3️⃣

# Section3. 서블릿, JSP, MVC 패턴

## 회원 관리 웹 애플리케이션 요구사항

### 회원 정보

- 이름: `username`
- 나이: `age`

### 기능 요구사항

- 회원 저장
- 회원 목록 조회

> 💡 실습 코드 작성
> DTO, Repository 작성

---

## 서블릿으로 회원 관리 웹 애플리케이션 만들기

> 💡 실습 코드 작성

- `MemberFormServlet`
- `MemberListServlet`
- `MemberSaveServlet`

### 템플릿 엔진

서블릿 덕분에 동적으로 원하는 HTML을 마음껏 만들 수 있다.

정적인 HTML 문서라면 화면이 계속 달라지는 회원의 저장 결과라던가, 회원 목록 같은 동적인 HTML을 만드는 일은 불가능 할 것이다.

자바 코드로 HTML을 만들어 내는 것 보다 차라리 HTML 문서에 동적으로 변경해야 하는 부분만 자바 코드를 넣을 수 있다면 더 편리할 것이다.

템플릿 엔진을 사용하면 HTML 문서에서 필요한 곳만 코드를 적용해서 동적으로 변경할 수 있다.

---

# JSP로 회원 관리 웹 애플리케이션 만들기

## JSP 라이브러리 추가

```
//JSP 추가 시작
implementation 'org.apache.tomcat.embed:tomcat-embed-jasper'
implementation 'jakarta.servlet:jakarta.servlet-api' //스프링부
implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.
implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl' /
//JSP 추가 끝
```

💡 실습 코드 작성

회원 등록 폼 JSP를 보면 첫 줄을 제외하고 완전히 HTML과 똑같다.

JSP는 서버 내부에서 서블릿으로 변환되는데, 우리가 만들었던 `MemberFormServlet` 과 거의 비슷한 모습으로 변환된다.

## JSP 사용

JSP는 자바 코드를 그대로 다 사용할 수 있다.

- `<%@ page import="hello.servlet.domain.member.MemberRepository" %>`
  자바의 import 문과 같다.
- `<% ~~ %>` 이 부분에는 자바 코드를 입력할 수 있다.
- `<%= ~~ %>` 이 부분에는 자바 코드를 출력할 수 있다.

💡 **※ 서블릿과 JSP의 한계 ※**

JSP를 사용한 덕분에 뷰를 생성하는 HTML 작업을 깔끔하게 가져가고, 중간중간 동적으로 변경이 필요한 부분에만 자바 코드를 적용했다.

몇가지 해결되지 않는 사항

코드의 **상위 절반**은 회원을 저장하기 위한 **비즈니스 로직**이고, 나머지 **하위 절반**만 결과를 HTML로 보여주기 위한 **뷰 영역**이다.

