



## **JSP 9강 – 모델2 기반의 MVC 패턴**

---

**양 명 속**

**[[now4ever7@gmail.com](mailto:now4ever7@gmail.com)]**



# 목차

---

- 모델2 와 MVC 패턴의 개요
- 서블릿에 사용자의 요청을 명령어로 전달 - 커맨드 패턴

1. 모든요청이 하나의 서블릿으로 가게해라-> web.xml 설정
2. 해당 서블릿의 매핑파일의 정보를 보고 담당자에게 넘겨줌(ex. 특정 명령어 클래스로 넘기는 등)

1. init에서 매핑파일 읽어서 변수에 저장하기
2. 담당자 지정 클래스에서 변수 읽어오기



# 모델2 와 **MVC** 패턴의 개요

---



# 모델1 구조 vs 모델2 구조

---

- jsp 기반 웹 어플리케이션의 구조

- 모델1 구조

- 웹 브라우저의 요청(request)을 받아들이고, 웹 브라우저에 응답(response)해 주는 처리에 대해 **jsp 페이지 단독으로 처리**하는 구조

- 모델2 구조

- 웹 브라우저의 요청을 하나의 서블릿이 받게 됨
    - 서블릿은 브라우저의 요청을 알맞게 처리한 후 그 결과를 보여줄 jsp 페이지로 포워딩함
    - 포워딩을 통해서 요청 흐름을 받은 jsp 페이지는 결과 화면을 클라이언트에 전송함
    - => 서블릿이 비즈니스 로직 부분을 처리하게 됨



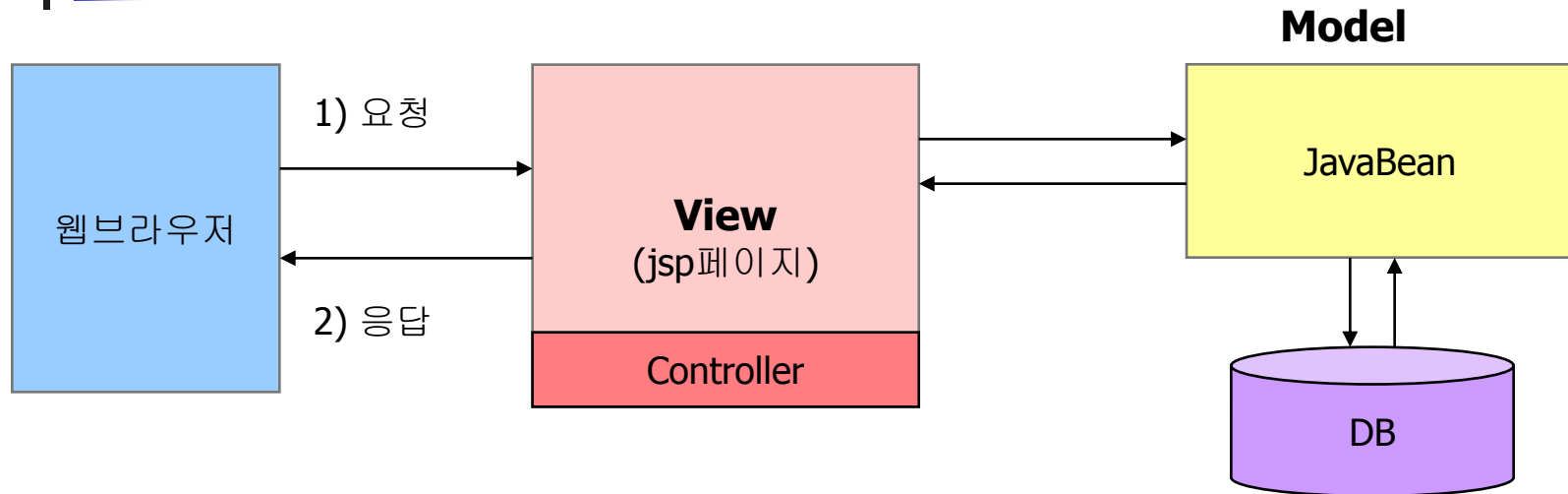
# 모델2 구조의 특징

---

- 모델2 구조의 특징
  - 브라우저의 모든 요청이 단일 진입점, 즉 하나의 서블릿에서 처리된다는 것
  - 하나의 서블릿이 웹 브라우저의 모든 요청을 받기 때문에, 서블릿은 브라우저의 요청을 구분할 수 있는 방법을 필요로 함
  - 서블릿은 브라우저의 요청을 처리한 후 브라우저에 보여줄 jsp 페이지를 선택하게 됨
  - => 모델 2 구조의 이러한 특징 때문에, MVC 패턴에 기반을 두어 웹 어플리케이션을 구현할 때는 모델 2 구조를 주로 사용함

# 모델1 구조

모델1 구조의 요청/응답 처리 방식



- 모델1 구조 - 뷰와 컨트롤러가 같은 jsp 페이지 안에서 실행되어짐
  - jsp 페이지가 뷰와 컨트롤러의 역할을 같이 하므로 모든 사용자 요청의 진입점이 요청되는 jsp 페이지가 됨
- jsp 페이지에서 브라우저가 요청한 것들을 처리 =>jsp페이지에 **비즈니스 로직**을 처리하기 위한 코드와 브라우저에 **결과를 보여줄 출력관련 코드**가 섞인다는 것(의존적)
- 모델 1 구조는 간단한 웹 어플리케이션을 구축할 때 적당.
- 중대형 프로젝트에서는 비즈니스 로직과 뷰 사이의 구분의 미비로 인한 개발자와 디자이너의 작업의 분리가 어려운 문제가 종종 발생함

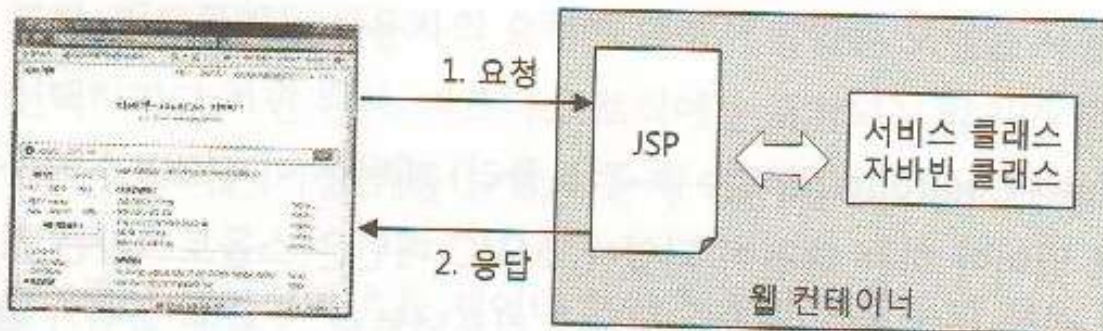
# 모델1 구조

## ■ 장점

- 단순한 페이지 흐름으로 인해 개발 기간이 단축됨
- MVC 구조에 대한 추가적인 교육이 필요 없고, 개발팀의 팀원의 수준이 높지 않아도 됨
- 중소형 프로젝트에 적합

## ■ 단점

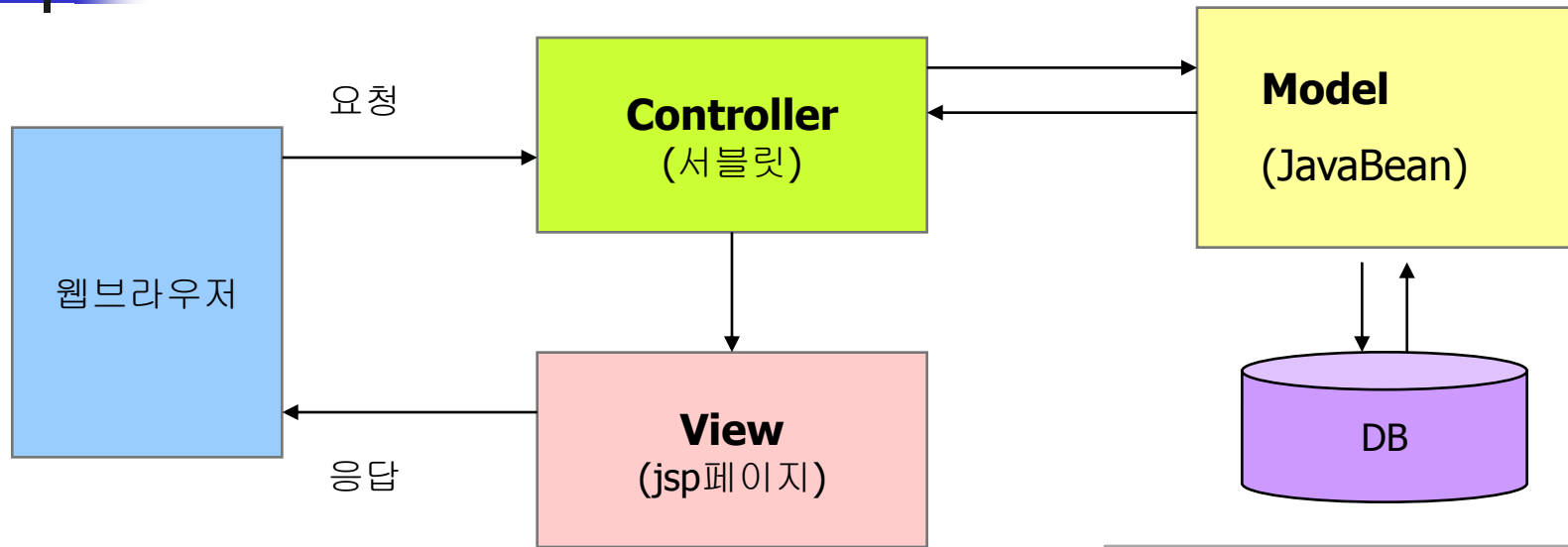
- 웹 어플리케이션이 복잡해질수록 유지 보수가 어려움
- 디자이너와 개발자 간의 원활한 의사 소통이 필요



