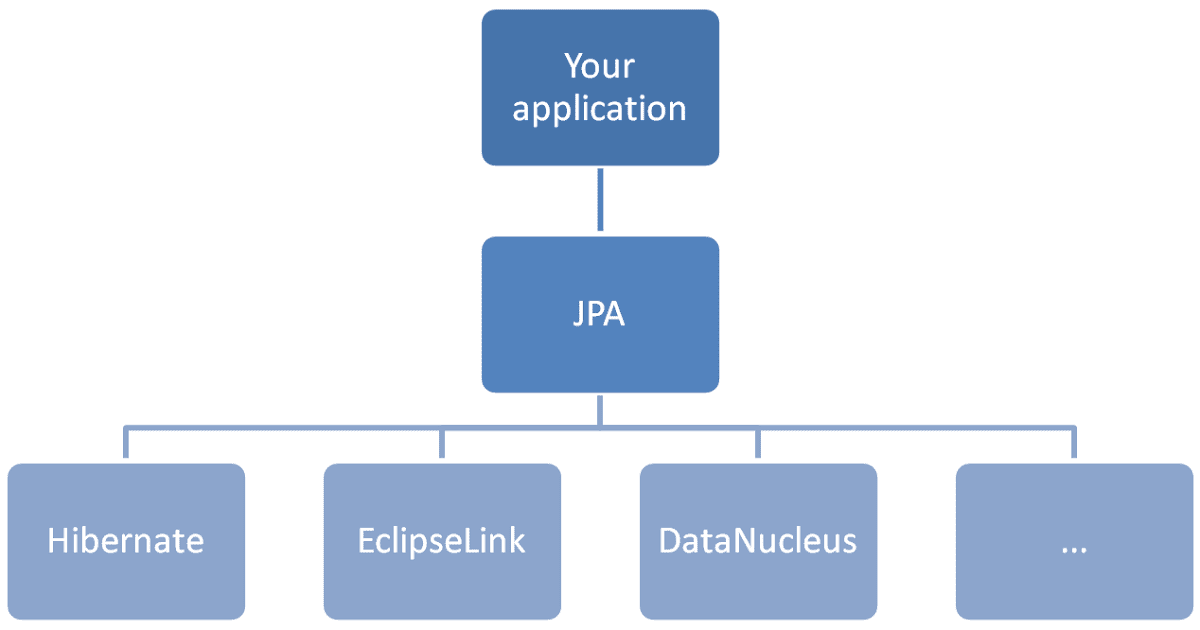
**JPA: Java Persistence API**

The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification. JPA is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces, and requires an implementation. There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use. JPA also requires a database to persist to.

JPA allows POJO (Plain Old Java Objects) to be easily persisted without requiring the classes to implement any interfaces or methods as the specification required. JPA allows an object's object-relational mappings to be defined through standard annotations or XML defining how the Java class maps to a relational database table. JPA also defines a runtime EntityManager API for processing queries and transaction on the objects against the database. JPA defines an object-level query language, JPQL, to allow querying of the objects from the database.



**Reasons for JPA**

1. It is a standard and part of EJB3 and Java EE.
2. Many free and open source products with enterprise level support.
3. Portability across application servers and persistence products (avoids vendor lock-in).
4. A usable and functional specification.
5. Supports both Java EE and Java SE.

**Spring Data**

Spring Data’s mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies.

**Features**

1. Powerful repository and custom object-mapping abstractions
2. Dynamic query derivation from repository method names
3. Implementation domain base classes providing basic properties
4. Support for transparent auditing (created, last changed)
5. Possibility to integrate custom repository code
6. Easy Spring integration via JavaConfig and custom XML namespaces
7. Advanced integration with Spring MVC controllers
8. Experimental support for cross-store persistence

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**

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

**Core Concepts**

The central interface in the Spring Data repository abstraction is Repository. It takes the domain class to manage as well as the ID type of the domain class as type arguments. 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.

1. **CrudRepository**

The **CrudRepository** provides sophisticated CRUD functionality for the entity class that is being managed.

public interface CrudRepository<T, ID extends Serializable>

extends Repository<T, ID> {

<S extends T> S save(S entity); // 1

Optional<T> findById(ID primaryKey); // 2

Iterable<T> findAll(); // 3

long count(); // 4

void delete(T entity); // 5

boolean existsById(ID primaryKey); // 6

// … custom methods allowed according to naming conventions.

}

1. Saves the given entity.
2. Returns the entity identified by the given ID.
3. Returns all entities.
4. Returns the number of entities.
5. Deletes the given entity.
6. Indicates whether an entity with the given ID exists.
7. **PagingAndSortingRepository**

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

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(PageRequest.of(1, 20));

**@EnableJpaRepositories**

This will enable the JPA repositories that are contained in the given package(s).

For instance, Enabling auto configuration support for Spring Data JPA required to know the path of the JPA the repositories. By default, it will scan only the main application package and its sub packages for detecting the JPA repositories.

Therefore, if the JPA repositories are placed under the main application package or its sub package, then it will be detected by the @EnableAutoConfiguration as a part of auto configuring the spring based configurations.

If the repository classes are not placed under the main application package or its sub package, then the relevant repository package(s) should be declared in the main application configuration class with @EnableJpaRepositories annotation. Then this will enable the JPA repositories contains in the given/declared package(s).

e.g:-

@EnableJpaRepositories(basePackages = "com.custom.repositories")

// or use just com. This will make Spring scan for your repository in all sub packages contained under com package. If still the repository isn’t found, Spring can’t inject the bean and result in a run time error.

**@EnableAutoConfiguration**

If you want to get the maximum advantage of spring boot’s auto configuration feature, it is expected to put all your class packages under spring boot main application package (directly in main package or indirectly as sub packages).

The @EnableAutoConfiguration will scan the main package and its sub packages when executing the spring boot auto configuration feature for class path dependencies. If any class or package that is outside from the main application package and it is required for completing auto configuration for some dependency, then should be declared in the main configuration class properly (with related annotation).

Then the @EnableAutoConfiguration will scan for those declared packages for detecting the required classes in the process of completing/doing the auto configuration for the application dependency declared in the class path.

**@EntityScan**

If the entity classes are not placed in the main application package or its sub package(s), then it is required to declare the package(s) in the main configuration class with @EntityScan annotation. This will tells spring boot to where to scan for detecting the entities for the application. Basically @EnableAutoConfiguration will scan the given package(s) for detecting the entities.

e.g:-

@EntityScan(basePackages = "com.entity")

**Query Lookup Strategies**

The JPA module supports defining a query manually as a String or having it being derived from the method name.

|  |  |  |
| --- | --- | --- |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

**Using @Query**

Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that executes them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

e.g:-

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.emailAddress = ?1")

User findByEmailAddress(String emailAddress);

}

**Using Named Parameters**

By default, Spring Data JPA uses position-based parameter binding. This makes query methods a little error-prone when refactoring regarding the parameter position. To solve this issue, you can use @Param annotation to give a method parameter a concrete name and bind the name in the query, as shown in the following example:

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")

User findByLastnameOrFirstname(@Param("lastname") String lastname,

@Param("firstname") String firstname);

}

**Native Queries**

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

public interface UserRepository extends JpaRepository<User, Long> {

@Query(value = "SELECT \* FROM USERS WHERE EMAIL\_ADDRESS = ?1", nativeQuery = true)

User findByEmailAddress(String emailAddress);

}

Oliver Gierke

Thomas Darimont

Christoph Strobl

Mark Paluch

Jay Bryant

Version 2.1.6.RELEASE,

2019-04-01

© 2008-2019 The original authors.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.