Internet Technologies

RESTful API Server

University of Cyprus
Department of Computer Science

How to build RESTful APIs?

- Develop JAVA RESTful API server:
  - **Spring Boot** (https://spring.io/guides/gs/rest-service/)
  - **Jersey** (https://eclipse-ee4j.github.io/jersey/)
- Develop JavaScript RESTful API server:
  - **Node.js** (https://nodejs.org/) and **Express** (https://expressjs.com/)
- Develop Python RESTful API server:
  - **Django** Framework (https://www.djangoproject.com/)
  - **FastAPI** (https://fastapi.tiangolo.com/)
  - **Flask** (https://flask.palletsprojects.com/)
    - **Flask** RESTful (https://flask-restful.readthedocs.io/en/latest/)
      - Flask extension for quickly building REST APIs
    - **Flask** RESTPlus (https://flask-restplus.readthedocs.io/en/stable/index.html)
      - Flask extension for quickly building REST APIs

Java or Python for RESTful APIs?

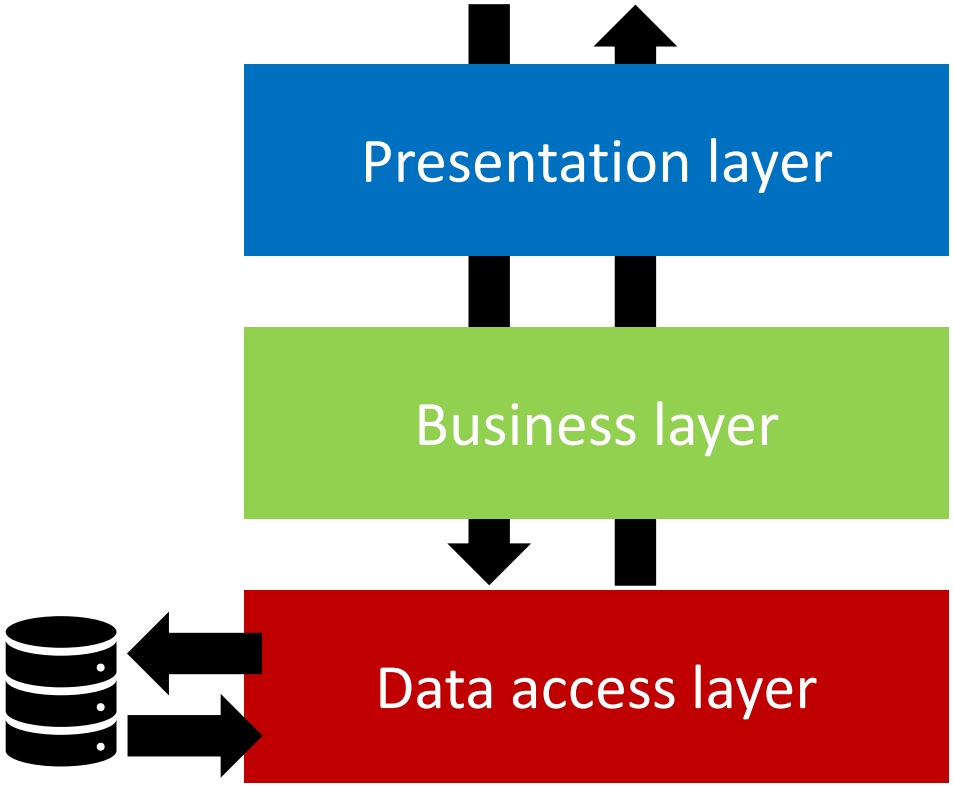
- Java is recommended for enterprise-level, high-load APIs
  - Slower development time
  - Heavier resource (RAM) usage
  - Easier application packaging (.jar)
  - Significant version dependence => expensive system support
- JavaScript is recommended for fast-prototyping, medium-load APIs
  - Use the same familiar syntax for both client and server-side tasks (faster development time)
  - Lightweight resource usage, ideal for real-time data processing
  - Slower than Java
- Python is recommended for fast-prototyping, low-load, personal-use APIs
  - Faster development time
  - No compilation, faster testing
  - Minimal version dependence (given than Python 2.x is deprecated and rarely used)

Case study - RESTful API using Spring Boot

- Create a system to store/deliver information about the books of a library along with their reviews
- Entities served by the RESTful API: books, reviews
  - API will be used to **C**reate, **R**etrieve, **U**pdate and **D**elete (CRUD) book and review data
- Steps:
  1. Create a DB table for each entity
  2. Create a **RESTful API in Spring Boot** to enable CRUD operations on each entity
  3. Test RESTful API using Postman

Spring Boot Application Architecture

- Spring Boot framework follows a layered architecture in which each layer communicates with the layer directly below or above

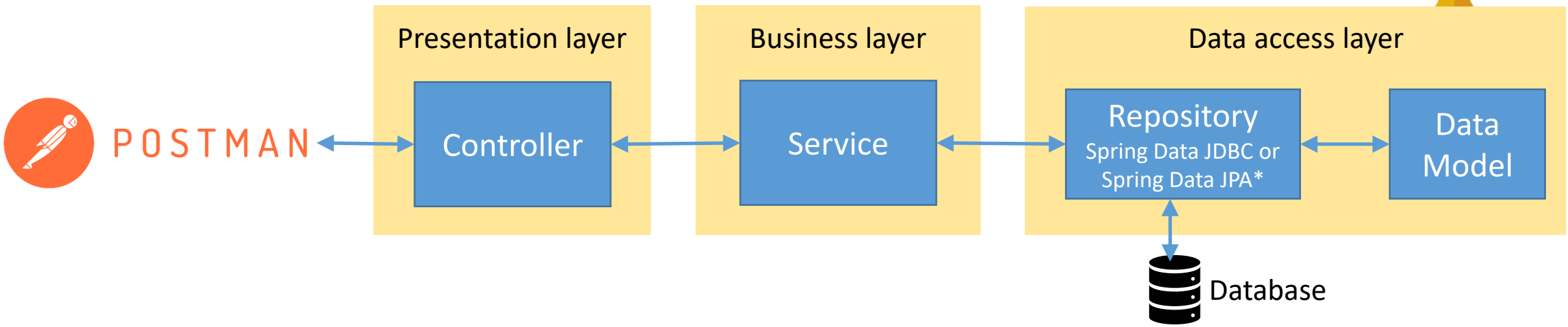


Handles incoming HTTP requests and transfers them to the business layer. It is responsible for converting (mapping) JSON messages to Java objects and vice-versa. This layer can perform authentication (verify user before allowing access to API resources). Java classes in this layer are called controllers.

Handles all the business logic which encodes real-world business rules (e.g. a user can borrow up to 3 books) that determine how entities can be created, retrieved, updated and deleted. Business layer uses functions provided by the data access layer to exchange entity-related information from/to DB. Java classes in this layer are called services.

Handles all the storage logic. It is responsible for performing CRUD (create, retrieve, update, delete) operations on the entities. **Database entities are mapped as Java classes**. Java classes in this layer are called models. Data access layer involves repository Java classes which provide functions to interact with the database.

Spring Boot Application Flow Architecture



1. RESTful API client (e.g POSTMAN) issues HTTP requests (e.g. GET, POST, PUT, DELETE)
   - Example: GET the book with id=3
2. The request handled by the controller object which transfers it to the service object
3. Service class calls one or more repository functions to retrieve the requested data
   - Repository is a mechanism for enabling CRUD operations on entities (tables of DB)
   - Example: call the specific repository function which returns the book based on its id
4. The retrieved entity (book) is mapped to a Java object (model)
5. Controller converts book object to JSON message and returns it to API client

(*) Spring Data JDBC is faster, gives control over SQL queries (allows for creating complex SQL queries) compared to Spring Data JPA. Spring Data JPA allows for faster development, hides complexity (no need to write SQL queries), offers lazy loading, data caching.

Create project using Spring initializr

- **Start from scratch**: Spring Initializr
  - Add dependencies:
    - **Spring Web**: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
    - **Spring Data JDBC**: Persists data in SQL stores (e.g. MariaDB) with plain JDBC using Spring Data.
    - **Spring Boot DevTools**: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
  - After we set the parameters (see next slide) we press Generate at the bottom of the page to download the zip folder of the project

Project

○ Gradle - Groovy ○ Gradle - Kotlin
● Maven

Language

● Java ○ Kotlin ○ Groovy

Spring Boot

○ 3.3.0 (SNAPSHOT) ○ 3.3.0 (M3) ○ 3.2.5 (SNAPSHOT) ● 3.2.4
○ 3.1.11 (SNAPSHOT) ○ 3.1.10

Project Metadata

Group	cy.ac.ucy.cs.epl425
Artifact	LMS
Name	LMS
Description	Demo project for Spring Boot
Package name	cy.ac.ucy.cs.epl425.LMS
Packaging	● Jar ○ War
Java	○ 22 ○ 21 ● 17

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

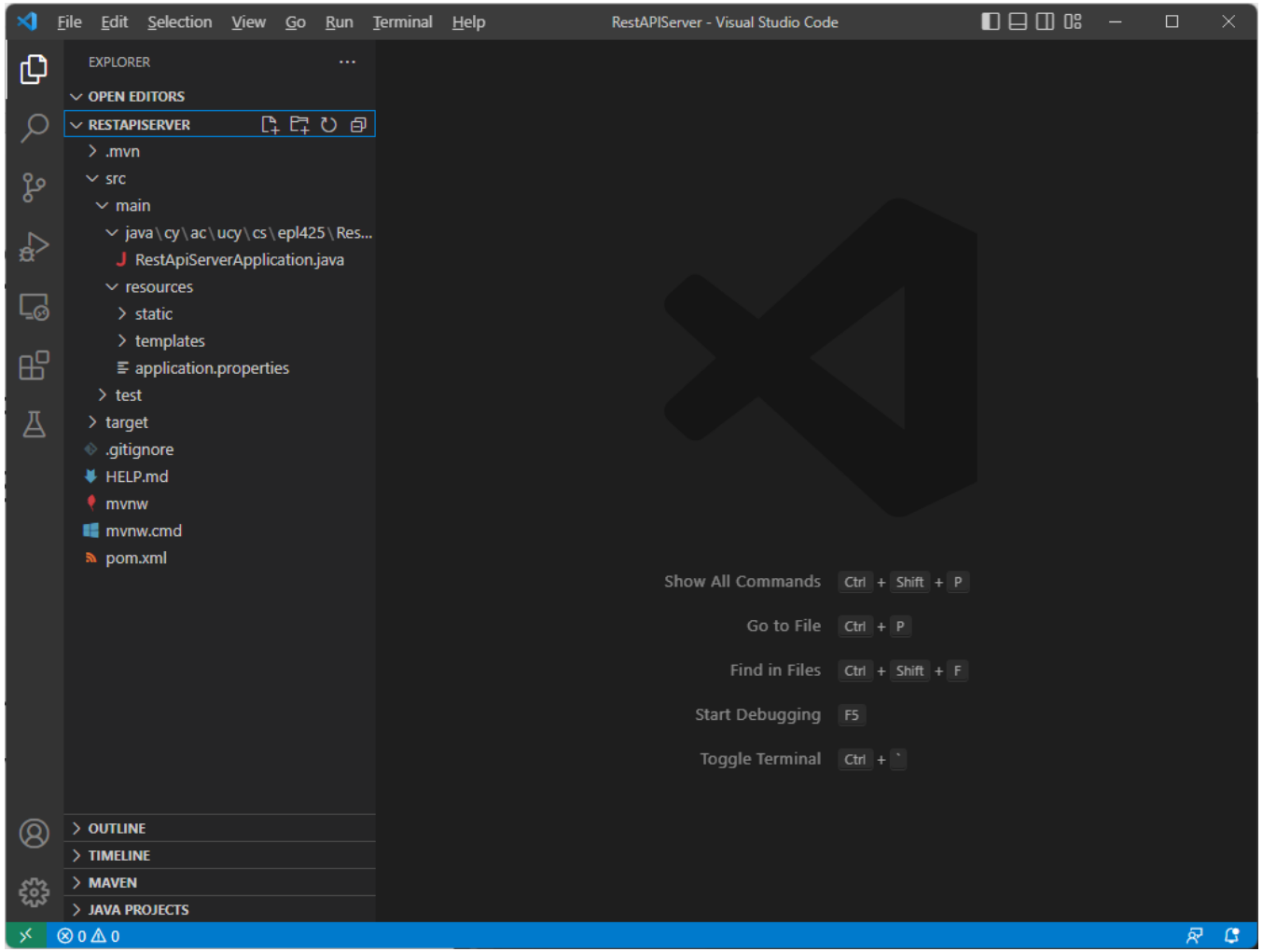
Spring Data JDBC SQL
Persist data in SQL stores with plain JDBC using Spring Data.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

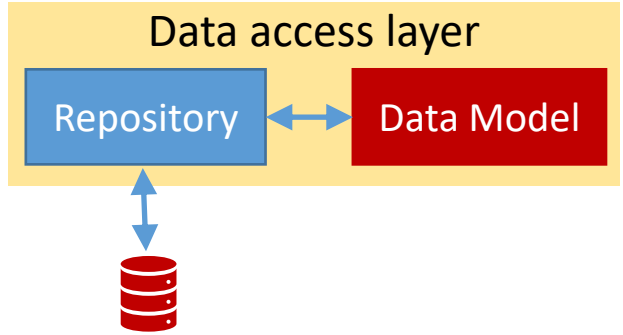
GENERATE CTRL + ⏎ EXPLORE CTRL + SPACE SHARE...