Presentation - Controller - Service - Dao - db

View - Controller - Model - db

# 모델2 구조



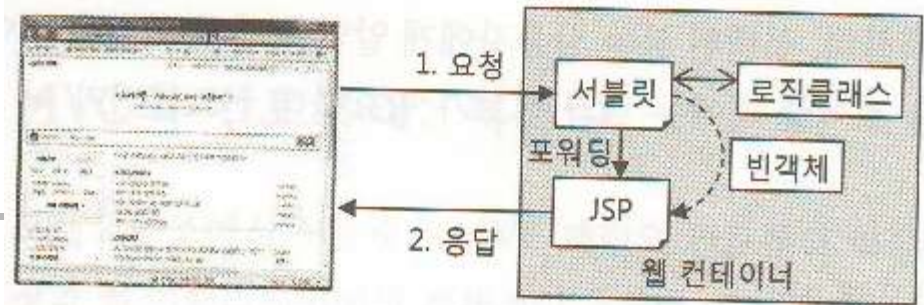
모델2 구조의 요청/응답 처리 방식

## ■ 모델 2 구조

- 요청 처리, 데이터 접근, 비즈니스 로직을 포함하고 있는 컨트롤 컴포넌트와 뷰 컴포넌트가 엄격히 구분되어 있음
- 뷰는 어떠한 처리 로직도 포함하고 있지 않음
- 사용자 요청의 진입점은 컨트롤러의 역할을 하는 서블릿이 담당하고 모든 흐름을 통제함
- 복잡한 중대형 규모의 프로젝트에 적합



# 모델 2 구조



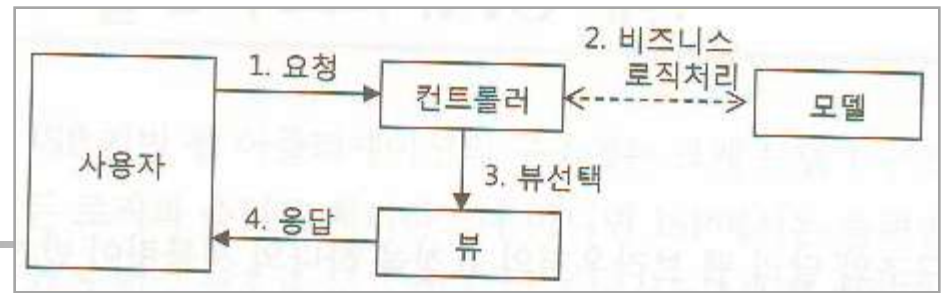
## ■ 장점

- 비즈니스 로직과 뷰의 분리로 인해 어플리케이션이 명료해지며 유지 보수와 확장이 용이(재사용 용이)
- 개발자와 디자이너의 작업이 분리되어져 역할과 책임 구분이 명확

## ■ 단점

- 개발 초기에 구조 설계를 위한 시간이 많이 소요되므로 개발 기간이 증가함
- MVC 구조에 대한 개발자들의 이해가 필요해서 개발팀의 팀원의 높은 수준이 요구됨
- 웹 어플리케이션을 개발할 때 모델1과 모델2 구조 중 어떤 것을 선택해야 하는가
  - 개발하려는 어플리케이션의 복잡도(규모), 유지보수의 빈도, 어플리케이션 컴포넌트의 재 사용성, 팀원의 수와 수준에 따라 결정해야 함
  - 웹 어플리케이션의 복잡도가 적고, 유지 보수가 빈번하지 않다면 모델1구조로 개발
  - 웹 어플리케이션의 복잡도가 크고, 유지 보수가 빈번하게 발생한다면 모델2 구조 사용

# MVC 패턴



- MVC 패턴(Model-View-Controller pattern)
  - 전통적인 GUI 기반의 어플리케이션을 구현하기 위한 디자인 패턴
  - 사용자의 입력을 받아서, 그 입력 혹은 이벤트에 대한 처리를 하고, 그 결과를 다시 사용자에게 표시하기 위한 최적화된 설계를 제시함
  - jsp 에서 모델 2 구조가 나오면서 웹 어플리케이션 개발 영역에서도 보편적으로 사용
- MVC 패턴의 구조
  - 뷰(View) – 화면에 내용을 표출하는 역할을 담당
    - 데이터가 어떻게 생성되고, 어디서 왔는지에 전혀 관여하지 않음
    - 단지 정보를 보여주는 역할만을 담당
    - JSP 기반 웹 어플리케이션에서는 jsp 페이지가 뷰에 해당



# MVC 패턴

---

- 모델(Model) – 로직을 가지는 부분
  - DB와의 연동을 통해서 데이터를 가져와 어떤 작업을 처리하거나 처리한 작업의 결과를 데이터로서 DB에 저장하는 일을 함
  - 모델은 어플리케이션의 수행에 필요한 데이터를 모델링하고 비즈니스 로직을 처리함
  - 데이터를 생성, 저장, 처리하는 역할만을 담당
  - 자바빈이 모델에 해당(비즈니스 로직 처리 클래스)
- 컨트롤러(Controller)
  - 어플리케이션의 흐름을 제어하는 것
  - 뷰와 모델사이에서 이들의 흐름을 제어함
  - 컨트롤러는 사용자의 요청을 받아서 모델에 넘겨주고, 모델이 처리한 작업의 결과를 뷰에 보내주는 역할을 함
  - 서블릿을 컨트롤러로 사용



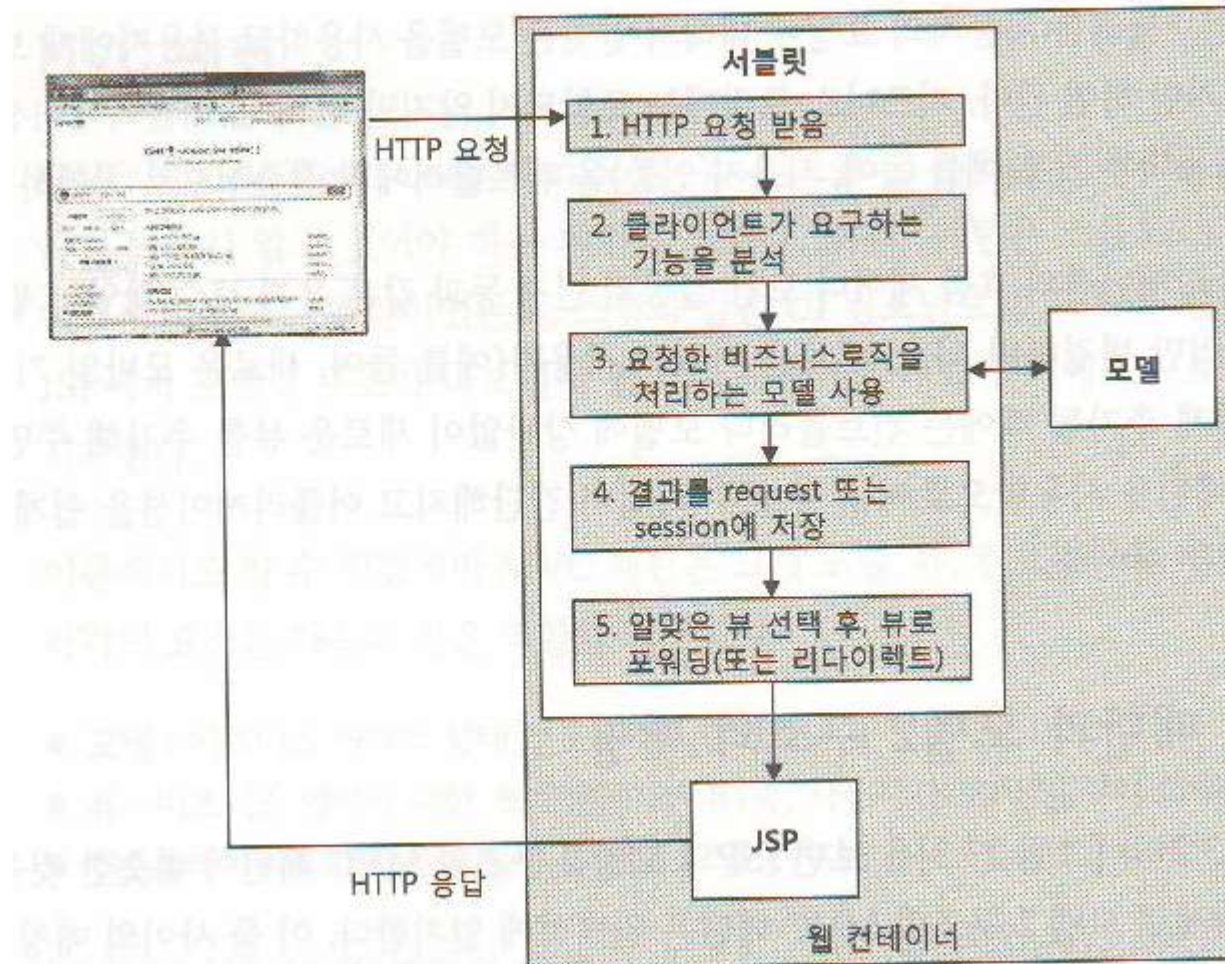
# MVC 패턴

---

- 모델 2 구조와 MVC 구조가 일치
  - 뷰 - jsp페이지 또는 서블릿
  - 모델 - 자바빈, 자바 객체 또는 EJB의 entityBean
  - 컨트롤러 - 서블릿, jsp 페이지 또는 EJB의 sessionBean
    - 컨트롤러에 서블릿을 많이 사용
    - 서블릿이 사용자의 요청을 받아서 비즈니스 로직을 처리하는 모델을 통해 수행 결과를 받아와 해당 jsp 페이지에 결과를 보내어 뷰가 사용자에게 응답을 보내는 구조

# MVC 패턴

- 컨트롤러 서블릿의 내부 동작 방식





# MVC 패턴

- 1. 컨트롤러 : 서블릿(Servlet)

- (1) 웹 브라우저의 요청을 받는다.

