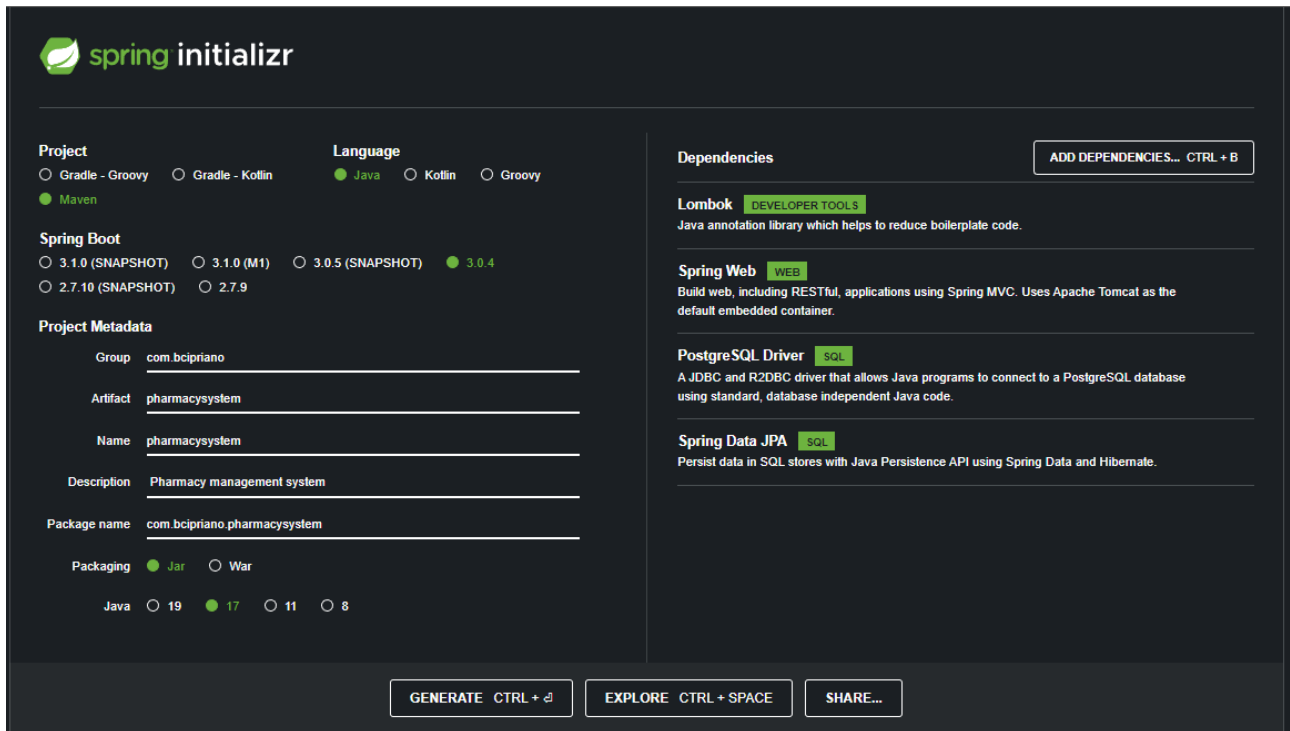<h1 style="text-align:center">(Chapter 1) About project</h1>

This is an educational project to make studies more pragmatic by developing a backend for a pharmacy management system. Attachment A, B and C contain the functional requirements, class model and architecture, respectively.

<h1 style="text-align:center">(Chapter 2)Spring boot project startup</h1>

To create a new project, go to http://start.spring.io/ and add the following settings:

In the left area are the initial settings of the project.

1- Project: This field specifies the type of project you want to create. You can choose between a Maven project or a Gradle project.

2- Language: This field specifies the programming language you want to use for your project. You can choose between Java, Kotlin, and Groovy.

3- Spring Boot version: This field specifies the version of Spring Boot you want to use. You can choose the latest version or a specific version.

4- Group: This field specifies the group ID for your project. This is typically a unique identifier for your organization or company.

5- Artifact: This field specifies the artifact ID for your project. This is typically the name of your project.

6- Dependencies: This field allows you to specify which dependencies you want to include in your project. Dependencies are libraries that your project will use. You can choose from a wide range of dependencies, including Spring Web, Spring Data, and Spring Security.

7- Packaging: This field specifies the packaging format for your project. You can choose between a JAR file or a WAR file.

8- Java Version: This field specifies the version of Java you want to use for your project.

