### JPA using Hibernate: Introduction

#### Definition

Java Persistence API (JPA) is a specification for accessing, persisting, and managing data between Java objects/classes and a relational database. Hibernate is an ORM (Object-Relational Mapping) framework that implements JPA, providing a way to map Java objects to database tables.

#### Necessity

- \*\*Abstraction over JDBC\*\*: JPA abstracts the boilerplate code required for database operations, making development faster and cleaner.

- \*\*Object-Relational Mapping\*\*: It bridges the gap between object-oriented programming and relational databases, allowing for the use of objects to represent data in a database.

- \*\*Consistency\*\*: JPA standardizes the way data persistence is handled, ensuring consistency across different projects and teams.

#### Features/Advantages

- \*\*Simplified Database Access\*\*: Using JPA and Hibernate, developers can perform complex database operations with simple Java methods.

- \*\*Automatic Table Creation\*\*: Hibernate can generate database tables based on entity mappings.

- \*\*Lazy Loading\*\*: Hibernate supports lazy loading, which fetches related data only when it’s accessed, improving performance.

- \*\*Caching\*\*: Hibernate provides first-level and second-level caching to reduce database access.

#### Example Scenario

Let's consider an example of an online bookstore. We'll use JPA with Hibernate to manage books and authors.

#### Programming Example

1. \*\*Entities\*\*

\*\*Book.java\*\*

```java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToOne;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String isbn;

@ManyToOne

private Author author;

// Getters and setters

}

```

\*\*Author.java\*\*

```java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToMany;

import java.util.List;

@Entity

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "author")

private List<Book> books;

// Getters and setters

}

```

2. \*\*Managing Entities\*\*

\*\*PersistenceConfig.java\*\*

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;

import javax.sql.DataSource;

@Configuration

@EnableTransactionManagement

public class PersistenceConfig {

@Bean

public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(dataSource);

em.setPackagesToScan(new String[] { "com.example.bookstore" });

// Add additional configuration settings if needed

return em;

}

@Bean

public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {

return new JpaTransactionManager(emf);

}

}

```

3. \*\*Querying Entities\*\*

\*\*BookRepository.java\*\*

```java

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface BookRepository extends JpaRepository<Book, Long> {

List<Book> findByTitle(String title);

List<Book> findByAuthorName(String authorName);

}

```

\*\*AuthorRepository.java\*\*

```java

import org.springframework.data.jpa.repository.JpaRepository;

public interface AuthorRepository extends JpaRepository<Author, Long> {

}

```

4. \*\*Entity Relationships\*\*

- \*\*One-to-Many Relationship\*\*: An `Author` can have multiple `Books`.

- \*\*Many-to-One Relationship\*\*: A `Book` belongs to one `Author`.

#### Example Code with Relationships

\*\*BookService.java\*\*

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

@Autowired

private AuthorRepository authorRepository;

public Book addBook(String title, String isbn, Long authorId) {

Author author = authorRepository.findById(authorId).orElseThrow(() -> new RuntimeException("Author not found"));

Book book = new Book();

book.setTitle(title);

book.setIsbn(isbn);

book.setAuthor(author);

return bookRepository.save(book);

}

public List<Book> getBooksByTitle(String title) {

return bookRepository.findByTitle(title);

}

public List<Book> getBooksByAuthorName(String authorName) {

return bookRepository.findByAuthorName(authorName);

}

}

```

#### Conclusion

JPA with Hibernate simplifies database operations, making it easier to manage complex relationships and perform CRUD operations with minimal code. The example above shows how entities can be mapped and managed using JPA annotations, with a focus on the online bookstore scenario to illustrate practical usage.

---

### Next Topic: Core Java 8: Introduction to Design Pattern

#### Definition

Design patterns are proven solutions to common problems in software design. They represent best practices and provide templates for solving recurring design problems.

#### Types of Design Patterns

- \*\*Creational Patterns\*\*: Deal with object creation mechanisms.

- \*\*Structural Patterns\*\*: Deal with object composition or structure.

- \*\*Behavioral Patterns\*\*: Deal with object interaction and responsibilities.

#### Necessity

- \*\*Reusability\*\*: Promote code reuse and improve code readability.

- \*\*Maintainability\*\*: Simplify code maintenance by providing a clear structure.

