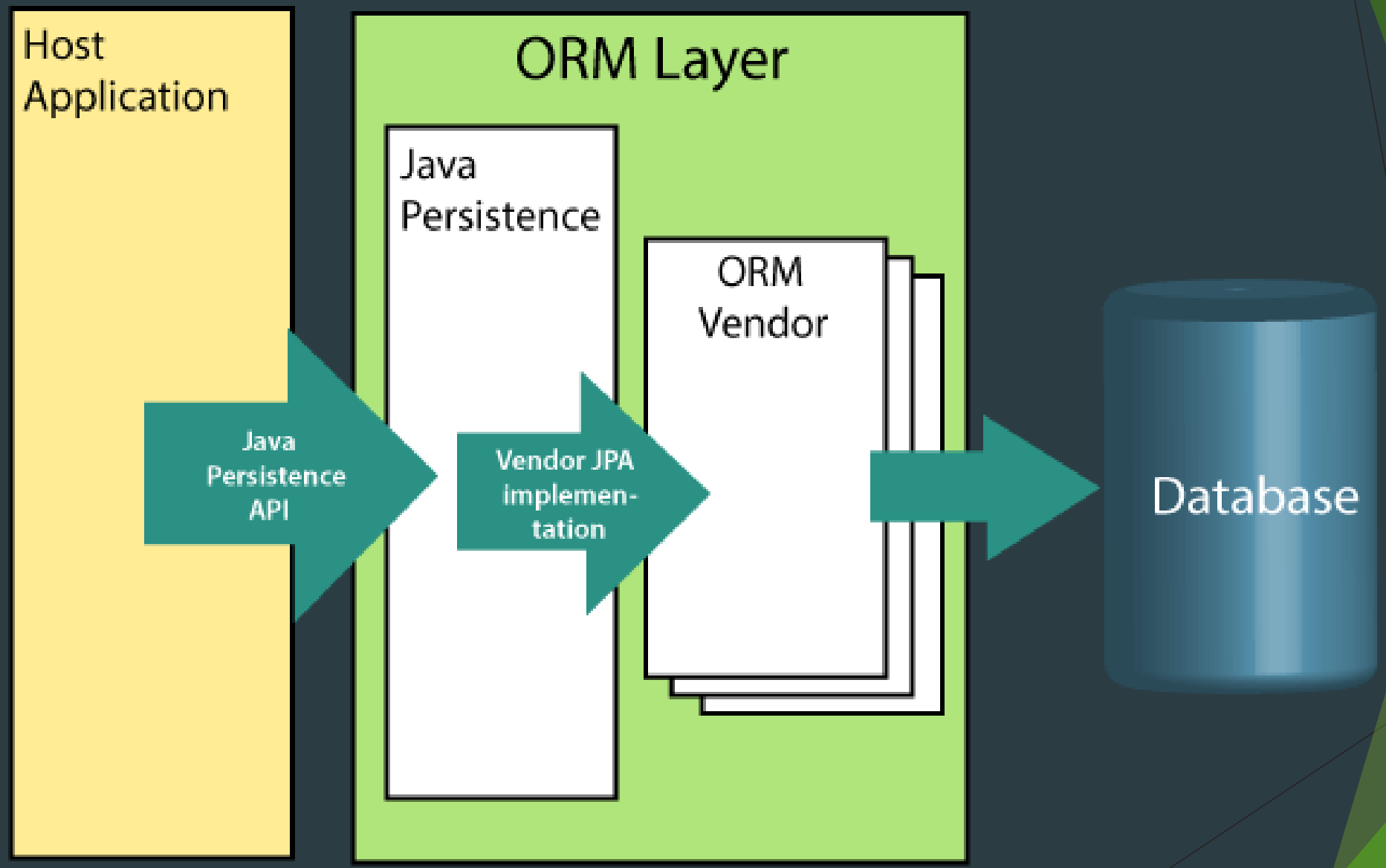
ORM

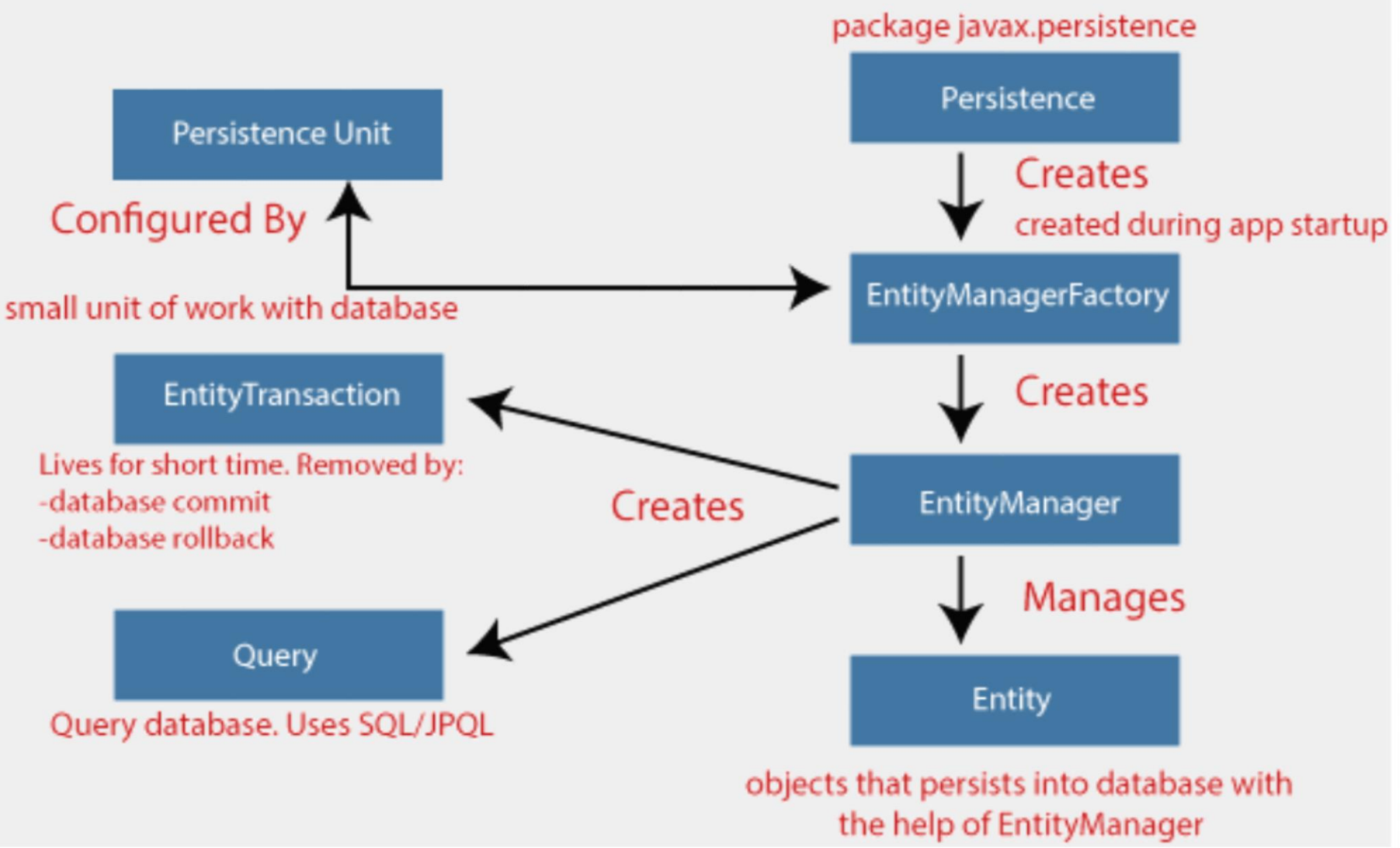
- The **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.
- **ORM Consists:-**
  - An API to perform basic CRUD operations on objects of persistent classes.
  - A language or API to specify queries that refer to classes and properties of classes.
  - A configurable facility for specifying mapping metadata.
  - A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.
- **Advantages:-**
  - Hides details of SQL queries from OO logic.
  - No need to deal with the database implementation.
  - Entities based on business concepts rather than database structure.
  - Transaction management and automatic key generation.
  - Fast development of application.
- **Java ORM Frameworks:-**
  - Hibernate, TopLink, EclipseLink, MyBatis, OpenJPA


Host Application
ORM Layer
Java Persistence
ORM Vendor
Java Persistence API
Vendor JPA implemen-tation
Database

