JDBC 구현

3. Todo

JDBC 에서 **Todo** 는 특정 작업을 수행하기 위해 필요한 코드나 기능을 의미하는 것이 아니라,
프로그래밍 개발 중에 **구현해야 할 부분**이나 **추가 작업을 나타내는 표시**이다.

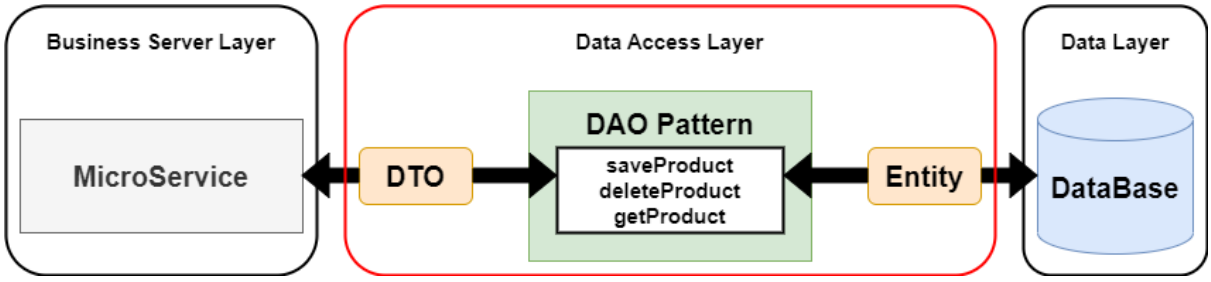
즉, **개발자가 나중에 구현해야 할 부분을 나타내는 표시로 사용**된다.

※ **사용 형식**은 Todo@@ 에서 **Todo 에 개발자가 "할 일" 을 작성**한다.
ex) BookDAO

예를 들어, 데이터베이스 연결 설정, 쿼리 작성, 결과 처리 등에 관련된 작업 중 아직 구현되지 않은 부분이나 추가로 처리해야 할 부분을 Todo 로 표시하여 개발자에게 알려준다.

TodoVo , TodoDTO , TodoDAO , TodoSerivce , TodoController 등이 있다.

가장 중요한 DAO 와 DTO 에 대해 설명하자면 다음과 같다.



The Data Access Object (DAO) pattern is a design pattern that provides an abstract interface for interacting with the database. The `DTO` pattern handles the transfer of data while the `DAO` pattern handles the CRUD logic.

DAO(데이터 액세스 개체) 패턴은 데이터베이스와 상호 작용하기 위한 추상 인터페이스를 제공하는 설계 패턴입니다. DTO 패턴은 데이터 전송을 처리하는 반면 DAO 패턴은 CRUD 로직을 처리합니다.

출처 : https://levelup.gitconnected.com/what-i-love-about-data-access-layer-dal-when-building-small-backend-projects-8ad694baee3a

위의 예시와 같이 **DTO 는 데이터 전송**을, **DAO 는 CRUD(Create, Read, Update, Delete)** 를 처리한다.

▼ TodoVO *(Todo Value Object)*

TodoVO는 **"할 일"을 나타내는 값 객체(Value Object)**이다.

주로 **데이터베이스나 외부 시스템과의 데이터 교환에 사용**된다.
TodoVO는 **제목, 내용, 우선순위 등과 같은 속성**을 가지고 있으며, **데이터의 변동 없이 읽기 전용으로 사용**되고, **데이터의 불변성과 읽기 전용 속성**을 가지고 있기 때문에 **안정적인 데이터 전달과 데이터 무결성을 유지하는 데 도움**이 된다.

주로 다른 계층(레이어) 간의 데이터 전달을 위해 사용된다.

※ **사용 예시 :** 데이터베이스에서 할 일을 조회할 때 TodoVO 객체에 데이터를 담아 반환한다.

lombok 을 사용하여 TodoVO 클래스를 추가하는 예시는 다음과 같다.

```
//VO 는 불변 객체이므로 다음과 같은 방법으로 객체를 불변으로 만들 수 있다.
  // 1. 필드를 private final로 선언
  // 2. 커스텀 생성자를 통한 초기화
    // ▶ 값을 입력받아 생성자에서 초기화하므로 객체는 불변이 된다.
  // 3. Getter 메서드만 제공
    // ▶ Getter만 제공하고, Setter 메서드를 제공하지 않는다.
        // 이로써 외부에서는 필드 값을 변경할 수 없게 된다.

//아래 코드에서는 @Builder 를 통해 2번에 해당하는 생성자를 생성하였다.
@Getter
@Builder
@ToString
public class TodoVo {
    private Long tno;
    private String title;
    private LocalDate dueDate;
    private boolean finished;
}
```

