**Spring Data JPA :** Managing data between java classes or objects.  The DAO layer usually contains a lot of boilerplate code that should be simplified in order to reduce the number of lines of code and make the code reusable.

**What is JPA? : JPA or Java Persistence API** is the Java specification for accessing, managing and persisting data between Java classes or objects and relational database. The specification was introduced as part of EJB 3.0.

JPA is not an implementation or product, it is just a specification. It contains set of interfaces which need to be implemented. It is a framework that provides an extra layer of abstraction on the JPA implementation. The repository layer will contain three layers as mentioned below.

* **Spring Data JPA: –** This provides spring data repository interfaces which are implemented to create JPA repositories.
* **Spring Data Commons: –** It provides the infrastructure that is shared between data store specific spring data projects.
* The JPA provider which implements the JPA persistence API.

Spring data JPA allows us not to write any boilerplate code by adding an additional repository layer.

What is **Hibernate**? : - Hibernate is a JPA provider. This means it provides an implementation for the JPA specification.

**JPA Specification? :-** The Java Persistence API (JPA) is a specification for mapping Java objects to database tables. Annotations like **@Entity**point to JPA provider (Hibernate) implementations of JPA specifications.

JPA specifies what an Entity is and Hibernate provides the implementation for that interface or specification.

This sounds a bit confusing because JPA originated from ORMs like Hibernate. JPA standardizes how ORM tools should look and behave.

**What is Spring Data JPA :** Spring Data JPA is an abstraction that makes it easier to work with a JPA provider. Specifically Spring Data JPA provides a set of interfaces for easily creating data access repositories.

Spring Data JPA is really a set of dependencies that makes it easier to work with a JPA provider. Hibernate is one of several JPA providers. This means you can use Spring Data JPA without using Hibernate (if you really wanted to).

**Spring Data JPA vs Hibernate: The Key Difference :** Consider the following implementation of a CrudRepository using Spring Data JPA:By simply extending the CrudRepository interface, Spring will automagically implement the following repository methods:

**save()**

**saveAll()**

**findById()**

**existsById()**

**findAll()**

**findAllById()**

**count()**

**deleteById()**

**delete()**

**deleteAll()**

making it super easy to persist objects to the database…

//create new author

Author author = new Author();

//save to database

repo.save(author);

//find all authors

List authors = repo.findAll();

If we didn't use Spring Data JPA, we'd have to write more boilerplate code to persist entities to the database...

EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Hibernate" );

EntityManager entitymanager = emfactory.createEntityManager( );

entitymanager.getTransaction( ).begin( );

Author author = new Author( );

entitymanager.persist( author );

entitymanager.getTransaction( ).commit( );

entitymanager.close( );

emfactory.close( );

**Conclusion :** Hibernate is a JPA provider and ORM that maps Java objects to relational database tables. Spring Data JPA is an abstraction that makes working with the JPA provider less verbose. Using Spring Data JPA you can eliminate a lot of the boilerplate code involved in managing a JPA provider like Hibernate.

**Native query**refers to actual sql queries (referring to actual database objects). These queries are the sql statements which can be directly executed in database using a database client. ... Named query is the way you define your query by giving it a name.

**Spring Data Repositories**

**Spring Data Commons** project provides repository abstraction which is extended by the datastore-specific subprojects.

We have to be familiar with the Spring Data repository interfaces as it will help us with the implementation of the interfaces. Let’s have a look at the interfaces.

**Spring Data Commons: –**Following interfaces are provided as part of this project:

[Repository<T, ID extends Serializable>](https://docs.spring.io/spring-data/commons/docs/current/api/index.html?org/springframework/data/repository/Repository.html" \t "https://www.javacodegeeks.com/2018/05/_blank)  :

* This interface is a marker interface.
* It captures the type of the managed entity and the type of the entity’s id.
* It helps the Spring container to discover the “concrete” repository interfaces when classpath is scanned.