- \*\*Efficiency\*\*: Provide tested and optimized solutions.

#### Features/Advantages

- \*\*Standardization\*\*: Provide a standard terminology and approach.

- \*\*Best Practices\*\*: Incorporate industry best practices.

- \*\*Improved Communication\*\*: Facilitate better communication among developers.

#### Example Scenario

Continuing with the online bookstore, let's consider a scenario where we need to create different types of books (e.g., EBooks, PaperBooks). We'll use the Factory Pattern to manage the creation of these objects.

#### Programming Example

\*\*Book.java\*\*

```java

public abstract class Book {

private String title;

private String isbn;

// Getters and setters

public abstract void read();

}

```

\*\*EBook.java\*\*

```java

public class EBook extends Book {

@Override

public void read() {

System.out.println("Reading EBook on an electronic device.");

}

}

```

\*\*PaperBook.java\*\*

```java

public class PaperBook extends Book {

@Override

public void read() {

System.out.println("Reading PaperBook by flipping pages.");

}

}

```

\*\*BookFactory.java\*\*

```java

public class BookFactory {

public static Book createBook(String type) {

if (type.equalsIgnoreCase("EBOOK")) {

return new EBook();

} else if (type.equalsIgnoreCase("PAPERBOOK")) {

return new PaperBook();

}

return null;

}

}

```

\*\*Main.java\*\*

```java

public class Main {

public static void main(String[] args) {

Book ebook = BookFactory.createBook("EBOOK");

ebook.setTitle("EBook Title");

ebook.setIsbn("1234567890");

ebook.read();

Book paperbook = BookFactory.createBook("PAPERBOOK");

paperbook.setTitle("PaperBook Title");

paperbook.setIsbn("0987654321");

paperbook.read();

}

}

```

#### Conclusion

Design patterns like the Factory Pattern provide a structured approach to solving common design problems. In this example, the Factory Pattern helps manage the creation of different types of books, making the code more modular and easier to maintain.

---

### Next Topic: Spring Core: Spring Core Introduction / Overview

#### Definition

Spring Core is the fundamental part of the Spring Framework, providing the core functionality and features required to build robust Java applications.

#### Overview

- \*\*Inversion of Control (IoC)\*\*: Manages the life cycle and configuration of application objects.

- \*\*Dependency Injection (DI)\*\*: Facilitates injecting dependencies into objects, promoting loose coupling.

- \*\*Aspect-Oriented Programming (AOP)\*\*: Provides support for separating cross-cutting concerns.

#### Necessity

- \*\*Simplification\*\*: Simplifies Java development by managing dependencies and object lifecycle.

- \*\*Modularity\*\*: Promotes modular application design through DI and AOP.

- \*\*Testability\*\*: Improves testability by allowing easy replacement of dependencies with mocks.

#### Features/Advantages

- \*\*Lightweight\*\*: The core container is lightweight, providing basic functionality without excessive overhead.

- \*\*Integration\*\*: Integrates seamlessly with various frameworks and technologies.

- \*\*Flexibility\*\*: Supports multiple configuration methods (XML, annotations, Java-based).

#### Example Scenario

Continuing with the online bookstore, let's manage the book and author services using Spring Core.

#### Programming Example

\*\*ApplicationConfig.java\*\*

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ApplicationConfig {

@Bean

public BookService bookService() {

return new BookService();

}

@Bean

public AuthorService authorService() {

return new AuthorService();

}

}

```

\*\*BookService.java\*\*

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class BookService {

@Autowired

private AuthorService authorService;

// Book related methods

}

```

\*\*AuthorService.java\*\*

```java

import org.springframework.stereotype.Service;

@Service

public class AuthorService {

// Author related methods

}

```

\*\*Main.java\*\*

```java

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(ApplicationConfig.class);

BookService bookService = context.getBean(BookService.class);

// Use bookService methods

}

}

```

#### Conclusion

Spring Core provides the essential features and functionalities to build scalable and maintainable Java applications. By using IoC and DI, it promotes modular and testable code. The example demonstrates how to manage services in a Spring application using annotations and Java-based configuration.

---

### Next Steps

We will continue with Spring MVC, Spring Boot, and Spring Data JPA in subsequent lessons.