Open Spring Boot Project in VS Code

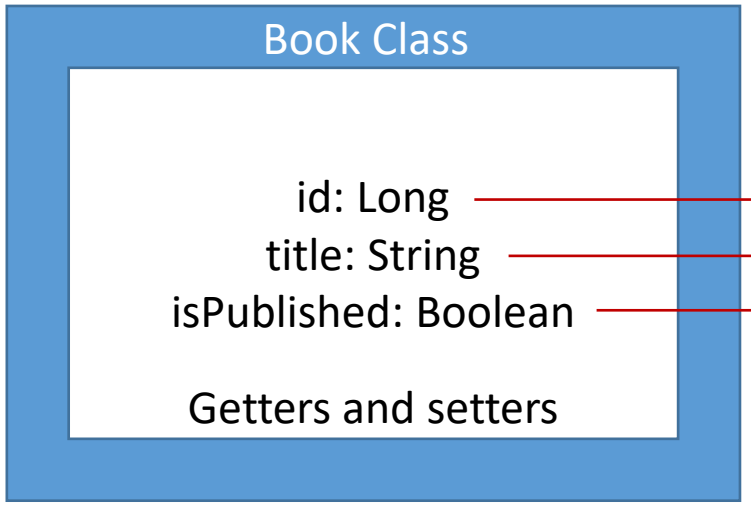
- Extract RestAPIServer.zip
  - Ensure that the RestAPIServer folder exists containing the src and target folders as well as pom.xml
- Open VS code
- Click on Explorer tab
- Click on Open RestAPIServer Folder
- Select the RestAPIServer directory



Book entity: Java class (data model) to DB table mapping

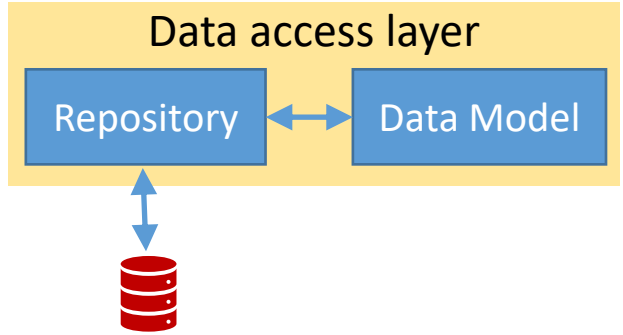


- In Spring Boot, book entity is modelled as a Java class with attributes: id, title, publishedDate. Also the class has getters and setters.
- In Database, book entity is modelled as a table




```
CREATE DATABASE `library`;
CREATE TABLE `library`.`books` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `title` VARCHAR(100) NOT NULL,
    `is_published` TINYINT(1) NOT NULL,
    PRIMARY KEY (`id`)
);
```

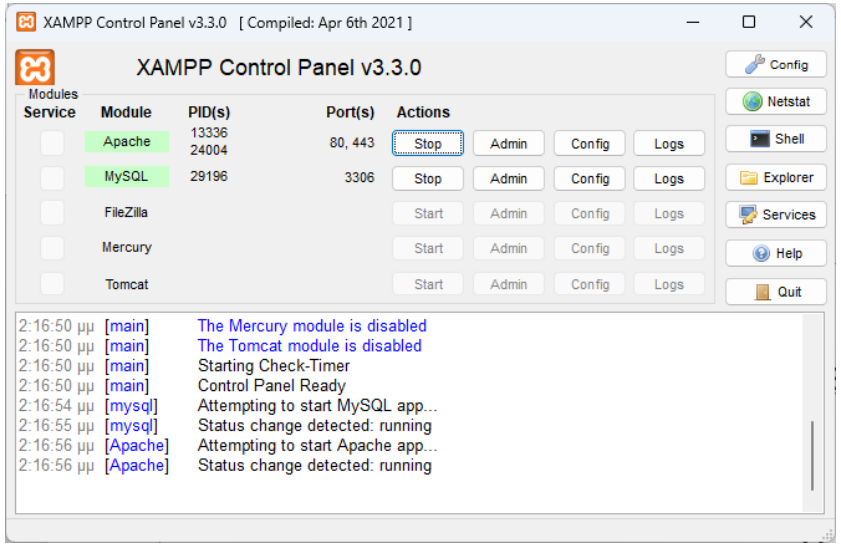
Spring Data JDBC uses, by default, a naming strategy that maps Java classes to relational database tables, and attributes to column names. By default, the Camel Case names of classes and attributes are mapped to snake case names of DB tables and columns, respectively. For example, attribute named *isPublished* is mapped to a table's column named *is_published*

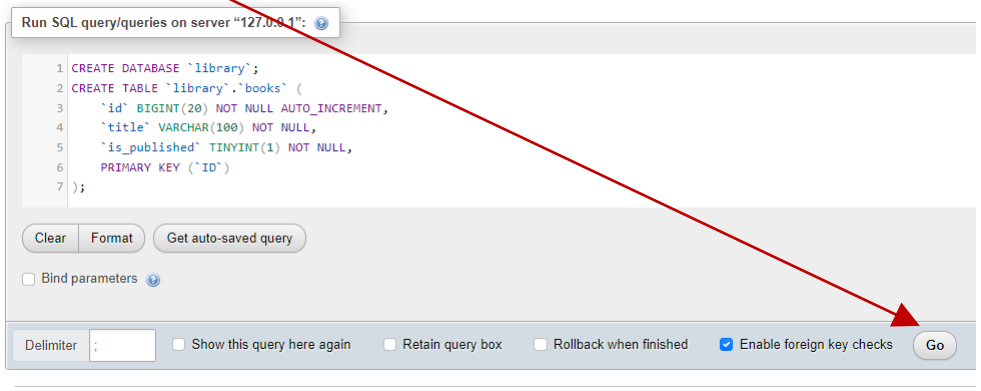


Book entity: Create DB table

- Create database and table using phpMyAdmin:
  - Launch XAMPP and start Apache and MySQL
  - Navigate to https://localhost/phpmyadmin
  - Open SQL tab
  - Copy the following script and click on Go:

```
CREATE DATABASE `library`;
CREATE TABLE `library`.`books` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `title` VARCHAR(100) NOT NULL,
    `is_published` TINYINT(1) NOT NULL,
    PRIMARY KEY (`id`)
);
```





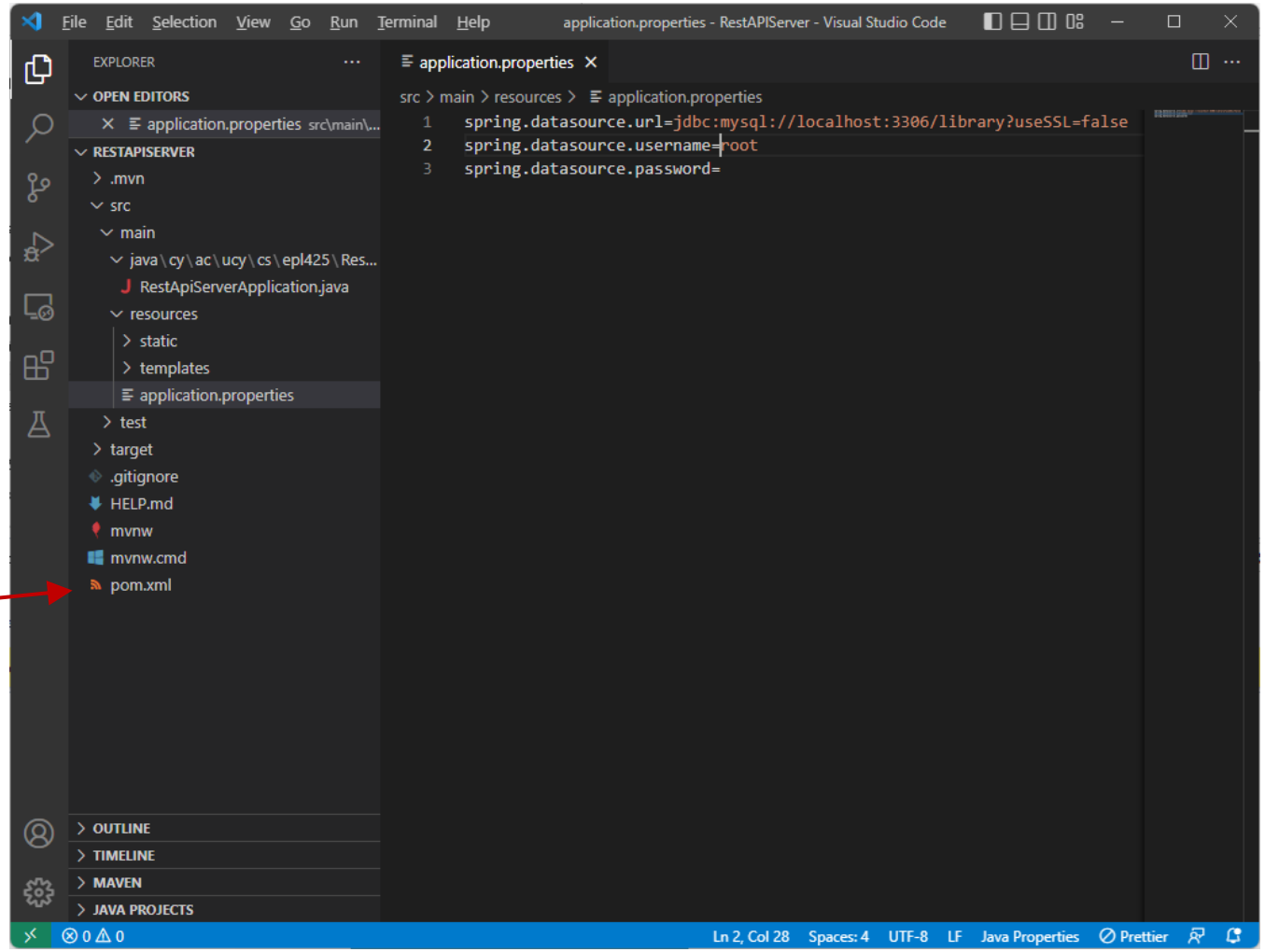
Set configuration properties

- Open resources folder
- Open application.properties file and insert the following

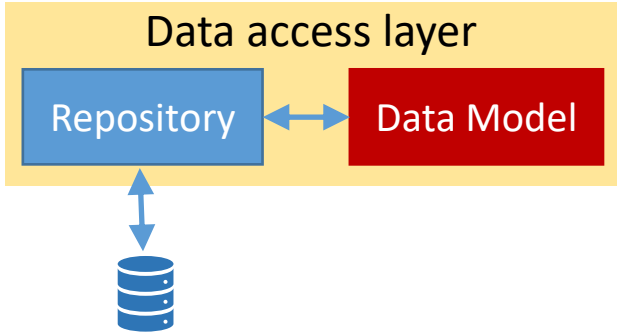
```
spring.datasource.url=jdbc:mariadb://localhost:3306/lib
rary?useSSL=false
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-
name=org.mariadb.jdbc.Driver
```

- Add one dependency to pom.xml for MariaDB (enable maven project to download related Java classes):

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
</dependency>
```

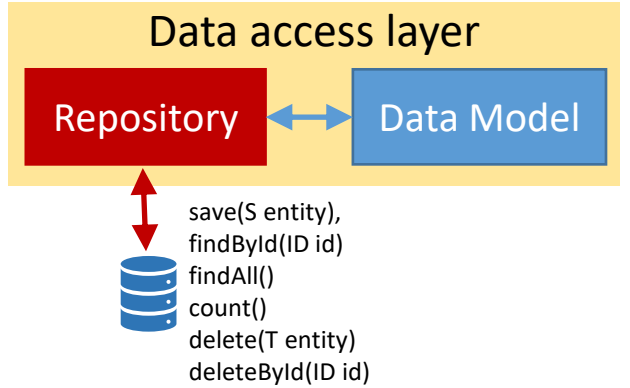


Book entity: Create Java class (data model)




```
// map Java class with books table with @Table annotation
@Table("books")
public class Book {

    // In Spring Data JDBC, the Java class is required to
    // have an @Id annotation to identify entities.
    @Id
    private Long id;

    // map attribute with column with @Column annotation
    @Column("title")
    private String title;

    @Column("is_published")
    private Boolean isPublished;

    public Book() {

    }

    public Book(String title, Boolean isPublished) {
        this.title = title;
        this.isPublished = isPublished;
    }

    // getters
    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public Boolean getIsPublished() {
        return isPublished;
    }

    // setters
    public void setTitle(String title) {
        this.title = title;
    }

    public void setIsPublished(Boolean isPublished) {
        this.isPublished = isPublished;
    }

    @Override
    public String toString() {
        return "Book [id=" + id + ", title=" + title + ",
published date=" + isPublished + "]";
    }
}
```

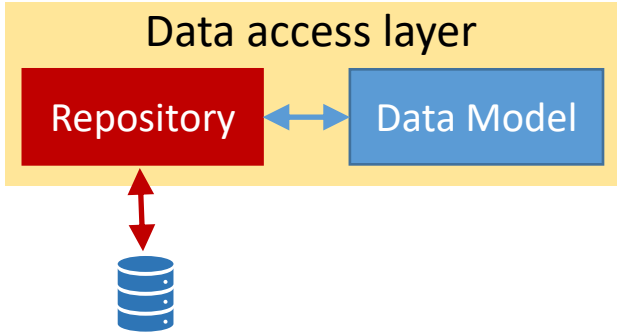
Book Repository



- Repository is used to interact with books table from the database
- BookRepository extends the CrudRepository class