- 서블릿의 서비스 메서드인 doGet() 또는 doPost() 메서드가 사용자의 요청을 받는다.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
public void doPost(HttpServletRequest request,
                  HttpServletResponse response)
```

- (2) 웹 브라우저가 요구하는 작업을 분석한다.

```
String type=request.getParameter("type");
```

- (3) 요청한 작업을 처리하기 위해서 비즈니스 로직을 처리하는 모델(JavaBean)을 사용



# MVC 패턴

```
Object result = null;  
if(type.equals("a"){  
    //처리  
}else if(type.equals("b"){  
    //처리  
}
```

- (4) 처리 결과를 request 의 속성에 저장

```
request.setAttribute("result", result);
```

- (5) 적당한 뷰(jsp 페이지)를 선택 후 해당 뷰로 포워딩

```
RequestDispatcher dispatcher = request.getRequestDispatcher("a.jsp");  
dispatcher.forward(request, response);
```

포워딩(forward) 처리를 하면 현재 요청(request)을 끝지 않고, a.jsp로 요청(request)이 전달됨



# MVC 패턴

---

- 컨테이너는 자신이 관리하는 컴포넌트들끼리 서로 호출할 수 있는 기능인 "요청처리 부탁(Request dispatching)" 메커니즘을 제공함
  - 이 기능을 이용해서 서블릿은 모델로부터 받은 정보를 Request 객체 안에 저장하고, jsp에 요청을 처리해줄 것을 부탁(dispatch)할 수 있음
- RequestDispatcher 클래스
  - javax.servlet 패키지에 있으며 클라이언트로부터 요청(request)을 받고 그것을 서버상의 어떤 리소스(Servlet, HTML, jsp page)로 보내는 작업을 할 때 사용됨
  - forward(request, response) 메서드 - 서블릿에서 다른 리소스(Servlet, HTML, jsp page)로 요청을 보냄. 이 때 다른 리소스와 request, response 객체를 공유함





# MVC 패턴

---

- 2. 뷰(View) : JSP 페이지
  - 비즈니스 로직을 가지고 있지 않은 점을 제외하고는 일반적인 jsp 페이지와 다를 바 없음
  - 서블릿에서 `dispatcher.forward(request, response)`로 해당 jsp 페이지와 `request`, `response` 객체를 공유한 경우, 해당 jsp 페이지에서 `request.getAttribute("result")`를 사용해서 객체 결과를 화면에 표출함
    - 이때 jsp 페이지의 `request`는 컨트롤러인 서블릿과 같은 객체로 공유되어짐



# MVC 패턴

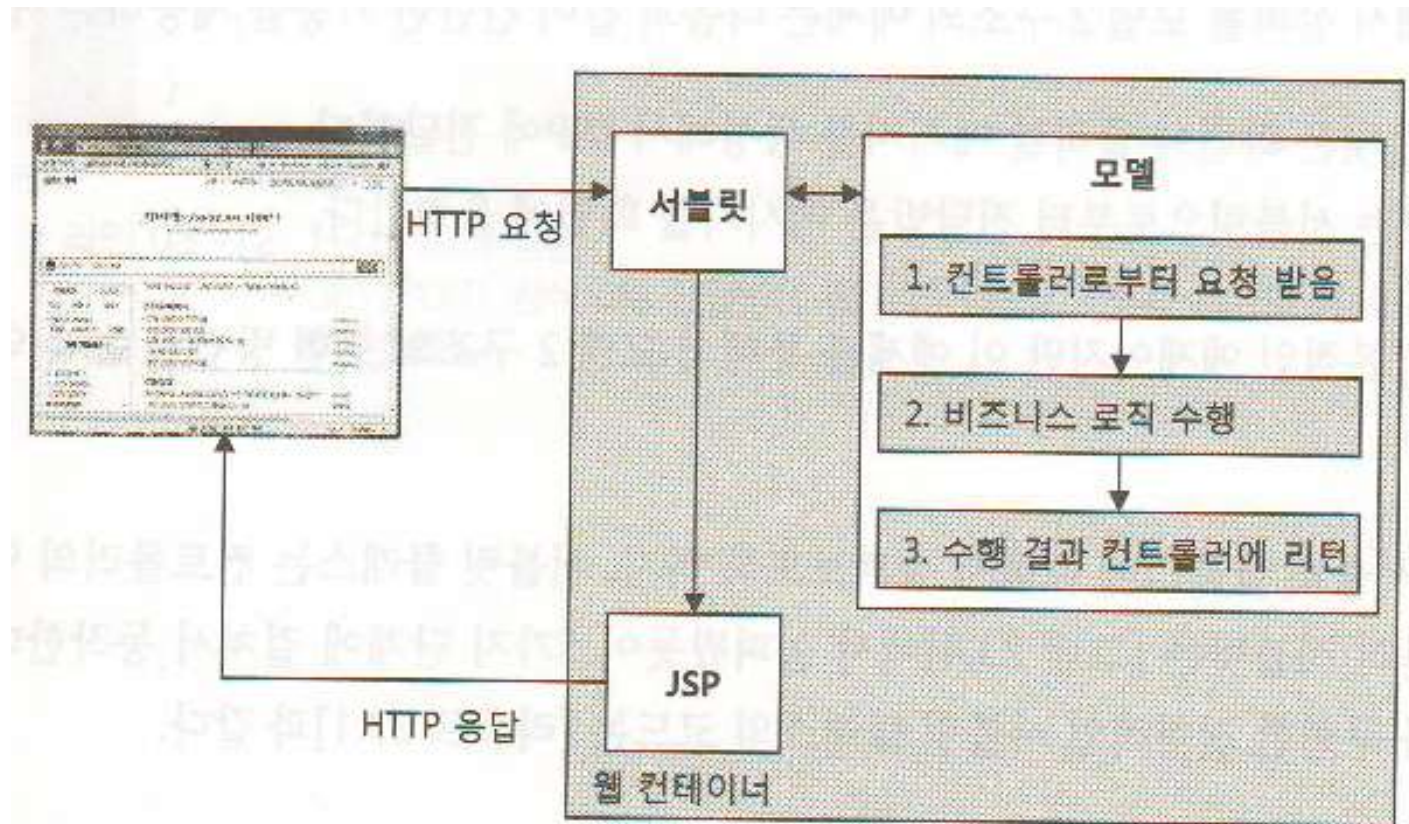
---

## ■ 3. 모델(Model) : 자바빈

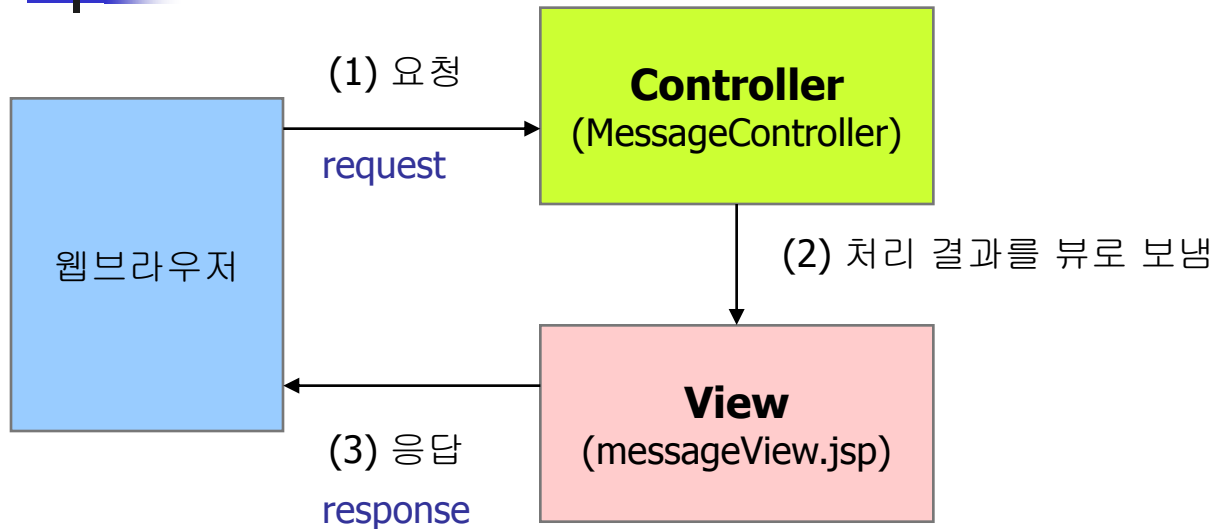
- jsp 페이지에서 비즈니스 로직의 처리를 요청 받아서 처리했던 것과 달라지는 것이 없음
  - 다만 요청을 하는 주체가 jsp 페이지에서 컨트롤러인 서블릿으로 바뀐 것 뿐임
  - 대부분 모델은 평이한 일반 자바 클래스(**POJO:Plain Old Java Object**)로 코딩함
- 
- (1) 컨트롤러의 요청을 받는다.
  - (2) 비즈니스 로직을 처리한다.
  - (3) 처리한 비즈니스 로직의 결과를 컨트롤러로 반환함

# MVC 패턴

- 컨트롤러 역할을 하는 서블릿과 모델간의 통신



# 예제-간단한 컨트롤러의 사용



- 컨트롤러인 MessageController 서블릿이 사용자의 요청을 받고 그 요청을 처리한 후 결과를 뷰인 messageView.jsp 페이지로 보내서 사용자의 요청에 응답함



# 예제-컨트롤러인 MessageController.java