JPA

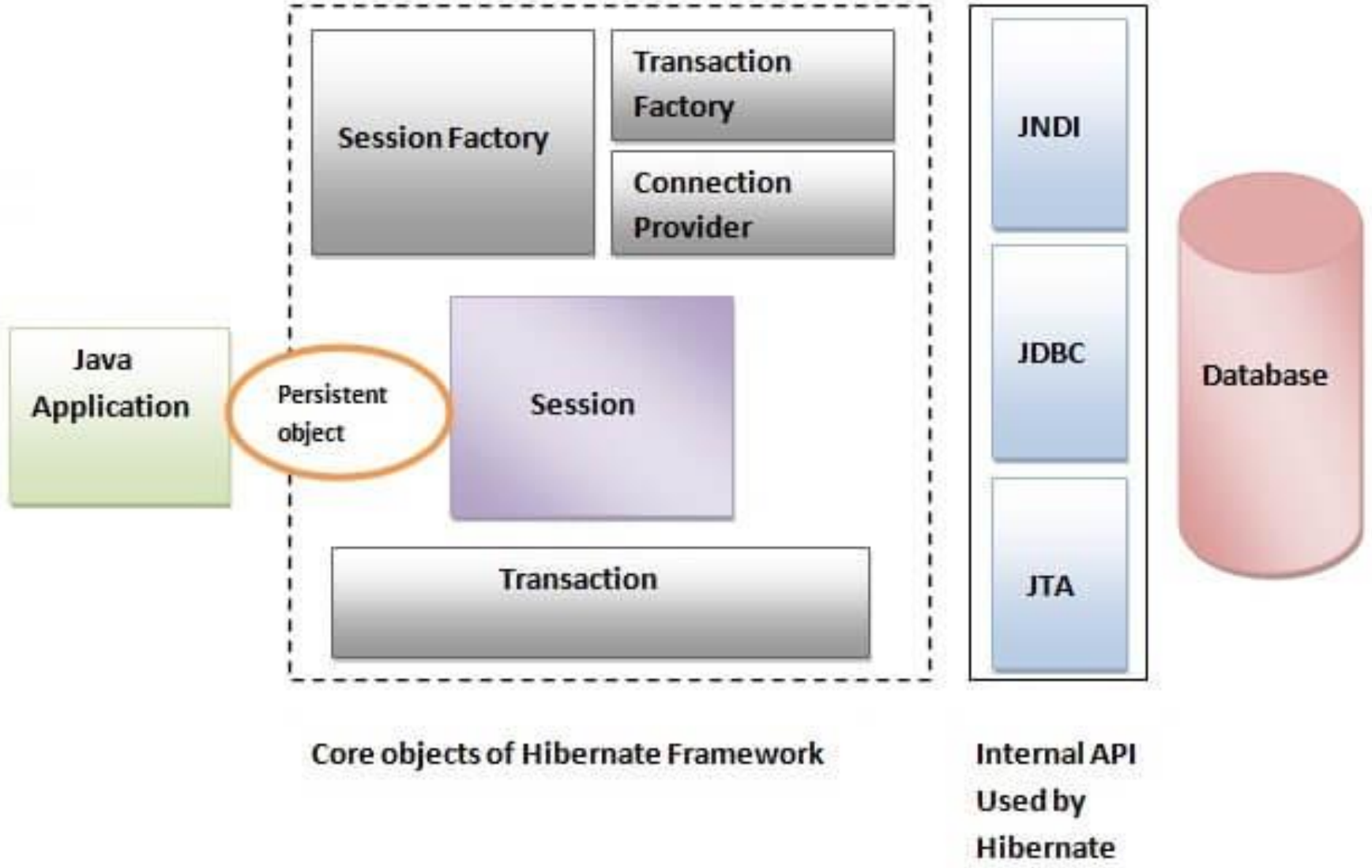
- The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.
- JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.
- The first version of Java Persistence API, JPA 1.0 was released in 2006 as a part of EJB 3.0 specification.

Architecture of Java Persistence API



Hibernate Framework

- Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.
- **Advantages of Hibernate**
  - Open Source and Lightweight
  - Fast Performance: because cache is internally used in hibernate framework.
  - Database Independent Query: It generates the database independent queries.
  - Automatic Table Creation
  - Simplifies Complex Join


Session Factory
Transaction Factory
Connection Provider
Java Application
Persistent object
Session
Transaction
JNDI
JDBC
JTA
Database
Core objects of Hibernate Framework
Internal API Used by Hibernate

Difference between JPA and Hibernate

JPA	Hibernate
JPA is a **Java specification** for mapping relation data in Java application.	Hibernate is an **ORM framework** that deals with data persistence.
JPA does not provide any implementation classes.	It provides implementation classes.
It uses platform-independent query language called **JPQL** (Java Persistence Query Language).	It uses its own query language called **HQL** (Hibernate Query Language).
It is defined in **javax.persistence** package.	It is defined in **org.hibernate** package.
It is implemented in various ORM tools like **Hibernate, EclipseLink,** etc.	Hibernate is the **provider** of JPA.
JPA uses **EntityManager** for handling the persistence of data.	In Hibernate uses **Session** for handling the persistence of data.

Spring Data JPA

- Spring Data JPA adds a layer on the top of JPA. It means, Spring Data JPA uses all features defined by JPA specification, especially the entity, association mappings, and JPA's query capabilities.
- Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.
- Spring Data JPA Features:-
  - **No-code repository:** It enables us to implement our business code on a higher abstraction level.
  - **Reduced standard code:** It provides the default implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.
  - **Generated Queries:** generate database query based on the method name.
- Spring Data Repository:-
  - **CrudRepository:** It offers standard **create**, **read**, **update**, and **delete** It contains method like **findOne()**, **findAll()**, **save()**, **delete()**,
  - **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the findAll methods. It allows us to **sort** and **retrieve** the data in a paginated way.
  - **JpaRepository:** It extends the both repository CrudRepository and PagingAndSortingRepository. It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context.

JPQL (Java Persistence Query Language)

- The JPQL is an object-oriented query language which is used to perform database operations on persistent entities. Instead of database table, JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.

Session Content (8)

- Types of Mapping (Relations)
- Mapping Directions
- **Generation Strategies**
- Entity Lifecycle
- JPA Cascading Operations
- JPA Fetch Type

Types of Mapping

- **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
- **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
- **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
- **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

Mapping Directions

- Mapping Directions are divided into two parts: -
- **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
- **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

- The main difference is that bidirectional relationship provides navigational access in both directions, so that you can access the other side without explicit queries. Also it allows you to apply cascading options to both directions.

