("/{id}")
9.                   .buildAndExpand(newBook.id)
10.                  .toUri()
11.            ResponseEntity.created(location).body(bookAssembler.toModel(newBook))
12.        } catch (e: IllegalArgumentException) {
13.            ResponseEntity.badRequest().build()
14.        }
15. }
```

Data for the new book is deserialized from JSON and passed as argument *book* to the function (line 3). We save the book to the database and return a description of the saved book (which has an id set). The response header *Location* value is set in lines (6-10) to the URI of the new book resource, and returned in the response. In case of an exception, the handler returns a response with HTTP Status Code BadRequest (400).

An handler function to update the information of an existing book is as follow.

```
@Operation(summary = "Update the information of a book")
@PutMapping("/books/{id}")
fun updateBook(@PathVariable("id") id: Long, @RequestBody book: Book): ResponseEntity<Any> {
    return try {
        val currBook = bookRepository.findById(id).get()
        currBook.title = book.title
        currBook.isbn = book.isbn
        currBook.cost = book.cost
        currBook.category = book.category
        currBook.description = book.description
        currBook.year = book.year
        bookRepository.save(currBook)
        ResponseEntity.noContent().build<Any>()
    } catch (e: NoSuchElementException) {
        ResponseEntity.badRequest().build()
    } catch (e: IllegalArgumentException) {
        ResponseEntity.badRequest().build()
    }
}
```

We use the HTTP PUT method for the operation. An handler function to remove a book is as follow.

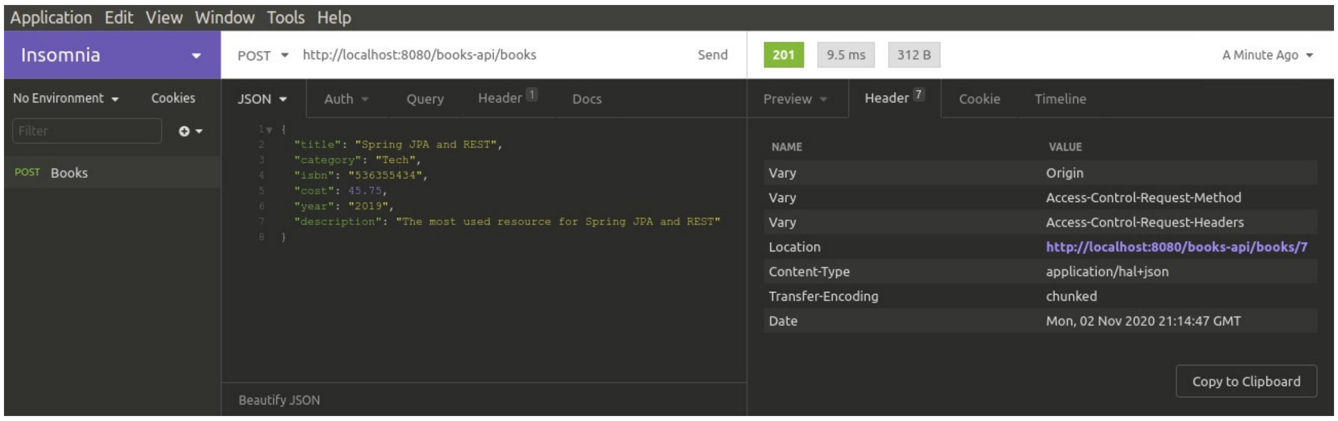
```
@Operation(summary = "Remove an book")
@DeleteMapping("/books/{id}")
fun deleteBook(@PathVariable("id") id: Long): ResponseEntity<Any> {
    return try {
        this.bookRepository.deleteById(id)
        ResponseEntity.noContent().build<Any>()
    } catch (e: IllegalArgumentException) {
        ResponseEntity.badRequest().build()
    }
}
```

Note that all authors of a book must first be deleted before you can delete the book.

Building and Running

You can run with command `./gradlew bootRun` from the project main folder and then issue requests to the API at the base URL http://localhost:8080/books-api using a tool such as Curl, Postman or Insomnia. You may also interact with the API with the Swagger UI at `http://localhost:8080/swagger-ui.html`.

The database is initially empty you will therefore need to add data first by issuing POST requests as shown here.



POST http://localhost:8080/books-api/books

```json
{
  "title": "Spring JPA and REST",
  "category": "Tech",
  "isbn": "536355434",
  "cost": 45.75,
  "year": "2019",
  "description": "The most used resource for Spring JPA and REST"
}
```

NAME	VALUE
Vary	Origin
Vary	Access-Control-Request-Method
Vary	Access-Control-Request-Headers
Location	http://localhost:8080/books-api/books/7
Content-Type	application/hal+json
Transfer-Encoding	chunked
Date	Mon, 02 Nov 2020 21:14:47 GMT

2. Angular Client

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

HTTP Client Module

Import the HttpClient Module to the AppModule. Edit `src/app/app.module.ts` and add the following.

