JSP2

Back-end Programming

김기정 (bangry313@gmail.com)

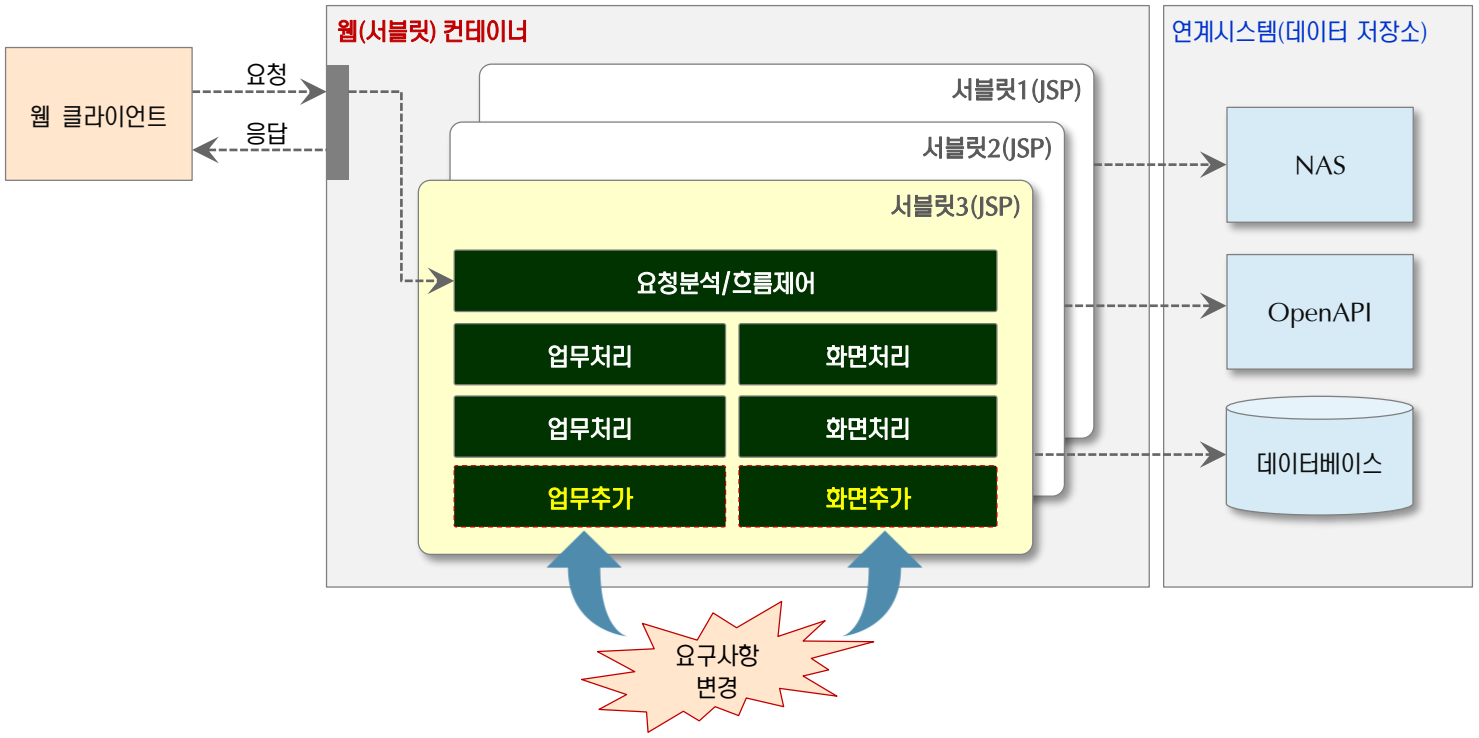
목차 (Table of Contents)

1. JSP 웹 프로그래밍 II

1.1 웹 프로그래밍 구조 (1/7) – 요구사항의 변화와 웹 애플리케이션 구조 (1/3)

✓ 웹 클라이언트 요청 시 하나의 웹 컴포넌트(Servlet 또는 JSP)에서 요청에 대한 모든 것을 처리하는 웹 애플리케이션 구조

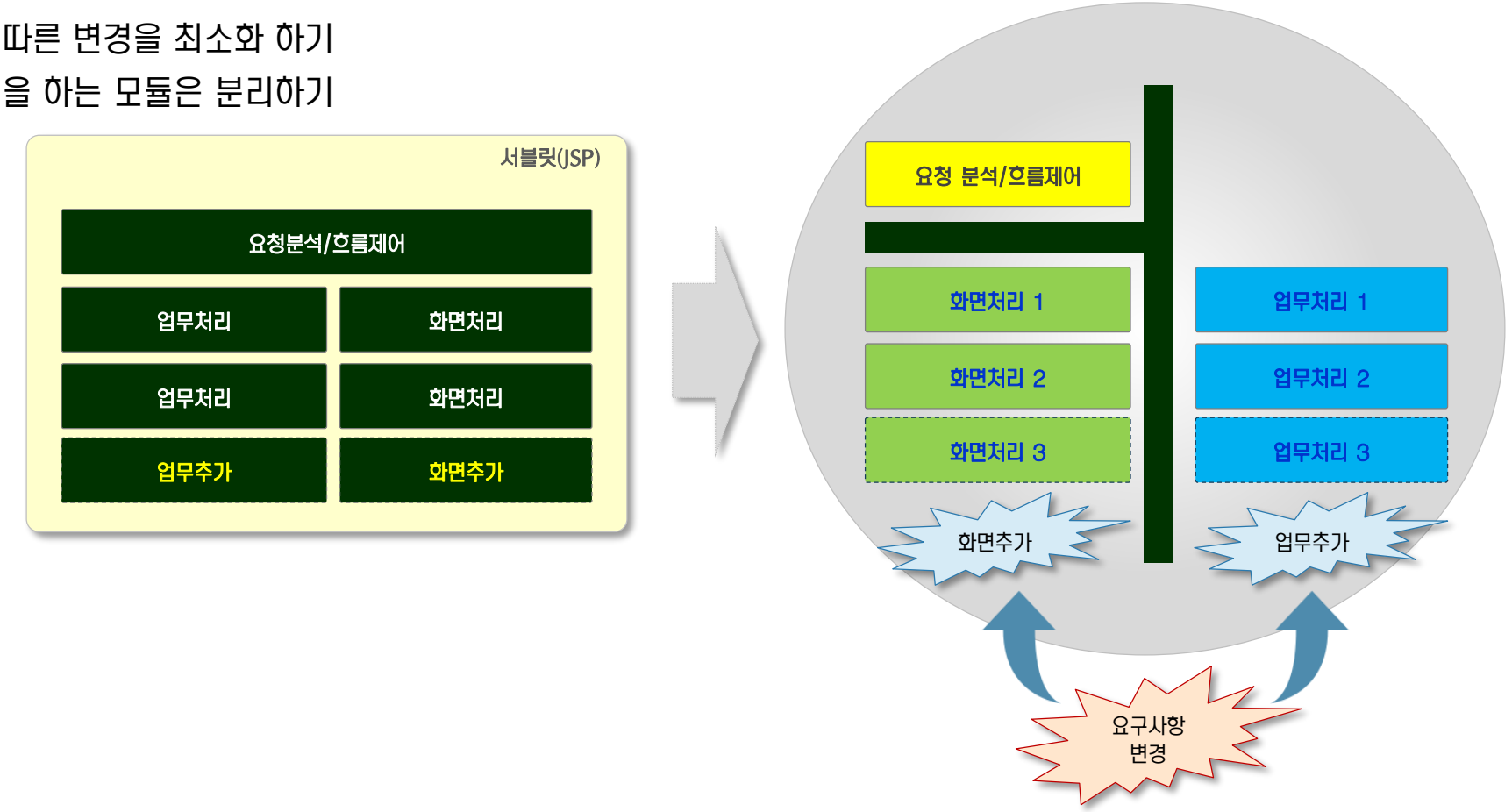
- **장점**
  - 웹 애플리케이션 개발 초창기 주로 사용한 구조로 소규모 프로젝트 진행 시 초기 개발 속도가 빠르고, 누구나 쉽게 배우고 다룰 수 있다.
- **단점**
  - 웹 클라이언트 요청 시 요청 분석 및 흐름을 제어하는 로직과 요청에 대한 내용을 처리하는 비즈니스 로직 그리고 사용자에게 보여줄 결과를 생성하는 화면 처리 로직이 하나의 웹 컴포넌트에 혼재되어 있다.
  - 프로젝트 규모가 커지거나 고객의 요구사항이 자주 변경되는 경우, 유지보수 및 확장이 어렵다.



1.1 웹 프로그래밍 구조 (2/7) – 요구사항의 변화와 웹 애플리케이션 구조 (2/3)

✓ 개선 사항

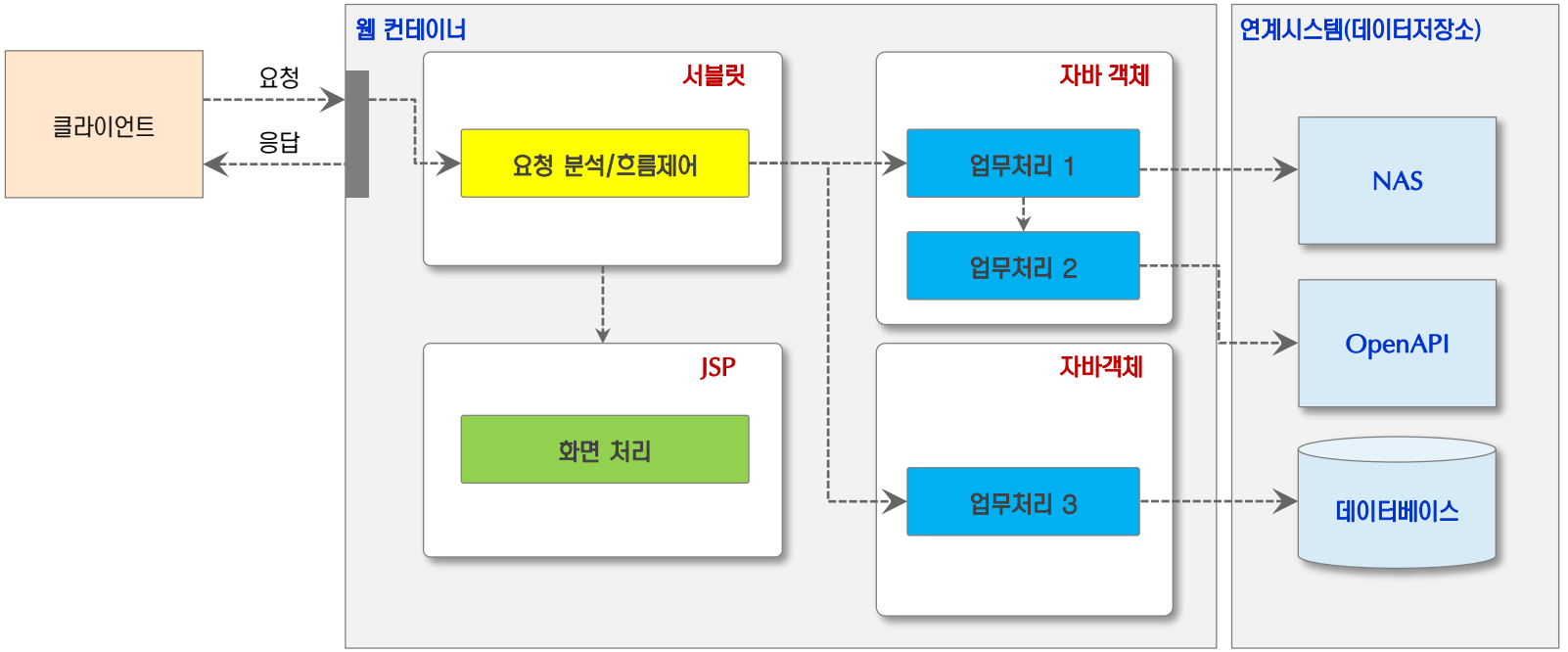
- 웹 애플리케이션은 변경되지 않는 부분과 자주 변경되는 부분을 분리하여, 확장에 열려 있고 변경에는 닫힌 구조로 설계 되어야 한다.
  -> SOLID의 OCP(개방 폐쇄 원칙) 적용
- 비슷한 일을 하는 모듈은 하나로 묶기
- 변화에 따른 변경을 최소화 하기
- 다른 일을 하는 모듈은 분리하기