VO 는 주로 읽기 전용으로 사용하는 경우가 많으므로 @Getter 를 추가했고,
객체 생성시 빌더 패턴을 이용하기 위해서 @Builder 를 추가했다.
@Builder 를 추가했기 때문에 `TodoVO.builder().build()` 와 같은 형태로 객체를 생성할 수 있고,
생성자를 따로 정의하지 않아도 **모든 객체가 불변객체로 유지**된다.

VO 는 **DTO** 와 **유사한 모습**이지만 **차이**가 있다.
DTO 를 **각 계층을 오고 가는데 사용하는 택배 상자**에 비유한다면,
VO 는 **데이터베이스의 엔티티를 자바 객체로 표현한 것**이라고 말할 수 있다.
쉽게말해 VO 는 불변 , DTO 는 가변 이라는 특성이 있다.

> 📎 **[값 객체*(Value Object)*]**
>
> 값 객체는 **데이터를 포함하고 있는 객체**로,
> **데이터의 속성**을 가지고 있으며 **변경 불가능(Immutable)한 특징**을 갖는다.
>
> 주로 데이터의 **전달**이나 **교환**을 목적으로 사용된다.

▼ TodoDTO *(Todo Data Transfer Object)*

TodoDTO는 **"할 일"과 관련된 데이터 전송을 위한 객체**이다.

TodoDTO는 TodoVO와 유사한 속성을 가지고 있지만, **목적이 다르다.**
TodoDTO는 주로 **클라이언트와 서버 간의 데이터 전송을 위해 사용**된다.
클라이언트에서 서버로 데이터를 전송할 때는 **클라이언트에서 TodoDTO 객체를 생성**하고,
서버에서는 **TodoDTO 객체를 받아 비즈니스 로직에서 처리**한다.

또한, 서버에서 클라이언트로 데이터를 전송할 때도 **TodoDTO 객체를 사용하여 데이터를 담아 전송**한다.

TodoDTO는 TodoVO와 달리 **데이터의 불변성이나 읽기 전용 속성을 강제하지 않는다.**
데이터 전송을 위해 필요한 속성과 메서드를 포함하고 있으며,
클라이언트와 서버 간의 **데이터 전송을 원활하게 수행할 수 있도록 돕는다.**

정리하자면, TodoDTO는 클라이언트와 서버 간의 **데이터 전송을 추상화**하고, **데이터의 일관성과 유효성을 유지**하는 데 도움이 된다.

> 📎 **[데이터 전송 객체*(Data Transfer Object)*]**
>
> 데이터 전송 객체는 **비즈니스 계층과 프레젠테이션 계층(UI) 사이에서 데이터를 전송하기 위해 사용되는 객체**이다.
> 주로 **데이터의 속성**을 가지고 있으며, 데이터의 **전달**을 목적으로 사용됩니다.

▼ TodoDAO *(Todo Data Access Object)*

TodoDAO는 **"할 일"과 관련된 데이터에 접근하기 위한 객체**이다.

TodoDAO는 주로 **데이터베이스와의 상호 작용을 처리**한다.
데이터베이스에 **새로운 할 일을 추가**하거나, **기존 할 일을 수정**하거나 **삭제**하는 등의
CRUD(Create, Read, Update, Delete) 작업을 수행한다.
이렇게 TodoDAO는 데이터베이스 연결, SQL 쿼리 실행, 데이터 변환 등의 기능을 제공하여 비즈니스 로직에서 **데이터베이스와의 상호 작용을 간단하게 처리**할 수 있도록 도와준다.

TodoDAO는 **데이터베이스와의 연결을 담당**하며, **데이터베이스에 대한 접근을 캡슐화**한다.
이를 통해 비즈니스 로직은 **데이터의 조작과 조회에 집중할 수 있다.**
또한, 트랜잭션 처리와 같은 **데이터베이스 관련 작업을 처리**하여 **데이터의 일관성과 안전성을 보장**합니다.