```
package com.tips.controller;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/MessageController")
public class MessageController extends HttpServlet {
    public void doGet(HttpServletRequest request, //1단계 : 사용자의 요청을 받는 서비스 메소드
        HttpServletResponse response) throws ServletException, IOException {
        requestPro(request, response);
    }
    public void doPost(HttpServletRequest request, //1단계 : 사용자의 요청을 받는 서비스 메소드
        HttpServletResponse response) throws ServletException, IOException {
        requestPro(request, response);
    }
    //사용자의 요청을 분석해서 요청에 따라 작업을 처리한 후 결과를 뷰인 jsp 페이지로 보냄
    private void requestPro(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        String message = request.getParameter("message");//2단계 : 사용자의 요청분석
        Object result = null;
```

http://localhost:9090/mymvc/MessageController?message=name

# 예제

- include 지시자, 액션 태그
  - forward 액션태그, **RequestDispatcher**
  - **web.xml**
- => 절대참조시 컨텍스트 경로가 자동으로 포함됨

```
if (message == null || message.equals("base")) { //3단계 : 사용자의 요청에 따른 작업처리
    result = "안녕하세요";
} else if (message.equals("name")) {
    result = "홍길동 입니다.";
} else {
    result = "타입이 맞지 않습니다.";
}
```

request.setAttribute("result", result); //4단계 : request의 속성에 처리결과 저장

// 5단계 : RequestDispatcher를 사용하여 해당 뷰로 포워딩

```
RequestDispatcher dispatcher = request.getRequestDispatcher("/tips/messageView.jsp");
//RequestDispatcher 를 사용하여 해당 뷰로 포워딩하기 위해서 RequestDispatcher 객체를 생성하
는데, 이때 포워드되는 페이지는 "/tips/messageView.jsp"
```

//서블릿과 경로가 다르므로 웹 어플리케이션을 기준으로 하여 절대 경로를 사용

