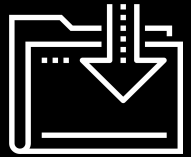


Java Accelerator 7

Lesson 3.2



Learning Objectives

01

Use Spring Data JPA to manipulate relational data.

02

Explain the different types of repositories.

03

Configure a connection to a MySQL database.

04

Automatically create the database with Spring Data JPA.

05

Set up and execute unit/integration tests for Spring Data JPA DAOs.

Spring Data JPA



Time to Code

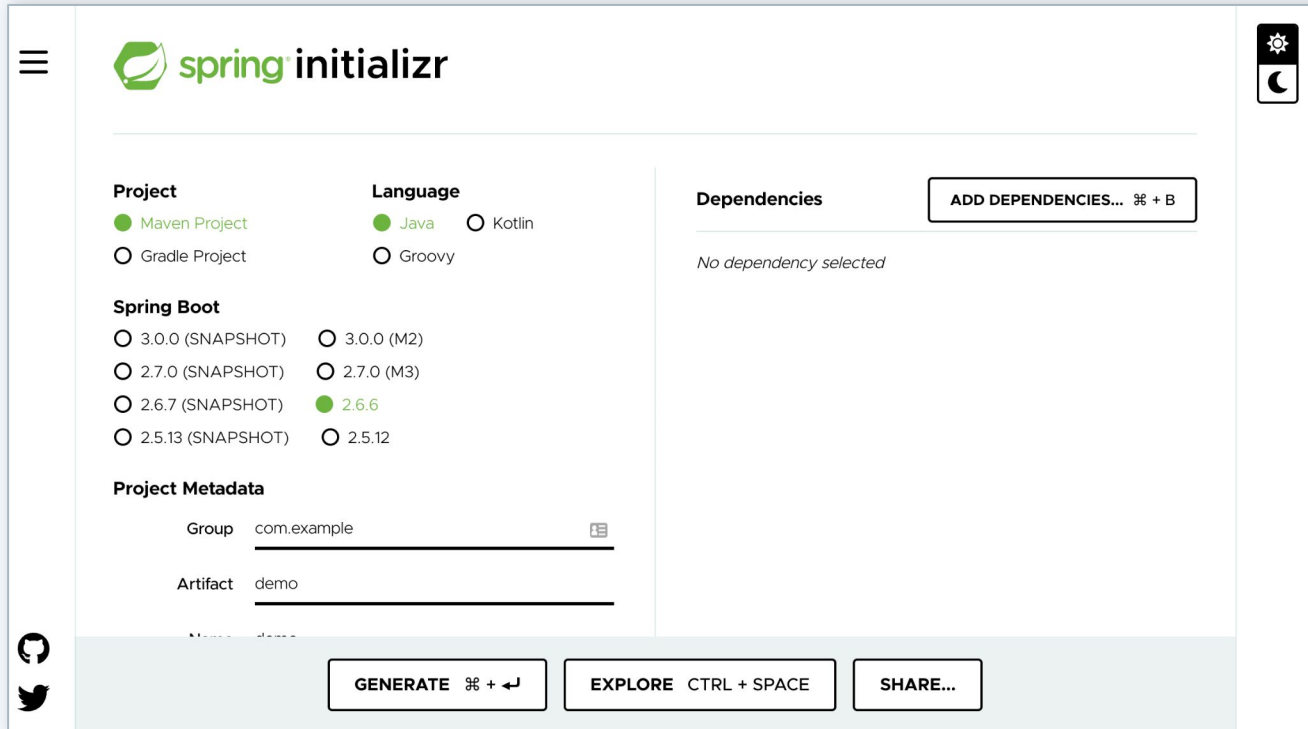
Simple CRM Project

Suggested Time:

Create the Project

Create the Project

Open your browser and navigate to start.spring.io.



The screenshot shows the Spring Initializr web application interface. The header features the Spring logo and the text "spring initializr". A hamburger menu is on the left, and a settings icon is on the right. The main content area is divided into three sections: "Project", "Language", and "Dependencies".