정리하자면, TodoDAO는 **데이터베이스와의 상호 작용을 추상화**하고, **비즈니스 로직과 데이터베이스 간의 결합도를 낮추는 데 도움**이 된다.
이를 통해 **애플리케이션의 유지 보수성과 확장성을 향상**시킬 수 있다.

※ DAO를 호출하는 객체는, DAO 가 **내부에서 어떤식으로 데이터를 처리하는지 알 수 없도록 구성**한다. 이 때문에 **'JDBC 프로그램을 작성한다'** 는 말은 **'DAO 를 작성한다'** 와 같은 말이 된다.

> 📎 **[데이터 접근 객체*(Data Access Object)*]**
>
> 데이터 접근 객체는 데이터베이스나 외부 소스와 같은 데이터 저장소에 접근하여 **데이터를 조작하고 조회하는 역할을 담당하는 객체**이다.
> 비즈니스 로직에서 **데이터베이스와의 상호 작용을 추상화**하고, **데이터의 영속성(Persistence)을 관리**한다.

TodoDAO 클래스를 선언하는 예시는 다음과 같다.

```
package Todo_Practice;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class TodoDAO {
    private Connection connection;

    // 생성자
    public TodoDAO(Connection connection) {
        this.connection = connection;
    }

    // TodoDTO 객체를 저장하는 메서드
    public void save(TodoDTO dto) throws SQLException {
        // SQL 쿼리문 작성 및 PreparedStatement 생성
        String sql = "INSERT INTO TBL_STUDENT (stuno, name, age, address) VALUES (?, ?, ?, ?)";
        PreparedStatement ps = connection.prepareStatement(sql);

        // TodoDTO 객체의 필드 값 설정
        ps.setInt(1, dto.getStuno());
        ps.setString(2, dto.getName());
        ps.setInt(3, dto.getAge());
        ps.setString(4, dto.getAddress());

        // SQL 실행
        ps.executeUpdate();

        // 리소스 해제
        ps.close();
    }

    // TodoDTO 객체를 조회하는 메서드
    public List<TodoDTO> findAll() throws SQLException {
        // SQL 쿼리문 작성 및 PreparedStatement 생성
        String sql = "SELECT * FROM TBL_STUDENT";
        PreparedStatement ps = connection.prepareStatement(sql);

        // SQL 실행 및 결과 처리
        ResultSet result = ps.executeQuery();

        // 결과를 담을 리스트 생성
        List<TodoDTO> todoList = new ArrayList<>();

        // 결과 순회
        while (result.next()) {
            // TodoDTO 객체 생성
            TodoDTO dto = new TodoDTO();

            // TodoDTO 객체의 필드 값 설정
            dto.setStuno(result.getInt("stuno"));
            dto.setName(result.getString("name"));
            dto.setAge(result.getInt("age"));
            dto.setAddress(result.getString("address"));

            // 리스트에 TodoDTO 객체 추가
```

```
            todoList.add(dto);
        }

        // 리소스 해제
        result.close();
        ps.close();

        return todoList;
    }

    // TodoDTO 객체를 삭제하는 메서드
    public void delete(int stuno) throws SQLException {
        // SQL 쿼리문 작성 및 PreparedStatement 생성
        String sql = "DELETE FROM TBL_STUDENT WHERE stuno = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 파라미터 설정
        ps.setInt(1, stuno);

        // SQL 실행
        ps.executeUpdate();

        // 리소스 해제
        ps.close();
    }
}
```

위와 같이 TodoDAO 클래스를 생성하여 **CRUD 를 더욱 쉽게 사용할 수 있다.**

※ DAO 를 작성하면 항상 **테스트** 코드를 이용해 동작에 문제가 없는지 확인해보는걸 추천한다.

📎 **[테스트 어노테이션]**

이클립스에서 테스트 어노테이션을 사용하는 방법은 다음과 같다.
[Build Path - ADD Library - JUnit - JUnit5] 를 통해 JUnit5 를 추가한다.
JUnit5 를 추가했다면 아래 어노테이션을 사용할 수 있다.