1.1 웹 프로그래밍 구조 (3/7) – 요구사항의 변화와 웹 애플리케이션 구조 (3/3)

✓ **웹 애플리케이션 내에서 관심사가 서로 다른 로직들은 서로 분리되어야 한다.**

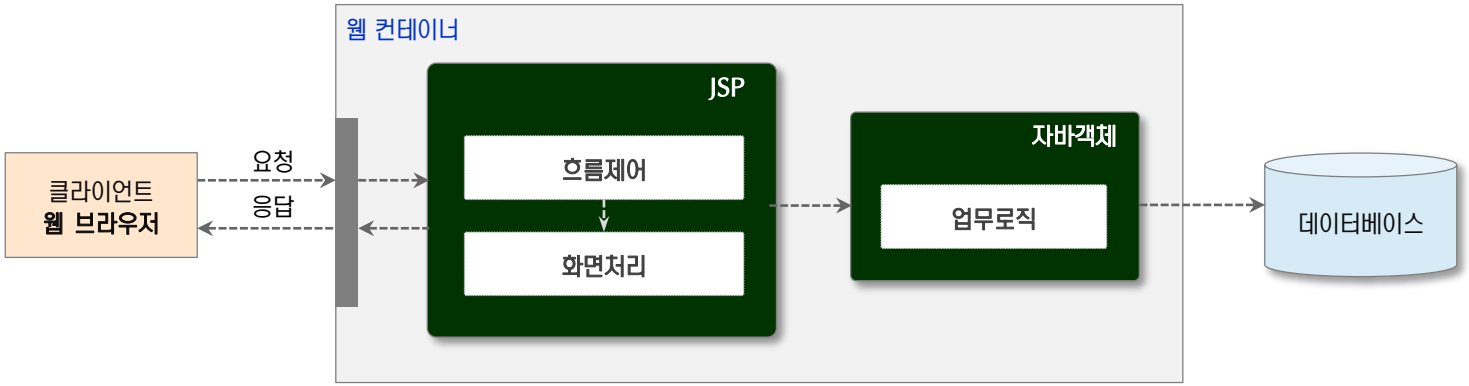
- Servelt : 웹 클라이언트 요청 분석 및 흐름 제어 담당
  - 자바 비즈니스(서비스) 객체와 데이터 접근 객체로 분리하여 개발한다
  - 객체를 분리하면 요구사항이 변화하더라도 관련된 부분만 변경되고 다른 모듈에는 영향을 주지 않는다
- Java 컴포넌트 : 업무 처리 담당
  - 자바 비즈니스 객체 (Service)와 데이터 접근 객체(DAO)로 분리하여 구현한다.
- JSP : 화면 처리 (동적 컨텐츠 생성)



1.1 웹 프로그래밍 구조 (4/7) – Model 1 개발방식

✓ Model 1

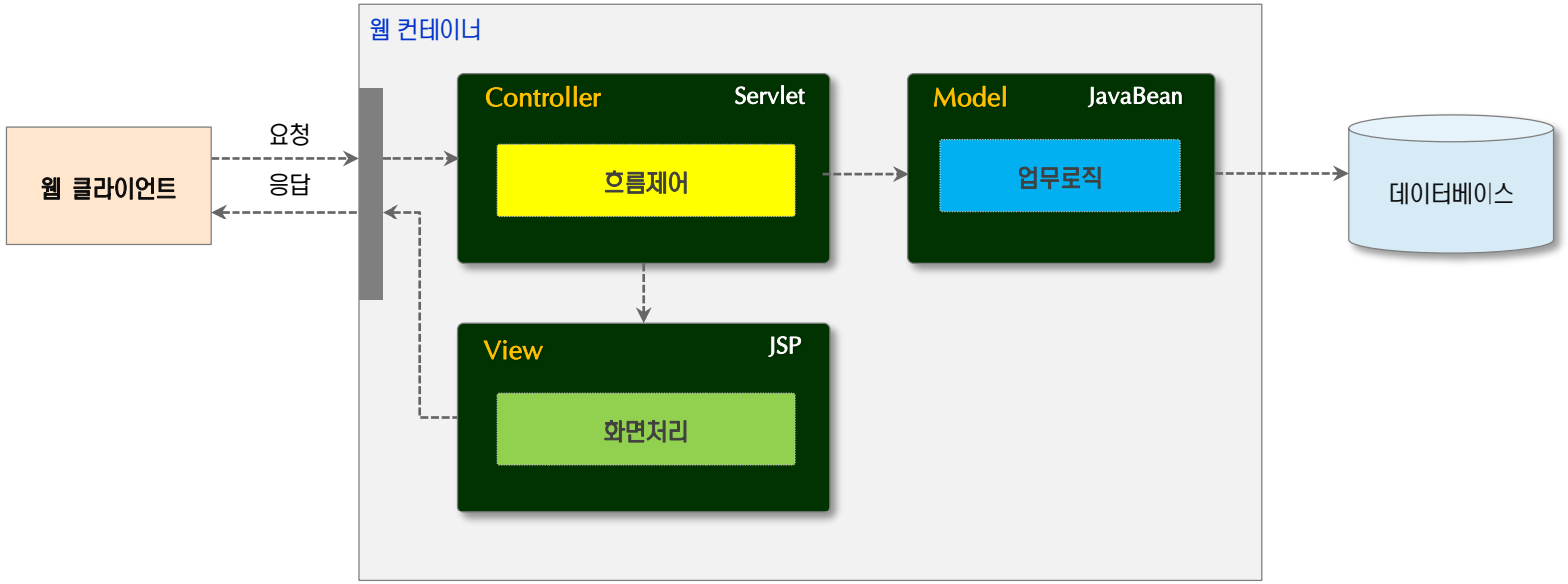
- 웹 애플리케이션 개발 초창기에 주로 사용된 개발 모델로 웹 클라이언트 요청에 대해 각각의 Servlet 또는 JSP가 요청 분석 및 흐름을 제어하고, 비즈니스 로직과 클라이언트에 대한 화면 출력까지 모두 담당하는 개발 초창기 개발 방식이다.
- 장점
  - 구조가 단순하기 때문에 규모가 작은 웹 애플리케이션을 개발할 때 적합하다.
  - JSP 중심의 개발 방법으로 프로젝트 초기 개발 속도가 빠르고, 누구나 쉽게 배우고 다룰 수 있다.
- 단점
  - 웹 클라이언트 요청에 대한 분석 및 흐름 제어로직, 비즈니스 로직, 화면 처리가 JSP안에 혼재되어 있어 애플리케이션 규모가 커질수록 코드 가독성이 떨어지고, 사용자의 많은 요구사항 변화에 대응하기 쉽지 않다.
  - HTML, CSS, JavaScript 코드와 Java 코드가 섞여 있으므로 웹 디자이너와 개발자 간에 원활한 협업이 어렵다.
  - 표준화되지 않은 웹 애플리케이션 구조로 인해 유지보수 및 확장이 어렵다.



1.1 웹 프로그래밍 구조 (5/7) – Model 2 개발방식

✓ Model 2

- MVC 디자인 패턴이 적용된 모델로 웹 클라이언트 요청 분석 및 흐름 제어로직, 비즈니스 로직, 화면처리를 서로 분리하여 개발하는 방식으로 MVC 모델이라고도 한다.(객체 지향적 개발 방법)
- MVC 디자인 패턴은 애플리케이션의 구성요소를 Model, View, Controller 3가지 영역으로 세분화하여 개발하는 방식이다.
  - Servlet(Controller) : 웹 클라이언트 요청 분석 및 흐름 제어를 담당
  - Java 컴포넌트(Model) : 요청에 대한 비즈니스 로직을 담당
  - JSP(View) : 화면처리 담당



1.1 웹 프로그래밍 구조 (6/7) – Model 2 개발방식

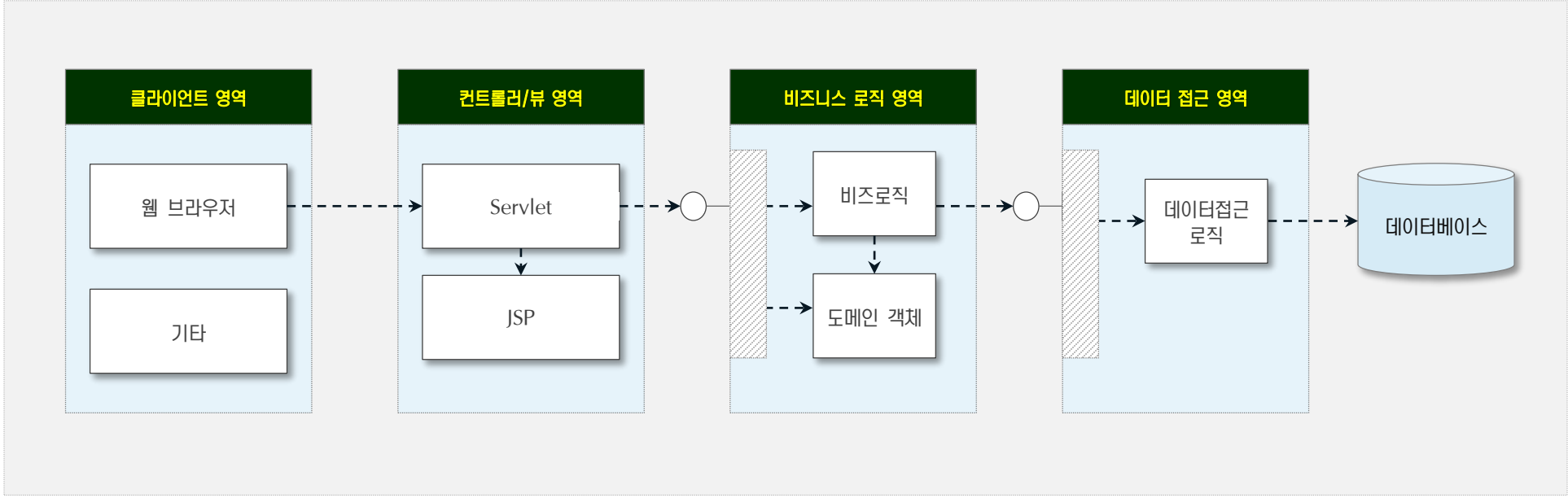
✓ Model 2

- 장점
  - 각각의 역할을 독립적인 모듈로 캡슐화 함으로 써 각 계층의 독립성이 보장되며, 계층별 유지보수가 쉽다.
  - 표준화된 개발이 이루어지므로 협업 작업이 용이하고, 확장성과 개발 생산성이 뛰어나다.
  - 디자이너와 개발자의 작업을 분리하여 원할하게 협업할 수 있다.
  - 특히 View의 경우 JSP 외에 다양한 Vew 기술(Freemarker, Sitemesh, Thymeleaf 등)들이 사용될 수 있다.
- 단점
  - 프로젝트 초기에 아키텍처 설계를 위한 시간이 소요되어 개발 기간이 늘어날 수 있으며, 구조가 복잡하여 개발자들의 이해가 필요하다.