```
dispatcher.forward(request, response); //해당 뷰로 포워딩하는 부분
```

```
}
```

```
}
```



# 예제-컨트롤러가 응답 결과를 보낼 messageView.jsp

```
<%@ page contentType = "text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
<title>간단한 컨트롤러(Controller)의 사용 예제</title>
</head>
<body>
<h2>View 페이지</h2>
결과 값:
<c:set var="result" value="${requestScope.result}" /> //생략하고 ${result}만 사용해도 됨
<c:out value="${result}" />
<hr>
<%
    String res = (String)request.getAttribute("result");
    out.println(" 기존 방식 이용 : "+res);
%>
</body>
</html>
```

• `dispatcher.forward(request, response)`와  
`<jsp:forward>` 액션 태그가 하는 일이 같다



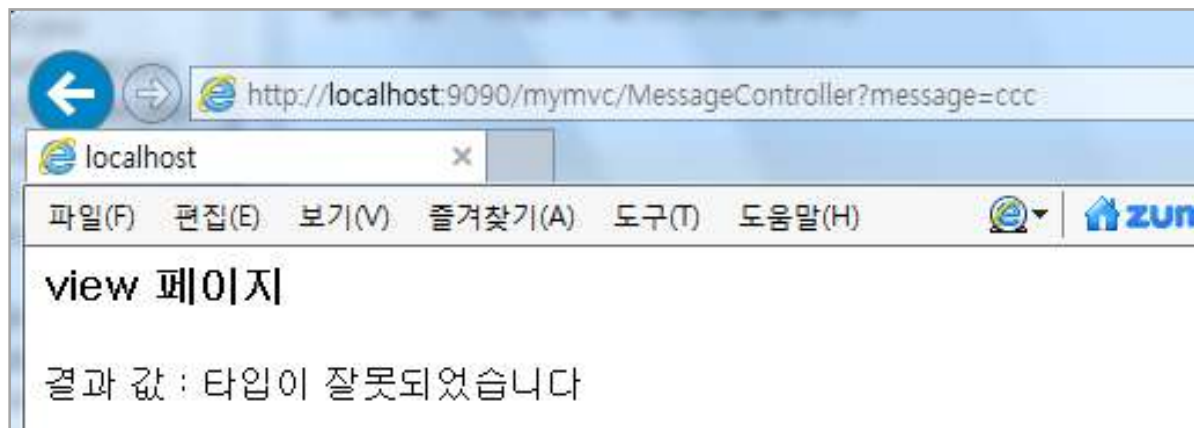
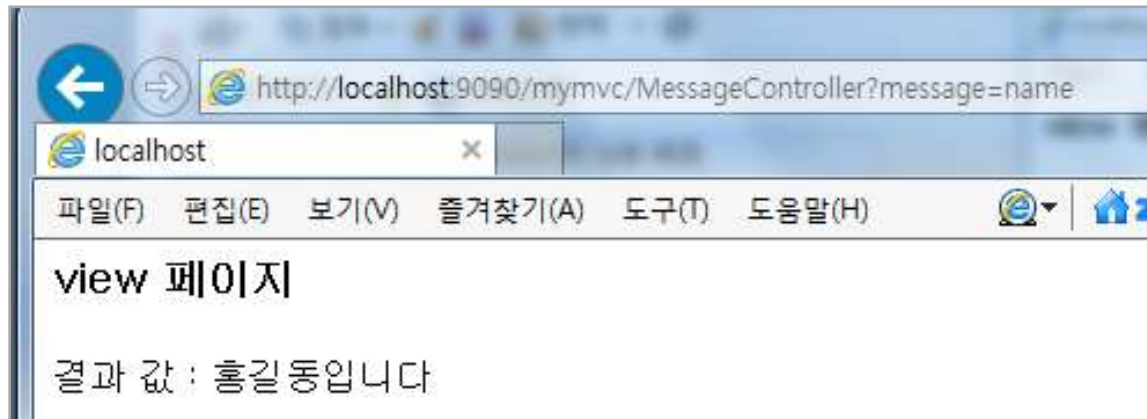
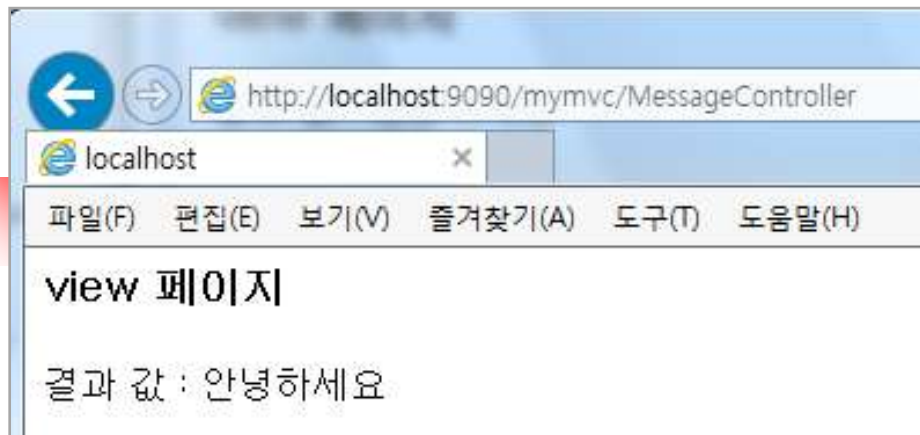
# web.xml

---

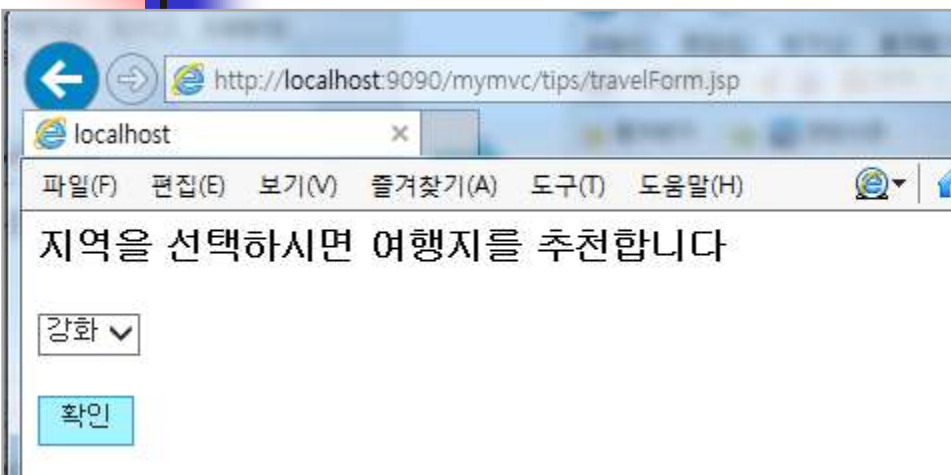
```
<servlet>
  <servlet-name>MessageController</servlet-name>
  <servlet-class>com.tips.controller.MessageController</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MessageController</servlet-name>
  <url-pattern>/MessageController</url-pattern>
</servlet-mapping>
```

@WebServlet("/MessageController") 을 지우고  
web.xml에 서블릿 매핑하고 사용할 수 있다





## 예제2



http://localhost:9090/mymvc/tips/travelForm.jsp

localhost

파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)

지역을 선택하시면 여행지를 추천합니다

강화 ▼

확인



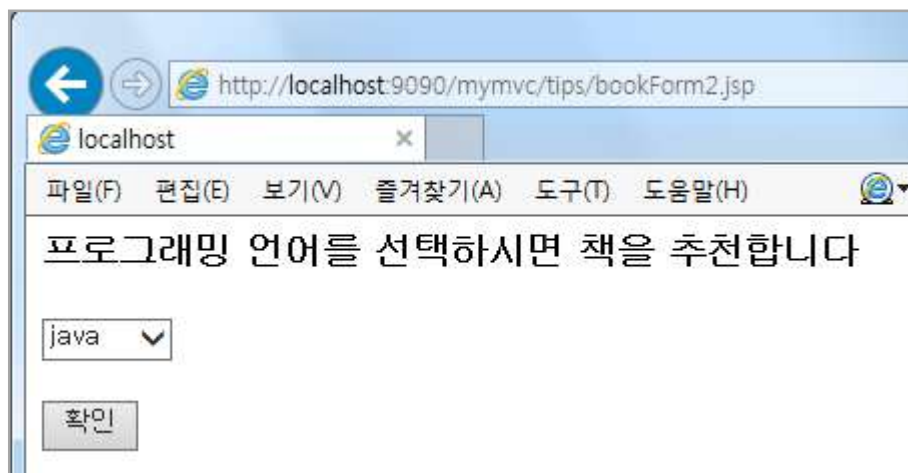
http://localhost:9090/mymvc/travel.do

localhost

파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)

View 페이지

추천하는 여행지 : 석모도



http://localhost:9090/mymvc/tips/bookForm2.jsp

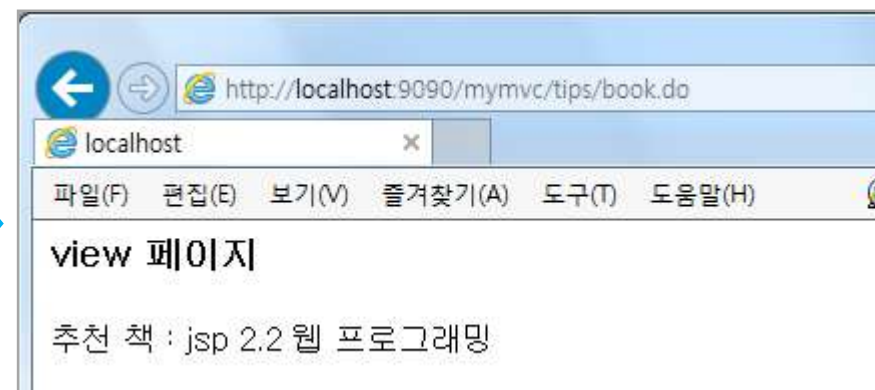
localhost

파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)

프로그래밍 언어를 선택하시면 책을 추천합니다

java ▼

확인



http://localhost:9090/mymvc/tips/book.do

localhost

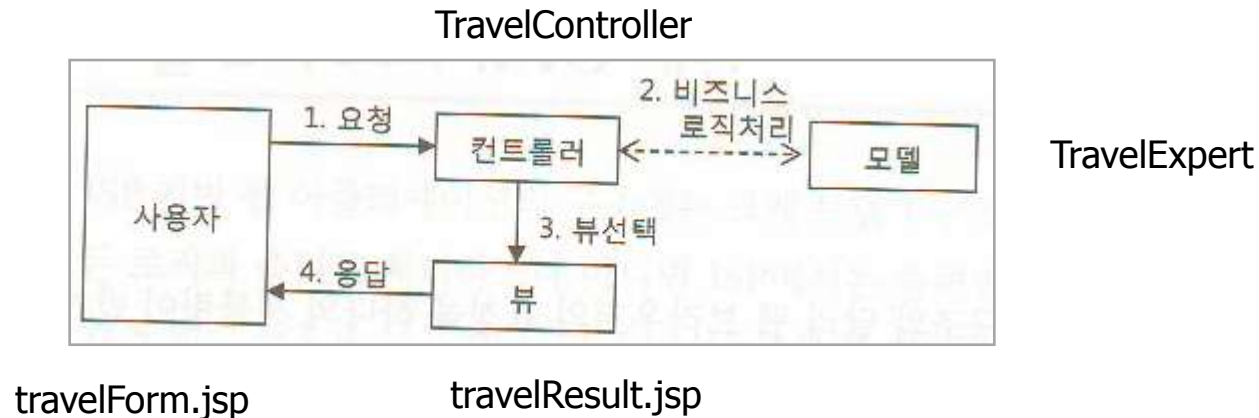
파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)

view 페이지

추천 책 : jsp 2.2 웹 프로그래밍

# 예제

- 사용자 요청                      컨트롤러                      모델                      뷰
- travelForm.jsp → TravelController.java → TravelExpert.java → travelResult.jsp (travelOk.jsp)
- bookForm.jsp → BookController.java → BookExpert.java → bookResult.jsp (bookOk.jsp)





# travelForm.jsp

---

<h3>지역을 선택하시면 좋은 여행지를 추천해드립니다</h3>

<form name="frm1" method="post"

action="<%=request.getContextPath() %>/travel.do">

<select name="city">

<option value="강화">강화</option>

<option value="강릉">강릉</option>

<option value="해남">해남</option>

<option value="거제">거제</option>

</select>

<br><br>

<input type="submit" value="확인">

</form>



# TravelController (컨트롤러)

---

```
package com.tips.controller;
```

```
public class TravelController extends HttpServlet {
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
        ServletException, IOException {
```

```
        requestProcess(request, response);
```

```
    }
```

```
    protected void doPost(HttpServletRequest request,
```

```
        HttpServletResponse response)
```

```
        throws ServletException, IOException {
```

```
        requestProcess(request, response);
```

```
    }
```



# TravelController(컨트롤러)

```
private void requestProcess(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException{
    //한글 인코딩
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");

    //[1] 요청 분석-파라미터
    String city= request.getParameter("city");

    //[2] 요청에 따른 처리 - 모델에 의뢰(db작업)
    TravelExpert tExpert = new TravelExpert();
    String result = tExpert.getTip(city);

    //[3] 처리 결과를 request 속성에 저장
    request.setAttribute("result", result);

    //[4] 뷰 페이지로 포워딩
    RequestDispatcher dispatcher
        = request.getRequestDispatcher("/tips/travelResult.jsp");
    dispatcher.forward(request, response);
}
```



# 컨트롤러를 서블릿으로

---

```
package com.tips.controller;
```

```
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.tips.model.BookExpert;
```

```
public class BookController extends HttpServlet {
    //[1] HTTP 요청받음
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
        requestProcess(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
        requestProcess(request, response);
    }
}
```



# 컨트롤러를 서블릿으로

```
public void requestProcess(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    //[2] 요청 분석
        request.setCharacterEncoding("utf-8");
        String lang = request.getParameter("lang");

    //[3] 모델을 사용하여 요청한 기능을 수행한다
        BookExpert book = new BookExpert();
        String result = book.getTip(lang);

    //[4] request나 session에 처리 결과를 저장
        request.setAttribute("result", result);

    //[5] RequestDispatcher를 사용하여 알맞은 뷰로 포워딩
        RequestDispatcher dis = request.getRequestDispatcher("/tips/bookResult.jsp");
        dis.forward(request, response);
    }
}
```





# TravelExpert.java (모델)

---

```
package com.tips.model;
```

```
public class TravelExpert {  
    public String getTip(String city){  
        String result="";  
        if(city.equals("강화")){  
            result="시월애 촬영지 석모도";  
        }else if(city.equals("강릉")){  
            result="바다와 가장 가까운 역 정동진";  
        }else if(city.equals("해남")){  
            result="한반도의 땅끝 땅끝마을";  
        }else if(city.equals("거제")){  
            result="남국의 파라다이스 외도";  
        }  
        return result;  
    }  
}
```



# travelResult.jsp (뷰)

---

<h2>여행지추천 결과</h2>

<%

String res = (String)request.getAttribute("result");

%>

좋은 여행지 : <%=res%>



# BookExpert

---