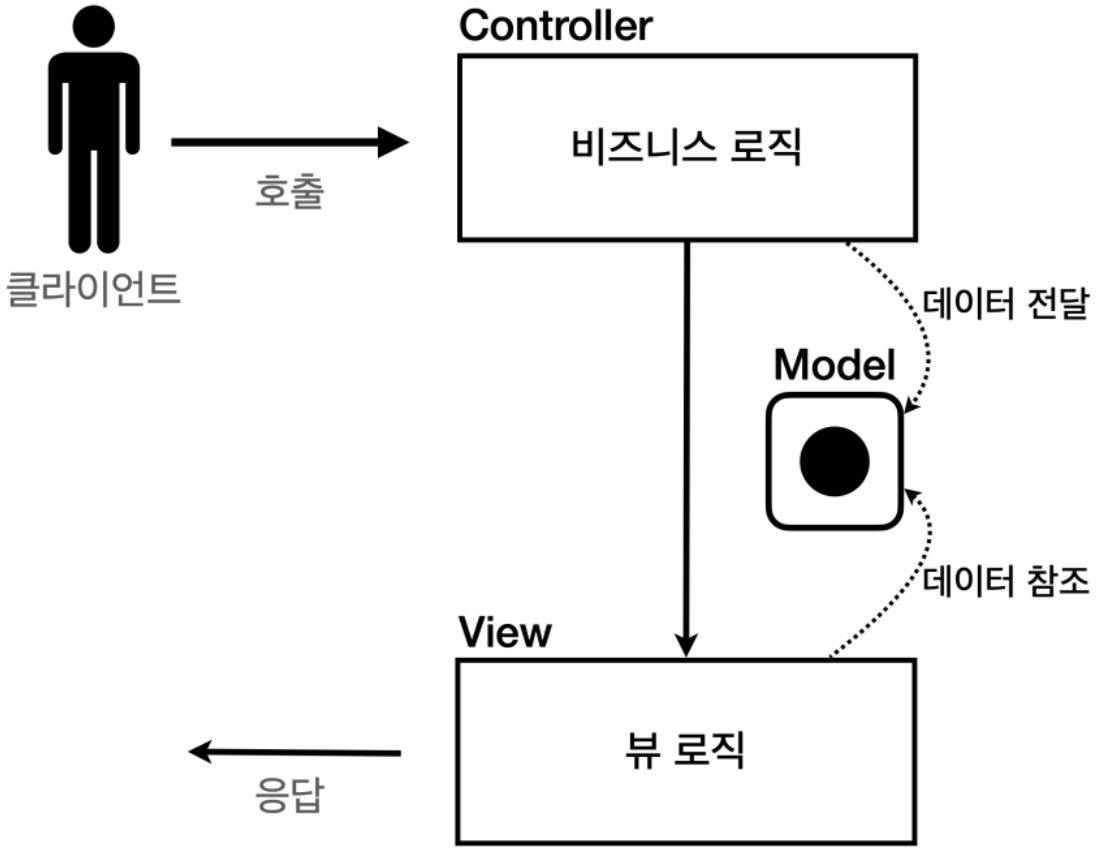
JSP가 너무 많은 역할을 한다.

→
MVC 패턴의 등장

비즈니스 로직은 서블릿 처럼 다른곳에서 처리하고, JSP는 목적에 맞게 HTML로 화면(View)을 그리는 일에 집중하도록 하자.

---

# MVC 패턴 - 개요




## 너무 많은 역할

하나의 서블릿이나 JSP만으로 비즈니스 로직과 뷰 렌더링까지 모두 처리하게 되면, 너무 많은 역할을 하게 되고, 결과적으로 유지보수가 어려워진다.

## 변경의 라이프 사이클

진짜 문제는 둘 사이에 변경의 라이프 사이클이 다르다는 점이다.

예를들어, UI를 일부 수정하는 일과 비즈니스 로직을 수정하는 일은 각각 다르게 발생할 가능성이 매우 높고 대부분 서로에게 영향을 주지 않는다.

→ 유지보수 관점에서 좋지 않다.

## 기능 특화

JSP 같은 뷰 템플릿은 화면을 렌더링 하는데 최적화 되어 있기 때문에 이 부분의 업무만 담당하는 것이 효과적이다.

## Model View Controller

MVC 패턴은 지금까지 학습한 것 처럼 하나의 서블릿, JSP로 처리하던 것을 컨트롤러와 뷰라는 영역으로 서로 역할을 나눈 것을 말한다.