1.1 웹 프로그래밍 구조 (7/7) – Layered Architecture

✓ Layered Architecture 기반 웹 애플리케이션 개발

- 웹은 단순한 정보만을 보여주는 정적인 웹 페이지에서 동적인 내용을 구성하는 웹 애플리케이션으로 발전하였다.
- 웹 애플리케이션은 웹 클라이언트 요청에 따라 정보를 가공하고 저장하며 다양한 형태로 보여준다.
- 다양한 요구사항으로 인해 웹 애플리케이션이 복잡해지는 것을 방지하려면 Layered Architecture 설계가 필요하다.

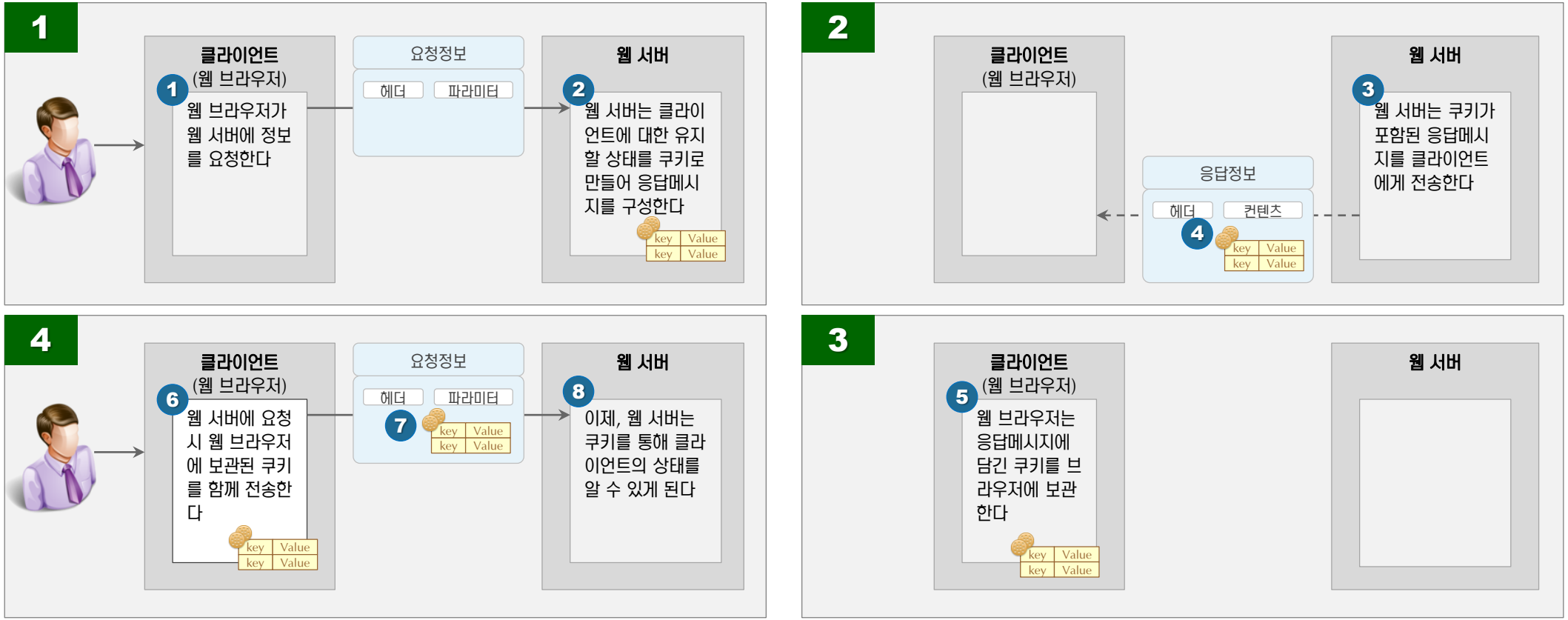


Layered(N-tier) Architecture

1.2 세션과 쿠키 (1/6) – Cookie

✓ 쿠키는 웹 서버에서 웹 클라이언트로 전송되는 작은 텍스트 조각으로 클라이언트의 상태/활동 정보를 저장하기 위하여 설계되었다.

- HTTP 프로토콜은 비연결지향(Stateless)으로 웹 서버가 웹 클라이언트의 상태를 유지할 수 없다.
- 웹 서버로부터 쿠키가 전송되면 웹 클라이언트는 쿠키를 저장해 두었다가 웹 서버 요청 시 다시 전달하여 클라이언트 상태를 유지한다



1.2 세션과 쿠키 (2/6) – Cookie 객체

✓ 쿠키는 웹 서버와 웹 클라이언트 간에 주고 받는 작은 텍스트 조각(이름:값의 문자열 쌍)이다.

- 웹 서버는 쿠키를 생성하여 웹 클라이언트로 쿠키를 보내고, 이후 웹 클라이언트는 매번 요청에 쿠키를 전송한다.
- request 객체를 통해 웹 클라이언트에서 전송된 쿠키 데이터를 조회할 수 있다.
- response 객체를 통해 클라이언트에 전송할 쿠키 데이터를 설정할 수 있다.

쿠키 객체 생성

```
Cookie cookie = new Cookie("username", name);
```

쿠키가 클라이언트에 얼마나 오랫동안 살아 있을지 설정

```
// 초 단위, 0을 설정하면 브라우저 종료 시 쿠키삭제
cookie.setMaxAge(60*60*24*30);
```

response 객체에 클라이언트로 보낼 쿠키 설정

```
response.addCookie(cookie);
```

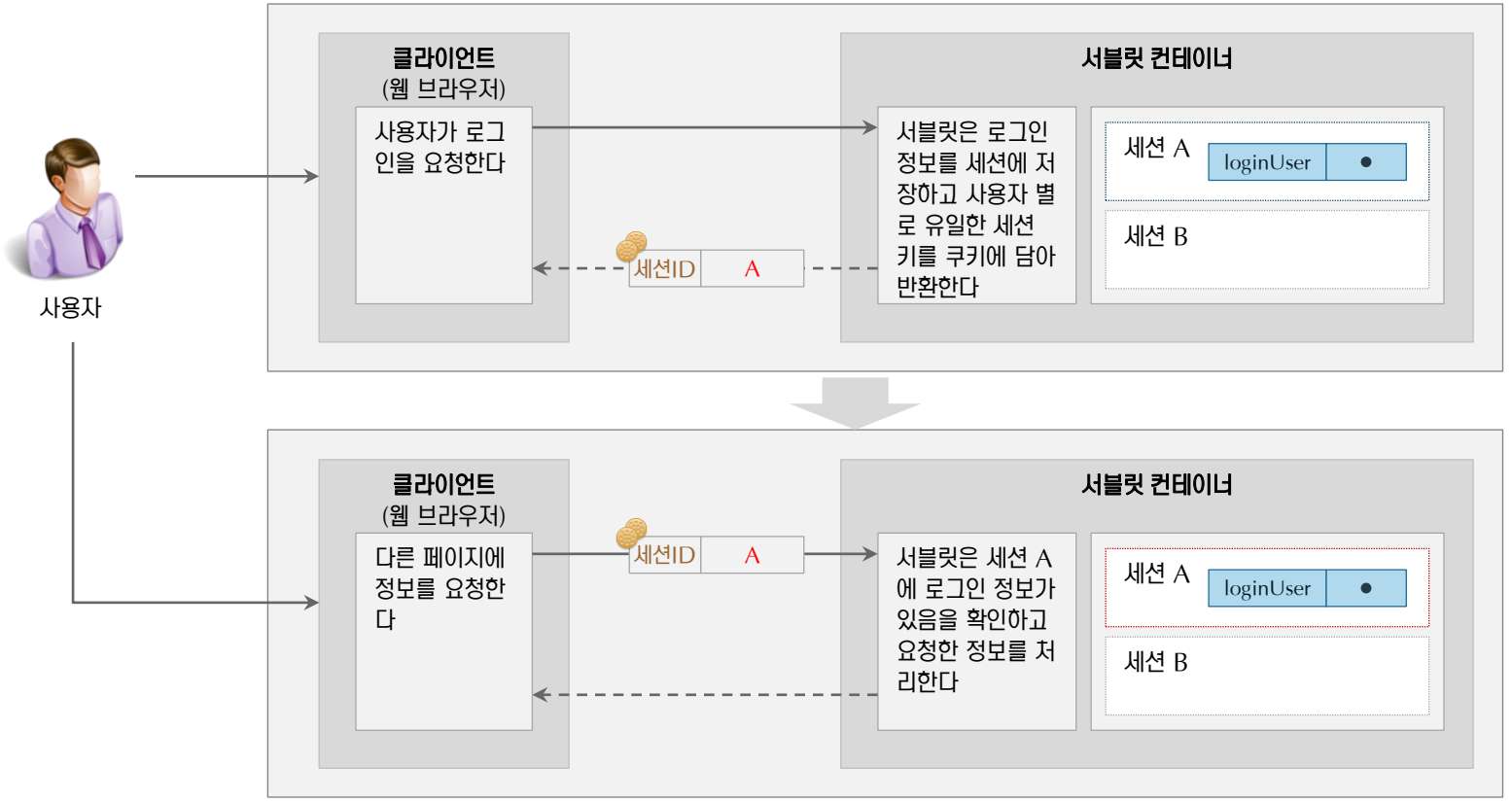
request 객체로 부터 클라이언트에서 전송된 쿠키 조회

```
Cookie[] cookies = request.getCookies();
for (Cookie cookie : cookies) {
    if ("username".equals(cookie.getName())) {
        System.out.println(cookie.getValue());
    }
}
```

1.2 세션과 쿠키 (3/6) – Session

✓ **세션은 사용자에 대한 상태/활동 정보를 웹 서버에 저장할 수 있는 방법을 제공하며 서블릿 컨테이너에 의해 관리된다.**

- 쿠키가 클라이언트 상태정보를 유지하는 방법을 제공하지만 웹 브라우저에 저장되므로 보안상 취약할 수 있다.
- 사용자(웹 브라우저) 식별을 위해 웹 서버에서 유일한 키인 세션 키(JSESSIONID)를 생성하여 쿠키에 담아 웹 브라우저로 전송한다.
- 웹 브라우저는 JSESSIONID 쿠키를 요청마다 서버로 전달하므로 웹 서버는 사용자 세션정보를 찾을 수 있다.



1.2 세션과 쿠키 (4/6) – HttpSession 객체

✓ 세션 객체는 HttpSession 타입으로 request 객체의 getSession() 메소드를 호출하여 생성한다.