[CrudRepository<T, ID extends Serializable>](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html" \t "https://www.javacodegeeks.com/2018/05/_blank) :

* It provides CRUD operations for the managed entity.

[PagingAndSortingRepository<T, ID extends Serializable>](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html" \t "https://www.javacodegeeks.com/2018/05/_blank) :

* This interface declares the methods that are used to sort and paginate entities that are retrieved from the database.

[QueryDslPredicateExecutor<T>](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/querydsl/QueryDslPredicateExecutor.html" \t "https://www.javacodegeeks.com/2018/05/_blank) :

* It is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [QueryDsl](http://www.querydsl.com/" \t "https://www.javacodegeeks.com/2018/05/_blank) Predicate objects.

**Spring Data JPA: –** This project provides the following interfaces:

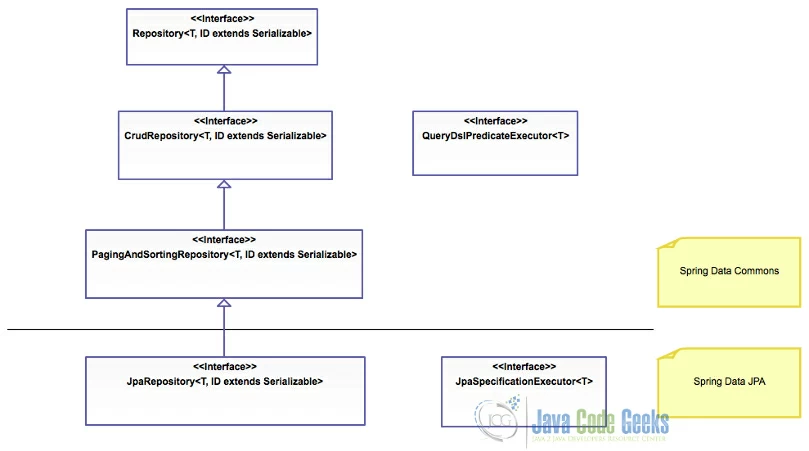
[JpaRepository<T, ID extends Serializable>](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html" \t "https://www.javacodegeeks.com/2018/05/_blank)  :

* This interface is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.

[JpaSpecificationExecutor<T>](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaSpecificationExecutor.html" \t "https://www.javacodegeeks.com/2018/05/_blank) :

* This is again not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [Specification<T>](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/Specification.html" \t "https://www.javacodegeeks.com/2018/05/_blank) objects that use the JPA criteria API.

**The repository hierarchy looks as follows:**



**Spring Data Custom Query :**

**Automatic Custom Queries:**The automatic custom query creation is also known as query creation from the method name.

Spring Data JPA has a built-in mechanism for query creation which can be used for parsing queries straight from the method name of a query method.

The mechanism first removes common prefixes from the method name and parses the constraints of the query from the rest of the method name. In order to use this approach, we have to make sure the method names of your repository interface are created by combining the property names of an entity object and the supported keywords.

The advantage of using this approach is that it makes it very easy to implement. But the limitation is if the query contains more than one parameter than the method name will be not readable.

**Manual Custom Queries:** Manual Custom queries are also known as queries creation using @Query tag.

The @Query annotation will be used to create queries using the JPA query language and for binding these queries directly to the methods of your repository interface.

When the query method is called, Spring Data JPA will execute the query specified by the @Query annotation.

The advantage of this approach is you can use JPA query language for the creation of a query. Also, the query stays in repository layer. Limitation of this approach is @Query can be only used when JPA query language is supported.

Steps :

<**dependency**>

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

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

        </**dependency**>

Entity/data model :

@Entity

@Table(name = "employee")

**public** **class** Employee **implements** Serializable {

**private** **static** **final** **long** serialVersionUID = -3009157732242241606L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    @Column(name="id")

**private** **long** id;

    @Column(name = "firstname")

**private** String firstName;

    @Column(name = "lastname")

**private** String lastName;

    @Column(name = "age")

**private** **int** age;

**@Entity:** Is used to define that the class is an Entity class.  
**@Table:** This annotation is used for specifying the table name which is defined in the database.  
**@Id:** Id annotation is used for specifying the Id attribute  
**@GeneratedValue:** This is used when we want to set automatically generated value. GenerationType is the mechanism that is used for generating the value for the specific column.  
**@Column:** This annotation is used for mapping the columns from the table with the attributes in the class.