Project

- ☒ Maven Project
- ☐ Gradle Project

Language

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

Spring Boot

- ☐ 3.0.0 (SNAPSHOT)
- ☐ 3.0.0 (M2)
- ☐ 2.7.0 (SNAPSHOT)
- ☐ 2.7.0 (M3)
- ☐ 2.6.7 (SNAPSHOT)
- ☒ 2.6.6
- ☐ 2.5.13 (SNAPSHOT)
- ☐ 2.5.12

Dependencies

No dependency selected

Project Metadata

Group

Artifact

Name

At the bottom, there are three buttons: "GENERATE ⌘ + ↵", "EXPLORE CTRL + SPACE", and "SHARE...".

Create the Project

Create a new project with the following settings:

Project Metadata—Group:	<code>com.company.</code>
Project Metadata—Artifact:	<code>simple-crm-api.</code>
Dependencies:	<code>Web, MySql Driver, Spring Data JPA.</code>



Be sure to select Java 8 instead of the default Java 11.

Create the Project

Add the following dependency in the `<dependencies> </dependencies>` section of the `pom.xml` file:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13</version>
  <scope>test</scope>
</dependency>
```


Discuss Model Annotations

Discuss Model Annotations

We use annotations to map the Java objects (`Customer` and `Note`) to database tables using a code-first approach.

<code>@Entity</code>	Class-level annotation that marks this as a persistable object. The class maps to a table.
<code>@Table</code>	Class-level annotation that specifies the name of the table this class maps to. This is optional. The table name will be the class name if this annotation is not present.
<code>@Id</code>	Property-level annotation that indicates that this is the primary key/identifier for this class.
<code>@GeneratedValue</code>	Property-level annotation that indicates that the value for the property is generated. We use <code>strategy=GenerationType.AUTO</code> because we want the database to autogenerate this value for us. This maps to <code>auto_increment</code> in MySQL.

Discuss Model Annotations, Continued

@OneToMany

Property-level annotation that indicates that this is the **one** side of a one-to-many relationship.

- The `mappedBy` attribute indicates the property in the associated object that acts as the foreign key (in our case, it is the `customerId` property of the `Note` class).
- The `CascadeType.ALL` attribute value indicates that we want the database to delete all associated instances of `Note` if we delete the `Customer`.
- The `FetchType.EAGER` attribute value indicates that we want Spring Data JPA to create all the `Note` objects associated with the `Customer` when the `Customer` object is instantiated. The other value for this attribute is `FetchType.LAZY`, which defers the retrieval of `Note` data and creation of the `Note` objects until they are accessed by other code. We generally want all the data up front in a web service because we are shipping the data back to a client of some sort, so we use `EAGER`.
- Our `Customer` class contains a set of `Note` objects. Spring Data JPA will take care of getting the data from the database and creating the `Note` objects for us.

Discuss Model Annotations, Continued

@JsonIgnoreProperties

Class-level annotation that specifies properties that should be ignored when serializing this object to JSON.

- This is not JPA-specific and is not necessary for DAO functionality, but it is needed when we use a JPA DAO in a REST web service where we want to exclude Hibernate-specific fields.



Instructor Demonstration

Customer Class

Customer Class

```
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Table(name="customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    private String firstName;
    private String lastName;

    private String company;
    private String phone;

    @OneToMany(mappedBy = "customerId", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Note> notes;

    // Getters, setters, equals, and hashCode left out of this listing for brevity
}
```

Create the Note Class

Create the Note Class

01

Create a new class called `Note`.

02

Put the new class in the `com.company.simplecrmapi.dto` package.

03

This class will be our DTO for the `note` table.

04

This class has the following properties:

- `Integer id`
- `String content`
- `Integer customerId`



Let's Review

Solution: Note Class

```
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Table(name="note")
public class Note {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    private String content;
    private String customerId;

    // Getters, setters, equals, and hashCode
    . . .
}
```

Database Configuration



Important



The database schema must be created before Spring Data JPA can automatically create the database tables. There are two ways to do so:

1. Run the schema creation statement (for example, `create schema if not exists simple_crm;`) in MySQL Workbench before starting the Java API.
2. In `application.properties` include the setting `"createDatabaseIfNotExist=true"` within `"spring.datasource.url"`.

For example:

```
"spring.datasource.url=jdbc:mysql://localhost:3306/simple_crm?useSSL=false&serverTimezone=UTC&createDatabaseIfNotExist=true&allowPublicKeyRetrieval=true"
```

Database Configuration

`spring.jpa.hibernate.ddl-auto` allows us to specify how we want Spring Data JPA to act when we start our application. Possible values include:

none	The default for MySQL databases. Spring Data JPA will not do anything to alter the database structure on startup.
update	Spring Data JPA will modify the database structure based on the annotations of the Java entity classes.
create	Spring Data JPA creates the database every time the application is started, but it does not drop the tables when the application quits.
create-drop	Spring Data JPA creates the data every time the application is started and drops all the tables when the application quits.



We use the `update` value because it will create the table if the table does not exist. Otherwise, it will leave the existing table in its current state.

Database Configuration

`spring.jps.show-sql=true` allows us to see the SQL statements that Spring Data JPA is executing.

Add the following config entries to the `../resources/application.properties` file to configure the datasource:

```
spring.datasource.url=jdbc:mysql://localhost:3306/simple_crm?useSSL=false&  
serverTimezone=UTC&createDatabaseIfNotExist=true&allowPublicKeyRetrieval=true
```

```
spring.datasource.username=root  
spring.datasource.password=password
```

```
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.show-sql=true
```



**Spring Data JPA can also start
with an existing database.**

Spring Data JPA with an Existing Database

```
create schema if not exists simple_crm;
use simple_crm;

create table if not exists customer (
  id int(11) not null auto_increment primary key,
  first_name varchar(50) not null,
  last_name varchar(50) not null,
  company varchar(50) not null,
  phone varchar(50) not null
);

create table if not exists note (
  id int(11) not null auto_increment primary key,
  content varchar(255) not null,
  customer_id int(11) not null
);

/* Foreign Keys: note */
alter table note add constraint fk_note_customer foreign key (customer_id) references customer(id);
```


Repository Overview

Repository Overview

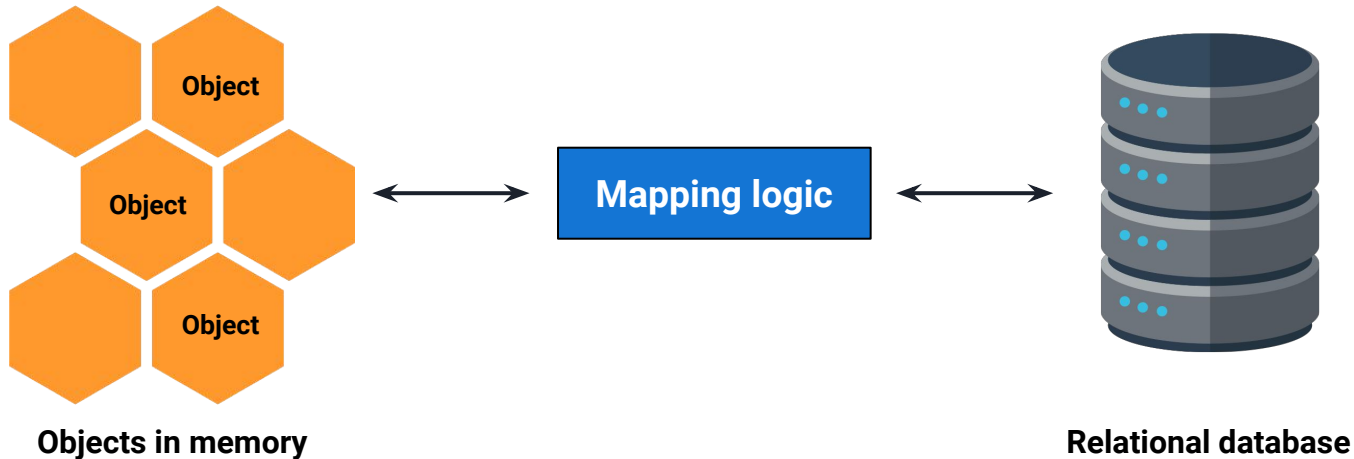
Spring Data JPA takes advantage of **object-relational mapping (ORM)** technology.



This technology automatically maps Java objects to database tables.



The framework that we'll be using is called Hibernate, but Spring Data JPA abstracts away the details of the underlying ORM technology for us so we don't have to worry about the details of Hibernate.



Repository Overview



Repositories are just interfaces—we do not provide an implementation.



The base interface (repository) is just a marker interface.



Don't confuse the repository base interface with the `@Repository` annotation. The annotation just makes the Spring framework aware of the interface.

Repository Overview

Several interfaces are derived from a repository:

CrudRepository	Has basic CRUD operations.
PagingAndSortingRepository	Adds paging and sorting capabilities.
JpaRepository	Adds JPA-specific features to PagingAndSortingRepository .

Repository Overview

The JavaDoc for the Crud repository shows the methods that we get for free:

- `count()`
- `delete()`
- `deleteAll()`
- `deleteById()`
- `existsById()`
- `findAll()`
- `findAllById()`
- `save()`
- `saveAll()`

Defining a Repository

Defining a Repository

01

We just need to declare an interface.

02

The first step is to decide which interface to extend. We will extend `JpaRepository`.

03

We need the `@Repository` annotation.

04

The `JpaRepository` is a parameterized interface (like `Map` or `List`). This means we have to tell the interface what it will be storing and what type the primary key or identifier will be.

In our case, we are going to declare two repositories:

- `Customers`—Customer ids are integers.
- `Notes`—Note ids are integers.

Defining a Repository

This is what it looks like—pretty simple!

(These interfaces are in the `com.company.simplecrmapl.repository` package.)

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
}

@Repository
public interface NoteRepository extends JpaRepository<Note, Integer> {
}
```