- getSession() 메소드는 기존 세션이 존재하면 그대로 반환하고, 없는 경우에만 새로운 세션을 생성하여 반환한다.
- 세션 객체의 isNew() 메소드를 호출하면 세션 객체가 이번에 신규로 생성되었는지 여부를 확인할 수 있다.
- 생성된 세션 ID는 응답객체에 쿠키로 저장되어 웹 클라이언트로 전달된다.

getSession() ➔ 세션이 없으면 새로운 세션을 생성하여 반환한다

```
HttpSession session = req.getSession();

if (session.isNew()) {
    System.out.println("This is new session.");
} else {
    System.out.println("This is previous session.");
}

System.out.println("Session ID : " + session.getId());
```

[기존 세션이 있으면] getSession(true) ➔ 기존 세션을 반환한다

```
HttpSession session = req.getSession(true);
System.out.println("This session may not be null.");
```

[기존 세션이 없으면] getSession(false) ➔ null 을 반환한다

```
HttpSession session = req.getSession(false);
System.out.println("This session may be null.");
```

1.2 세션과 쿠키 (5/6) – Session 관리

✓ **세션이 종료되는 경우는 3가지가 있으며 이 경우 세션에 저장된 속성들은 모두 삭제된다.**

① 서블릿에서 세션 객체의 invalidate() 메소드를 호출한 경우 (예: 로그아웃)

② web.xml 에 설정된 세션 타임아웃 시간 초과된 경우

③ 서블릿 컨테이너가 종료된 경우

세션 종료하기

```
HttpSession session = request.getSession(false);
if (session != null) {
    session.invalidate();
}
```

세션 타임아웃 설정 (web.xml)

```
<session-config>
    <!-- 분 단위 -->
    <session-timeout>15</session-timeout>
</session-config>
```

메소드	설명	언제 사용할까?
getCreationTime()	세션 생성시간을 Time 객체로 리턴	세션이 얼마나 오래 되었는지 알고 싶을 때
getLastAccessedTime()	이 세션으로 들어온 마지막 요청시간	클라이언트가 언제 마지막으로 세션에 접근했는지 알고 싶을 때
setMaxInactiveInterval()	해당 세션에 대한 요청과 요청간의 최대 허용 시간(초 단위)을 지정	클라이언트의 요청이 정해진 시간이 지나도 들어 오지 않을 경우, 해당 세션을 제거하기 위하여 사용함
getMaxInactiveInterval()	해당 세션에 대한 요청과 요청간의 최대 허용 시간(초 단위)을 리턴	세션이 얼마나 오랫동안 비활성화 상태였는지, 여전히 살아있기는 한지 알고 싶을 때. 세션이 invalidate() 되기까지 시간이 얼마나 남았는지 알기 위하여 사용
invalidate()	세션 종료. 현재 세션에 저장된 모든 세션 속성을 제거하는(unbind)작업이 포함됨	클라이언트가 비활성화이거나, 세션 작업이 완료되어 강제로 세션을 종료할 때 . invalidate()는 세션 ID가 더 이상 존재하지 않으니, 관련 속성을 세션 객체에서 제거하라는 의미

1.2 세션과 쿠키 (6/6) – HttpSession URL 재작성

✓ 서블릿 컨테이너는 쿠키에 담겨있는 JSESSIONID 쿠키를 이용하여 웹 클라이언트를 식별하고 세션정보를 관리한다.

✓ 웹 클라이언트에서 쿠키를 사용하지 않도록 설정되어 있는 경우에 세션을 관리할 수 없다.

- 해결책으로 JSESSIONID를 URL 뒤에 붙이는 방법을 사용한다 (URL Rewriting)
- 웹 클라이언트는 서버 요청 시 쿠키 대신에 URL 파라미터를 통해 JSESSIONID를 서버에 전송한다.
- URL 재작성을 위해 response 객체에는 encodeURL() / encodeRedirectURL() 메소드를 지원한다.

encodeURL(URL) : URL 문자열 뒤에 세션 ID를 추가하여 반환한다

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

HttpSession session = request.getSession();

out.println("<html><body>");

// URL 뒤에 세션 ID를 추가한다
out.println("<a href='" + response.encodeURL("/signup") + "'>Sign up</a>");
out.println("</body></html>");
```

encodeRedirectURL(URL) : 리다이렉트를 위한 목적으로 URL을 재작성 할 때 사용한다

```
String redirectURL = response.encodeRedirectURL("/signup");
response.sendRedirect(redirectURL);
```

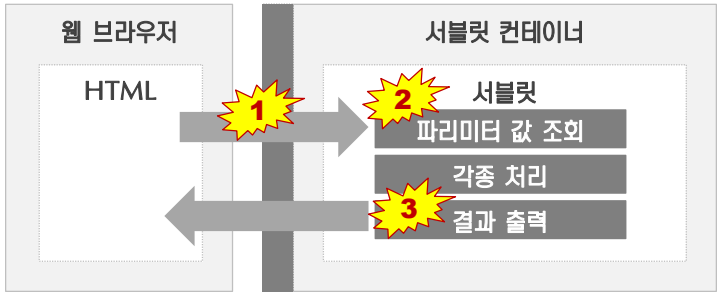
1.3 한글 깨짐 문제 해결

✓ 웹 애플리케이션을 개발 시 **한글 깨짐** 현상에 대한 **문제 해결 방법**

① GET 방식으로 전달되는 요청 파라메터가 있을 경우 웹 서버에서 문자 인코딩 설정이 필요하다.

② POST 방식으로 전달되는 요청 파라메터가 있을 요청 객체의 파라미터 값을 꺼내기 전에 문자 인코딩 설정이 필요하다.

③ 서블릿에서 응답 객체의 출력 스트림에 컨텐츠를 전송하기 전 Content-Type에 문자 인코딩 설정이 필요하다.



0	[공통] 모든 소스코드의 텍스트 인코딩은 UTF-8로 통일한다.
1	URL 파라미터로 전달되는 값의 문자 인코딩을 설정하기 위해 웹 서버의 URI Encoding을 UTF-8로 설정한다.
2	요청 객체에서 파라미터 값을 꺼내기 전에 문자 인코딩을 UTF-8로 설정한다.
3	응답 객체로 컨텐츠를 전송하기 전에 ContentType 헤더의 charset 속성 값을 UTF-8 로 설정한다.

[웹서버] GET 요청에 대한 server.xml URIEncoding 설정 (톰캣 기준)

```
<Connector port="80" protocol="HTTP/1.1" … URIEncoding="utf-8" />
```

[서블릿] POST 요청 시 request 객체에서 파라미터 값을 꺼내기 전에

```
request.setCharacterEncoding("utf-8");
```

[서블릿] 응답 객체의 출력 스트림에 HTML 문서를 출력하기 전

```
response.setContentType("text/html; charset=utf-8");
```

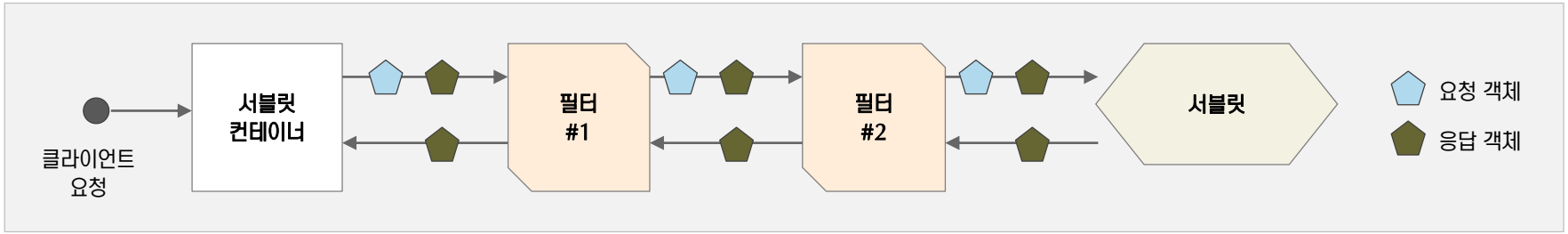
1.4 서블릿 필터 (1/3) – Servlet Filter 소개

✓ 서블릿 필터는 서블릿 컨테이너가 '서블릿으로 요청을 넘기기 전'과 '응답을 웹 서버로 전송하기 전'에 요청을 가로채어 어떤 공통적인 작업을 처리를 할 수 있도록 제공되는 API이다.

- 예) 요청 객체 및 응답 객체에 대한 공통적인 문자 인코딩 설정을 위하여 필터를 활용

✓ 서블릿 필터는 javax.servlet.Filter 인터페이스를 구현하여 작성한다.

- 서블릿 컨테이너는 web.xml(DD)에 선언한 정보를 바탕으로 필터를 언제 실행할지 결정한다.



Filter 인터페이스의 메소드	설명
init(FilterConfig)	init() 메소드는 필터가 서블릿 컨테이너에 등록될 때 최초 한번 초기화를 위하여 자동 호출된다. 필터 설정 초기화 파라미터는 FilterConfig 객체를 통해 조회할 수 있다.
doFilter(ServletRequest, ServletResponse, FilterChain)	doFilter() 메소드는 필터에서 수행하는 로직을 구현한다. 후속 필터(또는 서블릿)으로 처리를 위임하기 위해서 FilterChain 객체의 doFilter() 메소드를 반드시 호출해 주어야 한다.
destroy()	destroy() 메소드는 필터가 서블릿 컨테이너에서 소멸할 때 한번 호출된다.

1.4 서블릿 필터 (2/3) – Servlet Filter 활용

✓ Servlet Filter를 이용한 문자 인코딩 공통 처리

- 서블릿 별로 반복적으로 처리해야 하는 문자 인코딩 설정을 서블릿 필터를 구현하여 처리한다.
- 문자 인코딩 필터는 웹 클라이언트가 컨텐츠에 대한 MIME 타입을 알 수 있도록 ContentType 헤더를 설정한다.
- 문자 인코딩 필터는 CharacterEncodingFilter로 명명한다.

1. 문자 인코딩 필터 구현
2. 필터 등록 및 필터 매핑

1. init() 메소드는 필터 초기화 파라미터에서 encoding 값을 조회하여 필터의 인스턴스 변수 encoding에 할당한다. Init()은 서블릿 컨테이너에서 최초 한번만 호출된다.

2. doFilter() 메소드는 요청 객체와 응답 객체의 문자 인코딩을 encoding 값으로 설정한다. 그리고 브라우저를 위한 MIME 타입을 설정한다.

문자 인코딩 필터 구현 (project.web.common.CharacterEncodingFilter)

```
public class CharacterEncodingFilter implements Filter {
    private String encoding;

    @Override
    public void init(FilterConfig config) throws ServletException {
        this.encoding = config.getInitParameter("encoding");
    }

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp,
            FilterChain chain) throws IOException, ServletException {
        req.setCharacterEncoding(encoding);
        resp.setCharacterEncoding(encoding);
        resp.setContentType("text/html;charset="+encoding);

        chain.doFilter(req, resp);
    }

    @Override
    public void destroy() {  }
}
```

1.4 서블릿 필터 (3/3) – Servlet Filter 활용

✓ CharacterEncodingFilter를 web.xml에 등록한다.

- 문자 인코딩이 변경될 때 소스코드를 변경하지 않도록 인코딩은 필터 초기화 파라미터에 설정한다.
- 필터는 모든 요청에 대하여 수행하도록 필터를 매핑한다.
- 서블릿 3.0 부터는 @WebFilter 에노테이션을 통한 필터 등록 및 매핑을 지원한다.

1 문자 인코딩 필터 구현

2 필터 등록 및 필터 매핑

1. <filter> 요소를 사용하여 필터를 등록한다. <init-param> 요소에 필터 초기화 파라미터를 정의한다. 파라미터의 이름은 encoding, 값을 UTF-8로 설정한다.

2. <filter-mapping> 요소를 통해 문자 인코딩 필터를 매핑한다. url-pattern은 *.do 로 설정한다.

필터 등록 및 필터 매핑 (web.xml)

```
<filter>
  <description>문자인코딩 필터</description>
  <filter-name>characterEncodingFilter</filter-name>
  <filter-class>project.web.common.wharacterEncodingFilter
  </filter-class>
  <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>characterEncodingFilter</filter-name>
  <url-pattern>/*.do</url-pattern>
  <url-pattern>/some-path/*</url-pattern>
</filter-mapping>
```

1.4 서블릿필터 (3/3) – Servlet Filter 활용

✓ 서블릿 3.0 @WebFilter 에노테이션을 통한 필터 등록 및 매핑

문자 인코딩 필터 구현 (project.web.common.CharacterEncodingFilter)