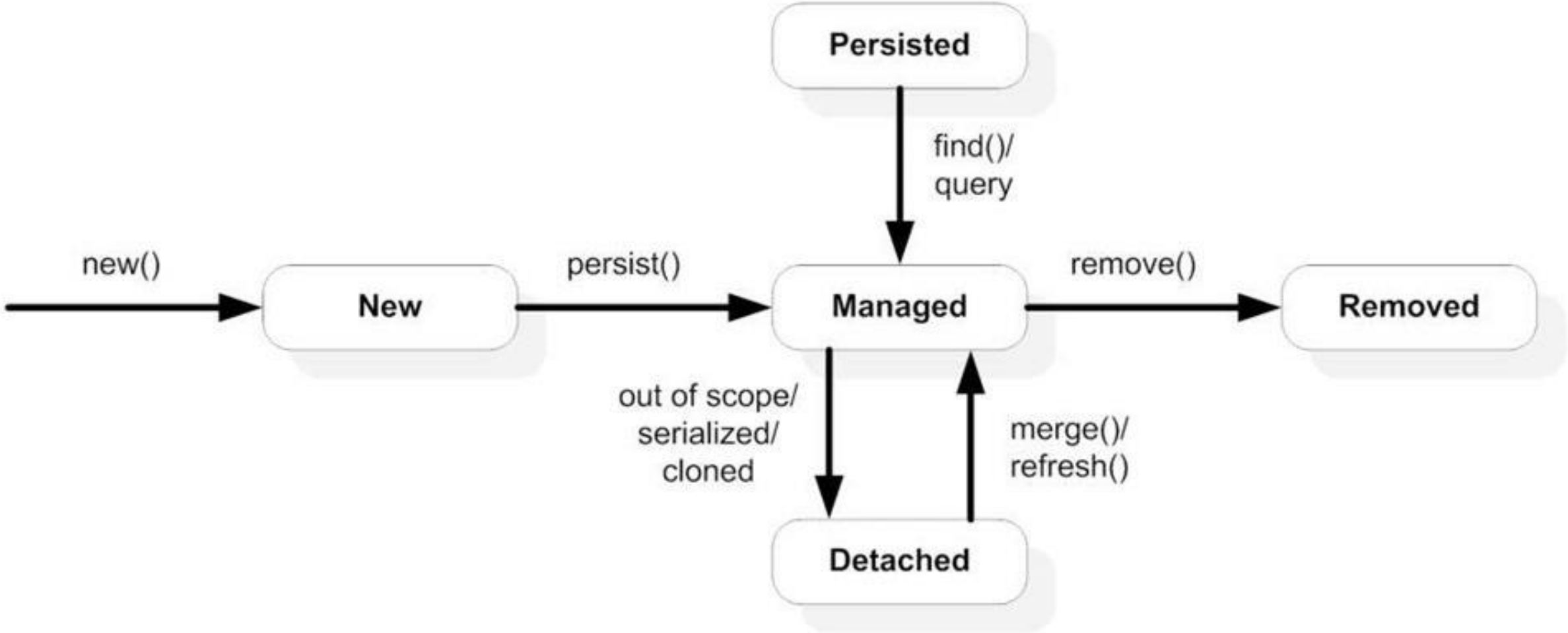
Hibernate/JPA - Primary Key Generation Strategies

- GenerationType.AUTO: Hibernate selects the generation strategy based on the used dialect,
- GenerationType.IDENTITY: Hibernate relies on an auto-incremented database column to generate the primary key,
- GenerationType.SEQUENCE: Hibernate requests the primary key value from a database sequence,
- GenerationType.TABLE: Hibernate uses a database table to simulate a sequence.

Entity Lifecycle Model in JPA & Hibernate

- All entity operations are based on JPA's lifecycle model. It consists of 4 states, which define how your persistence provider handles the entity object.
- **New:** The entity hasn't been persisted yet, so it doesn't represent any database record.
- **Managed:** All entity objects attached to the current persistence context are in the lifecycle state *managed*.
- **Detached:** An entity gets detached when you close the persistence context.
- **Removed:** When you call the remove method on your *EntityManager*, the mapped database record doesn't get removed immediately. The entity object only changes its lifecycle state to *removed*.

Entity Lifecycle Model in JPA & Hibernate



JPA Cascading Operations

- ***PERSIST*:-** The persist operation makes a new instance persistent, **Cascade Type *PERSIST* propagates the persist operation from a parent to a child entity,** When we save the *person* entity, the *address* entity will also get saved.
- ***MERGE*:-** The merge operation copies the state of the given object onto the persistent object with the same identifier. ***CascadeType.MERGE* propagates the merge operation from a parent to a child entity.**
- ***REMOVE*:-** the remove operation removes the row corresponding to the entity from the database and also from the persistent context, ***CascadeType.REMOVE* propagates the remove operation from parent to child entity.**
- ***DETACH*:-** The detach operation removes the entity from the persistent context, **When we use *CascadeType.DETACH*, the child entity will also get removed from the persistent context.**
- ***REFRESH*:-** Refresh operations **reread the value of a given instance from the database.** In some cases, we may change an instance after persisting in the database, but later we need to undo those changes, In that kind of scenario, this may be useful. **When we use this operation with Cascade Type *REFRESH*, the child entity also gets reloaded from the database whenever the parent entity is refreshed.**
- ***ALL*:-** ***CascadeType.ALL* propagates all operations from a parent to a child entity.**

JPA Fetch Type

- Lazy Fetch Type :- Data is not queried until referenced.
- Eager Fetch Type:- Data is queried up front.