**Repository Interface :** The repository interface is used for extending the CRUD interface. This interface adds the layer of a repository in the program. Spring Data JPA provides two major ways of creating queries. These queries are then used in the repository interface to fetch the data from the database.

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.CrudRepository;

**import** org.springframework.data.repository.query.Param;

**import** org.springframework.stereotype.Repository;

**import** com.tutorial.model.Employee;

@Repository

**public** **interface** EmployeeRepository **extends** CrudRepository<Employee, Long>{

    List findByLastName(String lastName);

@Query("SELECT e FROM Employee e WHERE e.age = :age")

**public** List findByAge(@Param("age") **int** age);

}

**Automatic Custom Query:**

List **findByLastName**(String lastName);

The method findByLastName contains the last name as the parameter which will be used for the search of data. Also, the query will be automatically created using JPA query builder.

**Manual Custom Query:**

**@Query**("SELECT e FROM Employee e WHERE e.age = :age")

public List **findByAge**(@Param("age") int age);

In this method we have manually defined a query to fetch the data based on age and the query is then bound with the findByAge method.

**Properties file :** As part of spring boot for connecting to the database, we will provide the details in the properties file. The properties file will look as shown below.

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

spring.datasource.username=postgres

spring.datasource.password=password

spring.jpa.generate-ddl=true

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

As part of the properties file, the database URL and credentials are provided. The property spring.jpa.show-sql=true shows the SQL query generated during the data manipulation by JPA.

**@RestController**

**@RequestMapping("/employee")**

**public** **class** WebController {

    @Autowired

    EmployeeRepository repository;

    @RequestMapping(value="/save",method = RequestMethod.POST)

**public** HttpStatus insertEmployee(@RequestBody Employee employee){

**boolean** status = repository.save(employee) != **null**;

**return** status? HttpStatus.CREATED : HttpStatus.BAD\_REQUEST;

    }

    @RequestMapping("/findall")

**public** List findAll(){

**return** (List) repository.findAll();

    }

    @RequestMapping("/findbyid")

**public** Optional findById(@RequestParam("id") **long** id){

        Optional result = repository.findById(id);

**return** result;

    }

    @RequestMapping("/findbylastname")

**public** List fetchDataByLastName(@RequestParam("lastname") String lastName){

**return** repository.findByLastName(lastName);

    }

    @RequestMapping("/findbyage")

**public** List fetchDataByAge(@RequestParam("age") **int** age){

**return** repository.findByAge(age);

    }

}

**@RestController** tag is used for defining the class as a rest controller class.  
**@RequestMapping** tag specifies the path mapping for the request. Value attribute specifies the URL mapping and the method attributes specifies the type of method it is like GET, POST, PUT, etc.  
If we don’t specify the method attribute by default the method is considered as GET method.In this class, we have autowired the repository and are using the methods available in the interface for data retrieval, insertion and deletion.

**What is JPA Entity Manager? :** **JPA EntityManager** is used to access a database in a particular application. It is used to manage persistent **entity** instances, to find **entities** by their primary key identity, and to query over all **entities**.

JPA EntityManager is at the core of Java Persistence API. [Hibernate](https://www.journaldev.com/3793/hibernate-tutorial) is the most widely used JPA implementation.

ORM consists of two concepts [object-oriented](https://www.journaldev.com/12496/oops-concepts-java-example) and relational programming. Hibernate is an ORM framework where programmer describes the way objects are represented in database. Hibernate handles the conversion automatically.

Hibernate provides implementation of JPA interfaces EntityManagerFactory and EntityManager.

JPA EntityManager is supported by the following set of methods.