```
@Repository
public interface BookRepository extends CrudRepository<Book, Long> {

}
```

- We can use CrudRepository's methods without implementing them:
  - save(S entity) : saves given entity in database
  - findById(ID id) : retrieves an entity identified by the given id
  - findAll() : returns all entities
  - count() : returns the number of entities available
  - delete(T entity) : deletes the given entity
  - deleteById(ID id) : deletes the entity with the given id



Book Repository

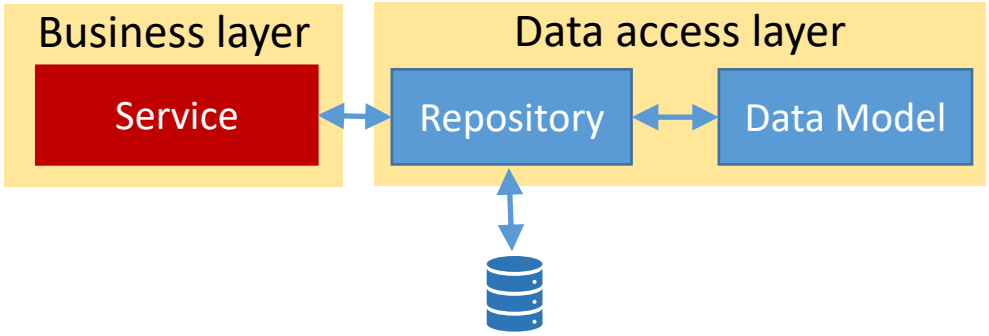
- Repository is used to interact with books table from the database
- BookRepository extends the CrudRepository class

```
@Repository
public interface BookRepository extends CrudRepository<Book, Long> {
    List<Book> findByTitleContaining(String val);
    List<Book> findByIsPublished(Boolean val);
}
```

- We can also define **additional custom** finder method **header**:
  - findByTitleContaining(String val): selects the title column
    - returns all Book entities from database where the title column contains the val value
      - Function calls this query: SELECT * FROM books WHERE title LIKE '%val%'
  - findByIsPublished(Boolean val): searches on the is_published column
    - returns all Book entities from database where the is_published column has the val value

Custom querying functions

- The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository – see more here
  - `Optional<User> findByUsername(String val);`
    - SELECT * FROM users WHERE username=val
  - `List<User> findAllBy`**`OrderBy`**`Username`**`Asc`**`();`
    - SELECT * FROM users ORDER BY username ASC
  - `List<User> findByRegistrationDate`**`Between`**`(LocalDate start, LocalDate end);`
    - SELECT * FROM users WHERE registration_date BETWEEN start AND end
  - `List<User> findByUsername`**`Containing`**`(String text);`
  - `List<User> findByUsername`**`And`**`Email(String username, String email);`
  - `List<User> findByUsername`**`ContainingAnd`**`Email`**`Containing`**`(String username, String email);`
  - `List<User> findByUsername`**`Or`**`Email(String username, String email);`
  - `List<User> findByUsername`**`IgnoreCase`**`(String username);`
  - `List<User> findByLevel`**`OrderBy`**`Username`**`Desc`**`(int level);`
  - `List<User> findByLevel`**`GreaterThan`**`Equal(int level);`
  - `List<User> findByUsername`**`Like`**`(String text);`
  - `List<User> findByUsername`**`StartingWith`**`(String start);`
  - `List<User> findByUsername`**`EndingWith`**`(String end);`
  - `List<User> findByActive(boolean active);`
  - `List<User> findByRegistrationDate`**`In`**`(Collection<LocalDate> dates);`
  - `List<User> findByRegistrationDate`**`NotIn`**`(Collection<LocalDate> dates);`



Book Service

- BookService.java class implements the business logic of the system
- Calls repository's functions to perform business logic operations

```
@Service
public class BookService {
    // Spring injects bookRepository object when BookService
    // object is created
    @Autowired BookRepository bookRepository;
    public List<Book> getAllBooks() {
        List<Book> books = new ArrayList<Book>();
        this.bookRepository.findAll().forEach(books::add);
        return books;
    }
```

Repository's function findAll() returns a list of book entities from database as an Iterate object. Using forEach, we add all books to a List of books which is returned.

```
    public Book getBookById(Long id) {
        return this.bookRepository.findById(id).get();
    }
```

Repository's function findById() returns the book entity with the given id from database. findById returns Optional, so you can get the book by get() method.

```
    public List<Book> getBooksByTitle(String title) {
        return this.bookRepository.findByTitleContaining(title);
    }
```

Repository's custom function findByTitleContaining() returns a list of book entities containing the given string in their title column from database.

```
    public List<Book> getPublishedBooks() {
        return this.bookRepository.findByIsPublished(true);
    }
```

Repository's custom function findByIsPublished() returns a list of book entities having true in their is_published column from database.

```
    public Book saveBook(Book book) {
        return this.bookRepository.save(book);
    }
```

Repository's function save() stores the given book in database.

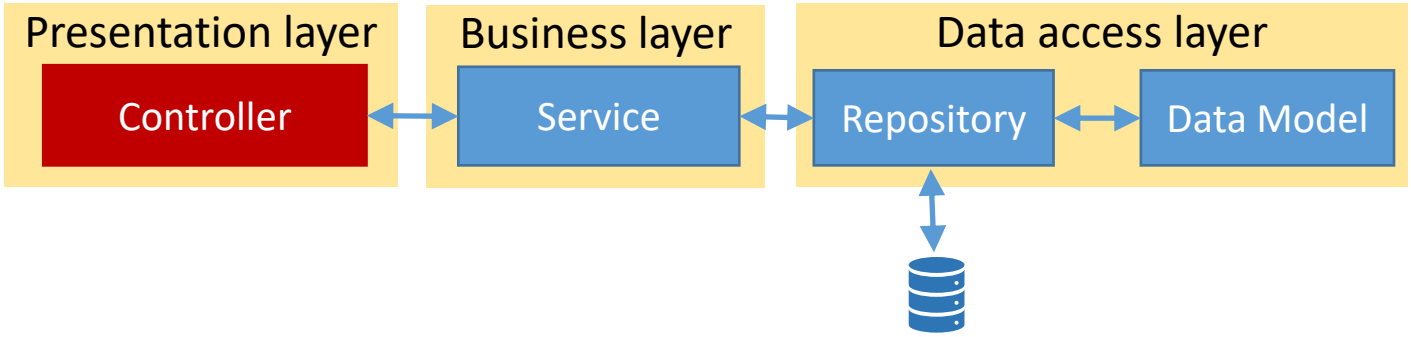
```
    public void deleteAllBooks() {
        this.bookRepository.deleteAll();
    }
```

Repository's function deleteAll() deletes all books from database.

```
    public void deleteBookById(Long id) {
        this.bookRepository.deleteById(id);
    }
}
```

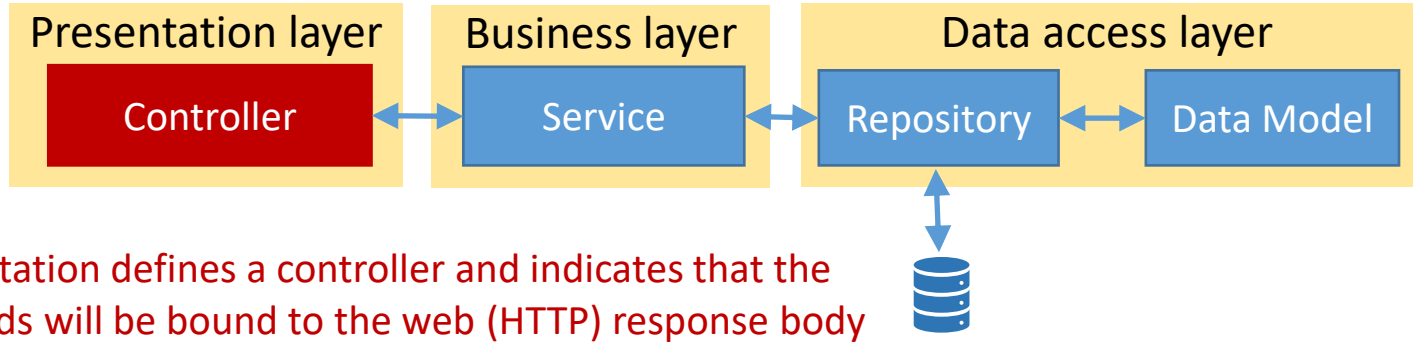
Repository's function deleteById() deletes the book with the given id from database.

Book Controller



Method	API Endpoint (URL)	Description
GET	`/api/books`	retrieve all Books
GET	`/api/books?title=[keyword]`	retrieve all Books whose title contains keyword
GET	`/api/books/published`	retrieve all published Books
GET	`/api/books/:id`	retrieve a Book by :id
POST	`/api/books`	create new Book
PUT	`/api/books/:id`	update a Book by :id
DELETE	`/api/books`	delete all books
DELETE	`/api/books/:id`	delete a Book by :id

For each API endpoint, a separate method needs to be implemented in the Controller class



Book Controller

@RestController annotation defines a controller and indicates that the return value of methods will be bound to the web (HTTP) response body

@RequestMapping annotation declares that all API endpoints' urls in BookController will start with /api

@GetMapping annotation declares that getAllBooks() handles a GET request matched with the /books

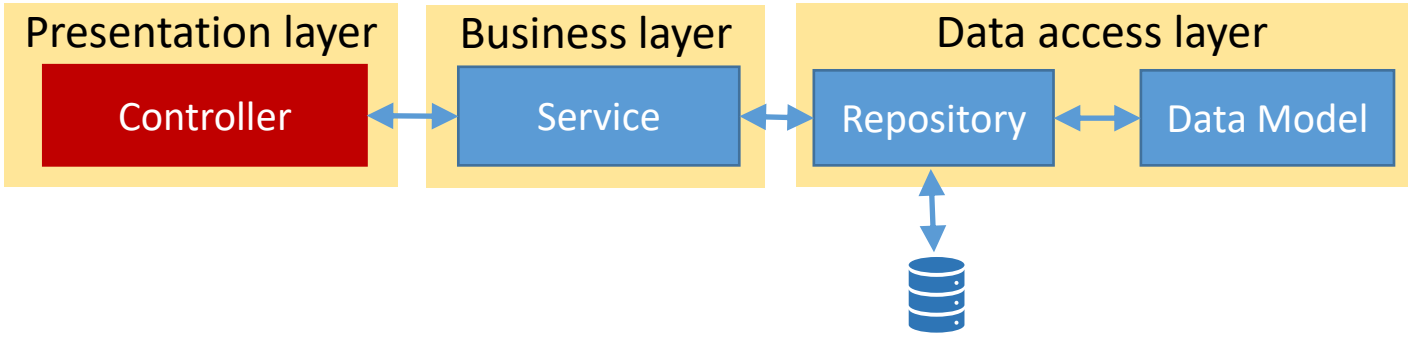
ResponseEntity represents the whole HTTP response: status code, headers, and body. This function returns a list of book objects within the body of the HTTP response message.

```
@RestController
@RequestMapping("/api")
public class BookController {

  @Autowired
  BookService bookService;

  @GetMapping("/books")
  public ResponseEntity<List<Book>> getAllBooks(@RequestParam(required = false) String title) {
    try {
      List<Book> books;

      if (title == null)
        books = bookService.getAllBooks();
      else
        books = bookService.getBooksByTitle(title);

      if (books.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
      }

      return new ResponseEntity<>(books, HttpStatus.OK);
    } catch (Exception e) {
      return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
  }
}
```

@RequestParam is used to extract query parameters (declared after ?) from the GET request. If, for example the request is `/api/books?title=test` the query parameter is `title` and its value is `test`. In order to extract the value of the title query param we need to define a function input parameter with the same name. The title query parameter is not required so the GET request `/api/books` can be also handled by the same function

- If no book entity is found, the HTTP response message "204 No Content" will be returned to API client
- Otherwise, the list of books is returned in the body of an HTTP response message "200 OK"
- In the unexpected event of an internal problem (e.g. in the communication with the database), an HTTP response message "500 Internal Server Error" is to be returned

Book Controller




