PROGRAM STUDI INFORMATIKA
UNIVERSITAS SANATA DHARMA

PRAKTIKUM PBO LANJUT

MODUL PENGANTAR SPRING-2

• TUJUAN PRAKTIKUM

1. Memahami penerapan **IoC** dan **DI** dalam skenario yang lebih kompleks.
2. Mengimplementasikan berbagai jenis **Injection** (Constructor Injection, Setter Injection, Field Injection).
3. Menggunakan anotasi **@Component**, **@Autowired**, dan **@Qualifier** untuk **Dependency Injection (DI)**.
4. Menggunakan **Stereotype Annotations**: **@Controller**, **@Service**, dan **@Repository**.

• DASAR TEORI

1. Injection (Dependency Injection)

Dependency Injection (DI) adalah salah satu bentuk implementasi dari prinsip **Inversion of Control (IoC)**, yang berarti bahwa kontrol pembuatan dan pengelolaan object diserahkan ke framework atau container, seperti Spring. Konsep DI sangat penting dalam pemrograman berbasis obyek karena memungkinkan kelas atau obyek untuk mendapatkan dependensi dari luar, daripada membuatnya sendiri.

Cara Kerja:

- **Constructor Injection:** Dependensi diinjeksikan melalui konstruktor object. Ini biasanya digunakan ketika dependensi bersifat wajib.
- **Setter Injection:** Dependensi diinjeksikan melalui metode setter. Cocok digunakan jika dependensi bersifat opsional.
- **Field Injection:** Dependensi langsung diinjeksikan ke dalam field tanpa melalui konstruktor atau setter. Ini bisa dilakukan dengan anotasi seperti `@Autowired` di Spring.

Manfaat DI:

- **Loose Coupling:** Kelas tidak perlu mengetahui detail tentang bagaimana dependensi dibuat, sehingga memudahkan pengelolaan dan pengujian.
- **Testability:** Kelas yang di-inject mudah untuk diuji karena kita bisa mengganti dependensi dengan mock atau dummy dalam pengujian unit.

Contoh:

- Tanpa DI: Object bertanggung jawab membuat dependensinya sendiri

```
public class Car {
    private Engine engine = new Engine();
}
```

- Dengan DI (Constructor Injection)

```
public class Car {
    private Engine engine;
```

```
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

2. Anotasi pada Spring

1. @Component

- @Component adalah anotasi di Spring yang digunakan untuk mendefinisikan sebuah kelas sebagai Spring Bean yang dikelola oleh Spring IoC Container.
- Kelas yang ditandai dengan @Component akan dipindai oleh Spring melalui mekanisme Component Scanning, yang secara otomatis mendaftarkan bean ini dalam konteks aplikasi.

Fungsi:

- Anotasi ini memberitahu Spring bahwa kelas tersebut adalah Spring Bean, dan bean tersebut harus dikelola oleh IoC container.
- Bisa digunakan di berbagai jenis komponen seperti service, repository, atau controller.

Contoh Penggunaan:

```
@Component
public class Engine {
    private String type = "V6";
    // kode lain
}
```

Dalam contoh ini, kelas `Engine` diidentifikasi oleh Spring sebagai bean, sehingga dapat di-inject ke komponen lain secara otomatis.

2. @Autowired

- @Autowired digunakan untuk melakukan Dependency Injection (DI) di Spring. Anotasi ini memberitahu Spring bahwa suatu dependensi perlu di-inject secara otomatis oleh IoC container.
- Spring akan mencari bean yang sesuai dalam konteks aplikasi untuk di-inject ke dalam field, konstruktor, atau metode setter yang ditandai dengan anotasi @Autowired.

Fungsi:

- Menerapkan injection otomatis tanpa perlu membuat objek secara manual.
- Spring akan secara otomatis memasukkan bean yang dibutuhkan di field atau metode yang ditandai dengan anotasi ini.

Jenis Injection yang Didukung:

- Constructor Injection: Dependensi disuntikkan melalui konstruktor.
- Setter Injection: Dependensi disuntikkan melalui metode setter.
- Field Injection: Dependensi disuntikkan langsung ke field.

Contoh Penggunaan:

```
@Autowired
private BookRepository bookRepository;
```

Di sini, Spring akan secara otomatis mencari bean `BookRepository` dan meng-inject-nya ke dalam `BookService`.

3. @Qualifier

- @Qualifier digunakan bersamaan dengan @Autowired untuk menyelesaikan ambiguitas ketika ada lebih dari satu bean yang memenuhi syarat untuk di-inject.
- Ketika terdapat beberapa bean dengan tipe yang sama, @Qualifier memungkinkan Anda untuk menentukan bean mana yang harus di-inject berdasarkan nama.

Fungsi:

- Menentukan secara spesifik bean mana yang akan digunakan ketika ada lebih dari satu bean yang cocok dengan tipe dependensi yang di-inject.
- Memecahkan masalah ambiguitas di mana beberapa implementasi dari satu interface tersedia di Spring IoC container.

Contoh Penggunaan:

```
@Autowired
@Qualifier("bookRepositoryImpl")
private BookRepository bookRepository;
```

Di sini, meskipun terdapat lebih dari satu implementasi `BookRepository`, Spring akan menggunakan bean dengan nama "bookRepositoryImpl".

Dalam aplikasi Spring Core, anotasi ini memungkinkan Dependency Injection yang mempermudah pengelolaan dependensi antar komponen. Berikut adalah bagaimana ketiga anotasi ini bekerja bersama:

1. @Component: Kelas yang ditandai dengan @Component akan didaftarkan oleh Spring sebagai bean yang dikelola oleh IoC container.
2. @Autowired: Spring secara otomatis akan memasukkan bean yang tepat ke dalam kelas yang membutuhkan dependensi tersebut.
3. @Qualifier: Jika ada lebih dari satu implementasi yang cocok untuk di-inject, @Qualifier digunakan untuk memilih bean yang tepat.

Contoh:

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    @Qualifier("v8Engine")
    public void setEngine(Engine engine) {
```

```
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving.");
    }
}
```

Dalam contoh ini:
- Car dan Engine keduanya didefinisikan sebagai komponen Spring dengan @Component.
- @Autowired digunakan untuk meng-inject dependensi `Engine` ke `Car`.
- @Qualifier memastikan bahwa Spring memilih implementasi Engine yang tepat (misalnya, "v8Engine" jika ada beberapa implementasi yang tersedia).

Dengan penggunaan ketiga anotasi ini, Anda dapat secara efisien menerapkan Dependency Injection dan membuat aplikasi yang modular dan mudah diuji.

3. Layer dalam Arsitektur Aplikasi

Arsitektur berbasis **layer** adalah pendekatan di mana aplikasi dipisahkan menjadi beberapa lapisan, dengan masing-masing lapisan bertanggung jawab atas fungsi tertentu dalam aplikasi. Pemisahan ini membuat aplikasi lebih modular, mudah dikelola, diuji, dan diatur. Setiap lapisan dalam arsitektur aplikasi berinteraksi dengan lapisan di atas dan di bawahnya melalui antarmuka yang terdefinisi dengan baik.

Manfaat Arsitektur Layered:
1. Modularitas: Memisahkan aplikasi menjadi lapisan-lapisan membuatnya lebih mudah untuk dipelihara dan dimodifikasi. Jika ada perubahan di satu lapisan, lapisan lainnya tidak akan terpengaruh selama antarmuka antar lapisan tetap konsisten.
2. Skalabilitas: Aplikasi dapat diperluas atau dioptimalkan pada level tertentu tanpa mempengaruhi keseluruhan sistem.
3. Reusability: Komponen dalam lapisan dapat digunakan kembali di bagian lain dari aplikasi atau di aplikasi yang berbeda.
4. Testability: Setiap lapisan dapat diuji secara independen, meningkatkan efisiensi pengujian unit dan integrasi.

Jenis-jenis Layer dalam Arsitektur Aplikasi
1. Presentation Layer (Lapisan Presentasi)
2. Business Logic Layer (Lapisan Logika Bisnis)
3. Service Layer (Lapisan Layanan)
4. Data Access Layer (Lapisan Akses Data)
5. Persistence Layer (Lapisan Persistensi)
6. Integration Layer (Lapisan Integrasi)

1. Presentation Layer (Lapisan Presentasi)
Fungsi:
- Lapisan presentasi adalah lapisan yang berhubungan langsung dengan pengguna. Ini adalah lapisan di mana antarmuka pengguna (UI) atau antarmuka aplikasi seperti aplikasi web, desktop, atau mobile dibuat.
- Lapisan ini menangani input pengguna dan menampilkan output dari data yang diproses oleh lapisan di bawahnya.