```
@WebFilter(
     description = "문자인코딩 필터",
     urlPatterns = {"/some-path/*", "*.do" },
     initParams = {
          @WebInitParam(name = "encoding", value = "utf-8")
})
public class CharacterEncodingFilter implements Filter {
    // 기존 코드와 동일
}
```

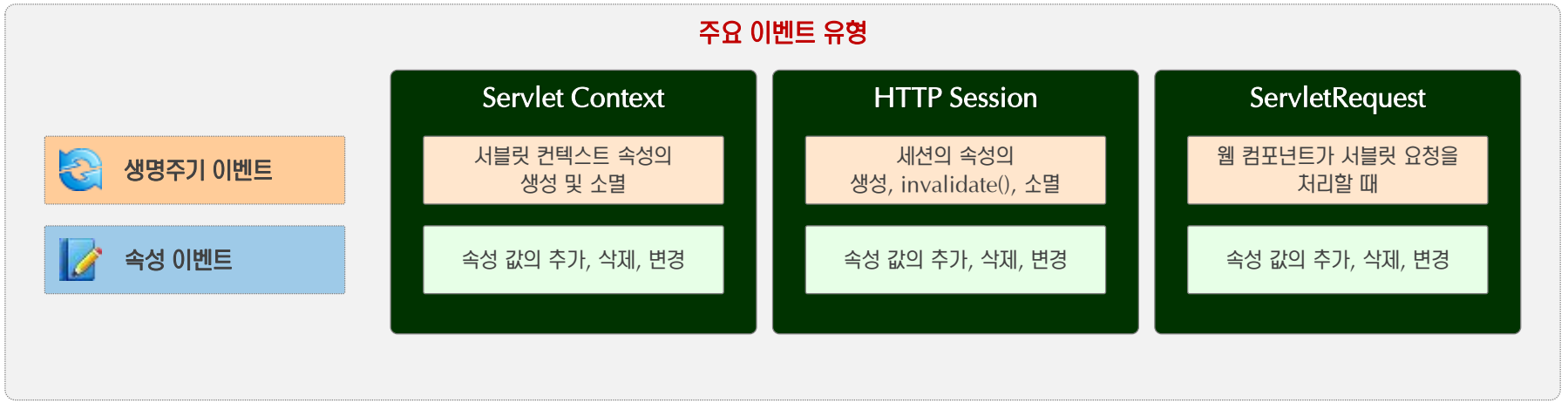
1.5 서블릿 리스너 (1/3) – 이벤트와 Lisenter 소개

✓ **웹 애플리케이션 안에서는 서블릿 생명주기 또는 스코프 객체의 속성 값의 변화 등과 관련된 다양한 시스템 이벤트가 발생**

- 웹 컨테이너는 웹 애플리케이션에서 발생하는 이벤트를 통지하여 특별한 처리를 할 수 있는 이벤트 Listener 인터페이스를 제공한다.

✓ **이벤트 Listener는 처리할 이벤트 종류에 따라 리스너 인터페이스를 구현하여 개발한다.**

- 이벤트 유형에 따라 ServletContext, HTTP Session, ServletRequest로 구분하며, 이벤트 종류에 따라 생명주기 이벤트, 속성값 변화 등으로 분류할 수 있다.



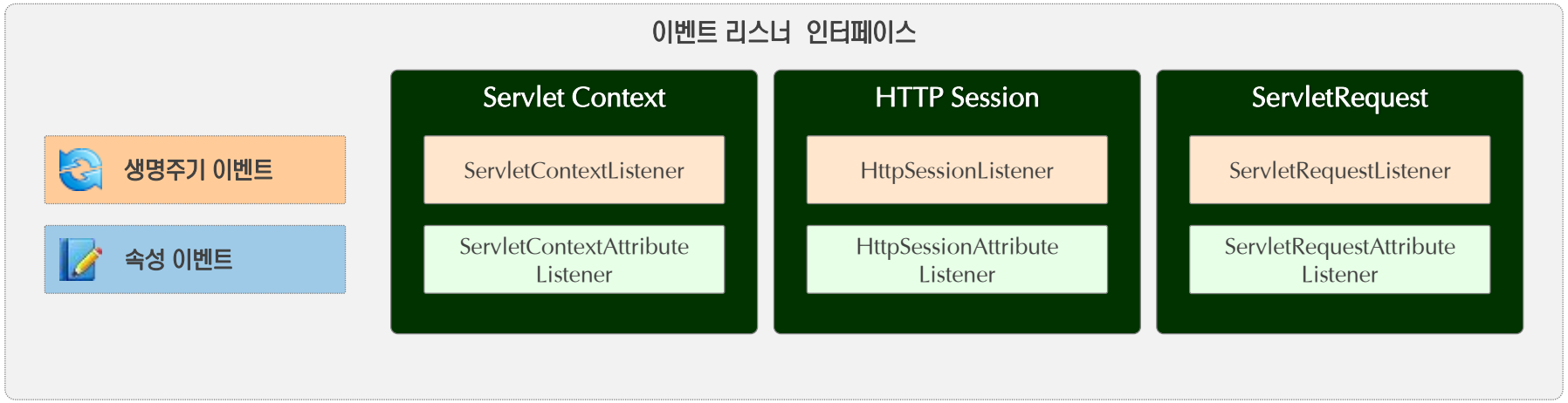
문제상황) 사용자가 로그인하거나 로그아웃 했을 때, 서버에 로그를 남기고 싶다. 그런데 문제가 있다. 로그인은 사용자의 서블릿 요청이므로 로그인을 담당하는 서블릿에서 로그를 남기면 될 것 같은데, 로그아웃의 경우 사용자의 요청에 의한 로그아웃 뿐만 아니라, 세션 타임아웃이 발생하거나, 서블릿 컨테이너가 종료 될 때 등 다양한 상황에서 발생한다. 좋은 해결 방법이 무엇일까?

해결책) 문제 상황은 이벤트 리스너를 활용하기에 적합한 상황이다. 로그인 정보는 보통 세션에서 관리되므로 HttpSession 속성 관련 이벤트를 처리하는 이벤트 리스너를 구현하여 로그인 정보를 담는 세션 속성의 추가 및 삭제 이벤트에 대해 로깅 처리를 구현하면 된다.

1.5 서블릿 리스너 (2/3) – 이벤트와 Listener 종류

✓ **웹 애플리케이션에서 발생하는 이벤트의 종류에 따라 다양한 이벤트 리스너가 인터페이스로 제공된다.**

- ServletContext*Listener는 서블릿 컨텍스트의 생명주기와 속성의 변화에 관한 이벤트를 처리한다.
- HttpSession*Listener는 HttpSession 객체의 생명주기 및 속성의 변화에 관한 이벤트를 처리한다.
- ServletRequest*Listener는 서블릿을 요청하거나, 요청객체 속성의 변화에 관한 이벤트를 처리한다.



1.5 서블릿 리스너 (3/3) – 이벤트와 Listener 활용

✓ 사용자가 로그인하거나 로그아웃 한 경우 로그를 남기도록 기능을 추가한다.

- 로그인 정보는 HttpSession 속성으로 관리되므로 HttpSessionAttributeListener 인터페이스가 적합한 후보이다.
- HttpSessionAttributeListener 인터페이스를 구현하여 UserAccessLoggingListener 클래스를 생성한다.
- 리스너 등록은 web.xml 에 정의하거나 서블릿 3.0 부터 제공되는 @WebListener 어노테이션을 사용한다.

세션 속성의 추가 및 삭제 이벤트를 처리하는 이벤트 리스너 구현

```
@WebListener
public class UserAccessLoggingListener implements HttpSessionAttributeListener {
    private static final String SESSION_ATTRIBUTE_NAME = "loginUser";

    @Override
    public void attributeAdded(HttpSessionBindingEvent evt) {
        if (SESSION_ATTRIBUTE_NAME.equals(evt.getName())) {
            User user = (User) evt.getValue();
            System.out.println(user.getName() + " is login.");
        }
    }