```
@GetMapping("/books/{id}")
public ResponseEntity<Book> getBookById(@PathVariable("id") long id) {
  Book book = bookService.getBookById(id);

  if (book != null) {
    return new ResponseEntity<>(book, HttpStatus.OK);
  } else {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }
}
```

Handles a GET request matched with the `api/books/id` endpoint where id is within the path (@PathVariable) of the API endpoint url and needs to be an integer number.

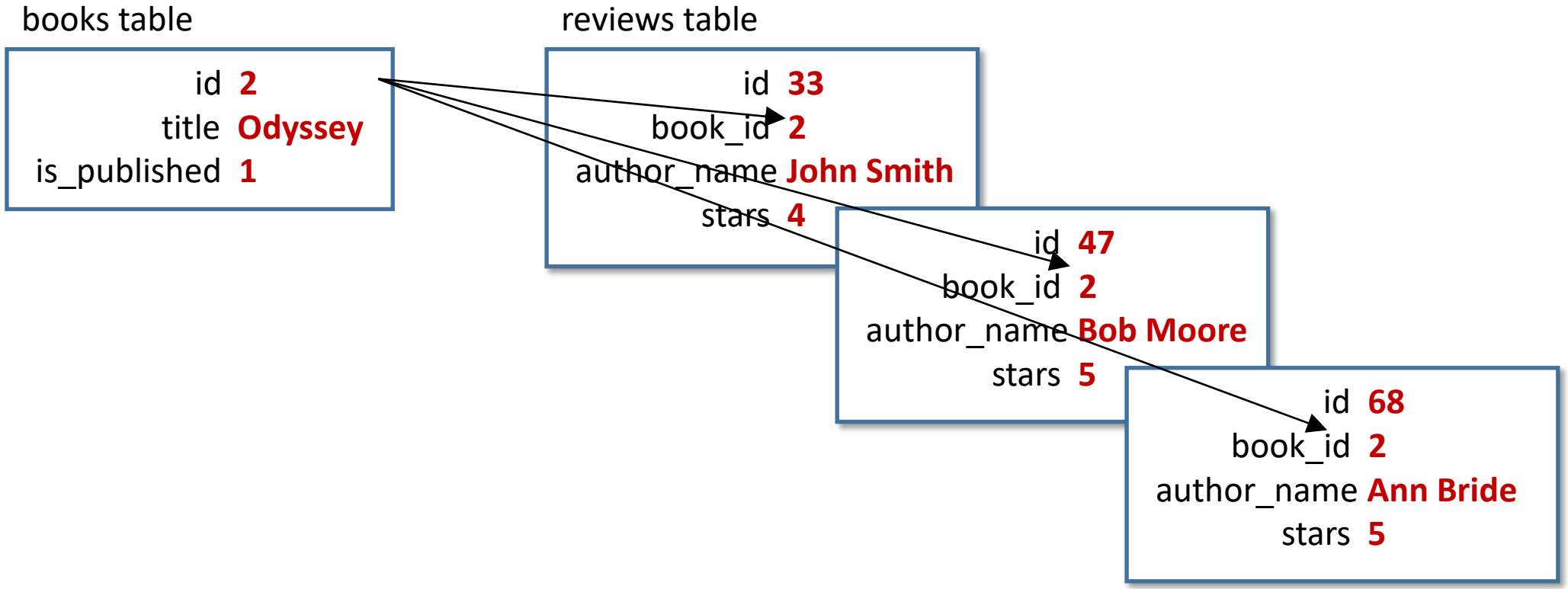
```
@PostMapping("/books")
public ResponseEntity<Book> createBook(@RequestBody Book book) {
  try {
    Book _book = bookService
        .saveBook(new Book(book.getTitle(), book.getIsPublished()));
    return new ResponseEntity<>(_book, HttpStatus.CREATED);
  } catch (Exception e) {
    return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

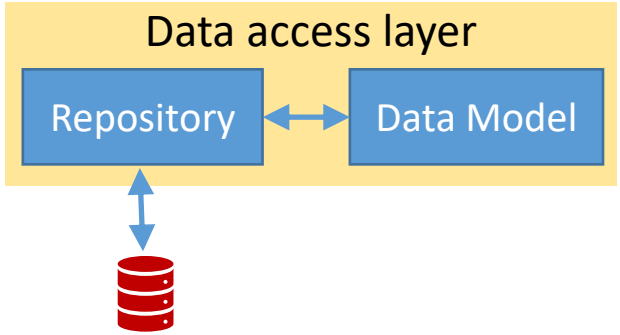
Handles a POST request matched with the `api/books` endpoint which contains a book entity as a JSON string in the body of the message (@RequestBody) as shown below:

```
{
  "title":"Orient Express",
  "isPublished": true
}
```

Add Book Reviews

- New entity needed: Review
- Each book can have one or more reviews: one-to-many relationship

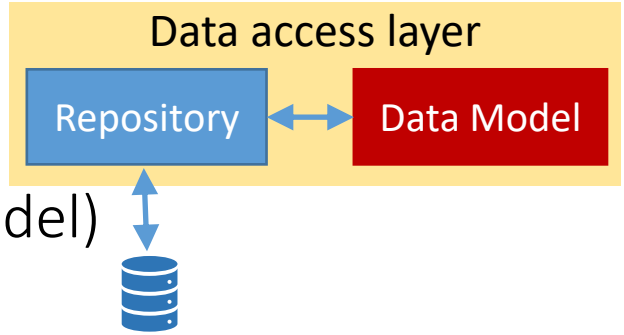




Review entity: Create DB table

- Create table using phpMyAdmin:
  - Copy the following script and click on Go:

```
CREATE TABLE `library`.`reviews` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `book_id` BIGINT(20) NOT NULL,
    `author_name` VARCHAR(30) NOT NULL,
    `title` VARCHAR(30) NOT NULL,
    `stars` INT NOT NULL,
    PRIMARY KEY (`id`),
    CONSTRAINT `review_id_fk`
     FOREIGN KEY (`book_id`)
     REFERENCES `books` (`id`)
     ON DELETE CASCADE
);
```



Review entity: Create Java class (data model)

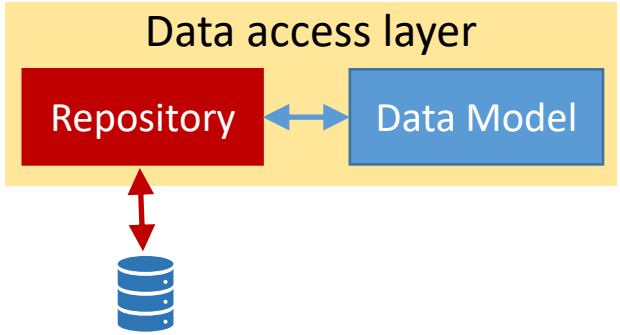
```
@Table("reviews")
public class Review {

    @Id
    private Long id;

    // Ignore the bookId when serializing object to JSON
    @JsonIgnore
    @Column("book_id")
    private Long bookId;

    @Column("author_name")
    private String authorName;

    @Column("title")
    private String title;

    @Column("stars")
    private Integer stars;

    public Review(Long bookId, ...) {
        ...
    }
    // getters
    ...

    // setters
    ...
```

Modify book data model

idColumn indicates the foreign key of the entity (review) referencing the id column of book entity

Set supports One-to-Many relationship. Creates a collection of the referenced entities.

```
@Table("books")
public class Book {
  ...

  // reference to book_id
  @MappedCollection(idColumn = "book_id")
  private Set<Review> reviews;

...
  // getters
  public Set<Review> getReviews() {
      return reviews;
  }
...

  // setters
  public void setReviews(Set<Review> reviews) {
      this.reviews = reviews;
  }

  public void setReview(Review review) {
      this.reviews.add(review);
  }
```

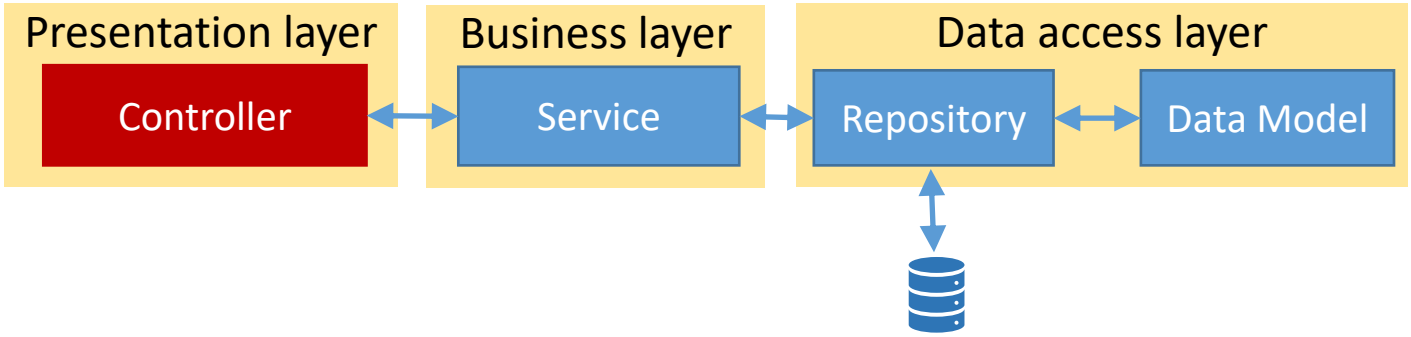


Review Repository

- ReviewRepository extends the CrudRepository

```
@Repository
public interface BookRepository extends CrudRepository<Book, Long> {
   List<Review> findByBookId(Long bookId);
   List<Review> findByBookIdAndAuthorNameContaining(Long bookId, String authorName);
   List<Review> findByBookIdAndTitleContaining(Long bookId, String val);
   List<Review> findByBookIdAndAuthorNameContainingAndTitleContaining(Long bookId,
String authorName, String title);
   List<Review> findByStarsEquals(Integer num);
   Long deleteByBookId(Long bookid);
}
```

Review Controller



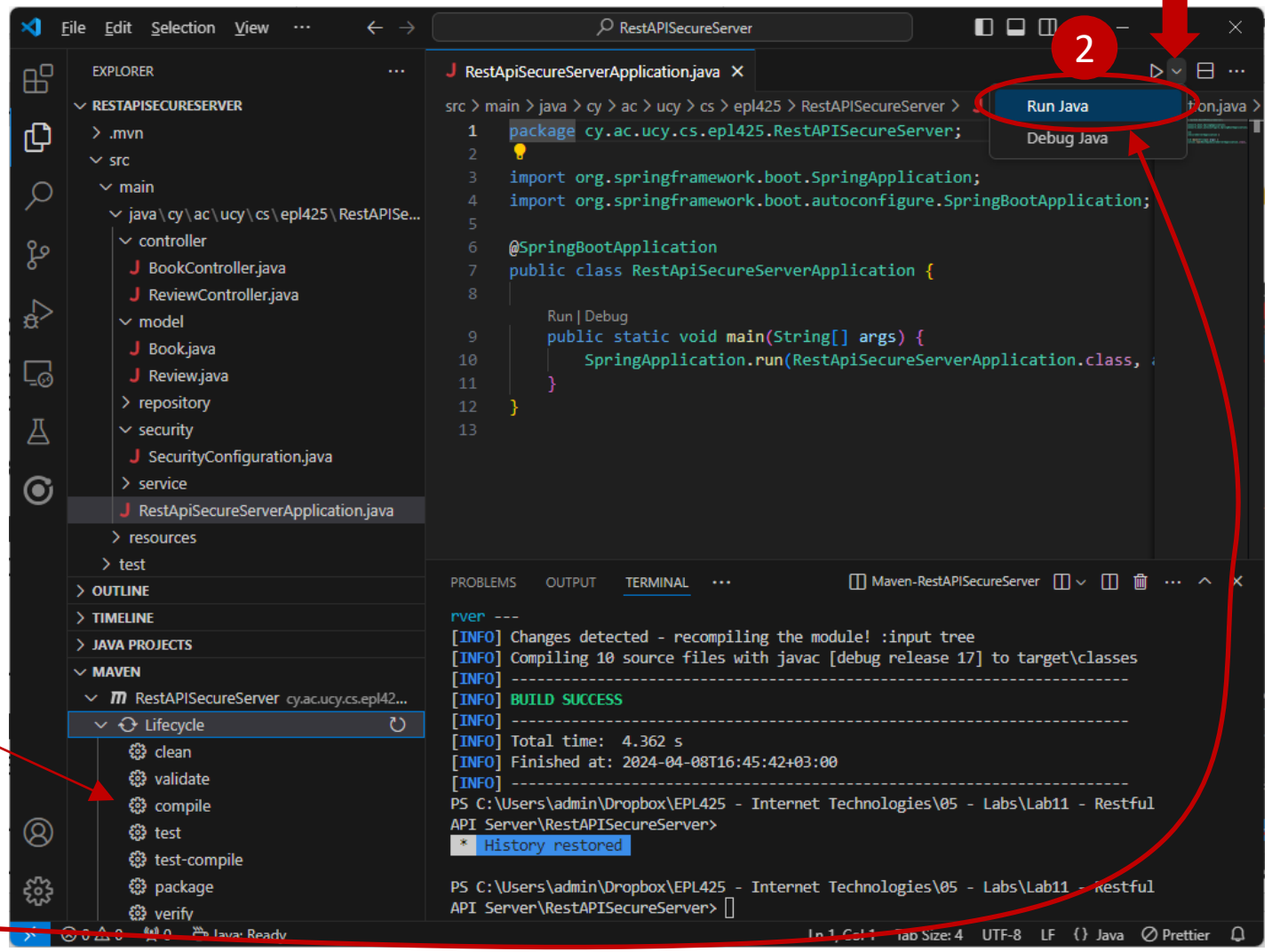
Method	API Endpoint (URL)	Description

GET	`/api/books/:bid/reviews`
	`/api/books/:bid/reviews?authorName=[keyword1]`
	`/api/books/:bid/reviews?title=[keyword2]`
		retrieve all Reviews of Book by :bid
		retrieve all Reviews by :bid whose authorName contains keyword1
		retrieve all Reviews by :bid whose title contains keyword2