```
package com.tips.model;

public class BookExpert {
    public String getTip(String lang){
        String result="";
        if(lang.equals("java")){
            result="자바의 정석";
        }else if(lang.equals("jsp")){
            result="jsp 2.0 웹 프로그래밍";
        }else if(lang.equals("ajax")){
            result="프로개발자를 위한 Ajax 완전정복";
        }else if(lang.equals("oracle")){
            result="뇌를 자극하는 오라클 프로그래밍";
        }
        return result;
    }
}
```



# web.xml

---

- bookForm.jsp

```
<form name="frmbook" method="post" action="<%=request.getContextPath() %>/book.do">
```

- web.xml

```
<servlet>
```

```
    <servlet-name>bookController</servlet-name>
```

```
    <servlet-class>com.tips.controller.BookController</servlet-class>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
    <servlet-name>bookController</servlet-name>
```

```
    <url-pattern>/book.do</url-pattern>
```

```
</servlet-mapping>
```

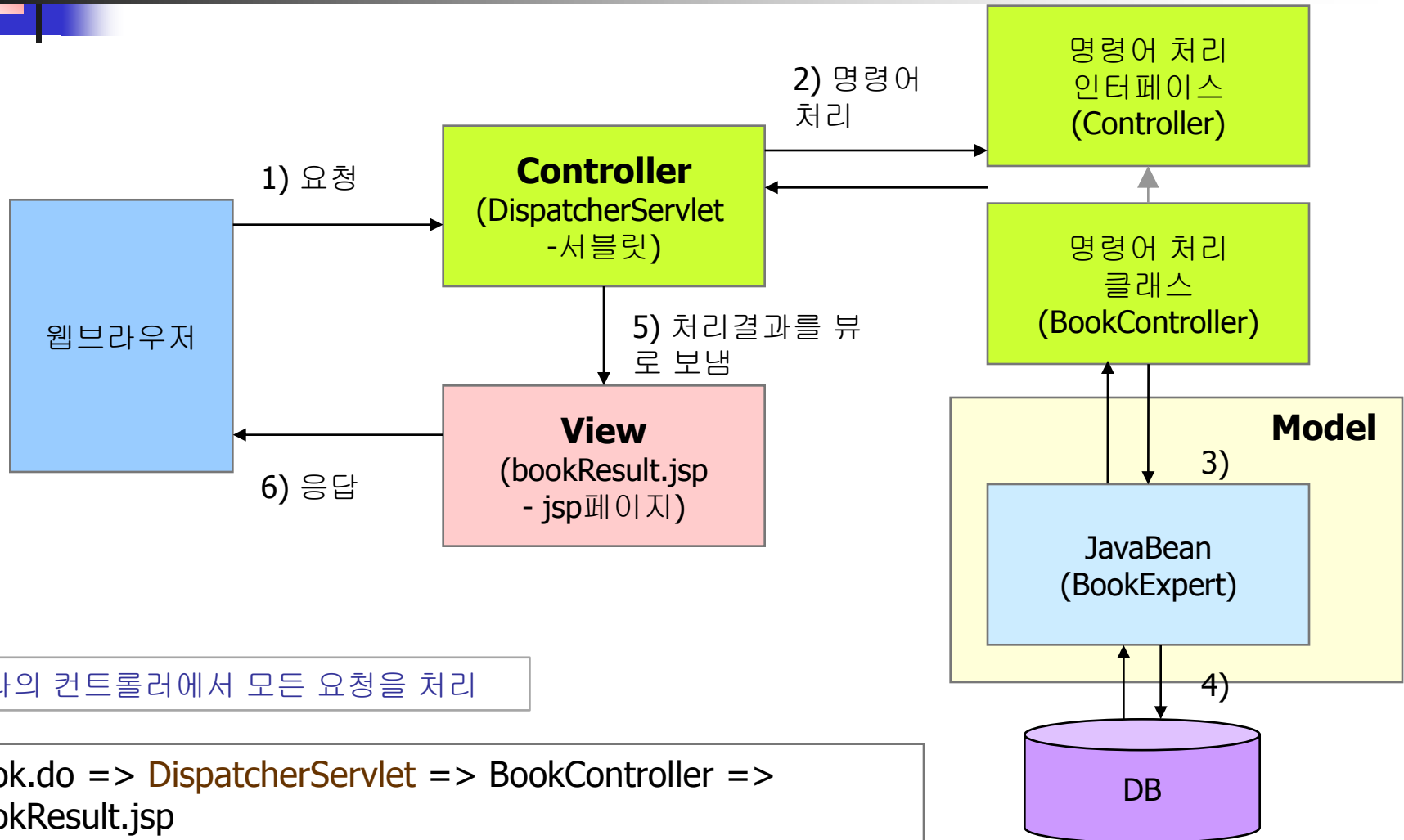


## 예제 - 보완

---

- 하나의 컨트롤러에서 모든 요청을 처리하도록 변경하자
- web.xml(DD, Deployment Descriptor)
  - 서블릿과 jsp를 어떻게 실행하느냐에 관한 많은 정보들이 들어 있다
    - URL과 서블릿을 매핑
    - 보안 역할 설정, 오류 페이지 설정, 항목 라이브러리, 초기화 구성 및 관련 정보 설정 등
  - DD - 작성한 소스코드를 바꾸지 않고, 중요한 것들을 수정할 수 있다

# 모델2기반의 MVC 패턴



하나의 컨트롤러에서 모든 요청을 처리

book.do => DispatcherServlet => BookController => bookResult.jsp



# 서블릿에 사용자의 요청을 명령어로 전달-커맨드 패턴

---

- 사용자가 어떤 요청을 했는지 판단하기 위한 가장 일반적인 방법
  - 명령어로 사용자의 요청을 전달하는 것
    - 예) 사용자가 글목록 보기 작업을 요청하면 "글목록"을 명령어로, 글삭제 작업을 요청하면 "글삭제"를 명령어로서 컨트롤러인 서블릿에 전달
    - 컨트롤러인 서블릿은 전달된 명령어를 분석해서 해당 작업을 처리하도록 한 후 결과를 뷰로 보내줌
- 컨트롤러인 서블릿에 사용자의 요청을 명령어로 전달하는 두 가지 방법
  - (1) 요청 파라미터로 명령어를 전달하는 방법
  - (2) 요청 URI 자체를 명령어로 사용하는 방법



# 서블릿에 사용자의 요청을 명령어로 전달-커맨드 패턴

- (1) 요청 파라미터로 명령어를 전달하는 방법
  - 컨트롤러인 서블릿에 요청 파라미터를 정보로 덧붙여서 사용하는 방법

```
http://localhost:9090/MessageController?message=name
```

- 컨트롤러인 MessageController 서블릿에 message라는 파라미터와 파라미터 값 name을 붙여서 사용자의 요청을 받음.
- 파라미터인 message가 가지는 값에 따라 컨트롤러인 MessageController 서블릿에서 처리되는 작업이 달라지고, 처리 결과도 달라짐

- 단점 - 간편하기 하지만 명령어가 파라미터로 전달되면 정보가 웹 브라우저를 통해 노출되어 악의적으로 사이트에 접근할 수 있는 빌미를 제공





# 서블릿에 사용자의 요청을 명령어로 전달-커맨드 패턴

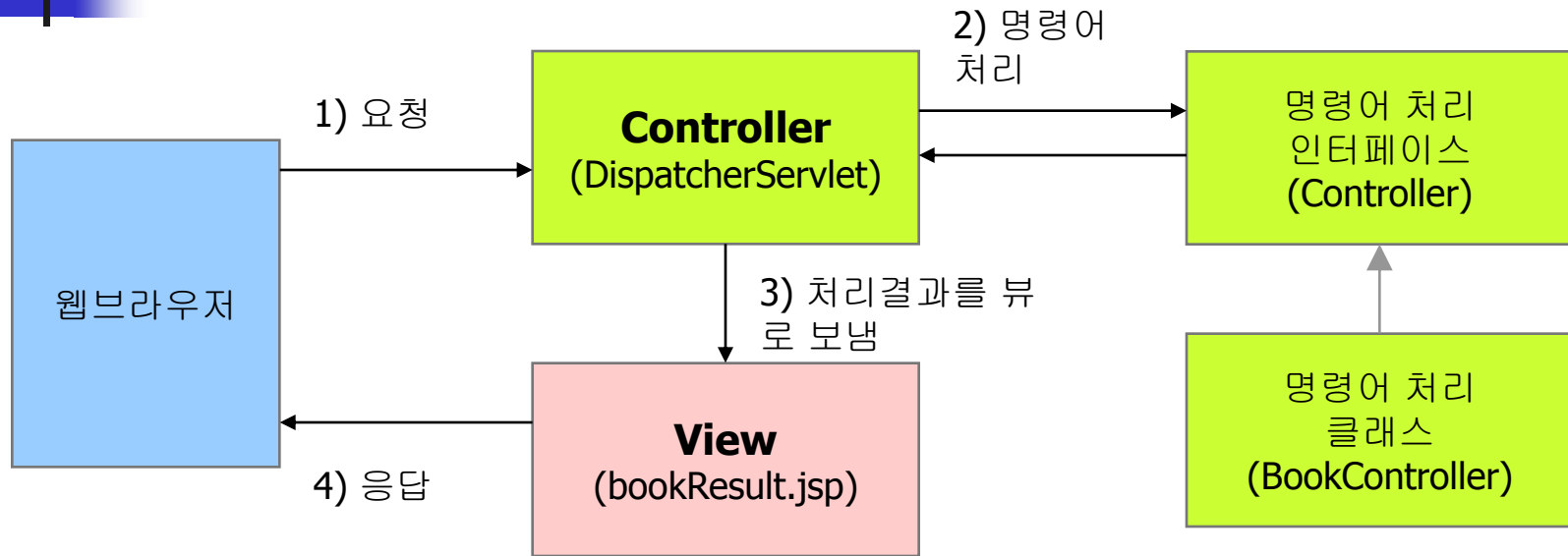
- (2) 요청 URI 자체를 명령어로 사용하는 방법
  - 사용자가 요청한 URI 자체를 명령어로 사용하는 방법