### 컨트롤러

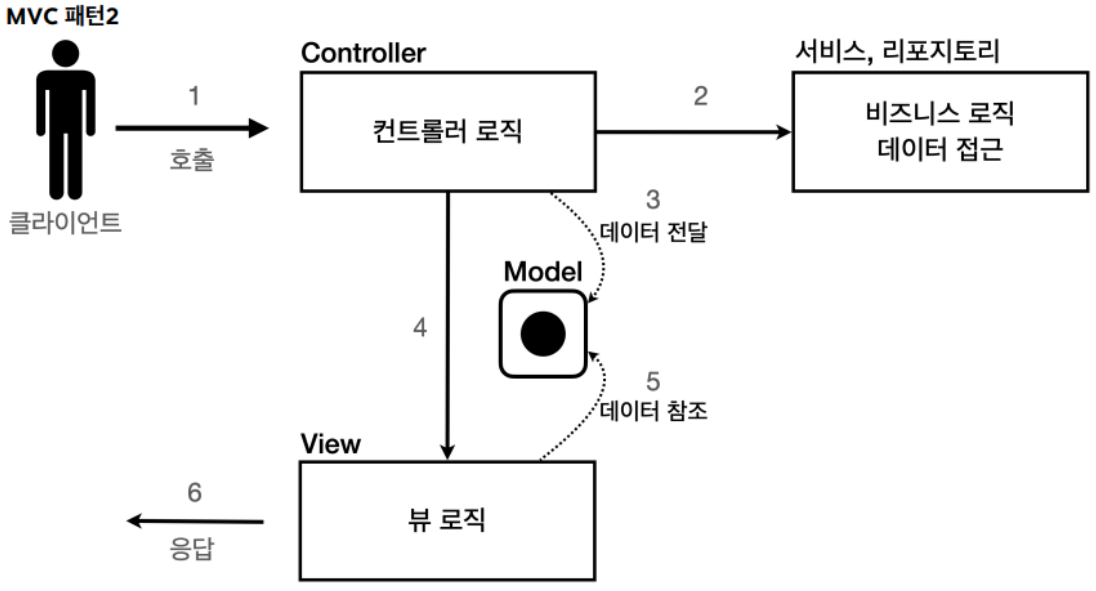
HTTP 요청을 받아서 **파라미터를 검증**하고 **비즈니스 로직을 실행**한다. 그리고 뷰에 전달할 결과 **데이터를 조회해서 모델에 담는다**.

### 뷰

모델에 담겨있는 데이터를 사용해서 화면을 그리는 일에 집중한다.

> 💡 컨트롤러에 비즈니스 로직을 둘 수도 있지만, 이러면 컨트롤러가 너무 많은 역할을 담당한다.
>
> 일반적으로 비즈니스 로직은 서비스라는 계층을 별도로 만들어서 처리한다. (컨트롤러는 서비스를 호출)
>
> BL이 변경되면 비즈니스 로직을 호출하는 컨트롤러의 코드도 변경될 수 있다.

## MVC 패턴2




---

# MVC 패턴 - 적용

서블릿을 컨트롤러로 사용하고, JSP를 뷰로 사용해서 MVC 패턴을 적용해보자.

Model은 `HttpServletRequest` 객체를 사용한다. `request`는 내부에 데이터 저장소를 가지고 있는데 `reqeust.setAttribute()`, `request.getAttribute()` 를 사용하면 데이터를 보관하고, 조

회할 수 있다.

> 💡 실습 코드 작성

```
@WebServlet(name = "mvcMemberFormServlet", urlPatterns = "/se
public class MvcMemberFormServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest request, HttpSe
        String viewPath = "/WEB-INF/views/new-form.jsp";
        RequestDispatcher dispatcher = request.getRequestDisp
        dispatcher.forward(request, response);
    }
}
```

- `dispatcher.forward()` : 다른 서블릿이나 JSP로 이동할 수 있는 기능. 서버 내부에서 다시 호출이 발생한다.

**/WEB-INF**
이 경로안에 JSP가 있으면 외부에서 직접 JSP를 호출할 수 없다. 우리가 기대하는 것은 항상 컨트롤러를 통해서 JSP를 호출하는 것이다

**redirect vs forward**

`리다이렉트`는 실제 클라이언트(웹 브라우저)에 응답이 나갔다가, **클라이언트가 redirect 경로로 다시 요청**한다. 따라서 클라이언트가 인지할 수 있고, URL 경로도 실제로 변경된다.