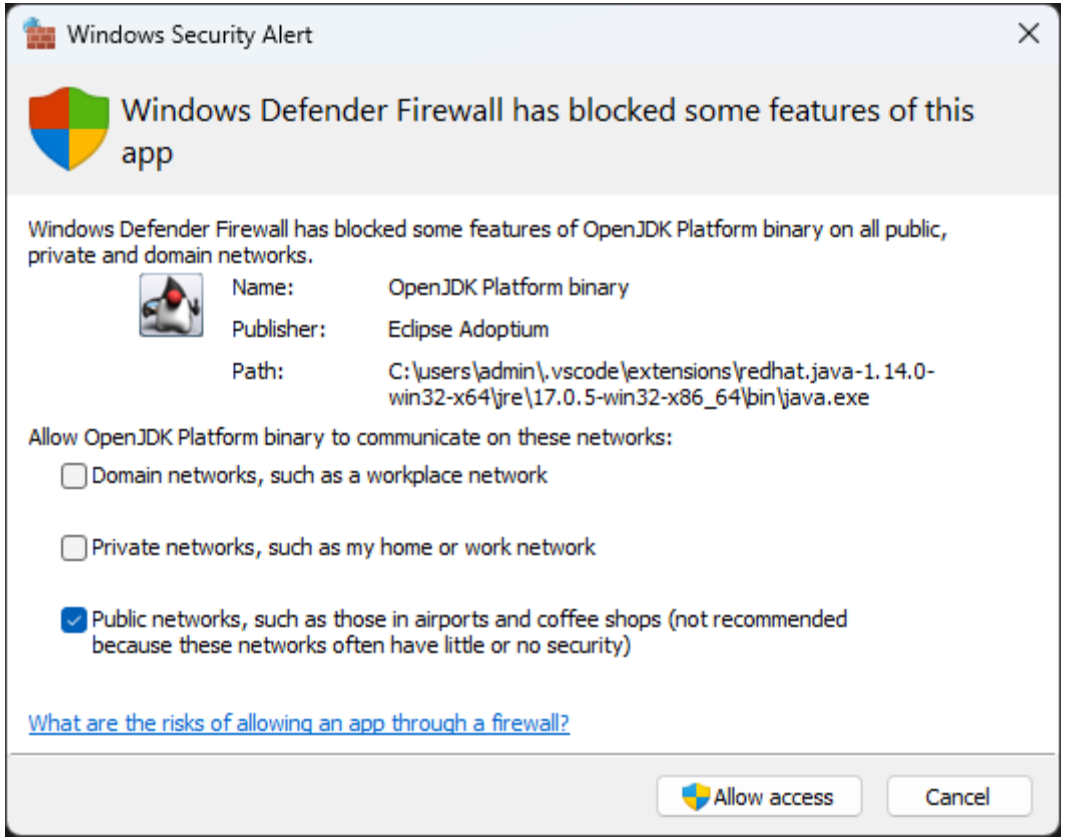
GET	`/api/books/:bid/reviews/:id`	retrieve the Review by :id of the Book by :bid
POST	`/api/books/:bid/reviews`	create new Review of the Book by :bid
PUT	`/api/books/:bid/reviews/:id`	update the Review by :id of the Book by :bid
DELETE	`/api/books/:bid/reviews`	delete all Reviews of the Book by :bid
DELETE	`/api/books/:bid/reviews/:id`	delete the Review by :id of the Book by :bid

Download, extract, compile, run application

- Download RestAPIServer.zip which contains the implementation of all API endpoints
- Extract it
- Open RestAPIServer folder via VS Code
- In Maven tab, click on compile to build the project
- Open the file RestApiServerApplication.java
- Run the project (click on Run Java, not Run Code if it exists)



Allow RESTful API Server to be accessible



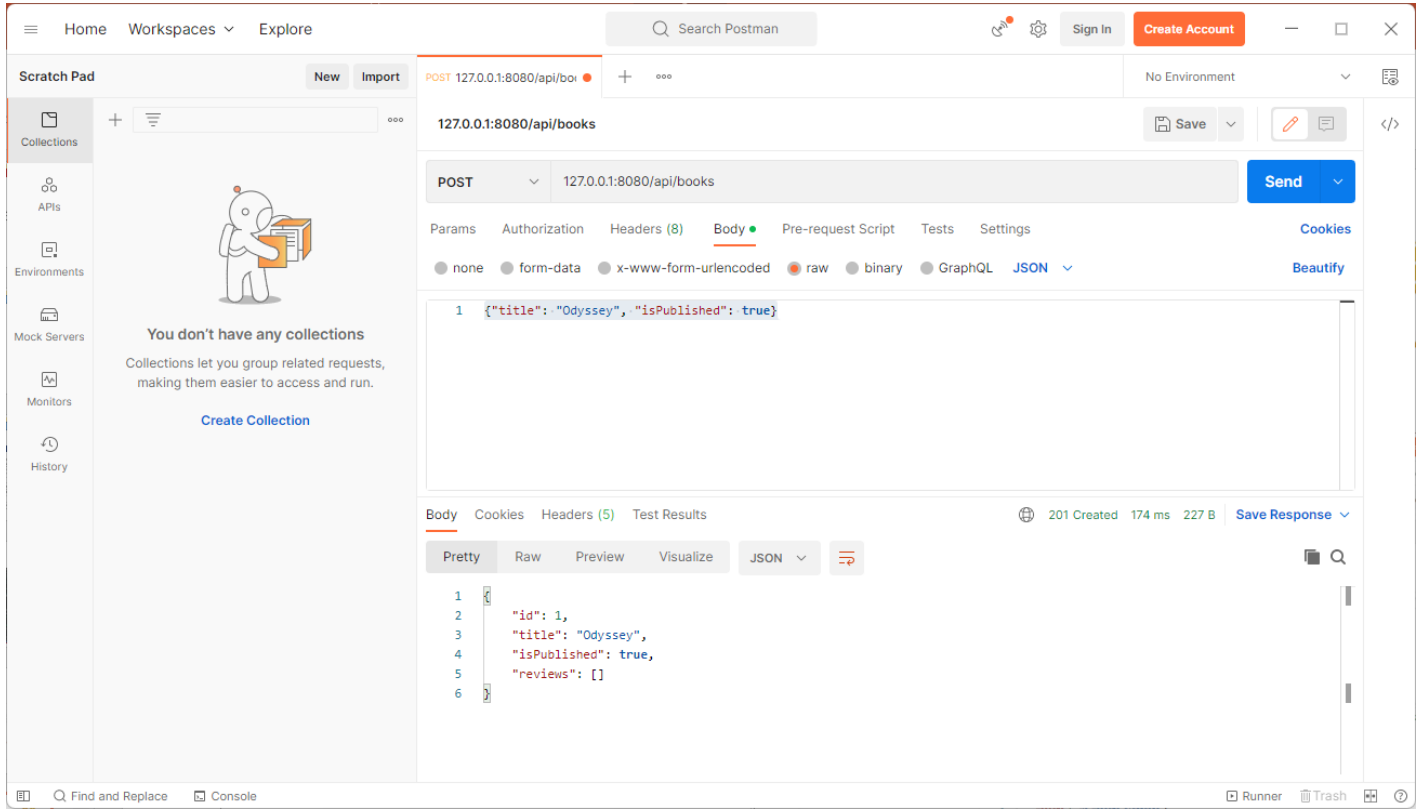
Embedded Application Server

- When running the application, Spring Boot starts up an embedded application server (servlet container), an Apache Tomcat instance by default, to host the RESTful API
  - We can use another servlet container such as Jetty by modifying pom.xml
- Test RESTful API endpoints by opening up a browser or a RESTful API client such as Postman and access the endpoint urls such as: http://localhost:8080/api/books

Use Postman to test RESTful API

- Create a book:
  - Send a POST message to `localhost:8080/api/books`
  - In message body provide a raw string in JSON format describing the entity to be created:

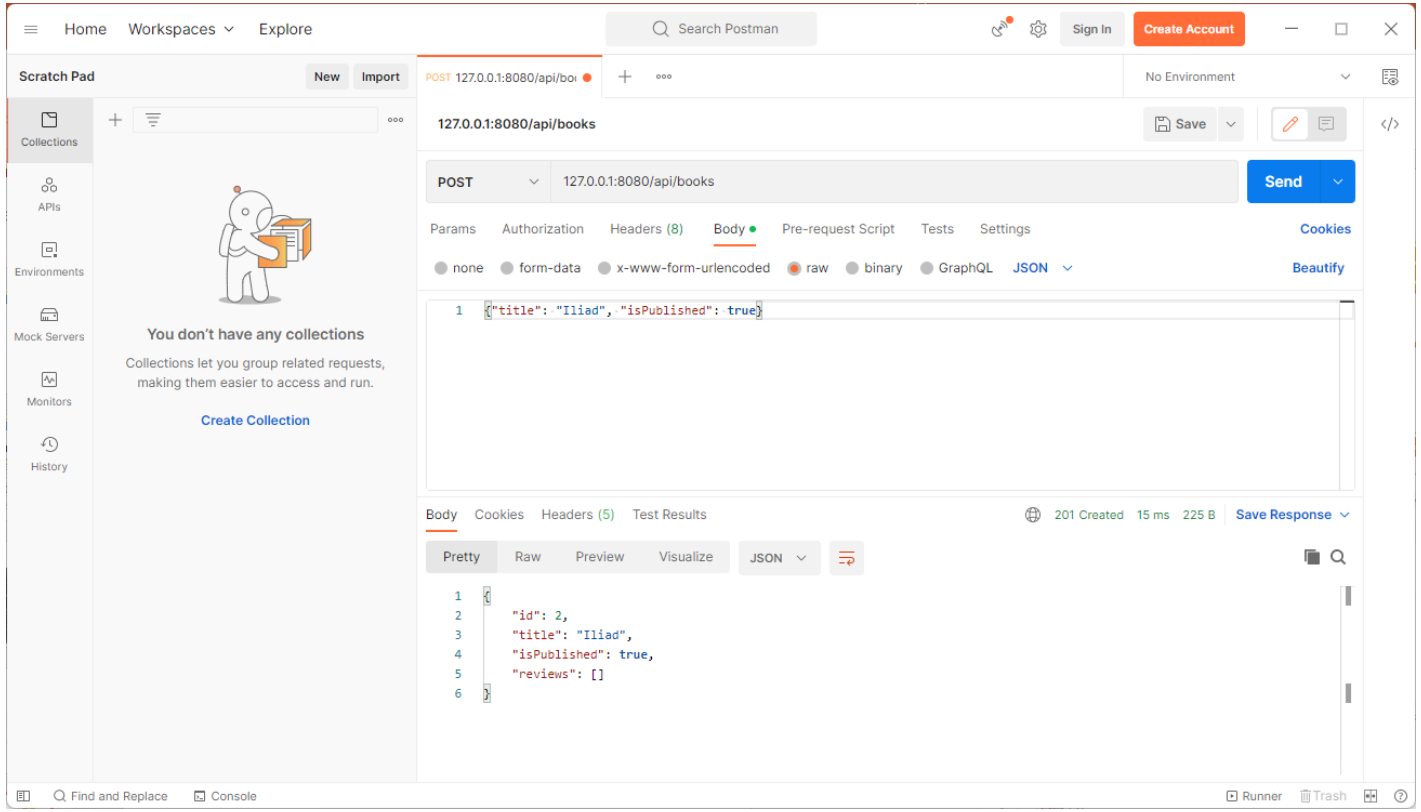
```
{"title": "Odyssey", "isPublished": true}
```



Use Postman to test RESTful API

- Create another book:
  - Send a POST message to `localhost:8080/api/books`
  - In message body provide a raw string in JSON format describing the entity to be created:

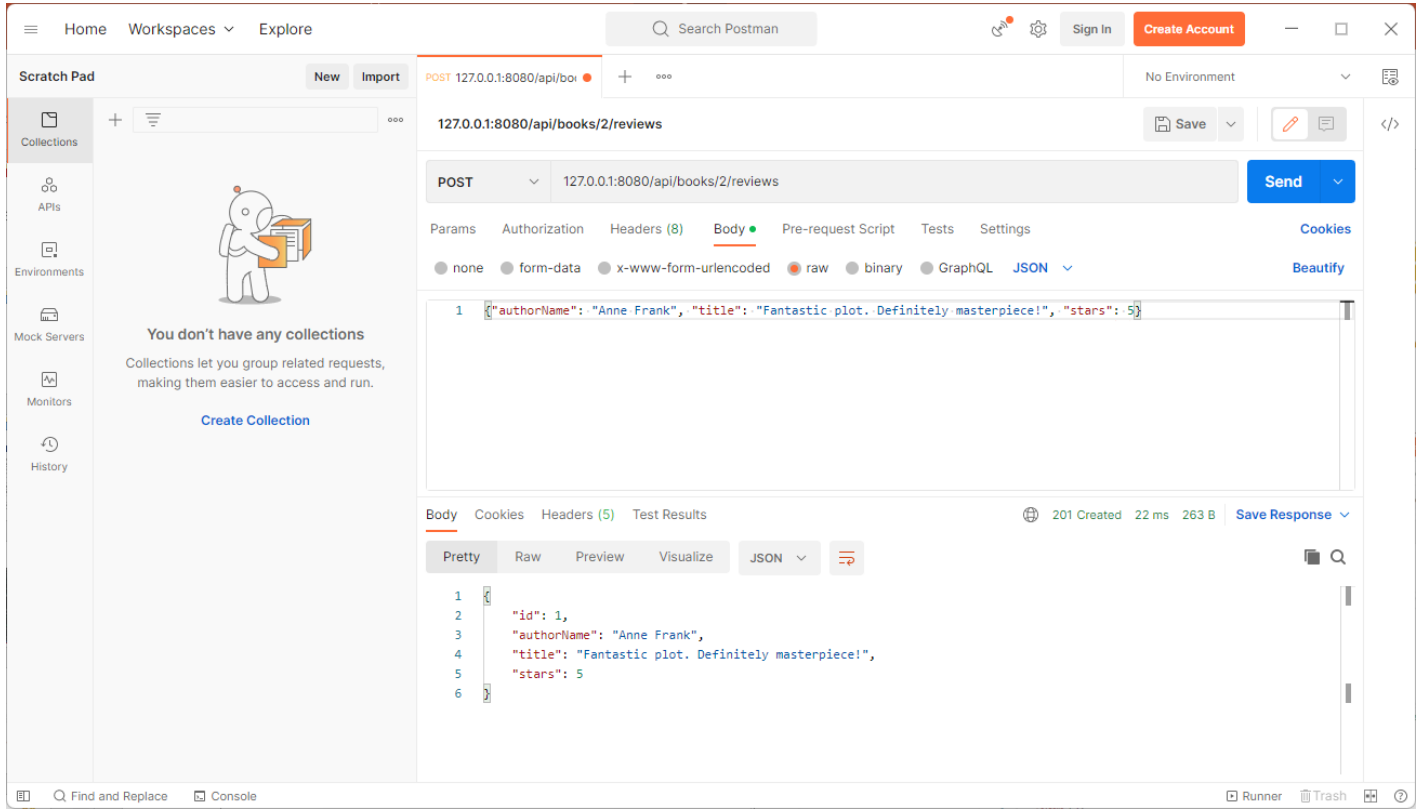
```
{"title": "Iliad", "isPublished": true}
```