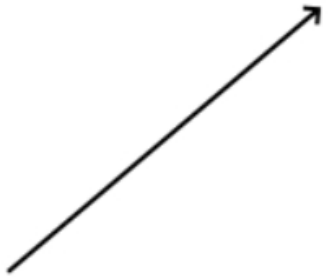
- JPA 2.1 Fetch Type Defaults.
  - OneToMany - Lazy
  - OneToOne - Eager
  - ManyToOne - Eager
  - ManyToMany - Lazy

FetchType.EAGER

```java
@Entity
public class Student
{
	@Id
	@GeneratedValue
	private long id;

	private String name;

	@OneToOne( fetch = FetchType.EAGER )
	private Passport passport;

}
```

```java
@Entity
public class Passport
{
	@Id
	@GeneratedValue
	private long id;

	private String number;

}
```

```java
public Student FetchTypeDemo()
{
	Student student = em.find( Student.class, 21 );
	Passport passport = student.getPassport();
	String number = passport.getNumber();
	return student;
}
```



```
Hibernate:
    select
      student0_.id as id1_3_0_,
      student0_.name as name2_3_0_,
      student0_.passport_id as passport3_3_0_,
      passport1_.id as id1_1_1_,
      passport1_.number as number2_1_1_
    from
      student student0_
    left outer join
      passport passport1_
        on student0_.passport_id=passport1_.id
    where
      student0_.id=?
```

FetchType.LAZY

```java
@Entity
public class Student
{
        @Id
        @GeneratedValue
        private long id;

        private String name;

        @OneToOne( fetch = FetchType.LAZY )
        private Passport passport;

        protected Student()
        {
        }

 }
```

```java
@Entity
public class Passport
{
        @Id
        @GeneratedValue
        private long id;

        private String number;
}
```

```java
public Student FetchTypeDemo()
{
        Student student = em.find( Student.class, 21 );
        Passport passport = student.getPassport();
        String number = passport.getNumber();
        return student;
}
```

```
Hibernate:
    select
        student0_.id as id1_3_0_,
        student0_.name as name2_3_0_,
        student0_.passport_id as passport3_3_0_
    from
        student student0_
    where
        student0_.id=?
```

```
Hibernate:
    select
        passport0_.id as id1_1_0_,
        passport0_.number as number2_1_0_
    from
        passport passport0_
    where
        passport0_.id=?
```

Session Content (9)

- JPA Repositories
- Saving Entities
- Entity State-detection Strategies
- Query Lookup / Derived Queries
- Declared Queries (@Query)
- JPA Named (Native) Queries
- Sorting & Pagination

JPA Repositories

- The goal of the Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
- 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.
- Spring Data JPA Features:-
  - **No-code repository:** It enables us to implement our business code on a higher abstraction level.
  - **Reduced standard code:** It provides the default implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.
  - **Generated Queries:** generate database query based on the method name.
- Spring Data Repository:-
  - **CrudRepository:** It offers standard **create**, **read**, **update**, and **delete** It contains method like **findOne()**, **findAll()**, **save()**, **delete()**,
  - **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the findAll methods. It allows us to **sort** and **retrieve** the data in a paginated way.
  - **JpaRepository:** It extends the both repository CrudRepository and PagingAndSortingRepository. It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context.

Saving Entities

- Saving an entity can be performed with the CrudRepository.save(…) method. It persists or merges the given entity by using the underlying JPA EntityManager. If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the entityManager.persist(…) method. Otherwise, it calls the entityManager.merge(…) method.

- Entity State-detection Strategies
- **Version-Property and Id-Property inspection (default):** By default Spring Data JPA inspects first if there is a Version-property of non-primitive type. If there is, the entity is considered new if the value of that property is null. Without such a Version-property Spring Data JPA inspects the identifier property of the given entity. If the identifier property is null, then the entity is assumed to be new. Otherwise, it is assumed to be not new.
- **Implementing Persistable**
- **Implementing EntityInformation**

Query Lookup / Derived Queries

- The JPA module supports defining a query manually as a String or having it being derived from the **Method Name**.
- Derived queries with the predicates IsStartingWith, StartingWith, StartsWith, IsEndingWith, EndingWith, EndsWith, IsNotContaining, NotContaining, NotContains, IsContaining, Containing, **Contains the respective arguments for these queries will get sanitized**.

- Derived **Find** Query : (findByLastName)
- Derived **Count** Query : (countByLastName)
- Derived **Delete** Query : (deleteByLastName)

JPA Named (Native) Queries

- **@NamedQuery** annotation. The queries for these configuration elements have to be defined in the JPA query language. Of course, you can use **@NamedNativeQuery** too. These elements let you define the query in native SQL by losing the database platform independence.
- Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined earlier instead of trying to create a query from the method name.