In the right area we have the dependencies that will be added to the base project:

Lombok: is a library that can help reduce boilerplate code in Java classes. It provides annotations that can generate getters, setters, constructors, and other common methods at compile-time. By using Lombok, you can write cleaner, more concise code and reduce the amount of repetitive code you need to write.

Spring Web: is a module in Spring Framework that provides support for building wAftereb applications. It includes features like HTTP request handling, URL mapping, view resolution, and more. With Spring Web, you can easily build RESTful web services and web applications.

PostgreeSQL DRIVER: is a popular open-source relational database. To interact with PostgreSQL from a Java application, you need to use a driver that implements the JDBC API. The PostgreSQL driver provides this functionality and allows you to connect to and interact with a PostgreSQL database.

Spring Data JPA: is a module in Spring Framework that provides support for working with relational databases using the Java Persistence API (JPA). With Spring Data JPA, you can easily perform common database operations like creating, reading, updating, and deleting records, without needing to write low-level SQL code. It provides a higher level of abstraction and reduces the amount of boilerplate code needed to interact with a database.

After generating the project, import it with your IDE and wait until all dependencies are installed. Initially, it will have the following configuration of folders.
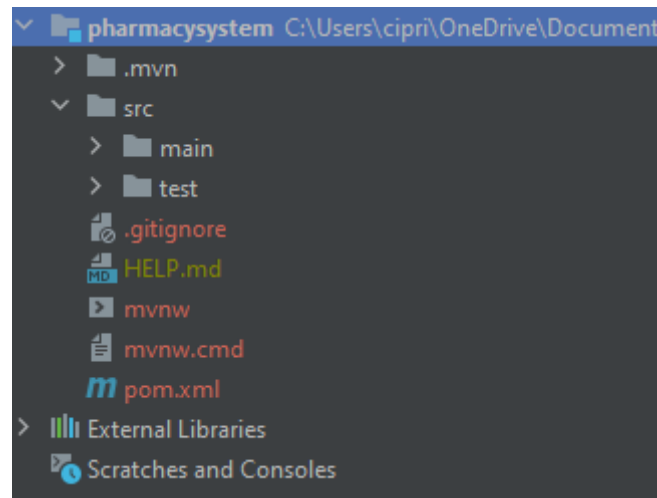
Data base config:

2.1) First, build a new PostgreSQL database:
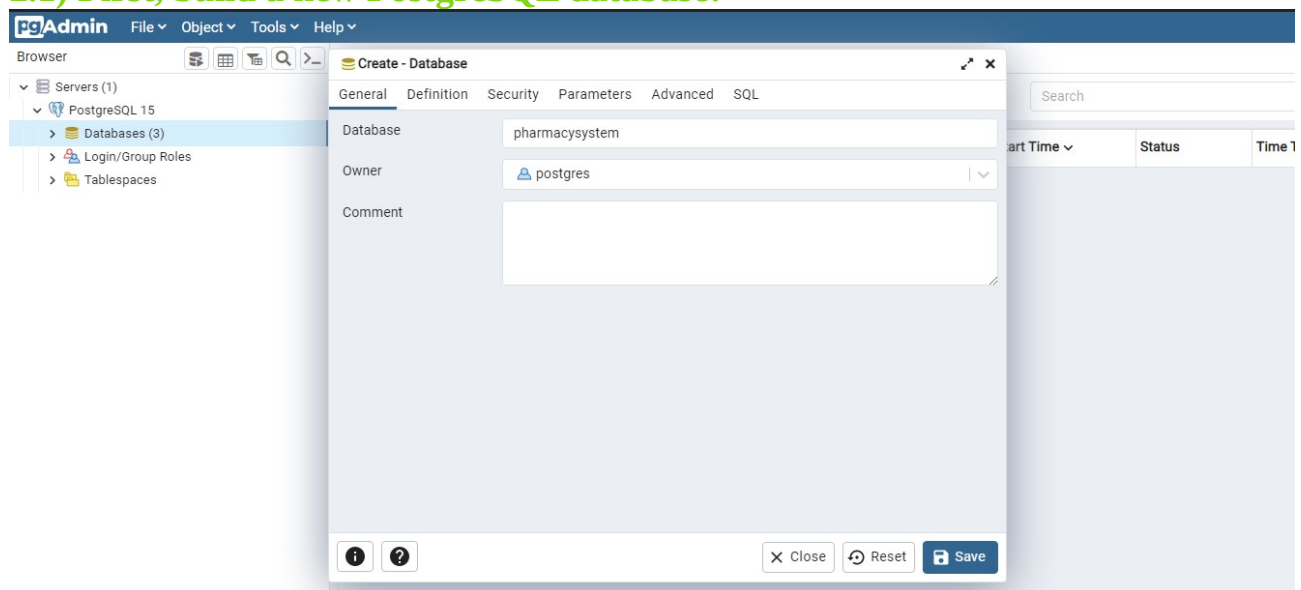
2.2) First, build a new PostgreSQL database:

2.3)Connection string configuration:

To make the connection with the PostgreSQL database, it is necessary to pass a String of connection inside the "application.properties" file which is inside the resources folder. See:


```
#Data Base Config:
spring.datasource.url=jdbc:postgresql://localhost:5432/pharmacysystem
spring.datasource.username=postgres
spring.datasource.password=?
spring.datasource.driver-class-name=org.postgresql.Driver


#Hibernate config:
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=update
```
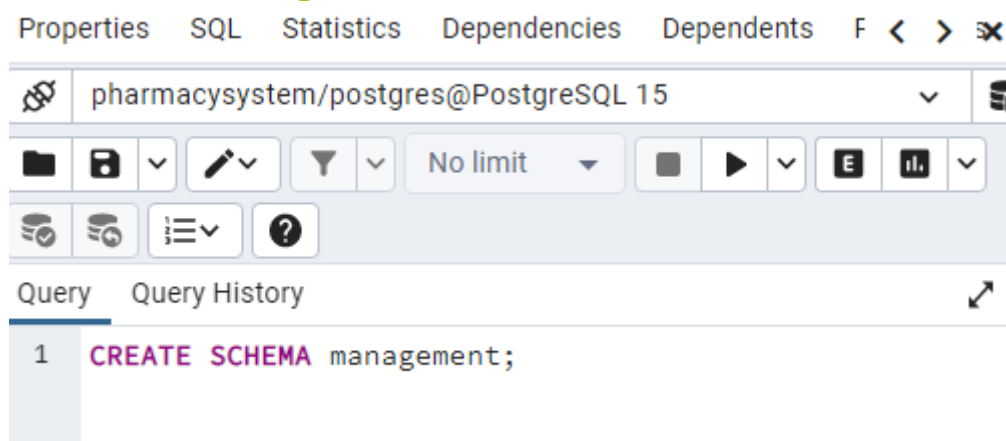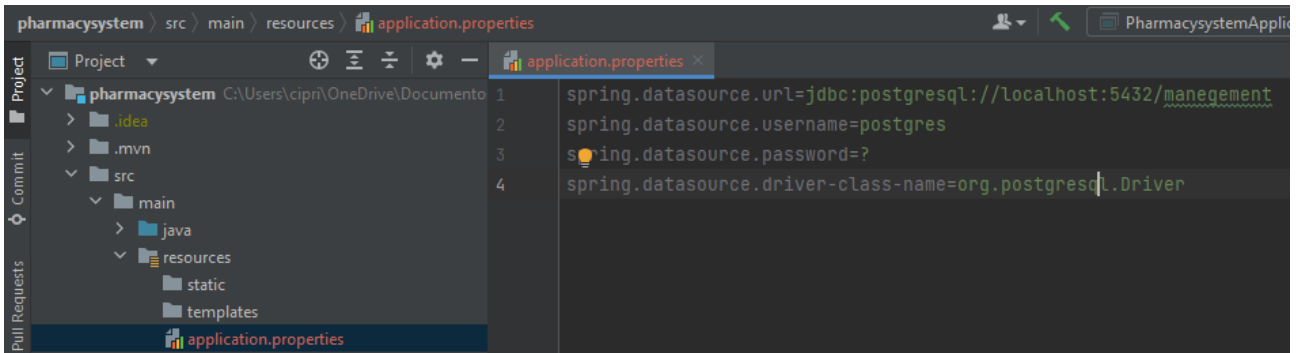
2.4)Connection test with database

After inserting the connection string and creating the database, run the file PharmacysystemApplicationTests" in the package /test/java/com/bcipriano/pharmacysystem/". Create a new package named databaseConnectionTest". Inside it, add a new java class with the following lines of code:

```java
package com.bcipriano.pharmacysystem.databaseConnectionTest;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.context.SpringBootTest;

import java.sql.*;

import static org.junit.jupiter.api.Assertions.assertTrue;
@SpringBootTest
public class DatabaseConnectionTest {
    @Value("${spring.datasource.url}")
    private String url;

    @Value("${spring.datasource.username}")
    private String username;

    @Value("${spring.datasource.password}")
    private String password;
```

```
    @Test
    public void testConnection() {
        try (Connection connection = DriverManager.getConnection(url, username, password)) {
            System.out.println("Successfully connected to the database.");
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT 1");
            assertTrue(resultSet.next());
            System.out.println("Successfully executed SELECT 1 statement.");
        } catch (SQLException e) {
            System.out.println("Failed to connect to the database: " + e.getMessage());
            assertTrue(false);
        }
    }
}
```