    @Override
    public void attributeRemoved(HttpSessionBindingEvent evt) {
        if (SESSION_ATTRIBUTE_NAME.equals(evt.getName())) {
            User user = (User) evt.getValue();
            System.out.println(user.getName() + " is logout.");
        }
    }
    @Override
    public void attributeReplaced(HttpSessionBindingEvent evt) {
    }
}
```

2. MVC 디자인 패턴 적용 Model 2 설계 및 구현

2.1 MVC 디자인 패턴 적용 (1/6) – 서블릿과 JSP 한계

✓ Servlet 만으로 웹 애플리케이션을 개발할 때는 뷰(View) 화면을 위해 HTML을 출력하는 작업이 자바 코드와 혼재하기 때문에 유지보수 및 확장이 용이하지 않다.

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

✓ 해결되지 않는 몇가지 문제

- 회원 저장을 위한 JSP의 경우, 회원 정보를 저장하기 위한 비즈니스 로직(자바 코드)이 대부분이고, 나머지는 결과를 HTML로 출력하기 위한 뷰 영역이다.
- 회원 목록의 경우에도 마찬가지다. 자바 코드 데이터를 조회하는 Dao 등등 다양한 코드가 모두 JSP에 노출되어 있다.
- JSP가 너무 많은 역할을 담당하고 있다.
- 대규모 프로젝트 수행 시 수백 줄이 넘어가는 JSP를 떠올려보면 정말 지옥과 같다(유지보수 지옥)

2.1 MVC 디자인 패턴 적용 (2/6) – MVC 디자인 패턴

✓ MVC 패턴의 등장

- 비즈니스 로직은 서블릿처럼 다른 곳에서 처리하고, JSP는 목적에 맞게 HTML로 화면(View)을 출력하는 일에만 집중하도록 한다.
- 과거 개발자들도 모두 비슷한 고민이 있었고, 그래서 등장한 것이 MVC 디자인 패턴이다.

✓ MVC 패턴 개요

- 너무 많은 역할
  - 하나의 서블릿이나 JSP만으로 비즈니스 로직과 뷰 렌더링까지 모두 처리하게 되면, 너무 많은 역할을 하게 되고, 결과적으로 유지보수가 어려워진다.
  - 비즈니스 로직을 호출하는 부분에 변경이 발생해도 해당 코드를 수정해야 하고, UI를 변경할 일이 있어도 비즈니스 로직이 함께 있는 해당 파일을 수정해야 한다.
- 기능 특화
  - 특히 JSP 같은 뷰 템플릿은 화면을 렌더링 하는데 최적화 되어 있기 때문에 이 부분의 업무만 담당하는 것이 가장 효과적이다.

2.1 MVC 디자인 패턴 적용 (3/6) – MVC 디자인 패턴

✓ Model, View, Controller

- MVC 패턴은 하나의 서블릿이나, JSP로 처리하던 것을 컨트롤러(Controller)와 뷰(View)라는 영역으로 서로 역할을 나눈 것을 말한다.
- 웹 애플리케이션은 보통 이 MVC 패턴을 적용하여 구현한다.

✓ Controller

- HTTP 요청을 받아서 요청 파라미터를 검증하고, 비즈니스 로직을 실행한다.
- 그리고 뷰에 전달할 결과 데이터를 조회해서 모델에 담는다.

✓ Model

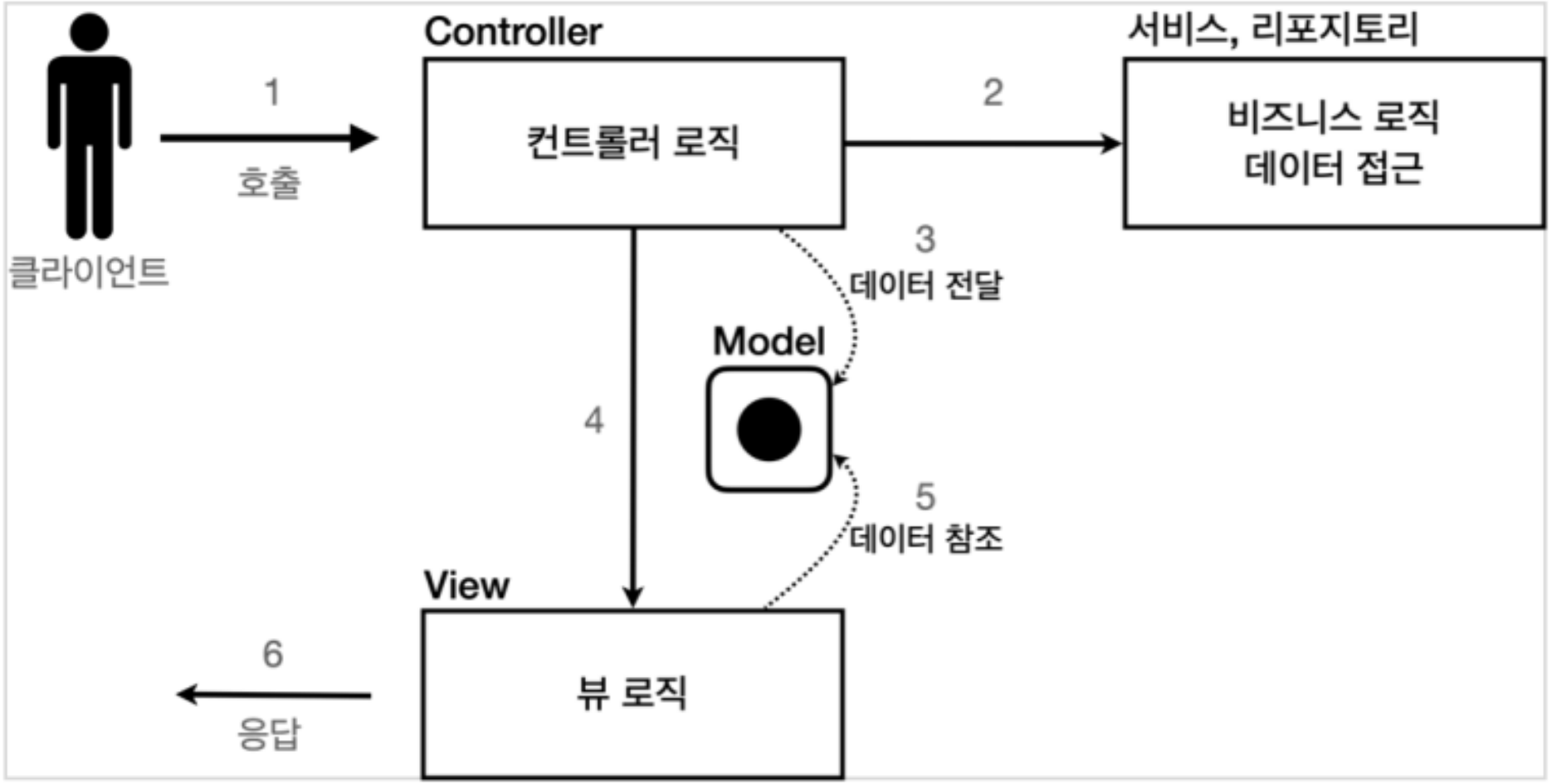
- 뷰에서 출력할 데이터를 담아두기 위한 모델로 스코프 객체를 사용한다.
- 뷰가 필요로 하는 데이터를 모두 모델에 담아서 전달해주는 덕분에 뷰는 비즈니스 로직이나 데이터 접근을 몰라도 되고, 화면을 렌더링 하는 일에만 집중할 수 있다.

✓ View

- 모델에 담겨있는 데이터를 사용해서 화면을 렌더링 일에만 집중한다(동적 컨텐츠 출력)

2.1 MVC 디자인 패턴 적용 (4/6) – MVC 디자인 패턴

✓ Model, View, Controller



2.1 MVC 디자인 패턴 적용 (5/6) – MVC 디자인 패턴 한계

✓ MVC 패턴 문제점

- MVC 패턴을 적용한 덕분에 컨트롤러의 역할과 뷰를 렌더링 하는 역할을 명확하게 구분할 수 있다.
- 특히 뷰는 화면을 그리는 역할에 충실한 덕분에, 코드가 간결하고 직관적이다.
- 단순하게 모델에서 필요한 데이터를 꺼내고, 화면을 렌더링 하면 된다.
- 하지만, 컨트롤러는 코드 중복이 많고, 필요하지 않는 코드들도 혼재해 있다.

✓ 세부 Controller의 단점

- forward 중복
  - View로 이동하는 코드가 항상 중복 호출되어야 한다.

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

- viewPath 중복

