# SpringData JPA

## Introduction

### JPA:

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Features:

* Sophisticated support to build repositories based on Spring and JPA
* Support for Querydsl predicates and thus type-safe JPA queries
* Transparent auditing of domain class
* Pagination support, dynamic query execution, ability to integrate custom data access code
* Validation of @Query annotated queries at bootstrap time
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing @EnableJpaRepositories.

### Hebernet

## SpringData JPA Dependencies and project build

### Dependencies

Due to the different inception dates of individual Spring Data modules, most of them carry different major and minor version numbers. The easiest way to find compatible ones is to rely on the Spring Data Release Train BOM that we ship with the compatible versions defined. In a Maven project, you would declare this dependency in the <dependencyManagement /> section of your POM, as follows:

*Example 1. Using the Spring Data release train BOM*

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-releasetrain</artifactId>

<version>Lovelace-SR4</version>

<scope>import</scope>

<type>pom</type>

</dependency>

</dependencies>

</dependencyManagement>

A working example of using the BOMs can be found in our [Spring Data examples repository](https://github.com/spring-projects/spring-data-examples/tree/master/bom). With that in place, you can declare the Spring Data modules you would like to use without a version in the <dependencies /> block, as follows:

*Example 2. Declaring a dependency to a Spring Data module*

<dependencies>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

</dependency>

<dependencies>

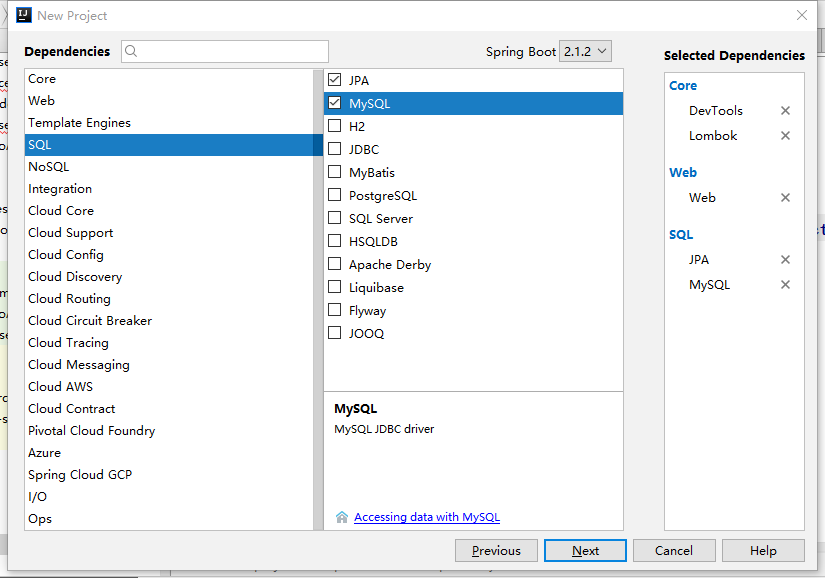
### Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.1.4.RELEASE or better.

### Dependency Management with Spring Boot

Spring Boot selects a recent version of Spring Data modules for you. If you still want to upgrade to a newer version, configure the property spring-data-releasetrain.version to the [train name and iteration](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#dependencies.train-names) you would like to use.

### Configuration when creating a project



*Example 3：*

<!—for mysql -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<!—jpa dependence include spring-data-jpa、spring-orm and Hibernate for JPA -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

### Database Configuration(yml )

### Mysql:

The simplest：

server:

#port

port: 8080

spring:

datasource:

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/test?serverTimezone=GMT%2B8&characterEncoding=utf8

username: root

password: root

jpa:

hibernate:

ddl-auto: update

show-sql: true

Complete configuration:

server:

#port

port: 8088

spring:

application:

#servr name

name: cms-dept

spring:

#data source and jpa

datasource:

#database url and encoding

url: jdbc:mysql://localhost:3306/crm?characterEncoding=utf8

#username

username: \*\*\*

#password

password: \*\*\*

spring:

#Connection pool

dbcp2:

# Initialize connection pool size

initial-size: 10

#Minimum number of connection pools

min-idle: 10

# maximum number of connection pools

max-idle: 30

# Configure the waiting time for a timeout connection

max-wait-millis: 30000

# Configure how often to perform a check to detect database connections that need to be closed

time-between-eviction-runs-millis: 200000

# Configure the minimum lifetime of the connection in the connection pool

remove-abandoned-on-maintenance: 200000

spring:

jpa:

# Configure the database type

database: MYSQL

# Configure whether to print SQL

show-sql: true

#Hibernate

hibernate:

# Configuration Cascade Level

ddl-auto: update

naming:

# Naming strategy

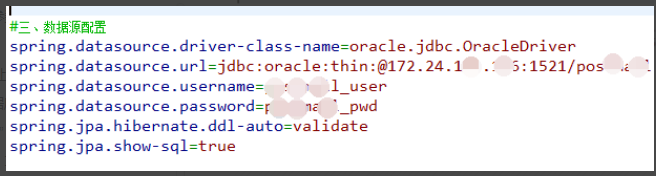
strategy: org.hibernate.cfg.ImprovedNamingStrategy

properties:

hibernate:

dialect: org.hibernate.dialect.MySQL5Dialect

* ORACLE



## Working with Spring Data Repositories

### CrudRepository

The central interface in the Spring Data repository abstraction is Repository. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one.

*Example 3. CrudRepository interface*

public interface CrudRepository<T, ID extends Serializable>

extends Repository<T, ID> {

<S extends T> S save(S entity);

Optional<T> findById(ID primaryKey);

Iterable<T> findAll();

long count();

void delete(T entity);

boolean existsById(ID primaryKey);

// … more functionality omitted.

}

|  |  |
| --- | --- |
|  | Saves the given entity. |
|  | Returns the entity identified by the given ID. |
|  | Returns all entities. |
|  | Returns the number of entities. |
|  | Deletes the given entity. |
|  | Indicates whether an entity with the given ID exists. |

### PagingAndSortingRepository

On top of the CrudRepository, there is a PagingAndSortingRepository abstraction that adds additional methods to ease paginated access to entities:

*Example 4. PagingAndSortingRepository interface*

public interface PagingAndSortingRepository<T, ID extends Serializable>

extends CrudRepository<T, ID> {

Iterable<T> findAll(Sort sort);

Page<T> findAll(Pageable pageable);

}

To access the second page of User by a page size of 20, you could do something like the following:

PagingAndSortingRepository<User, Long> repository = // … get access to a bean

Page<User> users = repository.findAll(new PageRequest(1, 20));

### Query methods

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:

1. define a entity.

@Table(name = "usertable")  
@Entity  
public class usertable {  
 @Id  
 @GeneratedValue(strategy= GenerationType.AUTO)

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

interface PersonRepository extends Repository<Person, Long> { … }

1. Declare query methods on the interface.

interface PersonRepository extends Repository<Person, Long> {

List<Person> findByLastname(String lastname);

}

1. Call the corresponding method

@RunWith(SpringRunner.class)  
@SpringBootTest  
public class testuser {  
 @Autowired  
 usertableIMP usertableimp;  
 @Test  
 public void TestUsertable() {

usertableimp.save(user1);

}

The sections that follow explain each step in detail:

Defining Repository Interfaces

Defining Query Methods

Creating Repository Instances

Custom Implementations for Spring Data Repositories

*names*

*Example 16. Query creation from method names*

interface PersonRepository extends Repository<User, Long> {

List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);

List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property

List<Person> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByLastnameOrderByFirstnameDesc(String lastname);

}

#### Special parameter handling

*Example 17. Using*Pageable*,*Slice*, and*Sort*in query methods*

Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);

#### Limiting Query Results

User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);

#### Streaming query results

The results of query methods can be processed incrementally by using a Java 8 Stream<T> as return type. Instead of wrapping the query results in a Stream data store-specific methods are used to perform the streaming, as shown in the following example:

*Example 19. Stream the result of a query with Java 8 Stream<T>*

@Query("select u from User u")

Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")

Stream<User> streamAllPaged(Pageable pageable);