This is the test method that verifies the connection to the database. The @Test annotation marks this as a test method. The try block establishes a connection to the database using the injected values, creates a Statement object, executes a test query, and verifies that the query returned a result using the assertTrue() method. If the test fails, it prints an error message to the console.

(Chapter 3) Build Models

Initially, we will start building the classes that compose the models, that is, the layer of the system that represents business objects such as a customer, a sale, a product, among others. These classes are mapped to tables in a relational database, and each entity property corresponds to a column in the table. All of these classes will be located in the model.entity package. To make this possible, SpringBoot uses JPA (Java Persistence API), which is a Java EE specification for database operations. Its main characteristics are:

Documentation: (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)

1- Object-Relational Mapping (ORM)
2- Avoids the creation of native SQL mixed with Java code.
3- Abstraction of the database layer
4- Provides simple ways to perform database operations through Java applications
5- Migrates to other DBMSs by just changing its reference in the pom.xml file.

Note: From the latest versions of Spring Boot, the annotations related to entities have been transferred from javax.persistence to jakarta.persistence. In short, there were no major changes, except that annotations such as "@Entity", "@Column", "@Table"... are now performed by jakarta.persistence.

3.1) The meaning of each of the annotations used in this layer:

@Entity:
The @Entity annotation is used to indicate that a class is a persistent entity in JPA (Java Persistence API). A persistent entity represents a table in a database, and each instance of the entity represents a row in the table. When a class is annotated with @Entity, it must have a public or protected no-argument constructor, and it must not be declared final or abstract. The class also needs to have a primary key, which is typically annotated with @Id.

@Table:
The @Table annotation is used to specify the details of the table that the entity represents. It can be used to specify the table name, schema name, and other details such as indexes and unique constraints. If the @Table annotation is not used, the name of the table will be inferred from the name of the entity.

@Id:
The @Id annotation is used to mark a field or property as the primary key of the entity. Each entity must have a primary key, which can be either a single field or a composite of multiple fields. The @Id annotation can be used with different types of primary keys, such as numeric, string, or UUID.

@GenerationValue:
Is used to specify the automatic generation of the primary key value for an entity in JPA (Java Persistence API). This means that instead of a programmer manually specifying the value of the primary key, it will be automatically generated by the database or the JPA persistence provider.

The @GeneratedValue annotation has some attributes that can be used to specify the strategy for generating the primary key value. Some common strategies include:
• GenerationType.AUTO: Lets the persistence provider choose the most appropriate strategy for generating the primary key value.
• GenerationType.IDENTITY: Uses an auto-increment column in the database to generate the primary key value.
• GenerationType.SEQUENCE: Uses a database sequence to generate the primary key value.
• GenerationType.TABLE: Uses a separate table to generate the primary key value.

@Column
Is annotation in Spring Boot is used to map a class field to a column in a database table. It is often used in conjunction with the @Entity annotation to create a persistent entity in JPA.Here are some of the attributes that can be used with the

@Column annotation:
name: specifies the name of the column in the database table. By default, the name of the field is used.

nullable: specifies whether the column can have null values. The default is true.

unique: specifies whether the column values must be unique. The default is false.

length: specifies the maximum length of the column. For string columns, this is the maximum number of characters. The default is 255.

precision: specifies the precision of a decimal column. The default is 0.

scale: specifies the scale of a decimal column. The default is 0.

insertable: specifies whether the column is included in SQL INSERT statements. The default is true.

updatable: specifies whether the column is included in SQL UPDATE statements. The default is true.

@ManyToOne:
The @ManyToOne annotation is used to establish a many-to-one relationship between two entities in a JPA-based data model. It is used to annotate a field or property that represents the "many" side of the relationship.

This annotation is typically used in conjunction with the @OneToMany annotation, which is used to establish the opposite "one-to-many" side of the relationship in the related entity.

@Convert(converter = Jsr310JpaConverters.LocalDateConverter.class)
The Jsr310JpaConverters.LocalDateConverter is a default date converter provided by Spring Data JPA that converts java.time.LocalDate objects into data types supported by the database.