Contoh:
- Dalam aplikasi web, ini bisa berupa HTML, CSS, dan JavaScript.
- Pada aplikasi desktop, ini bisa berupa elemen UI yang ditampilkan kepada pengguna.

Tugas Utama:
- Mengelola input pengguna dan mengirimkannya ke lapisan logika bisnis.
- Mengambil data dari lapisan logika bisnis dan menampilkannya dalam format yang dapat dimengerti oleh pengguna.

Controller adalah bagian dari Presentation Layer, yang berfungsi sebagai penghubung antara pengguna (melalui antarmuka atau API) dan lapisan logika bisnis. Ini adalah titik masuk pertama yang mengarahkan pengguna ke layanan yang sesuai.

2. Business Logic Layer (Lapisan Logika Bisnis)
Fungsi:
- Ini adalah inti dari aplikasi yang mengandung aturan bisnis, alur kerja, dan logika yang terkait dengan pengolahan data.
- Lapisan ini memutuskan bagaimana data diolah dan berinteraksi dengan lapisan data atau layanan lainnya.

Contoh:
- Dalam aplikasi e-commerce, logika bisnis mungkin termasuk perhitungan pajak, verifikasi inventaris, dan pengelolaan pembayaran.

Tugas Utama:
- Menyediakan logika bisnis yang memproses input dari lapisan presentasi.
- Menerapkan aturan-aturan khusus yang terkait dengan proses bisnis, seperti perhitungan diskon, validasi, dan pengelolaan transaksi.

3. Service Layer (Lapisan Layanan)
Fungsi:
- Lapisan layanan digunakan untuk mengabstraksi logika bisnis dari lapisan yang lebih rendah seperti lapisan akses data.
- Menyediakan antarmuka yang konsisten untuk berinteraksi dengan logika bisnis, sehingga logika bisnis tetap tersembunyi dari detail implementasi lapisan bawah.

Contoh:

- Pada sistem manajemen pelanggan (CRM), layanan untuk menambahkan pelanggan baru mungkin memanggil beberapa lapisan lain, seperti penyimpanan data atau validasi identitas.

Tugas Utama:
- Mengabstraksi akses ke logika bisnis sehingga lapisan presentasi atau kontroler tidak perlu mengetahui detail implementasi.

4. Data Access Layer (Lapisan Akses Data)
Fungsi:
- Lapisan ini menangani semua interaksi dengan sumber data, seperti database atau API eksternal.
- Semua operasi CRUD (Create, Read, Update, Delete) dilakukan di lapisan ini.

Contoh:
- Pada aplikasi yang menggunakan SQL, lapisan ini akan berisi query ke database seperti MySQL, PostgreSQL, atau MongoDB untuk mengakses atau memanipulasi data.

Tugas Utama:
- Mengambil, menyimpan, dan memperbarui data yang dibutuhkan oleh lapisan logika bisnis.
- Mengabstraksi detail akses data sehingga lapisan atas (logika bisnis dan presentasi) tidak perlu mengetahui detail implementasi database.

Repository, dalam konteks arsitektur layered, termasuk ke dalam **Data Access Layer.** Repository bertanggung jawab untuk menangani semua operasi yang berkaitan dengan interaksi dengan data atau database. Di lapisan ini, Anda bisa mengelola operasi **CRUD** (Create, Read, Update, Delete) dan menangani query ke database.

Fungsi Utama repository:

- **Abstraksi Akses Data:** Repository memberikan abstraksi dari detail implementasi database. Service layer atau controller tidak perlu tahu bagaimana query SQL atau operasi database dilakukan, mereka hanya perlu memanggil metode di repository.
- **Operasi CRUD:** Repository sering digunakan untuk melakukan operasi CRUD, seperti menyimpan objek ke database, mengambil objek, mengubah data, dan menghapus data.
- **Centralized Access to Data:** Semua operasi terkait data difokuskan di satu tempat (repository), membuat aplikasi lebih mudah untuk diuji, dikelola, dan diubah.

5. Persistence Layer (Lapisan Persistensi)
Fungsi:
- Lapisan ini bertugas menyimpan status objek aplikasi ke dalam database atau sistem penyimpanan yang bersifat persistensi.
- Menggunakan teknologi seperti ORM (Object-Relational Mapping) seperti Hibernate atau JPA.