- `@Test` : Test 메서드로 인식하고 테스트 한다.
  JUnit5 는 접근제한자가 Default 여도 된다. (JUnit4 까지는 public이어야 했었다.)
- `@BeforeEach` : 테스트 메서드 실행 **이전**에 수행된다.
- `@AfterEach` : 테스트 메서드 실행 **이후**에 수행된다.
- `@BeforeAll` : 해당 테스트 클래스를 **초기화할 때** 딱 한번 수행되는 메서드다.
  메서드 시그니쳐는 **static** 으로 선언해야한다.
- `@AfterAll` : 해당 테스트 클래스 내 **테스트 메서드를 모두 실행시킨 후**, 딱 한번 수행되는 메서드다.
  메서드 시그니쳐는 **static** 으로 선언해야한다.
- `@Disabled` : 본 어노테이션을 붙인 테스트 메서드는 **무시**된다.

▼ TodoSerivce

TodoService는 **"할 일"과 관련된 비즈니스 로직을 처리하는 서비스**이다.

TodoService는 **비즈니스 로직을 처리**하고, Todo의 **CRUD 기능을 제공**한다.
주로 **TodoController와 TodoDAO 사이에서 중간 계층으로 사용**되어, 클라이언트 요청을 받아 처리하고 **필요한 데이터 접근 객체(DAO)를 호출하여 데이터베이스에서 Todo 데이터를 조작**한다.

TodoService는 다음과 같은 역할을 수행할 수 있다.

- **데이터의 유효성 검사**
  클라이언트로부터 받은 데이터의 **유효성을 검사**하고, 필요한 경우 **예외 처리**를 수행한다.
  예를 들어, 필수 필드의 누락 여부를 확인하거나 데이터의 형식을 검증할 수 있다.
- **데이터 처리 및 조작**
  Todo 데이터의 **CURD(생성, 조회, 수정, 삭제)** 비즈니스 로직을 처리한다.
  TodoDTO / VO 와 같은 **데이터 객체를 활용**하여 **데이터의 변환과 가공** 작업을 할 수 있다.
- **트랜잭션 관리**
  여러 개의 데이터 조작이 **하나의 트랜잭션으로 묶여야 하는 경우**, TodoService에서 트랜잭션 관리를 수행한다. 이를 통해 **데이터의 일관성과 안전성을 보장**할 수 있다.
- **외부 시스템과의 상호 작용**
  필요한 경우 **외부 시스템과의 연동이 필요한 작업을 수행**할 수 있다.
  예를 들어, 이메일 발송, 알림 서비스와의 통신 등을 처리할 수 있다.

정리하자면, TodoService는 **비즈니스 로직의 중요한 부분을 담당**하므로, **테스트 가능하고 재사용 가능한 코드를 작성하는 것이 중요**하다.
이를 위해 의존성 주입(Dependency Injection), 인터페이스 활용, 단위 테스트 등의 개발 원칙과 기법을 적용하여 좋은 설계와 유지 보수성을 갖춘 서비스를 구현할 수 있다.

> 📎 **[서비스*(Service)*]**
>
> 서비스는 애플리케이션의 **비즈니스 로직을 처리하는 컴포넌트**이다.
> **데이터의 유효성 검사, 데이터 처리 및 조작, 외부 시스템과의 상호 작용 등**을 담당한다.
> 주로 **컨트롤러(Controller)와 데이터 접근 객체(DAO) 사이에서 중재자 역할**을 수행한다.

▼ TodoController

TodoController는 **사용자의 요청을 처리하고, 응답을 반환하는 컨트롤러**이다.

TodoController는 **클라이언트의 요청을 받아들여 해당 요청에 대한 작업을 수행**한다.
주로 **HTTP 프로토콜을 기반으로 동작**하며, **RESTful API를 구현하는 경우가 많다.**
클라이언트로부터 **전달받은 데이터를 검증**하고, TodoService를 호출하여 **비즈니스 로직을 수행**한 뒤, 클라이언트에게 **응답을 반환**한다.

TodoController는 다음과 같은 역할을 수행할 수 있다.