```
import {HttpClientModule} from '@angular/common/http';

@NgModule({
  ...
  imports: [
    ...
    HttpClientModule
  ],
   ...
})
export class AppModule { }
```

Books Service

Edit the books service class `src/app/books/service/books.service.ts` as follow.

```
import {Injectable} from '@angular/core';
import {Author, Book} from '../model/book';
import {HttpClient, HttpParams} from '@angular/common/http';
import {Observable} from 'rxjs';
import {map, tap} from 'rxjs/operators';

const Url = 'http://localhost:8080/books-api/';


@Injectable({
  providedIn: 'root'
})
export class BooksService {
  constructor(private http: HttpClient) {}

  public getBook(id: string): Observable<Book> {
    return this.http.get<Book>(Url + 'books/' + id);
  }

  public addBook(book: Book): Observable<Book> {
    return this.http.post<Book>(Url + 'books', book);
  }

  public addBookAuthor(id: number, author: Author): Observable<Author> {
    return this.http.post<Author>(Url + 'books/' + id + '/authors', author);
  }

  public getAuthorsNamed(firstName: string, lastName: string): Observable<any> {
    const options = {params: new HttpParams().set('firstName', firstName).set('lastName', lastName)};
    return this.http.get<any>(Url + 'authors', options).pipe(
      map(response => response._embedded ? response._embedded.authors : undefined )
    );
  }

  public updateBookAuthors(bookId: number, authorId: number): Observable<any> {
    return this.http.patch(Url + 'books/' + bookId + '/authors/' + authorId, {});
}
```

Books are now obtained by querying the books REST API. The Book component must now subscribe to the Observable returned by the service to get data for a book. Edit `src/app/books/book/book.component.ts` as follow.

```
import {Component, OnDestroy, OnInit} from '@angular/core';
import {ActivatedRoute} from '@angular/router';
import {Book} from '../model/book';
import {BooksService} from '../service/books.service';
import {Subscription} from "rxjs";

@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;
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 id = params.id;
22.        this.subscription = this.booksService.getBook(id).subscribe(
23.          (data: Book) => {
24.            this.selectedBook = data;
25.          },
26.          (_: any) => {
27.            this.selectedBook = null;
28.          });
29.      });
30.    }
31.
32.    ngOnDestroy(): void {
33.      this.subscription.unsubscribe();
34.    }
35.  }
```

The component calls the service function getBook and subscribes to the Observable returned (line 22) with a callback for a response and a callback for an error. The callback for a response sets a property with the book to be shown in the component HTML template.

Admin Component

Refer to the source on GitHub for the Admin component class and HTML template. We take a look here at the updated function for handling book creation.

```
onSubmit(): void {
  const book =  new Book(0,
    this.bookForm.value.category,
    this.bookForm.value.title,
    Number(this.bookForm.value.cost),
    [],
    Number(this.bookForm.value.year),
    this.bookForm.value.description);
  const authors = this.bookForm.value.authors;
  this.booksService.addBook(book).subscribe(
    (response) => {
      authors.forEach(
13.      (author: Author) => {
14.       this.booksService.getAuthorsNamed(author.firstName, author.lastName).subscribe(
15.        (authorList: Author[]) => {
16.         if (authorList === undefined || authorList.length === 0) {
17.          this.booksService.addBookAuthor(response.id, author).subscribe();
18.         } else {
19.          // *** Assumes unique firstName/LastName for Authors
20.          this.booksService.updateBookAuthors(response.id, authorList[0].id).subscribe();
21.         }
22.        }
23.       );
24.      }
25.     );
26.     this.showMessage('info', `The was successfully added with id ${response.id}`);
27.    },
28.   (_: any) => {
29.    this.showMessage('error', 'Unable to add the book');
30.   }
31.   );
32.   this.bookForm.reset();
33.   this.authors.clear();
34. }
```

At line 10, a call to the books service function addBook is performed. This issue a POST request with the data for the new book (except authors). In the callback for a response, we issue a request for each of the authors, that queries for the existence of authors with the same name (line 14). If no such author exist, we issue a query to add the author as new author for the book (line 17) otherwise, we add the existing author as an author of the book (line 20). Note that it is assumed here that authors names are unique.

Exercise

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

Add an "Authors" menu option to the navigation bar so that when clicked, the Content area displays a form allowing the user to enter an author id. Once submitted, the app should perform a query and display the author's information if found. Otherwise, an appropriate message should be displayed.