```
http://localhost:9090/mymvc/tips/book.do
```

- URI 자체를 명령어로 인식하도록 함
- **/tips/book.do** 만을 명령어로 사용
- 요청한 URI 를 읽어서 웹 어플리케이션의 위치 값을 알아낸 후 URI에서 웹 어플리케이션의 다음 문자열부터 명령어로 인식하게 하는 방법

- 장점
  - 요청되는 URI가 실제 페이지가 아니고 명령어이므로 악의적인 명령어로부터 사이트가 보호됨
  - 요청되는 URL이 좀 더 자연스러워짐

# 요청 URI 자체를 명령어로 사용하는 방법



- 컨트롤러인 DispatcherServlet 서블릿이 사용자의 요청 자체를 URI로 전달 받아서, 그 요청 URI를 자체 명령어로 사용하는 것에 대한 처리를 한다.
- 명령어 컨트롤러인 DispatcherServlet 서블릿에 사용자의 요청 자체를 URI로 전달받아서 명령에 해당하는 처리를 명령어 처리 클래스에 일임해서 처리하게 하고, 그 결과를 전달받아서 뷰인 bookResult.jsp 페이지로 보내서 사용자의 요청에 응답함

# 명령어와 명령어 처리 클래스를 매핑한 정보 파일 작성

- 1. 명령어와 명령어처리 클래스를 매핑한 정보파일인 Command.properties 파일을 저장

- Command.properties

설정파일에서 첫글자가 #으로 시작하면 주석으로 처리

# 컨트롤러 클래스 매핑

/tips/book.do=com.tips.controller.BookController

/tips/travel.do=com.tips.controller.TravelController

- /tips/book.do - 명령어,
- com.tips.controller.BookController - 해당 명령어를 처리할 클래스명

- bookForm.jsp, travelForm.jsp 변경하기

<form name="frmbook" method="post" action="/mymvc/tips/book.do">

<form name="frmTravel" method="post" action="/mymvc/tips/travel.do">



# 명령어를 처리하는 슈퍼 인터페이스 작성

- 2. 명령어 처리 클래스의 슈퍼 인터페이스인 Controller.java 파일

```
package com.controller;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
```

```
//요청 파라미터로 명령어를 전달하는 방식의 슈퍼 인터페이스
```

```
public interface Controller {
```

```
    public String requestProcess(HttpServletRequest request,
```

```
        HttpServletResponse response)throws Throwable;
```

```
}
```



# 명령어를 처리하는 슈퍼 인터페이스

- Controller 인터페이스는 서블릿으로부터 명령어의 처리를 지시받아 해당 명령에 대한 작업을 처리하고 작업 결과를 서블릿으로 리턴함
  - Controller는 인터페이스이므로 실제로 작업을 처리하는 것은 이 인터페이스를 구현하는 클래스가 수행
- 슈퍼 인터페이스를 만들고 그것을 구현하는 클래스를 만드는 이유
  - 이렇게 구현하면 좀 더 유연한 형태의 객체가 생성되고 컨트롤하기도 쉬워지기 때문

# Controller 인터페이스를 구현하는 명령어 처리 클래스

- 3. Controller 인터페이스를 구현하는 명령어 처리 클래스인 BookController.java 파일

```
package com.tips.controller;
```

```
public class BookController implements Controller{  
    public String requestProcess(HttpServletRequest request,  
        HttpServletResponse response)throws Throwable {
```

```
        request.setCharacterEncoding("utf-8");  
        String lang = request.getParameter("lang");
```

```
        BookExpert book = new BookExpert();  
        String result = book.getTip(lang);
```

```
        request.setAttribute("result", result);
```

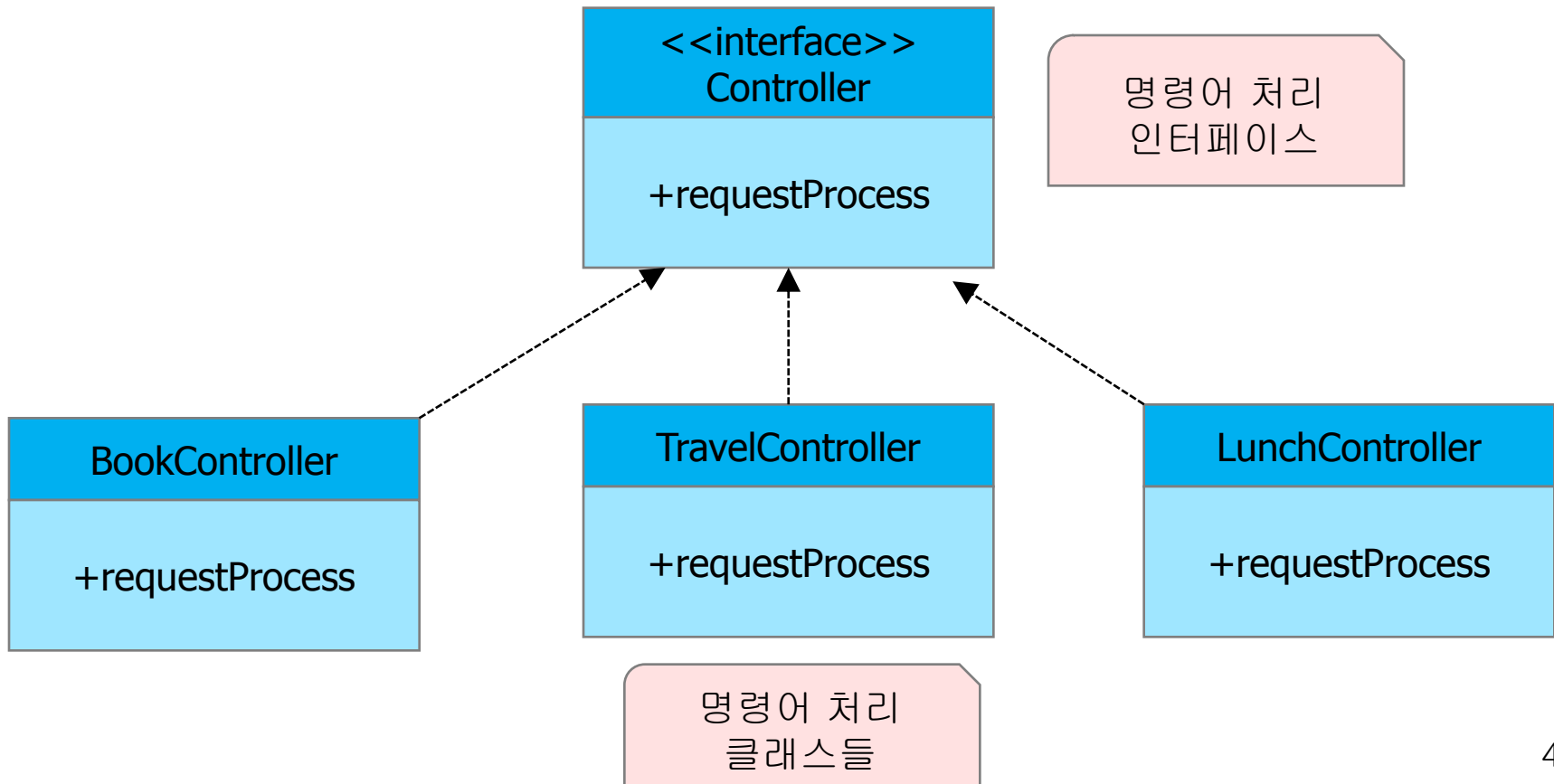
```
        return "/tips/bookResult.jsp";
```

```
    }  
}
```

- 서블릿으로부터 명령어의 처리를 지시 받는 것은 **Controller** 인터페이스이나 해당 명령에 대한 작업을 처리하고 작업 결과를 서블릿으로 리턴하는 실제 작업은 이 클래스가 수행

# 슈퍼 인터페이스인 Controller 와 명령어 처리 클래스들간의 관계

- 작업 처리에 대해 단일 진입점인 Controller 인터페이스에서 실제 작업을 처리하는 클래스로 제어가 이동하므로 제어와 관리가 편하고 확장성이 좋음
- 구현하는 클래스가 늘어나더라도 컨트롤러인 서블릿의 소스 코드를 고칠 필요가 없음



# DispatcherServlet.java

• 서블릿 라이프 사이클  
init() – service() – destroy()

```
<servlet>
  <servlet-name>DispatcherServlet</servlet-name>
  <servlet-class>com.controller.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>configFile</param-name>
    <param-value>/config/Command.properties</param-value>
  </init-param>
</servlet>
```