Use Postman to test RESTful API

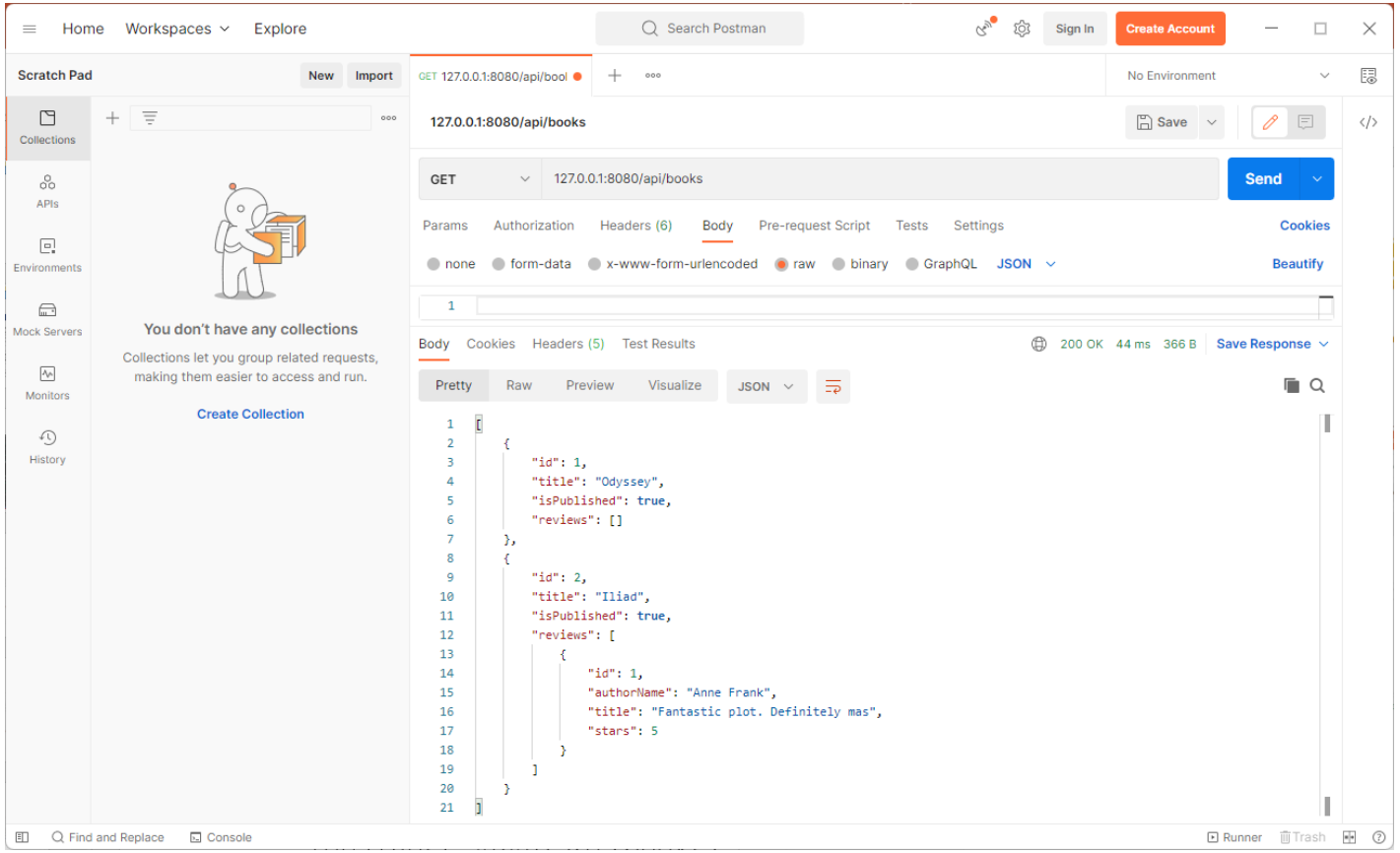
- Create a review for the book with id = 2:
  - Send a POST message to `localhost:8080/api/books/2/reviews`
  - In message body provide a raw string in JSON format describing the entity to be created:

```
{"authorName": "Anne Frank", "title": "Fantastic plot. Definitely masterpiece!", "stars": 5}
```



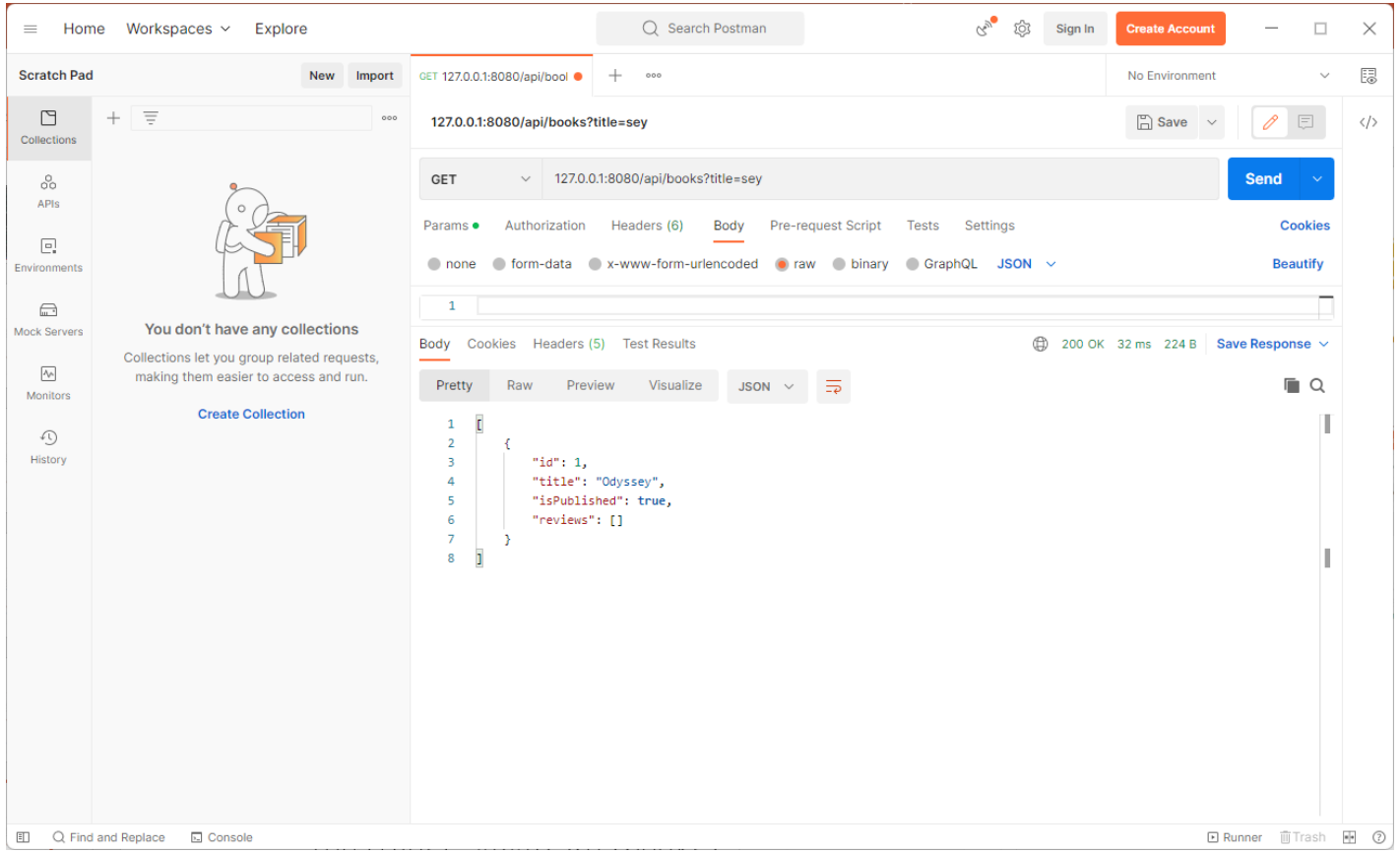
Use Postman to test RESTful API

- Retrieve all books
  - Send a GET message to `localhost:8080/api/books`
- Retrieve all books whose titles contain the phrase sey
  - Send a GET message to `localhost:8080/api/books?title=sey`
- Retrieve all reviews for the book with id = 2:
  - Send a GET message to `localhost:8080/api/books/2/reviews`



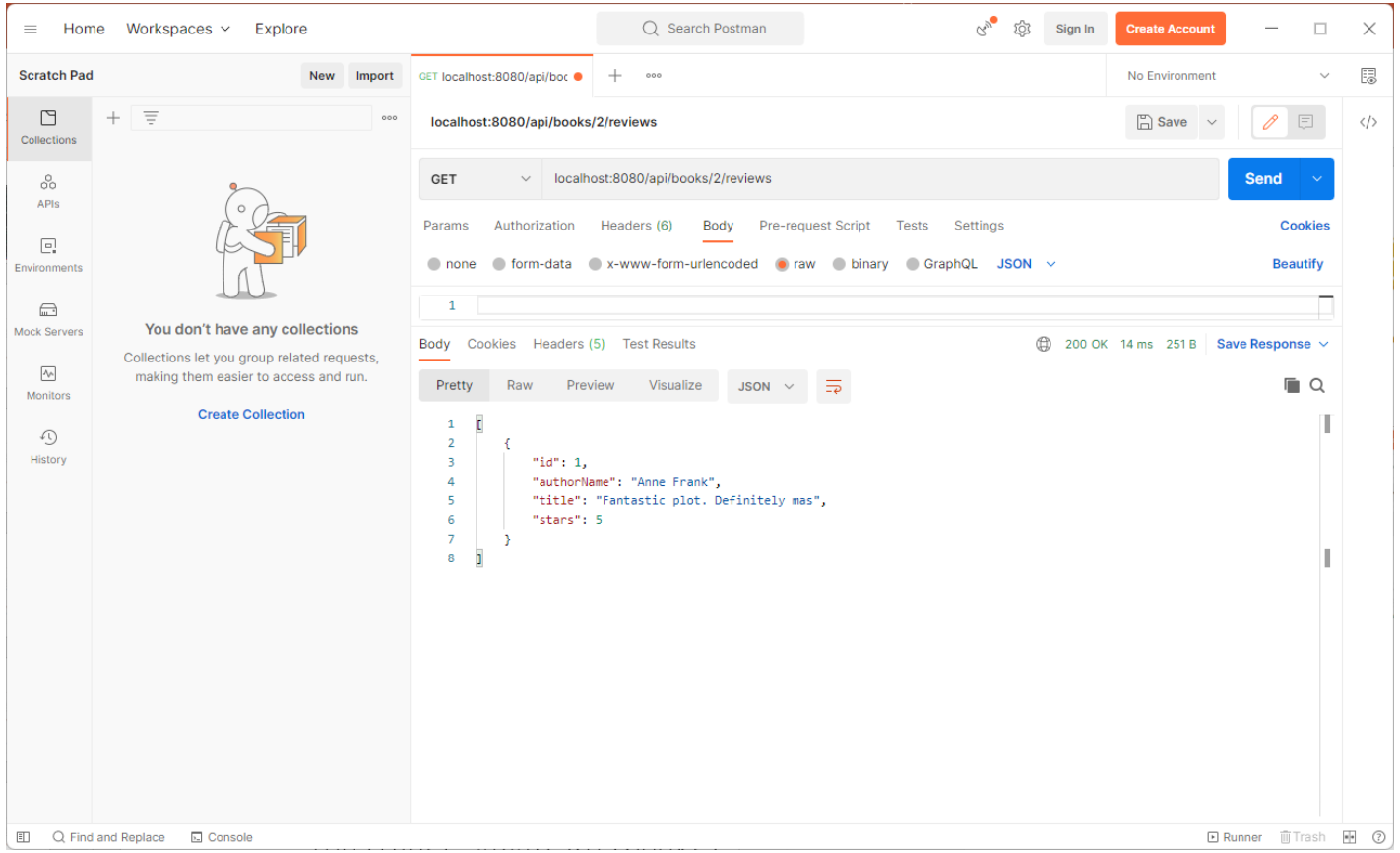
Use Postman to test RESTful API

- Retrieve all books
  - Send a GET message to `localhost:8080/api/books`
- Retrieve all books whose titles contain the phrase sey
  - Send a GET message to `localhost:8080/api/books?title=sey`
- Retrieve all reviews for the book with id = 2:
  - Send a GET message to `localhost:8080/api/books/2/reviews`