When it comes to accepting dates from the frontend, the default format accepted by LocalDateConverter is ISO 8601, which is an internationally recognized and accepted format for date and time representation. This format has the following appearance: "YYYY-MM-DD", where "YYYY" represents the year, "MM" represents the month, and "DD" represents the day.

For example, if the user types "2023-03-16" in a frontend form, the LocalDateConverter will convert this string into a LocalDate object with the equivalent value of "March 16, 2023".

It is important to remember that although LocalDateConverter can convert dates in other formats, it is a good practice to standardize the input date format on the frontend to avoid possible data conversion errors.

3.2) About lombok

We will use a tool to simplify the code called Lombok. As we know, every Java class has basic structures related to each attribute such as getters, setters, constructors, toString, equals, and hashCode.

Lombok is a Java library that makes writing repetitive Java code, such as getters, setters, constructors, and more, easier. It allows you to reduce the amount of code written and increase productivity. Some of the key concepts of Lombok include:

1- Annotations: Lombok uses annotations to identify the classes and methods that need to be modified. Some of the most common annotations are @Data, @Getter, @Setter, and @NoArgsConstructor.

2- Code generator: Lombok has a built-in code generator that automatically generates the necessary code based on the annotations used.

3- Validation: Lombok also provides validation annotations, such as @NotNull and @NonNull, that allow you to add validations to your code without writing a lot of code.

4- Convenience methods: In addition to getters, setters, and constructors, Lombok also provides other useful methods, such as equals(), hashCode(), and toString(). IDE plugin: Lombok is compatible with various IDEs, including IntelliJ IDEA, Eclipse, and NetBeans. You need to install a plugin in your IDE to recognize the annotations and generate code automatically.

Observations:
To use Lombok, it is necessary to include it in the pom.xml and have the Lombok.plugin extension installed in your IDE.

Annotations used in the project:

@Data: The @Data annotation from Lombok is a shorthand annotation that automatically generates getter and setter methods for all attributes, as well as toString, equals, hashCode, and other common methods of a Java class. With it, it's possible to write shorter and more readable classes without having to write all methods manually. Additionally, it also adds the @Getter and @Setter annotations for each attribute, making it possible to customize the method generation by adding other annotations to the attributes, such as @NonNull or @Builder.

@Builder: This annotation is used to generate an internal builder class for the annotated class. A builder class is a class that allows easy and readable creation of objects, especially when the objects have many attributes or are complex. The @Builder annotation generates a class constructor with all attributes and a "build" method that returns an instance of the class. It's also possible to define default values for some or all of the class attributes through the builder. Additionally, the @Builder annotation allows attributes to be defined fluently, making the code more readable and easy to understand.

@NoArgsConstructor: This annotation is used to automatically generate a no-arguments constructor for the class. This constructor is generated with the purpose of making it easier to create objects without passing any arguments to it, making it easier to work with this class. The constructor generated by the @NoArgsConstructor annotation is useful, for example, in creating objects that will be filled later with the necessary values. The @NoArgsConstructor annotation can be used in combination with other Lombok annotations to generate constructors with different configurations, such as constructors with arguments or constructors with advanced construction configurations.

@AllArgsContructor: is used to automatically create a constructor that receives all the class attributes as arguments. This can be useful in cases where you want to provide a simple way to create instances of the class, especially when there are many attributes. By adding this annotation to a class, Lombok will automatically generate a constructor that has an argument for each attribute in the class, in the order in which they appear.

3.2) Repositories

In the models class, we also have repositories, which are interfaces of JpaRepository, an interface provided by the Spring Data JPA framework, which is a sub-project of the Spring Framework. It provides a set of methods for performing CRUD (Create, Read, Update, Delete) operations on entities in a database using Java Persistence API (JPA).

JPA is a Java specification for Object-Relational Mapping (ORM) that provides a way to map Java objects to database tables and vice versa. Spring Data JPA builds on top of JPA to provide additional functionality and simplify common database operations.

The JpaRepository interface extends the PagingAndSortingRepository interface, which in turn extends the CrudRepository interface. These interfaces provide a set of generic methods for performing common database operations such as save, delete, and findById.

In addition to these methods, JpaRepository also provides methods for querying data using Spring Data's Query Creation mechanism, which allows you to generate queries based on method names and parameters. You can also write custom JPQL (Java Persistence Query Language) or native SQL queries using the @Query annotation.

https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html