반면에 `포워드`는 **서버 내부**에서 일어나는 호출이기 때문에 클라이언트가 전혀 인지하지 못한다

```
<%@ page contentType="text/html;charset=UTF-8" language="java
<html>
    <head>
     <meta charset="UTF-8">
     <title>Title</title>
    </head>
    <body>
    <!-- 상대경로 사용, [현재 URL이 속한 계층 경로 + /save] -->
        <form action="save" method="post">
```

```
            username: <input type="text" name="username" />
            age: <input type="text" name="age" />
            <button type="submit">전송</button>
        </form>
    </body>
</html>
```

여기서 form의 action을 보면 절대 경로( / 로 시작)가 아니라 상대경로( / 로 시작X)인 것을 확인할 수 있다. 이렇게 상대경로를 사용하면 폼 전송시 현재 URL이 속한 계층 경로 + save 가 호출된다.

- 현재 계층 경로: `/servlet-mvc/members/`
- 결과: `/servlet-mvc/members/save`

```
<%@ page import="hello.servlet.domain.member.Member" %>
<%@ page contentType="text/html;charset=UTF-8" language="java
<html>
  <head>
      <title>Title</title>
  </head>
  <body>
    성공
    <ul>
      <li>id=${member.id}</li>
      <li>username =${member.username}</li>
      <li>age = ${member.age}</li>
    </ul>
  <a href="/index.html">메인</a>
  </body>
</html>
```

`<%= request.getAttribute("member")%>` 로 모델에 저장한 member 객체를 꺼낼 수 있지만, 너무 복잡해진다.

JSP는 `${}` 문법을 제공하는데, 이 문법을 사용하면 `request`의 `attribute`에 담긴 데이터를 편리하게 조회할 수 있다.

---

# MVC 패턴 - 한계

MVC 패턴을 적용한 덕분에 컨트롤러의 역할과 뷰를 렌더링 하는 역할을 명확하게 구분할 수 있다.

- 뷰는 화면을 그리는 역할에 충실한 덕분에, 코드가 깔끔하고 직관적이다.
- 컨트롤러는 딱 봐도 중복이 많고, 필요하지 않는 코드들도 많이 보인다.

## MVC 컨트롤러의 단점

### 1. 포워드 중복

- view 로 이동하는 코드가 항상 중복 호출되어야 한다.
    - 물론 이 부분을 메서드로 공통화해도 되지만, 해당 메서드도 **항상 직접 호출**해야 한다.

```
RequestDispatcher dispatcher = request.getRequestDispatcher(v
dispatcher.forward(request, response);
```

### 2. ViewPath에 중복

```
String viewPath = "/WEB-INF/views/new-form.jsp";
```

- `prefix`: `/WEB-INF/views/`
- `suffix`: `.jsp`

  → 만약 jsp가 아닌 thymeleaf 같은 다른 뷰로 변경한다면 전체 코드를 다 변경해야 한다.

### 3. 사용하지 않는 코드

다음 코드를 사용할 대도 있고, 사용하지 않을 때도 있다. → response는 코드에서 사용하지 않는다.

```
HttpServletRequest request, HttpServletResponse response
```

### 4. 공통 처리가 어렵다

기능이 복잡해질 수록 컨트롤러에서 공통으로 처리해야 하는 부분이 점차 증가할 것이다.
단순히 공통 기능을 메서드로 뽑으면 될 것 같지만, 결과적으로 해당 메서드를 항상 호출..

호출하는 것 자체도 중복이다.

## 정리하면 공통 처리가 어렵다는 문제가 있다.

이 문제를 해결하려면 컨트롤러 호출 전에 먼저 공통 기능을 처리해야 한다.
소위 '
**수문장 역할'**을 하는 기능이 필요하다.

→ **프론트 컨트롤러 패턴(Front Controller)**을 도입하면 이런 문제가 해결된다.(입구를 하나로)