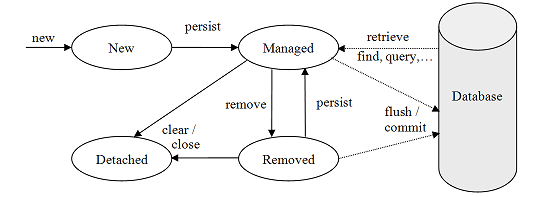
* **persist** – Make an instance managed and persistent.
* **merge** – Merge the state of the given entity into the current persistence context.
* **remove** – Remove the entity instance.
* **find** – Find by primary key. Search for an entity of the specified class and primary key. If the entity instance is contained in the persistence context, it is returned from there.
* **createQuery** – Create an instance of Query for executing a Java Persistence query language statement.
* **createNamedQuery** – Create an instance of Query for executing a Java Persistence named query language statement.
* **createNativeQuery** – Create an instance of Query for executing a native sql statement.
* **createNamedStoredProcedureQuery** – Create an instance of StoredProcedureQuery for executing a stored procedure in the database.
* **createStoredProcedureQuery** – Create an instance of StoredProcedureQuery for executing a stored procedure in the database.

Spring JdbcTemplate is the most important class in [Spring JDBC](https://www.journaldev.com/2593/spring-jdbc-example" \t "https://www.journaldev.com/17053/_blank) package.

* JDBC produces a lot of boiler plate code, such as opening/closing a connection to a database, handling sql exceptions etc.Implementing JDBC in the [Spring Framework](https://www.journaldev.com/16922/spring-framework" \t "https://www.journaldev.com/17053/_blank) takes care of working with many low-level operations (opening/closing connections, executing SQL queries, etc.).
* when working with the database in the Spring Framework, we only need to define the connection parameters from the database and register the SQL query, the rest of the work for us is performed by Spring.
* JDBC in Spring has several classes (several approaches) for interacting with the database. The most common of these is using the JdbcTemplate class. This is the base class that manages the processing of all events and database connections.
* The JdbcTemplate class executes SQL queries, iterates over the ResultSet, and retrieves the called values, updates the instructions and procedure calls, “catches” the exceptions, and translates them into the exceptions defined in the org.springframwork.dao package.
* Instances of the JdbcTemplate class are thread-safe. This means that by configuring a single instance of the JdbcTemplate class, we can then use it for several DAO objects.
* When using JdbcTemplate, most often, it is configured in the Spring configuration file. After that, it is implemented using bean in DAO classes.
* **@Configuration –** says that this class is configuration for Spring context.
* **@ComponentScan**(“com.journaldev.spring”)- specifies the package to scan for component classes.
* **@PropertySource**(“classpath:database.properties”)- says that properties will be read from database.properties file.

**Entity Object Life Cycle**

The life cycle of entity objects consists of four states: New, Managed, Removed and Detached.



When an entity object is initially created its state is **New**.  In this state the object is not yet associated with an [EntityManager](https://www.objectdb.com/api/java/jpa/EntityManager) and has no representation in the database.

An entity object becomes **Managed**when it is persisted to the database via an EntityManager’s [persist](https://www.objectdb.com/api/java/jpa/EntityManager/persist_Object)method, which must be invoked within an active transaction. On transaction commit, the owning EntityManager stores the new entity object to the database.Entity objects retrieved from the database by an EntityManager are also in the Managed state.

If a **managed** entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit.

A managed entity object can also be retrieved from the database and marked for **deletion**, using the EntityManager’s **[remove](https://www.objectdb.com/api/java/jpa/EntityManager/remove_Object)**method within an active transaction. The entity object changes its state from Managed to Removed, and is physically deleted from the database during commit.

The last state, **Detached**, represents entity objects that have been disconnected from the EntityManager. For instance, all the managed objects of an EntityManager become detached when the EntityManager is closed.

<https://www.stackchief.com/blog/Spring%20Data%20JPA>

<https://morioh.com/p/317a5fd76340>

<https://www.baeldung.com/jpa-join-types>