Start Up API and Confirm Database Tables

Start Up API and Confirm Database Tables



Start the API service and confirm that startup is successful by reviewing the application console log.



Confirm in the database that the customer and note tables have been properly created and the table columns match the entity model classes.

Custom Query Methods

Custom Query Methods



We can create custom query methods. We are just going to cover the basics here.



Constructing queries from method names is convenient for prototyping.



Add the following method definitions to your `CustomerRepository` interface:

- `List<Customer> findByLastName(String lastName);`
- `List<Customer> findByCompany(String company);`



Magic! JPA will create the methods for running these queries without any additional code from us.

Find by Last Name and Company

Find by Last Name and Company



Add a new method definition to the `CustomerRepository` interface to find customers by last name and company.



Hint: JPA seems semi-magical. Follow the same pattern as before, but include both last name and company.

Solution: Find by Last Name and Company

We don't have to write any code, and that JPA can interpret what needs be done from the method definition.

```
List<Customer> findByLastNameAndCompany(String lastName, String company);
```

Customer Unit Tests

Customer Unit Tests

How to set up and execute unit/integration tests for your Spring Data JPA DAOs:

01

We annotate the class with the following:

- `@RunWith`—standard with testing Spring components
- `@SpringBootTest`—also standard when testing Spring components

02

`@Autowired` the two repositories

03

Write tests!



Instructor Demonstration

Customer Unit Tests

Note Unit Tests

Note Unit Tests

Add the code necessary to the `createTest` method to test the `Note` repository:

01

Start by deleting all `Notes` in the repository.

02

Create two new `Note` objects:

- The first should have the content `"This is a test note"`.
- The second should have the content `"This is the SECOND test note"`.
- Set the `customerId` of each to the id from the previously created `Customer`.

03

Save both `Note` objects.

04

After the existing test of the size of the `customerList`, test the size of the `Note` set:

- Create a new set that is equal to the `Note` objects from the first element in the `customerList`.
- Write the test to check whether it has a `size()` of 2.



Instructor Demonstration

Demonstrate Repositories



Activity: Car Lot JPA Repository

In this activity you will create a Spring Data JPA Repository and database using a code-first approach.

Suggested Time:



Activity: Coffee Inventory

In this activity you will create a Spring Data JPA Repository and database using a code-first approach.

Suggested Time:





**This lesson showed how to use
Spring Data JPA to create
DAO components.**

Recap

The main takeaways from this lesson:



Spring Data JPA is built on top of ORM technology. We use Hibernate here.



Spring Data JPA DAOs are called repositories.



Creating a repository involves extending an existing repository interface.



We can create custom query methods by declaring them in our interface.



Java objects are mapped to relational tables using annotations.



Spring Data JPA can automatically create database tables for us based on the annotation on our object model.



Spring Data JPA can also be used with existing databases.

*The
End*