- **클라이언트의 요청 처리**
  클라이언트로부터 **HTTP 요청**을 받아들이고, 해당 요청을 **해석하여 처리**할 수 있다.
  요청의 **메서드**(GET, POST, PUT, DELETE 등)와 **경로, 요청 헤더, 쿼리 매개변수 등을 확인**하여 **적절한 동작을 결정**한다.
- **데이터 검증 및 변환**
  클라이언트로부터 **전달받은 데이터의 유효성을 검증**하고, **필요한 형식으로 변환**한다.
  이를 통해 **잘못된 데이터를 거르고,** 비즈니스 로직에 **적합한 형태로 전달**합니다.
- **비즈니스 로직 호출**
  TodoService나 다른 서비스 계층의 **메서드를 호출하여 비즈니스 로직을 수행**한다.
  필요한 데이터를 TodoDTO나 다른 **데이터 전송 객체로 변환**하여 **서비스 계층에 전달**한다.
- **응답 생성 및 반환**
  비즈니스 로직의 처리 결과를 바탕으로 클라이언트에게 **응답을 생성**하고, **반환**한다.
  주로 **JSON, XML 등의 형식으로 응답을 구성하여** 클라이언트에게 **전달**한다.
- **예외 처리**
  요청 처리 중 발생한 **예외를 적절하게 처리**하고, **오류 응답을 반환**한다.
  예를 들어, 유효하지 않은 요청이나 데이터 처리 중 오류가 발생한 경우 **적절한 상태 코드와 오류 메시지를 클라이언트에게 전달**한다.

정리하자면, TodoController는 **클라이언트와의 인터페이스 역할을 수행**하므로,
사용자 요청을 처리하는 데 **필요한 기능을 제공**하고,
응답을 반환하는데 **적절한 형식**과 **상태 코드를 사용**하는 것이 중요하다.
또한, TodoService와의 결합을 느슨하게 유지하여 유지 보수성과 테스트 용이성을 고려한 좋은 설계를 지향해야 한다.

> 📎 **[컨트롤러*(Controller)*]**
>
> **사용자의 요청을 받아 처리**하고,
> 비즈니스 로직이나 데이터 접근을 위한 **서비스 계층(Service)에 작업을 위임**한다.
> 그리고 **처리 결과를** 클라이언트에게 **응답으로 반환**한다.
> 주로 **클라이언트와 애플리케이션 사이의 인터페이스 역할**을 한다.

---

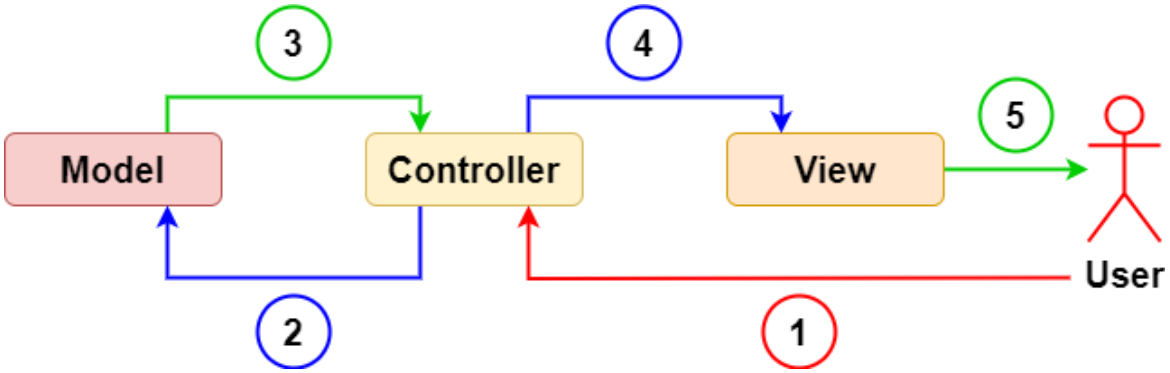
4. MVC *(Model-View-Controller)*

MVC *(Model-View-Controller)*는 **소프트웨어 개발에서 사용되는 디자인 패턴**으로, **애플리케이션의 구조를** 세 가지 주요 구성 요소인 **Model - View - Controller** 로 **분리**하여 관리하는 방법을 제공한다.

각각의 구성 요소는 **고유한 역할**을 수행하며 애플리케이션의 **유지보수성, 재사용성 및 확장성을 향상**시키는 데 도움을 준다.

예시를 보기 전에 MVC 의 구성요소를 우선적으로 간단히 설명하자면 다음과 같다.

- **Model - App의 데이터를 관리한다.**
- **View - Model의 시각적인 표현을 담당한다.**
- **Controller - User와 System간을 연결한다.**



<그림 1 : 상단> , <그림 2 : 우측>

우선 그림 1을 보자.