Declared Queries (@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 runs 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.
- The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following
- Spring Data JPA does not currently support dynamic sorting for native queries, because it would have to manipulate the actual query declared, which it cannot do reliably for native SQL. You can, however, use native queries for pagination by specifying the count query yourself

Sorting & Pagination

- Sorting can be done by either providing a PageRequest or by using Sort directly. The properties actually used within the Order instances of Sort need to match your domain model, which means they need to resolve to either a property or an alias used within the query. The JPQL defines this as a state field path expression.
- The **Pageable** interface contains the information about the requested page such as the size, the number of the page, or sort information with Sort object.
- So when we want to make paging and sorting (with or without filter) in the results, we just add Pageable to the definition of the method as a parameter.

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,

```
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);
}
```

Using SpEL Expressions

- Spring Data JPA supports a variable called entityName. Its usage is *select x from #{#entityName} x*. It inserts the entityName of the domain type associated with the given repository. The entityName is resolved as follows: If the domain type has set the name property on the @Entity annotation, it is used. Otherwise, the simple class-name of the domain type is used.

```
public interface UserRepository extends JpaRepository<User,Long> {

  @Query("select u from #{#entityName} u where u.lastname = ?1")
  List<User> findByLastname(String lastname);
}
```

Session Content (10)

- **Projections**
- **Transactional**
- **Fetch- and LoadGraphs**

Projections

- Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository. However, it might sometimes be desirable to create projections based on certain attributes of those types. Spring Data allows modeling dedicated return types, to more **selectively retrieve partial views** of the managed aggregates.
- **Interface-based Projections:** The easiest way to limit the result of the queries to only the name attributes is by declaring an interface that exposes accessor methods for the properties to be read.
- The query execution engine creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.
- Projections can be used recursively. If you want to include some of the Address information as well, create a projection interface for that and return that interface from the declaration of getAddress()
- **Interface Closed Projections:** A projection interface whose accessor methods all **match properties** of the target aggregate is considered to be a closed projection.
- **Interface Open Projections:** Accessor methods **not matched properties**, Accessor methods in projection interfaces can also be used to compute new values by using the @Value annotation

Class-based Projections

- Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved. These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied.
- If the store optimizes the query execution by limiting the fields to be loaded, the fields to be loaded are determined from the parameter names of the constructor that is exposed.

Transactional

- By default, CRUD methods on repository instances inherited from SimpleJpaRepository are transactional. For read operations, the transaction configuration readOnly flag is set to true. All others are configured with a plain @Transactional so that default transaction configuration applies. Repository methods that are backed by transactional repository fragments inherit the transactional attributes from the actual fragment method.
- If you need to change transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface.
- Another way to alter transactional behavior is to use a service implementation that (typically) covers more than one repository.
- Transactional query methods to let your query methods be transactional, use @Transactional at the repository interface you define with @Query.
- **@Transacional** annotation is used for indicating a method run inside a database transaction. It can also be annotated on the class level which applies as a default to all methods of the declaring class and its subclasses
- You can use rollbackFor to indicate which exception types must cause a transaction rollback. By default, they are unchecked exceptions including RuntimeException, Error and their subclasses
- Use noRollbackFor to indicate which exception types must not cause a transaction rollback

Session Content (11)

- Fetch- and LoadGraphs
- Some Best Practices
- Calculated Attributes

Configuring Fetch- and LoadGraphs

- The JPA 2.1 specification introduced support for specifying Fetch- and LoadGraphs that we also support with the @EntityGraph annotation, which lets you reference a @NamedEntityGraph definition. You can use that annotation on an entity to configure the fetch plan of the resulting query. The type (Fetch or Load) of the fetching can be configured by using the type attribute on the @EntityGraph annotation.

Notes

- Pay attention to remove operations, especially to removing child entities. The CascadeType.REMOVE and orphanRemoval=true may produce too many queries. In such scenarios, relying on bulk operations is most of the time the best way to go for deletions.
- Entity classes have different requirements than plain Java classes. That makes Lombok's generated equals() and hashCode() methods unusable and its toString() method risky to use.
- The bottom line is that you can use the @Getter, @Setter, and @Builder annotation without breaking your application. The only Lombok annotations you need to avoid are @Data, @ToString, and @EqualsAndHashCode.
- From a performance perspective, it is advisable to use findById(), find(), or get() instead of an explicit JPQL/SQL to fetch an entity by ID. That way, if the entity is present in the current Persistence Context, there is no SELECT triggered against the database and no data snapshot to be ignored
- if toString() need to be overridden, then pay attention to involve only the basic attributes fetched when the entity is loaded from the database

The most efficient way to delete all entities via a bulk deletion can be done via the built-in deleteAllInBatch() or

```
@Transactional
@Modifying(flushAutomatically = true, clearAutomatically = true)
@Query("DELETE FROM Book b WHERE b.author.id=?1")
public int deleteByAuthorIdentifier(Long id);

@Transactional
@Modifying(flushAutomatically = true, clearAutomatically = true)
@Query("DELETE FROM Book b WHERE b.author.id IN ?1")
public int deleteBulkByAuthorIdentifier(List<Long> id);

@Transactional
@Modifying(flushAutomatically = true, clearAutomatically = true)
@Query("DELETE FROM Book b WHERE b.author IN ?1")
public int deleteBulkByAuthors(List<Author> authors);
```