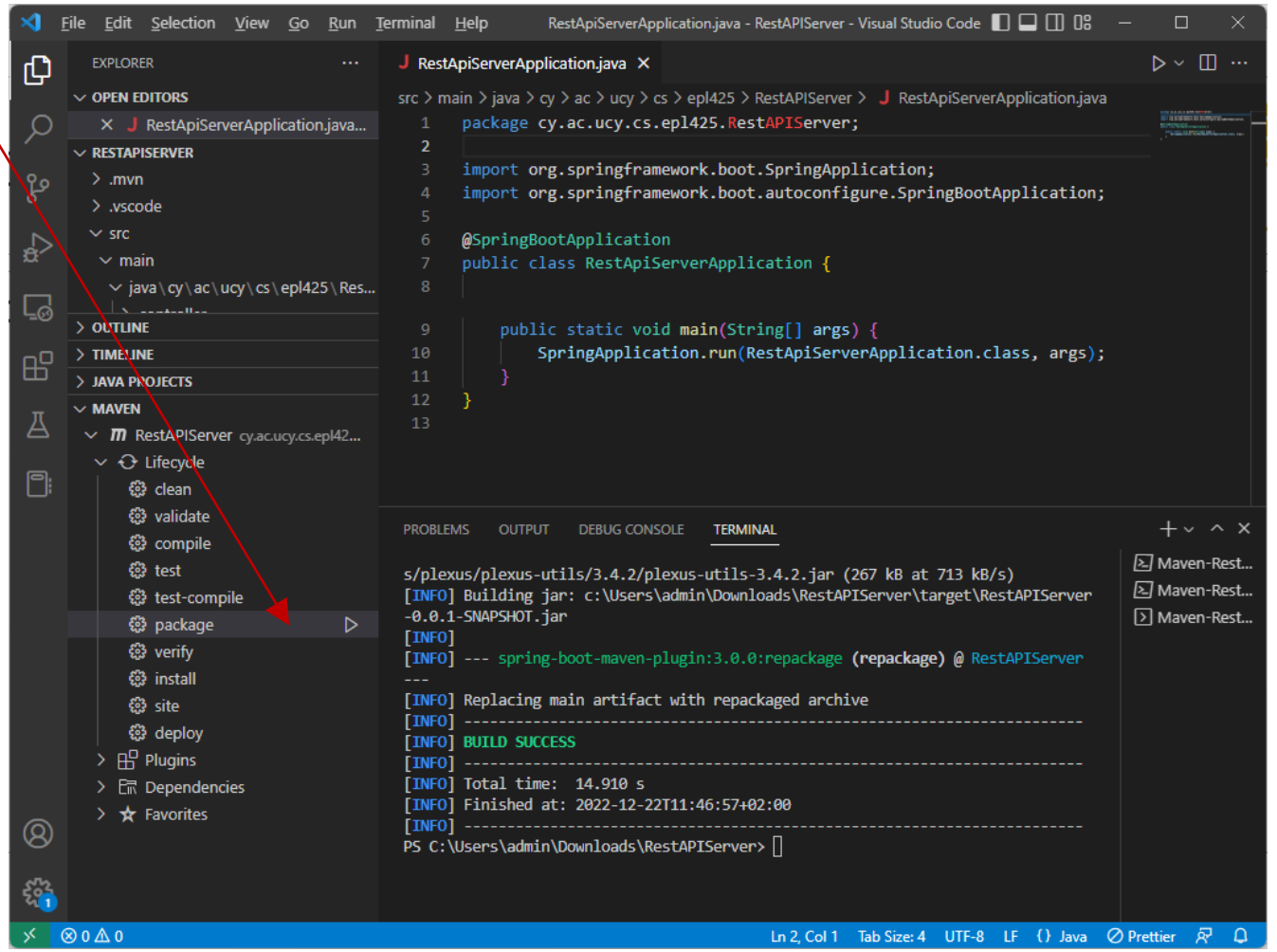
Use Postman to test RESTful API

- Retrieve all books
  - Send a GET message to `localhost:8080/api/books`
- Retrieve all books whose titles contain the phrase sey
  - Send a GET message to `localhost:8080/api/books?title=sey`
- Retrieve all reviews for the book with id = 2:
  - Send a GET message to `localhost:8080/api/books/2/reviews`



Package and deploy

- In Maven tab, execute package command to create a .jar file
  - .jar includes all dependencies and application server
- Open RestAPIServer/target folder to locate the .jar file
- You can launch the RESTful API from command line (cmd) using the following command: `java -jar myfile.jar`



Securing your RESTful API

- Sooner or later everyone needs to add security to a RESTful API
- In Spring ecosystem this is performed with the help of the Spring Security library
- Spring Security is a set of servlet filters that help you add authentication and authorization to your web application

Authentication & Authorization

- Authentication
  - Verifies the identity of a user. Typically done with a username & password check
    - Other available authentication mechanisms provided by Spring Security are shown here.
  - If user not authenticated, **HTTP 401 Unauthorized** response message is issued
- Authorization
  - Determines user rights: verifies what each user has access to
    - In simpler applications, authentication might be enough: As soon as a user authenticates, can access every part of an application
    - Most applications have the concept of permissions (or roles) e.g.: simple users who have access to read resources and administrators who have access to create/update/remove resources
  - If user doesn't have access permission, **HTTP 403 Forbidden** response message is issued

Servlet filters

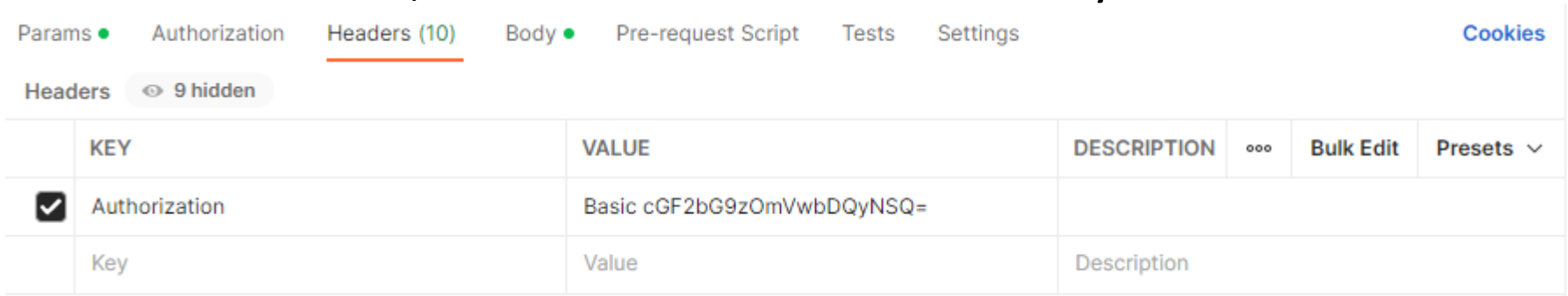
- Filters are placed *in front* of @RestController servlets and are configured to authenticate and authorize every incoming HTTP request before it hits the servlet
- A chain of filters can be used to handle every incoming request to:
  1. Extract a username/password from the request. It could be via a Basic Auth HTTP Header (**default security mechanism**, see next slide), or login form fields, or a cookie, etc.
  2. Validate that username/password combination against *something*, like a database, an in-memory (RAM) storage, etc. [Authentication filter]
  3. Allow access to resources based on user role [Authorization filter]

Default filter: Basic Authentication

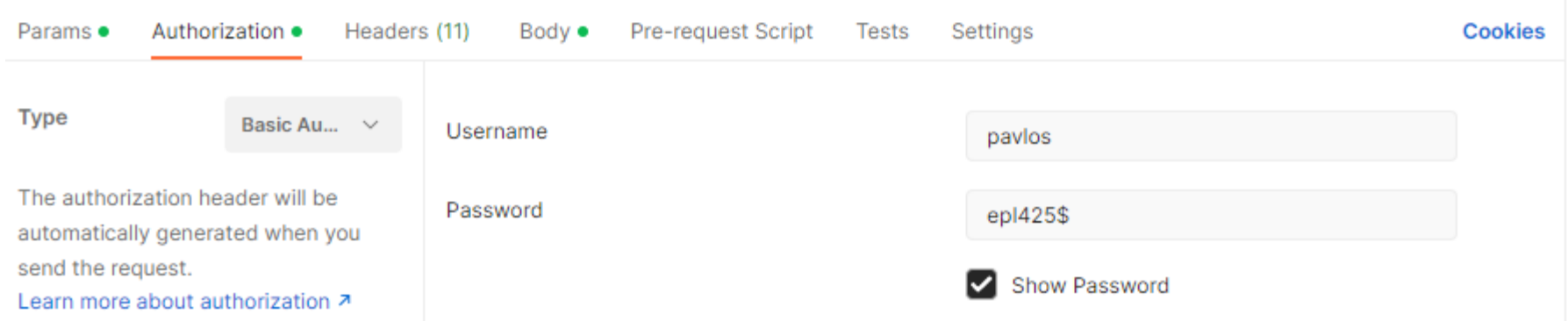
- Simplest method of securing HTTP requests through a special req. header: `Authorization: Basic <credentials>`
- API client can generate the `credentials` token, using username and password (given by API server), joined by the semicolon character and encode the resulting string with Base64
  - Example: If username is `pavlos` and password is `epl425$` the credentials string is `pavlos:epl425$`. When we encode it with Base64 we get `cGF2bG9zOmVwbDQyNSQ=` The header's value is: `Authorization: Basic cGF2bG9zOmVwbDQyNSQ=`
- API clients need to provide this header field in every request

Basic Authentication when sending a request using Postman

- Add the header manually
  - Go to headers tab, set "*Authorization*" as the key and the credentials token as value



- Or use the Authorization tab (recommended)
  - select "*Basic Auth*" as the authorization type and insert username password



Add Spring Security to secure your Restful API

- Spring Security can be added to Spring Boot project via pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