① User 가 Controller 를 조작한다.

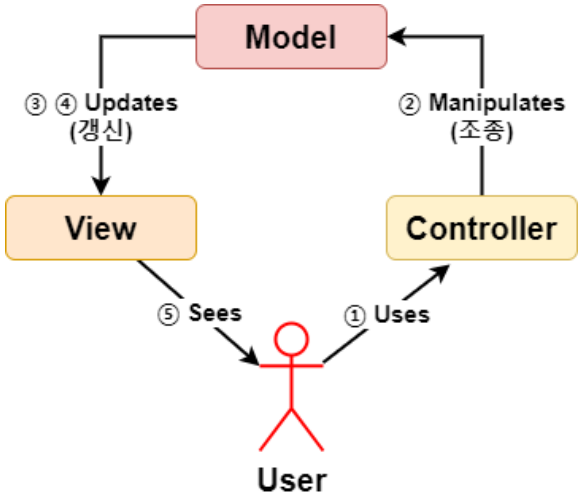
② Controller 가 Model 에게 명령한다.

③ Model 이 Controller 로 데이터를 가져온다.

④ Controller 가 View 에게 데이터를 전달한다.

⑤ View 가 User 에게 데이터를 전달한다.

그림 2의 내용은 이해를 돕기 위해 중간 ③ , ④ 의 내용을 '하나의 Updates' 로 합친 내용이다.



이제 MVC 의 구성 요소인 **Model - View - Controller** 에 대해 알아보도록 하겠다.

[Model]

Model 은 애플리케이션의 **데이터**와 **비즈니스 로직**을 담당하며,
데이터의 구조, 유효성 검사, 데이터베이스와의 상호작용 등을 다룬다.

여기에서 **데이터**는 애플리케이션의 상태나 정보를 나타내는데 사용되며,
비즈니스 로직은 데이터를 조작하고 처리하는 규칙과 동작을 정의한다.

Model 은 독립적이며, 사용자 인터페이스나 특정 플랫폼에 의존하지 않고,
데이터의 유지, 업데이트, 검색, 삭제 등을 처리하고, 비즈니스 로직을 실행하는 역할을 수행한다.

💡 *Model 은 다음과 같은 **규칙**을 지켜야 합니다.*

1. **단일 책임 원칙 *(Single Responsibility Principle)***
   한 가지 주요 역할에 집중하여 특정 도메인이나 비즈니스 개념을 관리해야 한다.
2. **독립성 *(Independence)***
   다른 컴포넌트나 프레임워크에 의존하지 않고 독자적으로 작동할 수 있어야 한다.
3. **추상화 *(Abstraction)***
   도메인에 관련된 개념과 데이터를 추상화하여 표현하며, 도메인의 복잡성을 감추고 이해하기 쉽게 만들어야 한다.
4. **일관성 *(Consistency)***
   데이터의 유효성 검사, 업데이트, 조회 등을 일관된 방식으로 처리하여 데이터 일관성과 비즈니스 규칙 준수를 유지해야 한다.
5. **테스트 용이성 *(Testability)***
   모델은 테스트 가능하도록 설계되어야 하며, 테스트 케이스를 작성하여 모델의 동작을 검증할 수 있어야 한다.
   이를 위해 모델을 분리하고 의존성 주입 등의 기법을 활용할 수 있다.

[View]

View 는 **모델의 데이터를 사용자에게 시각적으로 표시하는 역할**을 하며,
사용자 **인터페이스를 구성**하고, 데이터의 표현 방식이나 **레이아웃**을 담당한다.
이때 일반적으로 사용자의 입력을 받지 않으며, 단순히 데이터를 표시하는 역할만 수행한다.

하나의 Model 에 여러 개의 View 가 존재할 수 있으며, 각각의 View는 독립적으로 동작할 수 있다.