Example

```
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(name = "author_book",
        joinColumns = @JoinColumn(name = "author_id"),
        inverseJoinColumns = @JoinColumn(name = "book_id")
)
private Set<Book> books = new HashSet<>();

public void addBook(Book book) {
    this.books.add(book);
    book.getAuthors().add(this);
}

public void removeBook(Book book) {
    this.books.remove(book);
    book.getAuthors().remove(this);
}

public void removeBooks() {
    Iterator<Book> iterator = this.books.iterator();

    while (iterator.hasNext()) {
        Book book = iterator.next();

        book.getAuthors().remove(this);
        iterator.remove();
    }
}
```

In Auther Entity

```
@ManyToMany(mappedBy = "books")
private Set<Author> authors = new HashSet<>();
```

In Book Entity

The Best Way To Implement A Bidirectional with @ManyToMany

- Choose an owning and a mappedBy side
- Materialize the relationships collections via Set not List
- Use helper methods on the owner of the relationship to keep both sides of the association in sync
- On the owner of the relationship set up join table
- @ManyToMany is lazy by default keep it this way!

Prefer Set Instead of List in @ManyToMany Associations

inefficient

```
========================================
Removing a book (List case) ...
========================================
```

Removes all *book*-rows associated to the given *author* and reinserts the remaining ones back afterward

```
delete from author_book_list where author_id=?
insert into author_book_list (author_id, book_id) values (?, ?)
insert into author_book_list (author_id, book_id) values (?, ?)
```

efficient

```
========================================
Removing a book (Set case) ...
========================================
```

Single DELETE

```
delete from author book set where author id=? and book id=?
```

Difference between getOne and findById in Spring Data JPA?

- Both findById() and getOne() methods are used to retrieve an object from underlying datastore. But the underlying mechanism for retrieving records is different for both these methods, infact getOne() is lazy operation which does not even hit the database.
- **getOne() method:-** returns a reference to the entity with the given identifier.
- **findById() method:-** This method will actually hit the database and return the real object mapping to a row in the database.
- **Which one to choose:-** The only real difference between these methods is about the performance. Lazily loaded getOne() method avoids database roundtrip from the JVM as it never hits the database until the properties of returned proxy object are actually accessed.
- There are scenarios when you just want to get retrieve an entity from database and assign it as a reference to another object, just to maintain the relationship (OneToOne or ManyToOne).

Calculated Attributes

- **Calculate Non-Persistent Property via JPA @Transient**
  - annotate the non-persistent field and property with @Transient
- **Calculate Non-Persistent Property via JPA @PostLoad**
  - annotate the non-persistent field and property with @Transient
  - define a method annotated with @PostLoad that calculates this non-persistent property based on the persistent entity attributes.
- **Calculate Non-Persistent Property via Hibernate @Formula**
  - annotate the non-persistent field with @Formula
  - as the value of @Formula add the SQL query expression that calculates this non-persistent property based on the persistent entity attributes

Spring Framework 6.0 M1 released

- https://spring.io/blog/2021/12/16/spring-framework-6-0-m1-released

Session Content (12)

- JPA Auditing
- Base Layer
- JPA Logging
- Data source Properties
- Hibernate Soft Deletes

JPA Auditing

- Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change happened. To benefit from that functionality, you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface. Additionally, auditing has to be enabled either through Annotation configuration or XML configuration to register the required infrastructure components.
- We provide @CreatedBy and @LastModifiedBy to capture the user who created or modified the entity as well as @CreatedDate and @LastModifiedDate to capture when the change happened.
- The annotations capturing when changes were made can be used on properties of type Joda-Time, DateTime, legacy Java Date and Calendar, JDK8 date and time types, and long or Long.

Hibernate Soft Deletes

- Soft deletion implementation can be Hibernate-centric. Start by defining an abstract class annotated with @MappedSuperclass and containing a flag-field named deleted. This field is true for a deleted record and false (default) for an available record.
- Marked with Hibernate-specific @Where annotations, @Where(clause = "deleted = false"); this helps Hibernate filter the soft deleted records by appending this SQL condition to entity queries.
- Marked with Hibernate-specific @SQLDelete annotations to trigger UPDATE SQL statements in place of DELETE SQL statements; removing an entity will result in updating the deleted column to true instead of a physical delete of the record.

Session Content (13)

- Hibernate Validators
- Custom Validator
- JPA Specifications
- Call Stored Procedure

Hibernate Validators

- Bean Validation works by defining constraints to the fields of a class by annotating them with certain annotations.
- @Validated annotation is a class-level annotation that we can use to tell Spring to validate parameters that are passed into a method of the annotated class.
- @Valid annotation on method parameters and fields to tell Spring that we want a method parameter or field to be validated.
- @NotNull: to say that a field must not be null.
- @NotEmpty: to say that a list field must not empty.
- @NotBlank: to say that a string field must not be the empty string (i.e. it must have at least one character).
- @Min and @Max: to say that a numerical field is only valid when it's value is above or below a certain value.
- @Pattern: to say that a string field is only valid when it matches a certain regular expression.
- @Email: to say that a string field must be a valid email address.