**JPA using Hibernate: Entities, Managing Entities, Querying Entities [ Big blunder avoid for now ]**

**Entities**

**Definition**

In JPA, an entity is a lightweight, persistent domain object that typically represents a table in a relational database. Each instance of the entity corresponds to a row in the table.

**Necessity**

* **Mapping**: Entities are used to map Java objects to database tables.
* **Persistence**: They manage the state of objects in the context of the database lifecycle.
* **Relationships**: Entities allow defining relationships between different tables (e.g., One-to-Many, Many-to-Many).

**Features**

* **Annotations**: Use annotations to specify the mapping between the class and the database table.
* **Primary Key**: Every entity must have a primary key.
* **Lifecycle Callbacks**: Define methods that can act on certain lifecycle events (e.g., @PrePersist, @PostLoad).

**Example**

**Book.java**

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToOne;

@Entity

public class Book {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String title;

    private String isbn;

    @ManyToOne

    private Author author;

    // Getters and setters

}

Author.java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToMany;

import java.util.List;

@Entity

public class Author {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")

    private List<Book> books;

    // Getters and setters

}

**Managing Entities**

**Definition**

Managing entities involves creating, reading, updating, and deleting (CRUD) entities in the database.

**Necessity**

* **Data Persistence**: Ensures data is saved and retrieved from the database.
* **Transactions**: Manage database transactions to ensure data consistency.
* **Session Management**: Handle the session/EntityManager lifecycle to manage entity states.

**Features**

* **EntityManager**: Primary JPA interface to interact with the persistence context.
* **Transactions**: Manage transactions using @Transactional annotation.
* **CRUD Operations**: Simplify database operations through repository interfaces.

**Example**

**PersistenceConfig.java**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;

import javax.sql.DataSource;

@Configuration

@EnableTransactionManagement

public class PersistenceConfig {

    @Bean

    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {

        LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

        em.setDataSource(dataSource);

        em.setPackagesToScan(new String[] { "com.example.bookstore" });

        return em;

    }

    @Bean

    public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {

        return new JpaTransactionManager(emf);

    }

}

BookService.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service

public class BookService {

    @Autowired

    private BookRepository bookRepository;

    @Transactional

    public Book addBook(String title, String isbn, Long authorId) {

        Author author = authorRepository.findById(authorId).orElseThrow(() -> new RuntimeException("Author not found"));

        Book book = new Book();

        book.setTitle(title);

        book.setIsbn(isbn);

        book.setAuthor(author);

        return bookRepository.save(book);

    }

    public List<Book> getAllBooks() {

        return bookRepository.findAll();

    }

    public void deleteBook(Long id) {

        bookRepository.deleteById(id);

    }

}

**Querying Entities**

**Definition**

Querying entities involves retrieving data from the database based on certain criteria using JPQL (Java Persistence Query Language) or Criteria API.

**Necessity**

* **Data Retrieval**: Fetch data from the database as needed by the application.
* **Dynamic Queries**: Create flexible queries to filter and sort data based on user input.
* **Efficiency**: Retrieve only the necessary data to optimize performance.

**Features**

* **JPQL**: A SQL-like query language for JPA.
* **Criteria API**: Type-safe way to create queries programmatically.
* **Named Queries**: Define static queries at the entity level using @NamedQuery.

**Example**

**BookRepository.java**

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface BookRepository extends JpaRepository<Book, Long> {

    List<Book> findByTitle(String title);

    List<Book> findByAuthorName(String authorName);

}

BookService.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service

public class BookService {

    @Autowired

    private BookRepository bookRepository;

    @Autowired

    private AuthorRepository authorRepository;

    @Transactional

    public Book addBook(String title, String isbn, Long authorId) {

        Author author = authorRepository.findById(authorId).orElseThrow(() -> new RuntimeException("Author not found"));

        Book book = new Book();

        book.setTitle(title);

        book.setIsbn(isbn);

        book.setAuthor(author);

        return bookRepository.save(book);

    }

    public List<Book> getAllBooks() {

        return bookRepository.findAll();

    }

    public List<Book> getBooksByTitle(String title) {

        return bookRepository.findByTitle(title);

    }

    public List<Book> getBooksByAuthorName(String authorName) {

        return bookRepository.findByAuthorName(authorName);

    }

    public void deleteBook(Long id) {

        bookRepository.deleteById(id);

    }

}