💡 *View 는 다음과 같은 **규칙**을 지켜야 합니다.*

1. **데이터 표시 *(Data Presentation)***
   모델의 데이터를 사용자에게 명확하게 표현하여 이해 가능하도록 보여줘야 한다.
   이를 위해 데이터의 포맷팅, 정렬, 필터링 등을 적용할 수 있다.
2. **독립성 *(Independence)***
   다른 컴포넌트나 모듈에 의존하지 않고 독립적으로 작동해야 한다.
   자체적으로 데이터를 표시하고 관리할 수 있는 구조여야 하며, 모델의 변경에 영향을 받지 않고 재사용이 가능해야 한다.
3. **재사용성 *(Reusability)***
   재사용 가능한 구성 요소로 설계되어야 한다.
   로직과 표현 방식을 분리하고, 템플릿이나 컴포넌트화 등의 기법을 활용하여 재사용성을 높일 수 있다.
4. **일관성 *(Consistency)***
   일관된 디자인과 사용자 경험을 제공해야 한다.
   이를 위해 통일된 레이아웃, 스타일, 색상 등을 사용하여 사용자가 일관된 인터페이스를 경험할 수 있도록 할 수 있다.
5. **반응성 *(Responsiveness)***
   사용자의 상호작용에 실시간으로 반응해야 한다.
   사용자 입력에 즉각적인 피드백이 제공되어야 하며, 데이터 변경 시 동적으로 업데이트되어야 한다.

[Controller]

Controller 는 **입력의 유효성 검사**, 요청의 분배, 비즈니스 로직의 실행 등을 담당하며,
사용자의 입력을 처리하고 모델과 뷰 사이의 상호작용을 조정하는 역할을 수행한다.
쉽게 말해, 사용자의 요청을 받아 해당하는 작업을 수행하고, 모델을 업데이트하거나 뷰를 업데이트하는 역할을 수행한다는 것이다.

즉, Controller 는 **Model 이나 View 에 대한 변경 사항을 통지하고, 각각의 역할을 적절하게 업데이트하여 일관성을 유지하는 역할**을 수행한다.

💡 *Controller 는 다음과 같은 **규칙**을 지켜야 합니다.*

1. **사용자 입력 처리 *(User Input Handling)***
   사용자의 입력을 적절하게 처리해야 한다.
   사용자의 요청을 수신하고, 필요한 데이터를 추출하고, 해당 요청에 대한 적절한 액션을 수행하는 역할을 담당한다.
2. **비즈니스 로직 처리 *(Business Logic Handling)***
   비즈니스 로직을 적절히 처리해야 한다.
   사용자의 요청에 따라 모델을 업데이트하거나 쿼리를 수행하여 필요한 데이터를 추출하고 가공하는 등의 비즈니스 로직을 수행한다.
   이를 통해 요청에 대한 적절한 결과를 생성하고 전달한다.
3. **모델과 뷰의 연결 *(Model-View Interaction)***
   모델과 뷰 사이의 상호작용을 관리해야 한다.
   모델과 뷰 간의 데이터 흐름을 조정하여 사용자에게 정확하고 일관된 정보를 제공한다.
4. **의존성 관리 *(Dependency Management)***
   필요한 의존성을 관리해야 한다.
   이를 통해 컨트롤러는 독립적으로 테스트하고 재사용할 수 있는 구조를 유지할 수 있다.
5. **분리된 역할 *(Separation of Concerns)***
   관심사를 분리해야 한다.
   이를 통해 코드의 가독성, 유지보수성, 확장성을 향상시킬 수 있다.

---

5. MVC 의 사용

MVC 는 DAO , DTO 클래스를 사용하여 DBMS에 접근한다.

사용 방법의 관계성에 대해 설명하기 앞서, 우선 MVC 의 3가지 구성요소의 역할에 대해 설명하겠다.

- **Model** - DAO , DTO , VO 클래스에 접근한다.
  1. Controller 로 부터 request 를 받는다.
  2. 요청값을 받는다. request.getParameter()
  3. 결과값을 담는다. request.setAttribute()
- **view** - 웹 언어(예시는 JSP)로 사용자 인터페이스 중심으로 화면에 출력되는 공간이다.
  1. Controller 에게 request 값을 받는다.
  2. 화면을 출력한다.
- **Controller** - 모델과 뷰 사이에서 서로를 중계하는 역할을 수행한다.
  1. 사용자의 요청값에 따라