JPA Specification

- Spring Data JPA Specifications allow us to create dynamic database queries by using the JPA Criteria API. It defines a specification as a predicate over an entity.
- Spring has a wrapper around the JPA criteria API (that uses predicates) and is called the specification API.
- Spring Data JPA repository abstraction allows executing predicates via JPA Criteria API predicates wrapped into a Specification object. To enable this functionality, you simply let your repository extend JpaSpecificationExecutor.

Call Stored Procedure

- https://www.baeldung.com/spring-data-jpa-stored-procedures
- https://www.appsdeveloperblog.com/calling-a-stored-procedure-in-spring-boot-rest-with-jpa/

Session Content (15)

- SQL VS NoSQL
- MongoDB Installation
- MongoDB Repository
- MongoDB Template

SQL VS NoSQL

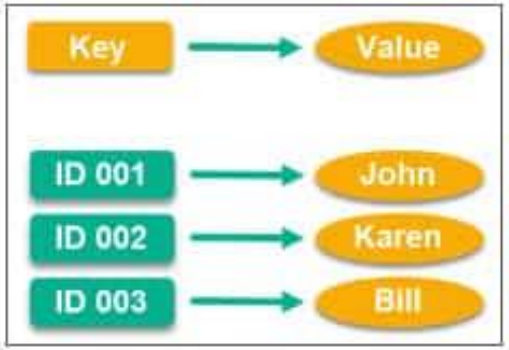
SQL Databases

Table

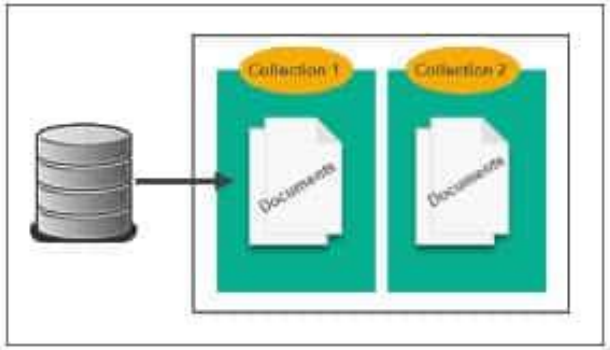
ID	Name	Grade	GPA
001	John	Senior	4.00
002	Karen	Freshman	3.67
003	Bill	Junior	3.33

NoSQL Databases

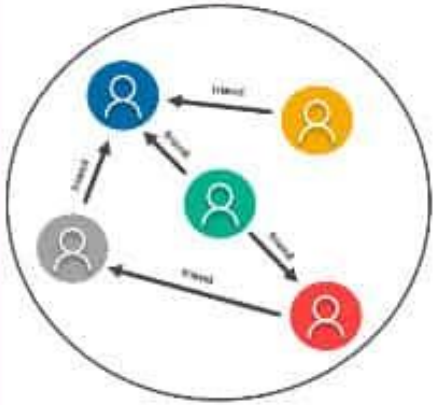
Key-value



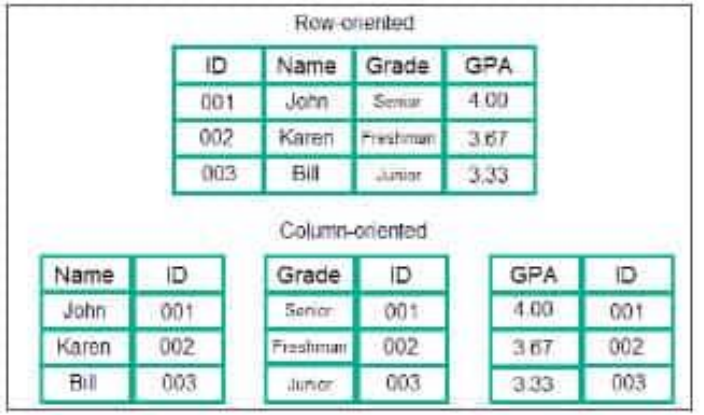
Document



Graph



Wide-column



	SQL	NoSQL
Type	Relational	Non-Relational
Data	Structured Data stored in Tables	Un-structured stored in JSON files but the graph database does supports relationship
Schema	Static	Dynamic
Scalability	Vertical	Horizantal
Language	Structured Query Language	Un-structured Query Language
Joins	Helpful to design complex queries	No joins, Don't have the powerful interface to prepare complex query
OLTP	Recommended and best suited for OLTP systems	Less likely to be considered for OLTP system
Support	Great support	community depedent, they are expanding the support model
Integrated Caching	Supports In-line memory(SQL2014 and SQL 2016)	Supports integrated caching
flexible	rigid schema bound to relationship	Non-rigid schema and flexible
Transaction	ACID	CAP theorem
Auto elasticity	Requires downtime in most cases	Automatic, No outage required