Contoh:
- Dalam sistem perbankan, informasi transaksi pelanggan akan disimpan ke dalam database menggunakan ORM.

Tugas Utama:
- Mengelola siklus hidup entitas data, termasuk menyimpan, memperbarui, dan mengambil data dari penyimpanan jangka panjang.

6. Integration Layer (Lapisan Integrasi)
Fungsi:
- Lapisan ini menangani komunikasi dengan sistem eksternal, seperti layanan web, sistem ERP, atau sistem perusahaan lain.
- Menerjemahkan format data antar sistem dan menangani detail implementasi integrasi.

Contoh:
- API gateway dalam sistem microservices yang mengatur permintaan dari satu layanan ke layanan lain.
- Integrasi dengan layanan pihak ketiga seperti pembayaran online (PayPal, Stripe).

Tugas Utama
- Mengelola interaksi antara aplikasi dan sistem eksternal, memastikan data yang dipertukarkan sesuai dengan format yang diharapkan.

Kesimpulan

- **Lapisan Presentasi** berfungsi untuk berinteraksi dengan pengguna.
- **Lapisan Logika Bisnis** menangani aturan bisnis dan keputusan.
- **Lapisan Layanan** menyediakan antarmuka yang menghubungkan logika bisnis dengan lapisan lainnya.
- **Lapisan Akses** Data mengelola komunikasi dengan database atau sistem penyimpanan data.
- **Lapisan Persistensi** menjaga data yang bersifat permanen.
- **Lapisan Integrasi** menangani komunikasi dengan layanan eksternal atau sistem pihak ketiga.

• LATIHAN

Bagian 1: Setup Proyek dan Struktur Aplikasi (30 Menit)
Langkah 1.1: Membuat Proyek Maven

1. **Buat Proyek Maven Baru:**
   - Buka IDE Anda, buat proyek Maven baru seperti yang sudah Anda lakukan pada praktikum sebelumnya.
   - Nama proyek: library-management.
2. **Tambahkan Spring Core Dependensi di `pom.xml`:**

- Tambahkan dependensi Spring Core yang diperlukan untuk menjalankan proyek di pom.xml

```
<dependencies>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.10</version>
      </dependency>
</dependencies>
```

Langkah 1.2: Struktur Proyek

Buat struktur package berikut dalam proyek Anda:

com.example.springcoreadvanced

- controller
- service
- repository
- model

Bagian 2: Membuat Aplikasi Sederhana dengan Layered Architecture

Pada bagian ini, Anda akan membuat aplikasi sederhana untuk **manajemen buku perpustakaan** dengan menggunakan lapisan-lapisan berikut: **Controller**, **Service**, **Repository**, dan **Model**.

Langkah 2.1: Membuat Model

- Buat class `Book` di package `model`.

```
package com.example.springcoreadvanced.model;

public class Book {
    private String isbn;
    private String title;
    private String author;

    public Book(String isbn, String title, String
author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }

    // Getters and Setters
    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn =
isbn; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title =
title; }
```

```
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author
= author; }

    @Override
    public String toString() {
        return "Book [ISBN=" + isbn + ", Title=" +
title + ", Author=" + author + "]";
    }
}
```

Langkah 2.2: Membuat Repository

- Buat interface `BookRepository` di package `repository`.

```
package com.example.springcoreadvanced.repository;

import com.example.springcoreadvanced.model.Book;
import java.util.List;

public interface BookRepository {
    void save(Book book);
    List<Book> findAll();
}
```

- Buat implementasi `BookRepositoryImpl`.

```
package com.example.springcoreadvanced.repository;

import com.example.springcoreadvanced.model.Book;
import java.util.ArrayList;
import java.util.List;
import org.springframework.stereotype.Repository;

@Repository
public class BookRepositoryImpl implements
BookRepository {
    private List<Book> books = new ArrayList<>();

    @Override
    public void save(Book book) {
        books.add(book);
        System.out.println("Book saved: " + book);
    }

    @Override
    public List<Book> findAll() {
        return books;
    }
}
```

Langkah 2.3: Membuat Service

- Buat interface `BookService` di package `service`.

```
package com.example.springcoreadvanced.service;

import com.example.springcoreadvanced.model.Book;
import java.util.List;

public interface BookService {
    void addBook(Book book);
    List<Book> getAllBooks();
}
```