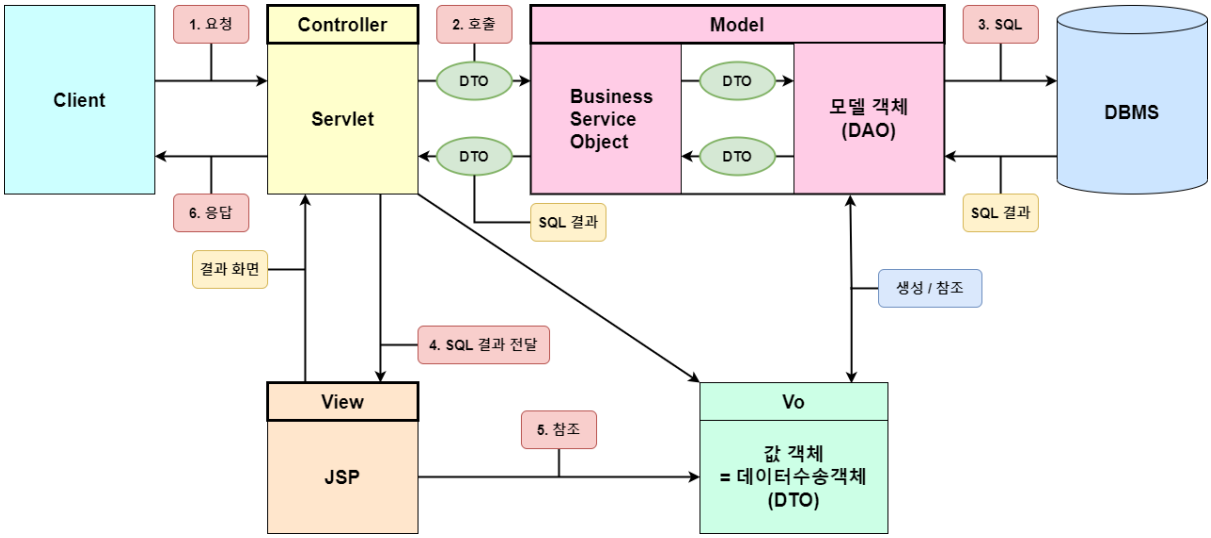
2. Model을 찾는다
3. Model에게 받은 request 값을 받는다.
4. 받은 request 값에 따라 View를 찾는다.

다음으로 DAO , DTO 의 역할에 대해 설명하겠다.

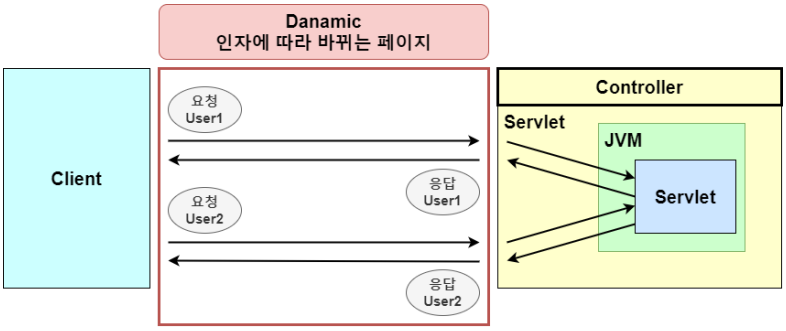
- **DAO** - 데이터베이스와의 상호작용을 추상화하여 Controller 와의 인터페이스 역할을 한다.
- **DTO** - 데이터 전달을 위한 객체로 사용된다.
  - Todo 항목의 필드와 해당 값을 저장하는 역할
  - Model 과 View 사이에서 데이터 전달을 위한 중간 객체로 사용
- **VO** - Todo 항목의 데이터를 저장하기 위한 객체로 사용된다.

이렇게 MVC 구조를 적용하면, 각자의 공간에서 각자의 언어만을 사용하여 각자의 역할을 수행한다.
즉, **언어를 구분할 수 있다.**

위와 같은 역할을 토대로 예시 다이어그램을 살펴보겠다.



1. Client가 Controller 역할을 하는 servlet 에 요청한다.
   이때, Client 와 Servelt 의 관계는 동적으로 작동되며, 다음과 같다.



2. Controller는 DTO 클래스를 사용하여 사용자의 요청값에 따라 해당하는 Model을 찾는다.
3. Model 은 DAO 클래스를 통해 DBMS 로 SQL 문을 입력하고, 결과를 다시 Controller 로 반환한다.
4. Controller는 View 로 SQL 결과를 전달한다.
5. View 는 Vo 를 참조하여 결과 '화면'을 Controller 로 전달한다.
6. Controller는 Client로 화면을 출력한다.