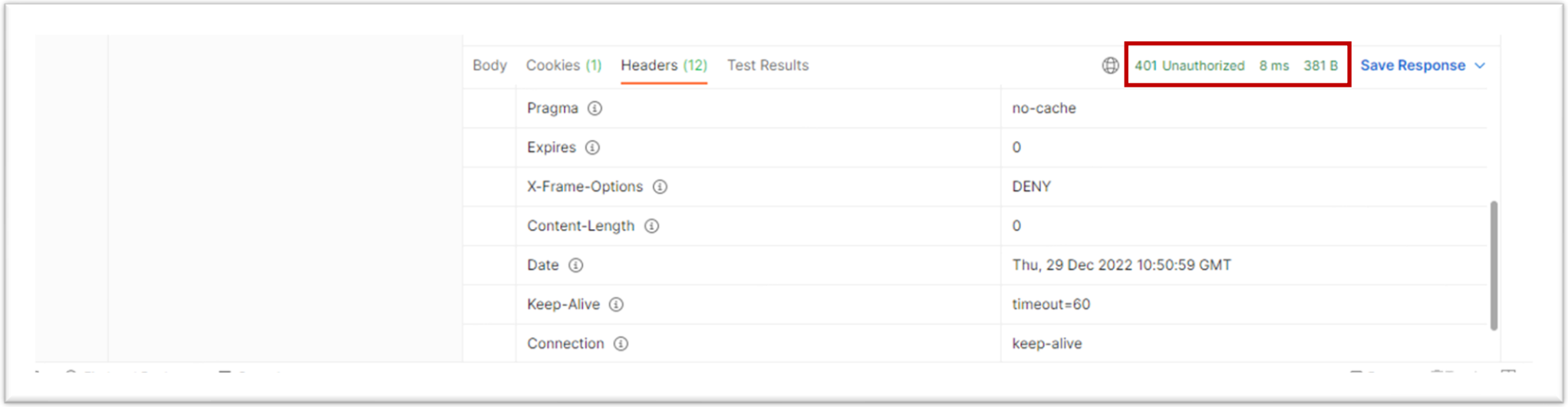
- … then all API endpoints will not be accessible anymore
- When you re-run the web application, on VSCode terminal you can see (among other) the following message:

```
Using generated security password: e0901288-da22-4bde-b73f-13d93cc01d8d

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

- This is the auto-generated password for the Basic Authentication scheme
- Is there any username? Yes, the default username is **user**

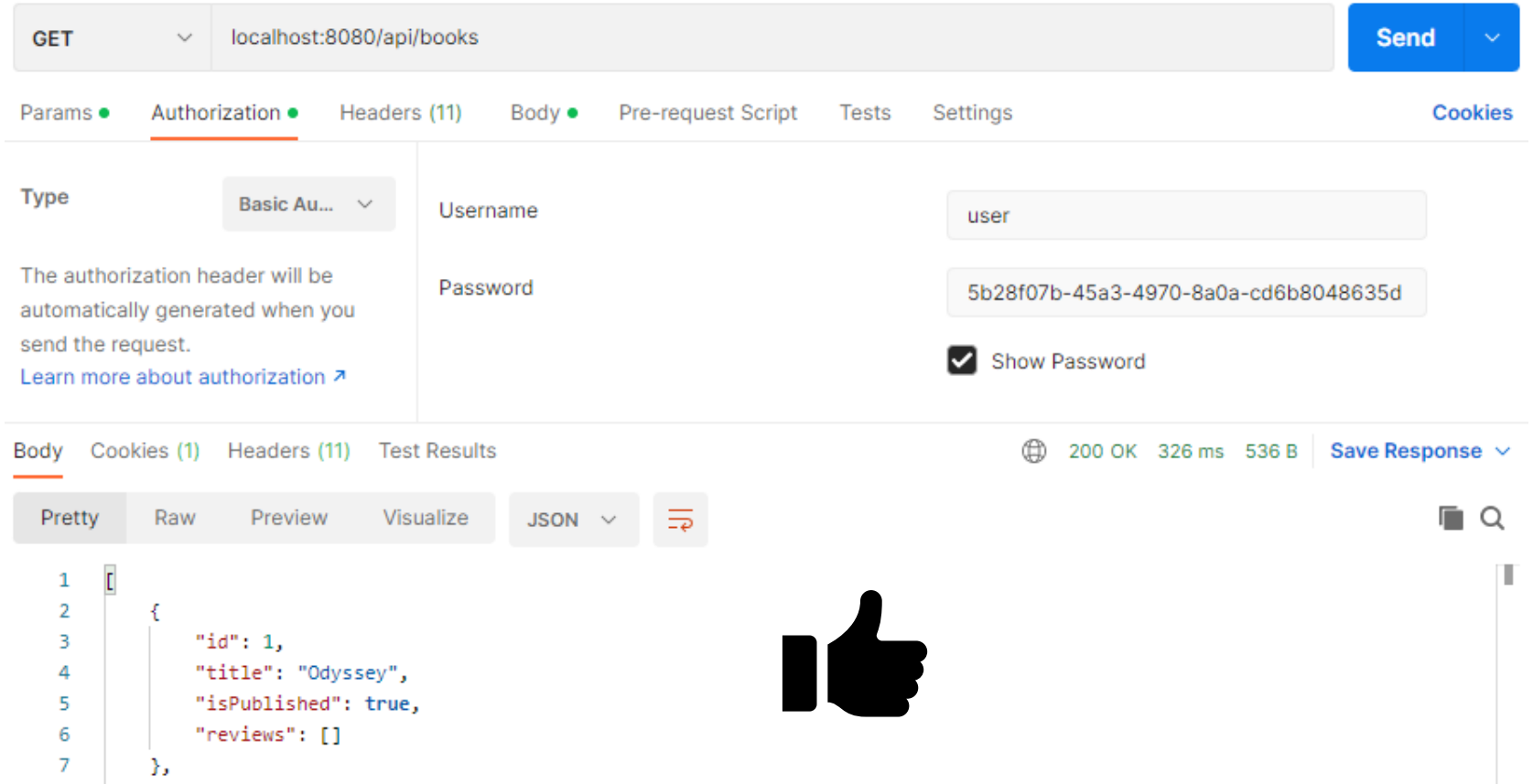
Securing your RESTful API

- When you try to access: `localhost:8080/api/books`
- HTTP 401 Unauthorized response code returned
  - Indicates that the client request has not been completed because it lacks valid authentication credentials (username and password) for the requested resource.



Securing your RESTful API

- Access API endpoint `localhost:8080/api/books` by taking into account the username (user) and the generated password



Overriding default credentials

- We can override the default username and auto-generated password provided by the Spring Security mechanism
- Custom user credentials can be set in any of the following ways:
  1. manually via the `application.properties` file
     - `spring.security.user.name=pavlos`
     - `spring.security.user.password=epl425$`
  2. using a dedicated Java class that retrieves credentials from:
     a) memory (Java variables) or
     b) database

Credentials in Memory

In Spring, the objects that form the backbone of your application and that are managed by Spring are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by Spring and it is globally available to the entire application.

- userDetailsManager() method **creates in-memory** users
  - To start with we have added two users: **employee** and **manager**
- passwordEncoder() returns an instance of BCryptPasswordEncoder which is used to encode the password

```
@Bean
public InMemoryUserDetailsManager userDetailsManager() {
    List<UserDetails> userDetailsList = new ArrayList<>();
    userDetailsList.add(User
        .withUsername("employee")
        .password(passwordEncoder().encode("epl425$"))
        .roles("EMPLOYEE")
        .build());

    userDetailsList.add(User
        .withUsername("manager")
        .password(passwordEncoder().encode("password"))
        .roles("EMPLOYEE", "MANAGER")
        .build());

    return new InMemoryUserDetailsManager(userDetailsList);
}

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Adding roles

- userDetailsManager() method creates in-memory users **with specific roles** in order to enable authorization
  - employee is assigned the **EMPLOYEE** role and manager both **EMPLOYEE** and **MANAGER** roles

```
@Bean
public InMemoryUserDetailsManager userDetailsManager() {
   List<UserDetails> userDetailsList = new ArrayList<>();
   userDetailsList.add(User
        .withUsername("employee")
        .password(passwordEncoder().encode("epl425$"))
        .roles("EMPLOYEE")
        .build());

   userDetailsList.add(User
        .withUsername("manager")
        .password(passwordEncoder().encode("password"))
        .roles("EMPLOYEE", "MANAGER")
        .build());

   return new InMemoryUserDetailsManager(userDetailsList);
}

@Bean
public PasswordEncoder passwordEncoder() {
   return new BCryptPasswordEncoder();
}
```

Security Filter Chain

- How does Spring Security know that we want to **require all users to be authenticated**?
- How does Spring Security know we want to **support Basic authentication**?
- How does Spring Security know where to **find user credentials for user authentication**?
- How does Spring Security know how to **authorize access to content based on the different roles**?
- Actually, there is a configuration class (called SecurityFilterChain) that is being invoked behind the scenes.

Security Filter Chain **default** configuration

- Ensures that any request to our application requires the user to be authenticated
- Lets users authenticate with form based login
- Lets users authenticate with HTTP Basic authentication

```
@Bean
public SecurityFilterChain
filterChain(HttpSecurity http) throws Exception {
    http
            .authorizeRequests(authorize -> authorize
                    .anyRequest().authenticated()
            )
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults())
);
    return http.build();
}
```

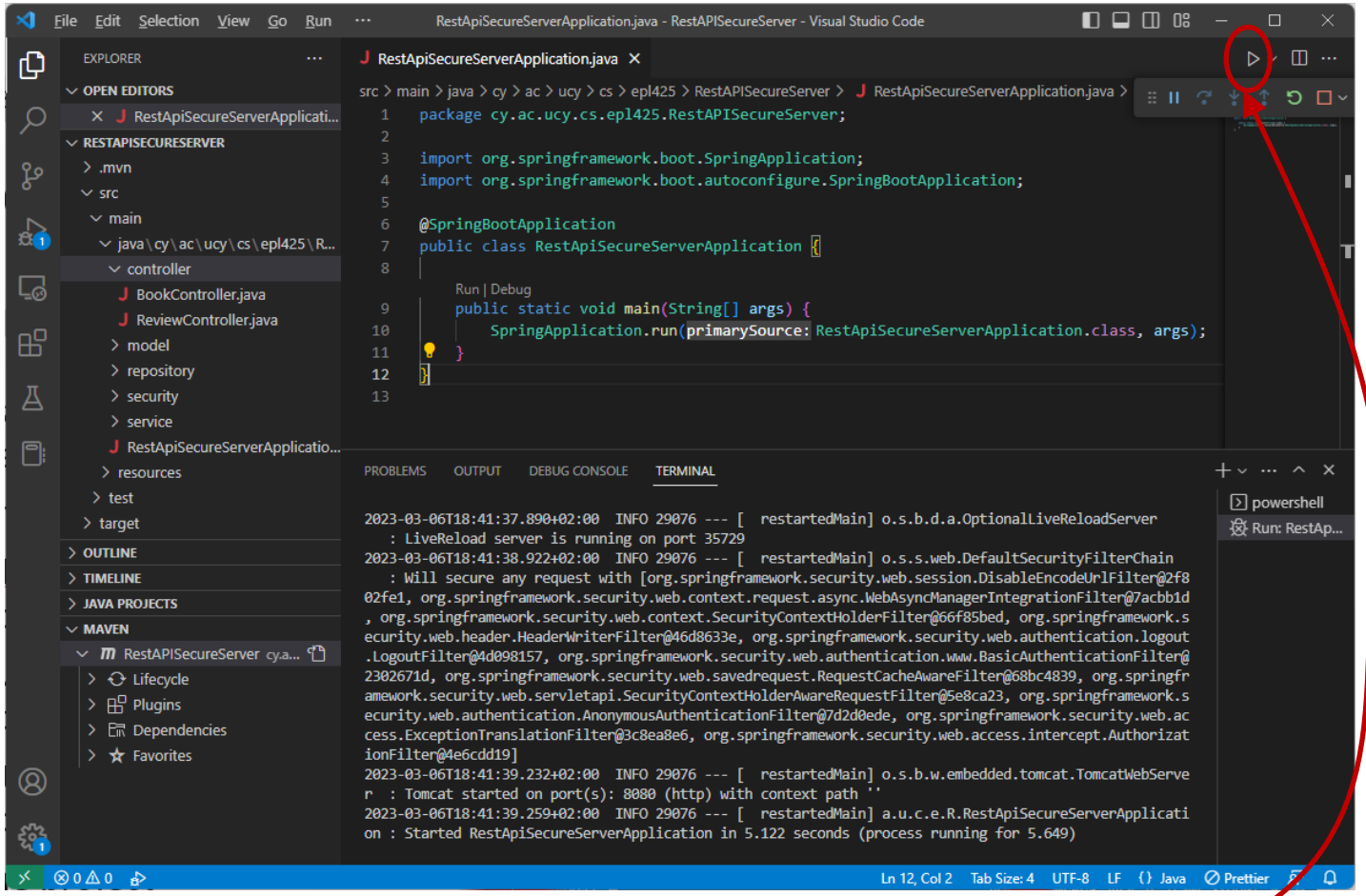
Security Filter Chain

- We can create an instance of the SecurityFilterChain to define a custom filter chain in and define what to apply on each request
  - In this example, any request is required to be authenticated and access to content is authorized on the basis of HTTP method and roles
  - In this example, authentication manager relies on In-Memory credentials
  - In this example, users authenticate with Basic Authentication

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {

   AuthenticationManagerBuilder authenticationManagerBuilder =
http.getSharedObject(AuthenticationManagerBuilder.class);
   authenticationManagerBuilder.userDetailsService(userDetailsManag
er());
   authenticationManager = authenticationManagerBuilder.build();

   http
    .csrf(csrf->csrf.disable())
    .httpBasic(Customizer.withDefaults())
    .authorizeHttpRequests(authorize -> authorize
      .requestMatchers(HttpMethod.POST,
"/api/books/**").hasAnyRole("MANAGER")
      .requestMatchers(HttpMethod.PUT,
"/api/books/**").hasAnyRole("MANAGER")
      .requestMatchers(HttpMethod.DELETE,
"/api/books/**").hasAnyRole("MANAGER")
      .requestMatchers(HttpMethod.GET, "/api/books/**").permitAll()
      .anyRequest().authenticated()
    )
    .authenticationManager(authenticationManager);

    return http.build();
}
```

If you want to disable CORS when testing API from localhost see here

Download, extract, compile, run application



- Download RestAPISecureServer.zip which contains the implementation of all secure API endpoints for Books and Reviews
- Extract it
- Open RestAPISecureServer folder via VS Code
- In Maven tab, click on compile to build the project
- Open the file RestApiServerApplication.java
- Run the project

Appendix

Replace Tomcat with Jetty

- Modify pom.xml as shown below:

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
           <exclusions>
              <exclusion>
                 <groupId>org.springframework.boot</groupId>
                 <artifactId>spring-boot-starter-tomcat</artifactId>
              </exclusion>
           </exclusions>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jetty</artifactId>
   </dependency>
</dependencies>
```

exclude the default added spring-boot-starter-tomcat dependency

add the dependency for spring-boot-starter-jetty

Disable CORS on API for localhost

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

   AuthenticationManagerBuilder authenticationManagerBuilder = http.getSharedObject(AuthenticationManagerBuilder.class);
   authenticationManagerBuilder.userDetailsService(userDetailsManager());
   authenticationManager = authenticationManagerBuilder.build();

   http
      .csrf(csrf->csrf.disable())
      .httpBasic(Customizer.withDefaults())
      .cors(cors -> cors.configurationSource(corsConfigurationSource()))
   ...
}

CorsConfigurationSource corsConfigurationSource() {
   CorsConfiguration configuration = new CorsConfiguration();
   configuration.setAllowCredentials(true);
   configuration.setAllowedOrigins(Arrays.asList("http://localhost"));
   configuration.setAllowedMethods(Arrays.asList(CorsConfiguration.ALL)); // e.g. GET, POST, PATCH, PUT, DELETE, OPTIONS, HEAD
   configuration.setAllowedHeaders(Arrays.asList(CorsConfiguration.ALL));
   //configuration.setMaxAge(3600L);
   UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
   source.registerCorsConfiguration("/**", configuration);
   return source;
}
```