- Buat implementasi `BookServiceImpl`.

```
package com.example.springcoreadvanced.service;

import com.example.springcoreadvanced.model.Book;
import
com.example.springcoreadvanced.repository.BookReposito
ry;
import
org.springframework.beans.factory.annotation.Autowired
;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class BookServiceImpl implements BookService {

    @Autowired
    private BookRepository bookRepository;

    @Override
    public void addBook(Book book) {
        bookRepository.save(book);
    }

    @Override
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }
}
```

Langkah 2.4: Membuat Controller

- Buat class LibraryController di package controller.

```
package com.example.springcoreadvanced.controller;

import com.example.springcoreadvanced.model.Book;
```

```
import
com.example.springcoreadvanced.service.BookService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class LibraryController {

    @Autowired
    private BookService bookService;

    public void addNewBook(String isbn, String title,
String author) {
        Book book = new Book(isbn, title, author);
        bookService.addBook(book);
    }

    public void displayAllBooks() {
        for (Book book : bookService.getAllBooks()) {
            System.out.println(book);
        }
    }
}
```

Bagian 3: Mengonfigurasi dan Menjalankan Aplikasi

Langkah 3.1: Konfigurasi Aplikasi dengan XML

- Buat file applicationContext.xml di folder src/main/resources.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xmlns:context="http://www.springframework.org/schema/c
ontext"

xsi:schemaLocation="http://www.springframework.org/sch
ema/beans

http://www.springframework.org/schema/beans/spring-
beans.xsd

http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context.xsd">
    <!-- Aktifkan scanning komponen -->
```

```
    <context:component-scan base-
    package="com.example.springcoreadvanced" />

    </beans>
```

Langkah 3.2: Menjalankan Aplikasi

- Buat class App.java di package utama.

```
package com.example.springcoreadvanced;

import
com.example.springcoreadvanced.controller.LibraryControl
ler;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicat
ionContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml")
;

        LibraryController libraryController =
context.getBean(LibraryController.class);
        libraryController.addNewBook("978-1234567897",
"Spring Framework", "Craig Walls");
        libraryController.addNewBook("978-9876543210",
"Java for Beginners", "James Gosling");

        System.out.println("All books in library:");
        libraryController.displayAllBooks();
    }
}
```

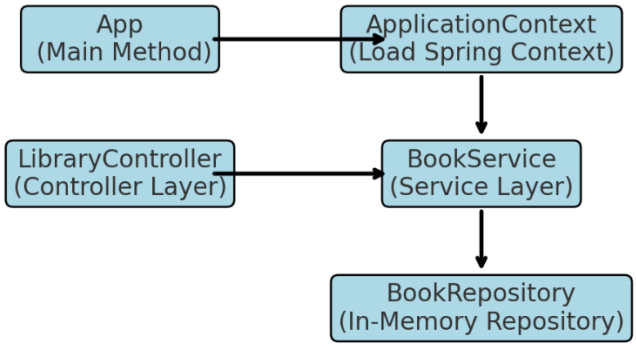
Langkah 3.3: Jalankan Proyek

- Klik kanan proyek dan pilih **Run**. Output akan menampilkan daftar buku yang berhasil disimpan dan ditampilkan menggunakan service dan repository.

Alur pengerjaan aplikasi

- **App (Main Method)**: Eksekusi dimulai dari metode `main()`, yang menjalankan aplikasi.
- **ApplicationContext (Load Spring Context)**: Spring IoC Container diinisialisasi untuk memuat semua bean yang diperlukan dari konfigurasi Spring.

- **LibraryController (Controller Layer)**: Setelah konteks Spring dimuat, controller (`LibraryController`) digunakan untuk memulai eksekusi proses aplikasi. Dengan menerima permintaan dan mengarahkan logika bisnis.
- **BookService (Service Layer)**: Controller memanggil service (`BookService`) untuk memproses logika bisnis, seperti menambahkan buku atau menampilkan semua buku.
- **BookRepository (In-Memory Repository)**: Service berinteraksi dengan repository (`BookRepository`), yang menyimpan data buku di memori (menggunakan `List<Book>`).



- **LAPORAN**

1. Penjelasan kode program, capture GUI untuk setiap soal (9 x (@10%)

Referensi Utama:

1. Craig Walls, "Spring in Action," 6th Edition, Manning Publications, 2022.
2. Juergen Hoeller, "Spring Framework Reference Documentation," Spring.io.