```
String viewPath = "/WEB-INF/views/member/signup.jsp";
```

  - Prefix(접두사): /WEB-INF/views/, suffix(접미사): .jsp
  - 그리고 만약 jsp가 아닌 tiles, thymeleaf 같은 다른 뷰로 변경할 경우 전체 코드를 다 변경해야 한다.
- 공통 코드 처리의 한계
- 기능이 복잡해 질수록 세부 컨트롤러에서 공통으로 처리해야 하는 부분이 점점 더 많이 증가한다.

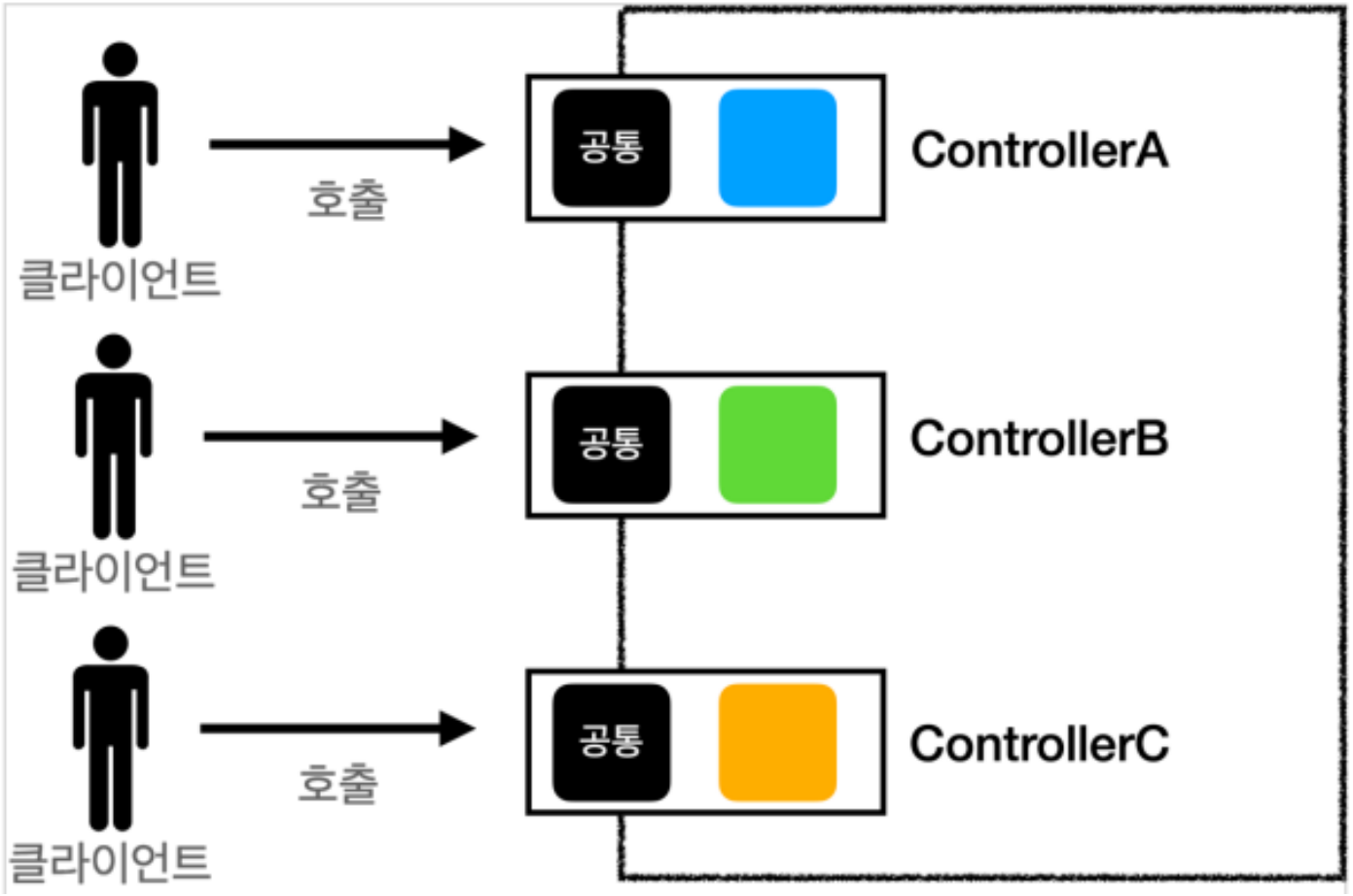
2.1 MVC 디자인 패턴 적용 (6/6) – MVC 디자인 패턴 한계

✓ **세부 컨트롤러의 문제 해결**

- 문제를 해결하기 위해 세부 컨트롤러 호출 전에 공통 기능을 처리하는 소위 **수문장 역할**을 담당하는 기능이 필요하다.
- 디자인 패턴 중 프론트 컨트롤러(Front Controller) 패턴을 도입하면 이런 문제를 해결할 수 있다.
- 웹 클라이언트의 요청 입구를 하나로(단일 창구) 단일화 한다.
- 추후 학습하는 스프링 MVC 프레임워크의 핵심도 바로 이 프론트 컨트롤러에 있다.

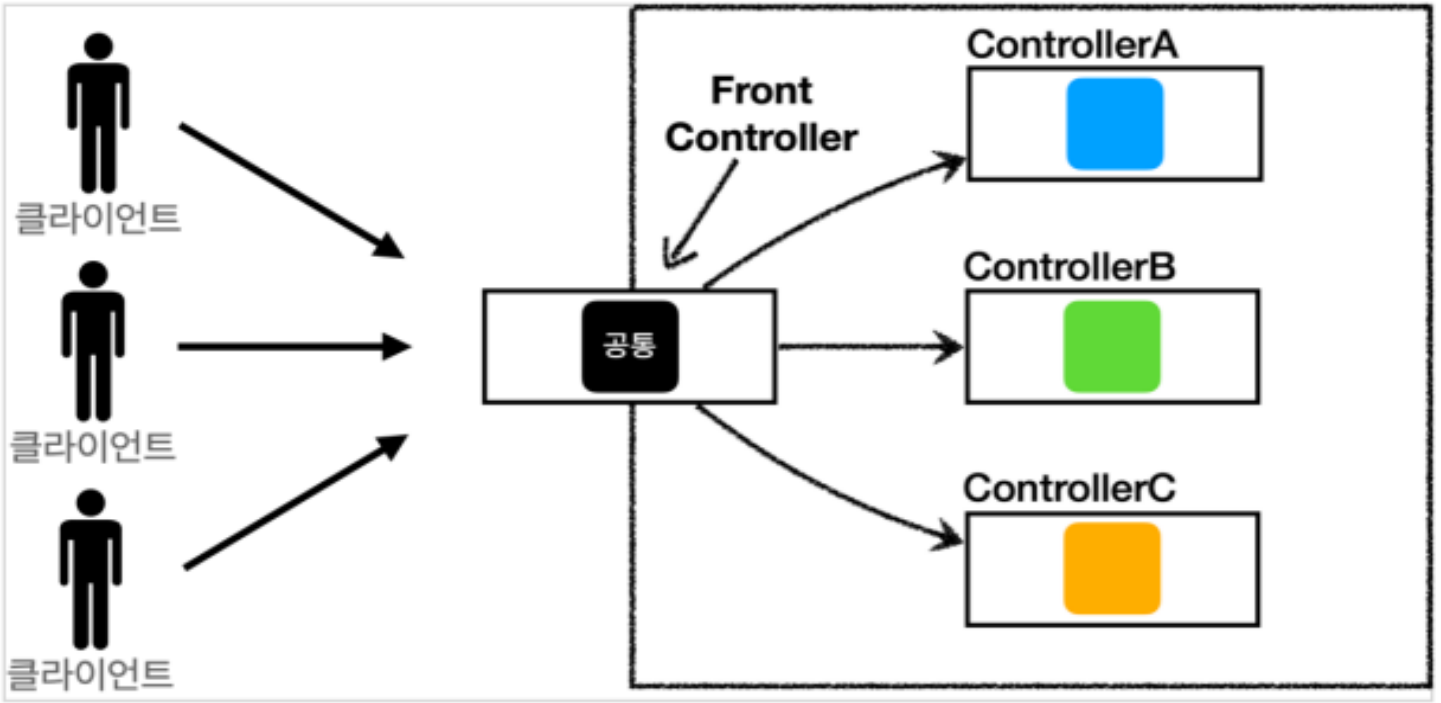
2.2 MVC 프레임워크 만들기 – 프론트 컨트롤러 패턴 적용

✓ 프론트 컨트롤러 패턴 도입 전



2.2 MVC 프레임워크 만들기 – 프론트 컨트롤러 패턴 적용

✓ 프론트 컨트롤러 도입 후

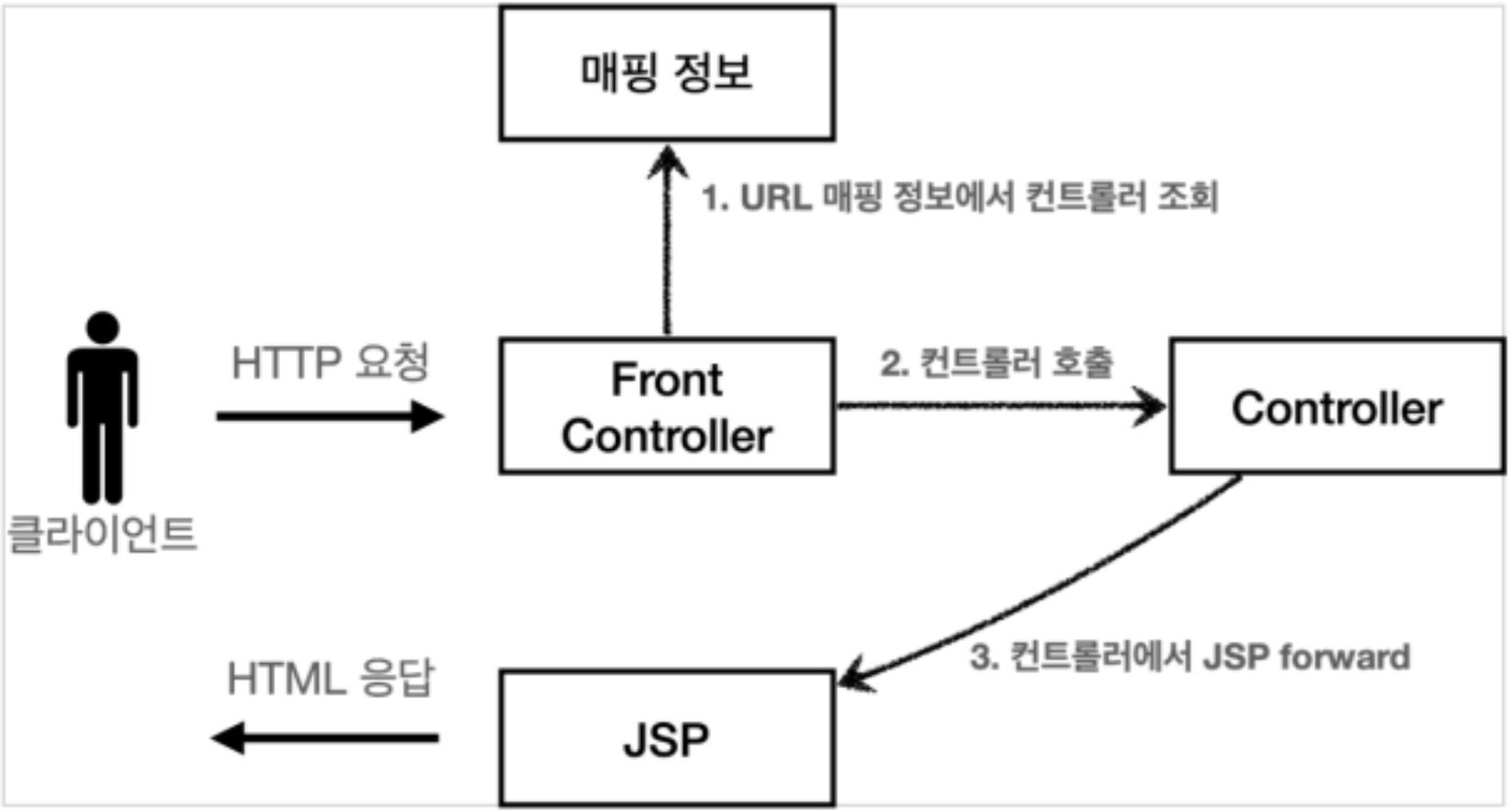


✓ 프론트 컨트롤러 패턴 특징

- 프론트 컨트롤러(서블릿) 하나로 모든 웹 클라이언트의 요청을 받는다(요청 입구를 하나로!!!)
  - 웹 클라이언트 요청에 대한 공통 처리 가능
- 프론트 컨트롤러가 웹 클라이언트 요청에 맞는 세부 컨트롤러를 찾아서 실행한다.
- 프론트 컨트롤러를 제외한 나머지 세부 컨트롤러들은 서블릿으로 구현하지 않아도 된다.

2.2 MVC 프레임워크 만들기 – 프론트 컨트롤러 도입 V1

✓ V1 구조

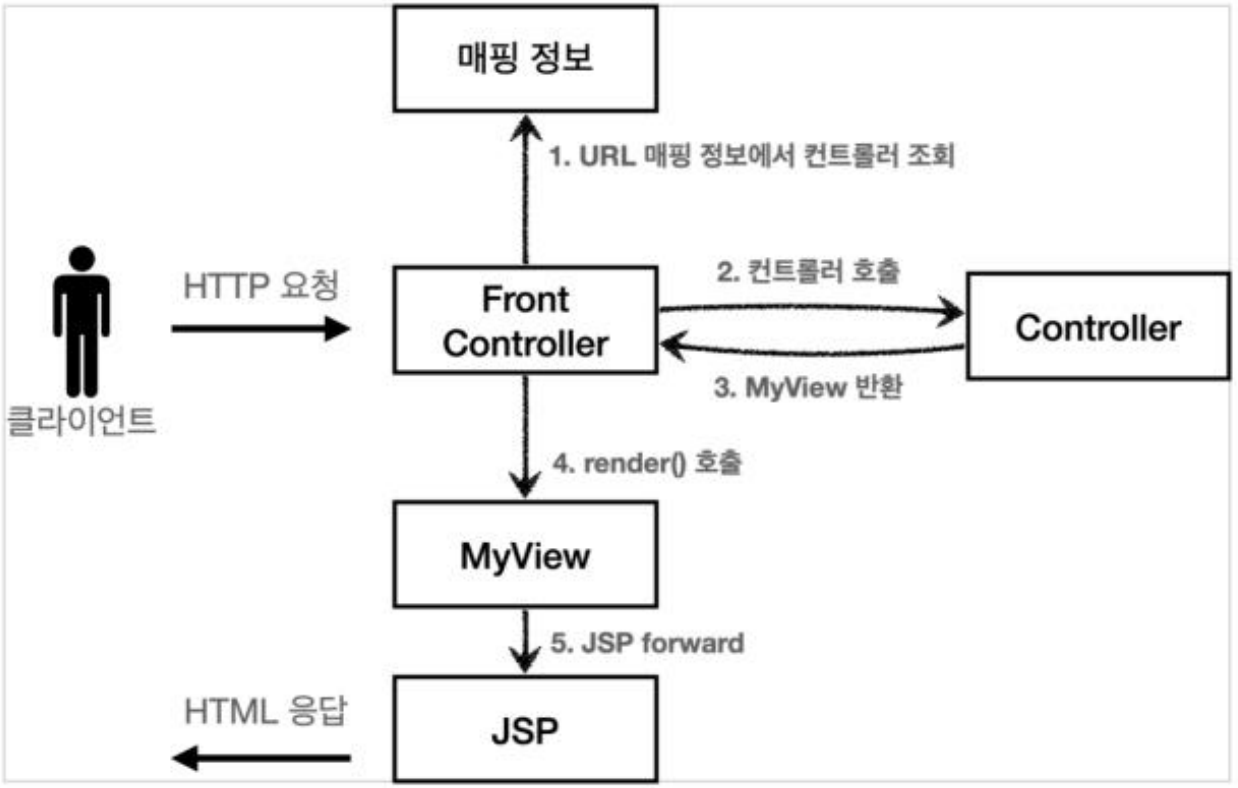


2.2 MVC 프레임워크 만들기 – View 분리 V2

✓ 모든 세부 컨트롤러에서 뷰로 위임하는 부분에 중복 코드가 존재한다.

```
String viewPath = "/WEB-INF/views/member/signup.jsp";
RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
dispatcher.forward(request, response);
```

- 이 부분을 분리하기 위해 별도로 뷰를 처리하는 클래스를 만든다.



2.2 MVC 프레임워크 만들기 – Model 추가 V3

✓ 세부 컨트롤러에서 서블릿 종속성 제거

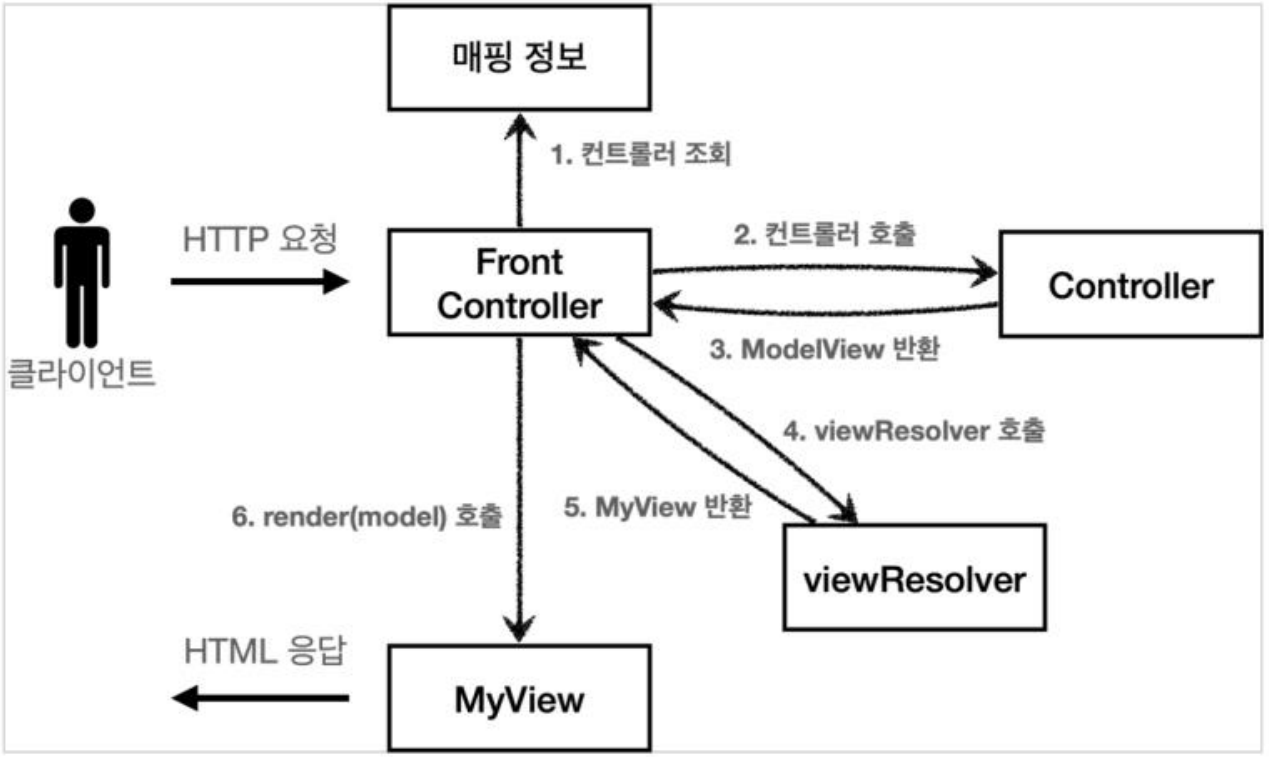
- 세부 컨트롤러 입장에서 HttpServletRequest, HttpServletResponse은 꼭 필요할까?
- 웹 클라이언트의 요청 파라미터를 자바의 Map으로 전달하면 세부 컨트롤러에서 서블릿 관련 기술을 몰라도 구현할 수 있다.
- 또한 스코프 객체인 request를 Model로 사용하는 대신에 별도의 Model 객체를 만들어서 반환한다.
- 세부 컨트롤러 구현 코드가 매우 단순해지고, 단위 테스트도 용이해 진다.

✓ 뷰 이름 중복 제거

- 세부 컨트롤러에서 지정하는 뷰 이름에 중복이 있는 것을 확인할 수 있다.
- 세부 컨트롤러는 물리적 경로가 아닌 뷰의 논리적 이름만 반환하고, 실제 물리적 경로는 프론트 컨트롤러에서 처리하도록 단순화 한다
- 추후 뷰의 물리적 경로가 변경되더라도 프론트 컨트롤러에서만 변경하면 된다.
- 예) WEB-INF/views/member/signup.jsp -> signup

2.2 MVC 프레임워크 만들기 – Model 추가 V3

✓ 세부 컨트롤러에서 서블릿 종속성 제거



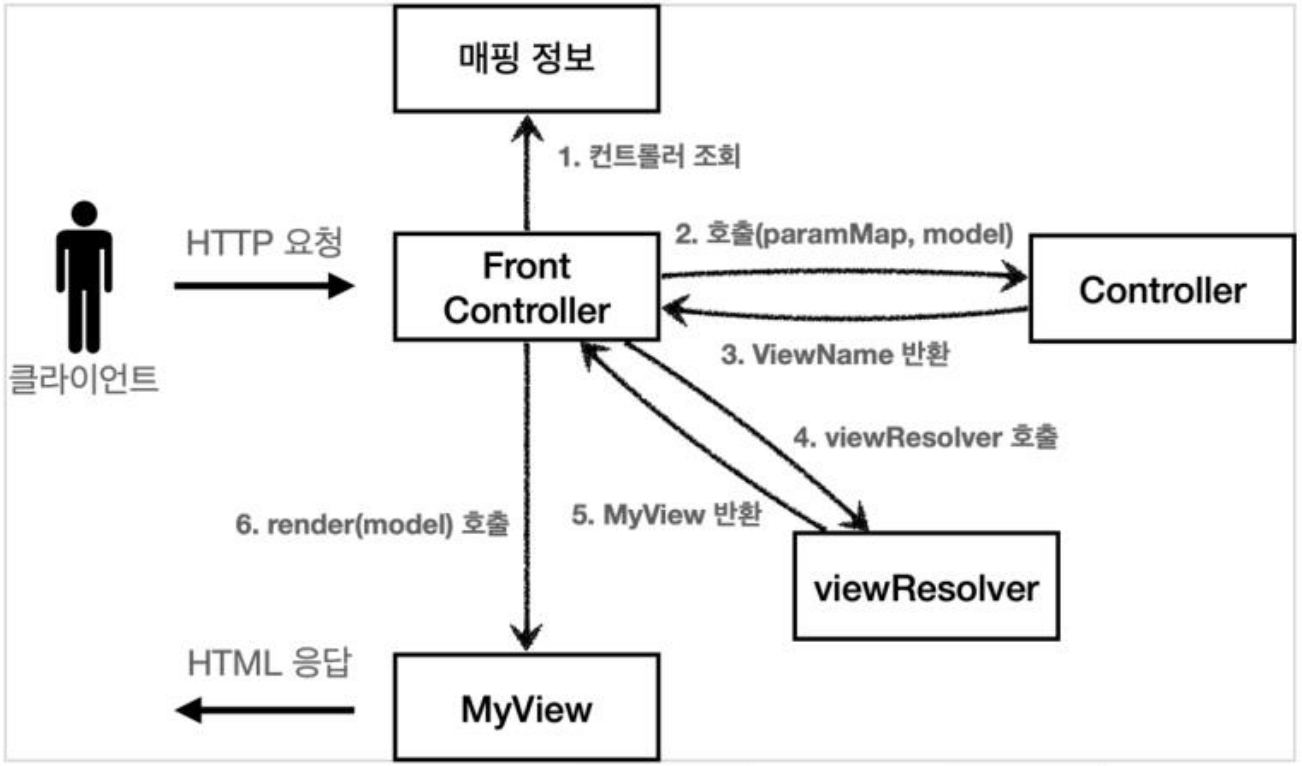
✓ ModelAndView

- 이전까진 세부 컨트롤러에서 서블릿에 종속적인 HttpServletRequest를 사용했다.
- 그리고 Model도 request.setAttribute() 메소드를 이용하여 데이터를 저장하고 뷰에 전달했다.
- 서블릿의 종속성을 제거하기 위해 데이터 저장을 위한 Model을 직접 만들고, 추가적으로 View 이름까지 전달할 수 있는 ModelAndView 클래스를 정의한다

2.2 MVC 프레임워크 만들기 – 단순하고 실용적인 세부 컨트롤러 V4

✓ 세부 컨트롤러의 문제점

- 개발자 입장에서 보면, 세부 컨트롤러는 항상 ModelAndView 객체를 생성하고 반환해야 하는 부분이 번거롭다.
- 개발자가 좀 더 쉽고, 편리하게 세부 컨트롤러를 구현할 수 있도록 변경해야 한다.



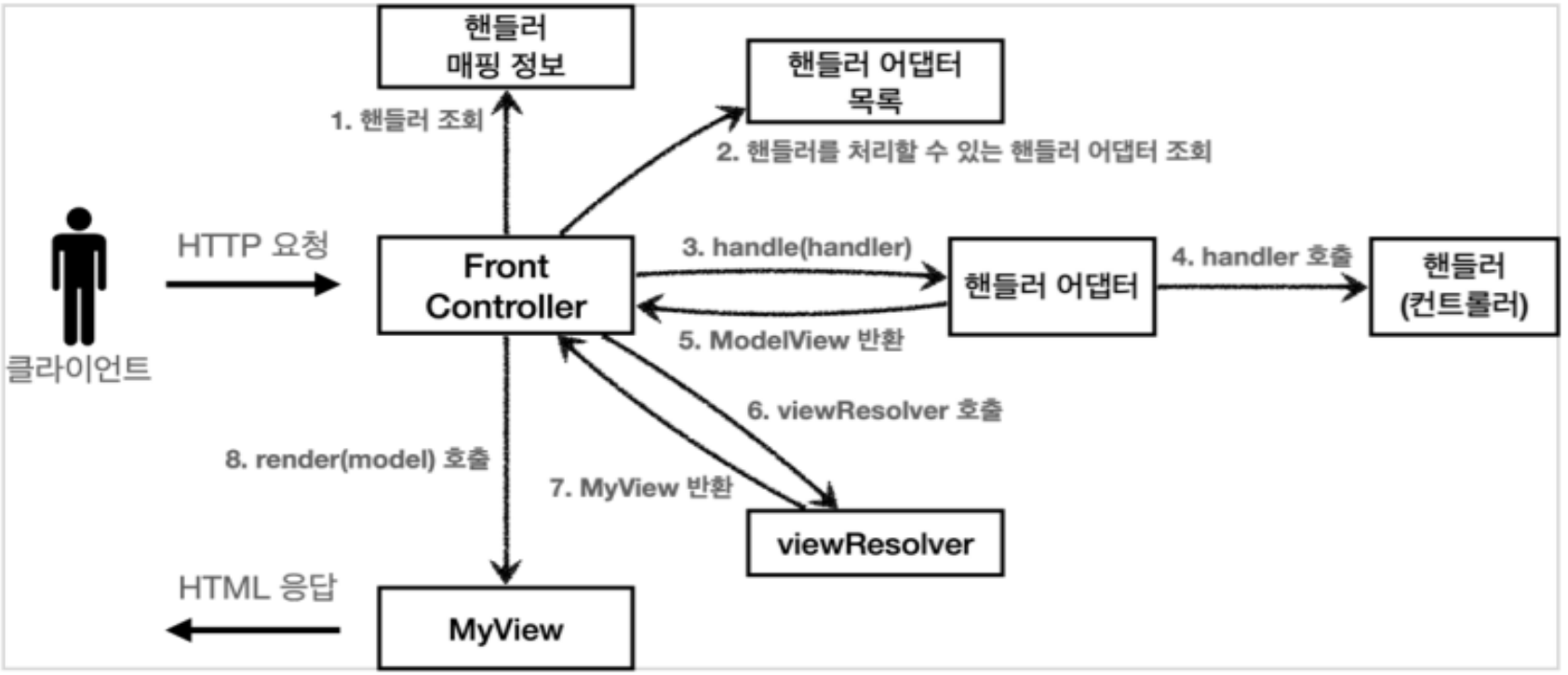
✓ 세부 컨트롤러 개선

- 세부 컨트롤러가 ModelAndView를 반환하지 않고, 논리적 뷰 이름만 반환하도록 한다.

2.2 MVC 프레임워크 만들기 – 유연한 세부 컨트롤러 V5

✓ 프론트 컨트롤러에 어댑터(Adapter) 패턴 적용

- 현재까지의 프론트 컨트롤러는 한가지 방식의 세부 컨트롤러 인터페이스만 사용할 수 있다.
- ControllerV3 , ControllerV4는 완전히 다른 인터페이스이고, 서로 호환이 불가능하다.
  - 마치 ControllerV3는110v이고, ControllerV4는 220v 전기 콘센트 같다.
- 어댑터 패턴을 적용하면 프론트 컨트롤러가 다양한 방식의 세부 컨트롤러를 처리할 수 있다.



2.2 MVC 프레임워크 만들기 – 유연한 세부 컨트롤러 V5

✓ **핸들러 어댑터**

- 프론트 컨트롤러와 세부 컨트롤러 중간에 어댑터 역할을 하는 클래스를 추가한다.
- 어댑터 역할을 해주는 덕분에 다양한 종류의 컨트롤러를 호출할 수 있다.

✓ **핸들러**

- 세부 컨트롤러의 이름을 더 넓은 범위의 이름인 핸들러로 변경한다.
- 컨트롤러의 개념 뿐만 아니라 어떠한 것이든 해당하는 종류의 어댑터만 있으면 다 처리할 수 있기 때문이다.

End of Document

✓ Q&A

감사합니다...