```
package com.controller;

public class DispatcherServlet extends HttpServlet {
    private Properties props;
    String configFile;
    String realConfigFile;
    //해당 서블릿이 요청될때 최초로 한번만 호출되는 메서드
    public void init(ServletConfig config){
        //매핑 파일을 읽어서 Properties 컬렉션에 담아 놓는다
        //web.xml에서 <init-param>의 값 읽기-CommandPro.properties 파일
        configFile
        = config.getInitParameter("configFile"); //=>/config/CommandPro.properties
        //매핑 파일의 실제 경로 구하기
        //realConfigFile = config.getServletContext().getRealPath(configFile);
        realConfigFile
        = "D:\\lecture\\jsp\\workspace\\mymvc\\WebContent\\config\\CommandPro.properties";
        props = new Properties(); //명령어와 처리클래스의 매핑정보를 저장할 Properties객체 생성(key, value 쌍)
        FileInputStream fis=null;
        try {
            fis = new FileInputStream(realConfigFile); //CommandPro.properties파일의 내용을 읽어옴
            props.load(fis); //CommandPro.properties파일의 정보를 Properties객체에 저장
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```

    }finally{
        try {
            if(fis!=null) fis.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    requestPro(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    requestPro(request, response);
}

```

//사용자의 요청이 들어올때마다 호출됨

```

private void requestPro(HttpServletRequest request,HttpServletResponse response) throws IOException {
    //매핑파일이 들어있는 Properties 컬렉션에서 사용자의 URI(/tips/book.do)에 해당하는
    //직원 Controller(BookController)를 구해서, 직원 Controller에게 일시킨다
    //(직원 Controller의 메서드 호출)
    //=> 그리고 나서, 결과를 리턴 받아서 해당 뷰 페이지로 포워딩시킨다
    //한글처리
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");
    //사용자의 URI 읽어오기 => /mymvc/tips/book.do
    String uri = request.getRequestURI();
    //Context Path 제거하기
    String contextPath = request.getContextPath(); //=>/mymvc

```

```
if(uri.indexOf(contextPath)==0){  
    uri = uri.substring(contextPath.length()); //=>/tips/book.do  
}
```

```
/*PrintWriter out = response.getWriter();  
out.print("configFile : " + configFile+"<br>");  
out.print("realConfigFile : " + realConfigFile+"<br>");  
out.print("uri : " + uri+"<br>");*/
```

```
System.out.println("WnWnconfigFile : " + configFile);  
System.out.println("realConfigFile : " + realConfigFile);  
System.out.println("uri : " + uri);
```

```
//매핑파일 : /tips/book.do=com.tips.controller.BookController2  
//key : /tips/book.do, value : com.tips.controller.BookController2  
String command = props.getProperty(uri); //BookController2
```

```
try {  
    Class commandClass = Class.forName(command);//해당 문자열을 클래스로 만든다.  
    Controller controller = (Controller)commandClass.newInstance(); //해당클래스의 객체를 생성  
    String viewPage = controller.requestProcess(request, response);  
  
    System.out.println("controller : " + command);  
    System.out.println("viewPage : " + viewPage);  
  
    //뷰페이지로 포워딩시킨다  
    RequestDispatcher dispatcher = request.getRequestDispatcher(viewPage);  
    dispatcher.forward(request, response);
```

```

    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (Throwable e) {
        e.printStackTrace();
    }
}

```

```

}

```

```

<servlet>
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>com.controller.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>configFile</param-name>
    <param-value>/config/Command.properties</param-value>
  </init-param>
</servlet>

```



# DispatcherServlet

- 웹 브라우저가 get방식으로 보내든, post 방식으로 보내든 컨트롤러는 1단계에서 일단 requestProcess() 메서드로 웹 브라우저의 요청을 전달함
- 새로운 명령어가 추가되더라도 컨트롤러 서블릿 클래스의 코드를 변경하지 않기 위해, <명령어, 명령어 핸들러 클래스>의 매핑 정보를 설정 파일에 저장해서 사용
  - command.properties 파일
  - 명령어=핸들러 클래스의 이름

```
/tips/book.do=com.tips.controller.BookController  
/tips/travel.do=com.tips.controller.TravelController
```

- 컨트롤러 서블릿은 이 설정 파일로부터 명령어에 해당하는 핸들러 클래스 인스턴스를 미리 생성해두었다가 requestProcess()메서드에서 사용

# DispatcherServlet

- 컨트롤러 서블릿에서 설정파일을 읽어오기 가장 좋은 위치는 init()메서드
  - init() - 서블릿을 생성한 후 초기화할 때 단 한번 호출되는 메서드
- 요청 URI에서 request.getContextPath()부분을 제거

```
String command = request.getRequestURI(); // =>/tips/book.do
if (command.indexOf(request.getContextPath()) == 0) {
    command = command.substring(request.getContextPath().length());
}
```

- 웹 어플리케이션 내에서의 요청 URI만을 사용하기 위함
- 예) <http://localhost:9090/mymvc/tips/bookForm.jsp>
  - => 전체 요청 URI는 /mymvc/tips/bookForm.jsp
  - 웹 어플리케이션 경로를 제외한 나머지 URI는 /tips/bookForm.jsp
- 요청 URI인 /tips/book.do를 명령어로 사용하고 뷰로 bookResult.jsp가 출력



# DispatcherServlet

---

- `init()` 메서드
  - 서블릿이 실행될 때 가장 먼저 한번만 실행되는 메서드로 자동으로 실행됨
- `init()` 메서드에서 하는 일
  - 명령어와 처리 클래스가 매핑되어 있는 매핑 파일(`Command.properties`)을 읽어서 Map 객체인 `Properties` 컬렉션에 저장함
- `config.getInitParameter("configFile")`
  - `web.xml`에서 `propertyConfig`에 해당하는 `init-param`의 값을 읽어옴
    - `<param-value>` 태그의 body인 `"Command.properties"` 을 읽어 옴
- `FileInputStream` 객체 `fis`로 `Command.properties` 파일의 내용을 읽어옴
  - 외부 데이터를 자바 클래스로 읽어오려면 입력 스트림을 사용해서 읽어옴, `FileInputStream`은 파일을 읽어올 때 사용하는 스트림 객체
- `doGet()`, `doPost()` – 서비스 메서드로 사용자의 요청을 받아서 `requestProcess()` 메서드 호출



# DispatcherServlet

- `request.getRequestURI()` – 사용자가 요청한 URI가 <http://localhost:9090/mymvc/tips/book.do> 일 경우 웹 어플리케이션부터 요청 페이지까지를 얻어냄
  - 즉, `/mymvc/tips/book.do` 가 결과로 나오고, 이것이 `command` 변수에 저장됨
- `request.getContextPath()` – 현재의 웹 어플리케이션 루트인 `/mymvc` 를 얻어냄
- `command` 변수값 : `/mymvc/tips/book.do`
- `command.indexOf(request.getContextPath())` : " `/mymvc/tips/book.do` " 에 "`/mymvc`" 가 있으면 시작 인덱스 번호 리턴 => 0을 리턴
- `request.getContextPath().length()` : "`/mymvc`"의 문자열 개수 리턴 => 6
- `command.substring(request.getContextPath().length())` : `command` 변수에서 인덱스번호 6부터 끝까지 문자열추출 => `/tips/book.do`
  - 이 부분만 명령어로 사용하기 위해서 요청 URI에서 웹 어플리케이션 루트의 문자열을 제거한 것



# java.util.Properties 컬렉션

---

- public class Properties extends  
Hashtable<Object, Object>
- void load(InputStream inStream)
  - Reads a property list (key and element pairs) from the input byte stream.
- void load(Reader reader)
- String getProperty(String key)
  - Searches for the property with the specified key in this property list.
- Object setProperty(String key, String value)





# web.xml 수정

- 5. Command.properties 파일을 컨트롤러인 DispatcherServlet 에서 읽어올 수 있도록 설정 정보를 저장할 web.xml 수정
- <web-app>태그 바로 다음에 제일 먼저 나오도록 아래 문장 추가

```
<servlet>
```

```
  <servlet-name>dispatcherServlet</servlet-name>
```

```
  <servlet-class>com.controller.DispatcherServlet</servlet-class>
```

```
  <init-param>
```

```
    <param-name>configFile</param-name>
```

```
    <param-value>/config/Command.properties</param-value>
```

```
  </init-param>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
  <servlet-name>dispatcherServlet</servlet-name>
```

```
  <url-pattern>*.do</url-pattern>
```

```
</servlet-mapping>
```

특정 확장자(.do)를 가진 요청을 DispatcherServlet 컨트롤러 서블릿이 처리하도록 지정

=> \*.do로 오는 요청은 DispatcherServlet 컨트롤러 서블릿으로 전달됨



# web.xml

- <servlet-name>태그에서 서블릿명 Controller로 기술
- <servlet-class> 태그에서 실제 서블릿의 경로 기술
- <init-param>태그 – initial parameter를 설정하는 부분으로 이 태그는 config.getInitParameter("configFile") 메소드를 사용해서 파라미터의 이름을 읽어 들여서 해당 파라미터의 값을 얻어 낼 수 있음
- <param-name> 태그 – 파라미터명을 configFile로 설정
- <param-value> 태그 – 파라미터의 값을 설정
- <servlet-mapping>태그 : <url-pattern>태그의 값인 \*.do 와 같이 사용자의 요청이 \*.do로 오는 경우<servlet-name> 태그의 값인 서블릿 DispatcherServlet이 그 요청을 받아서 처리해 주게 함



# 응답 결과 뷰 페이지

---

- 6. 컨트롤러가 응답 결과를 보낼 bookResult.jsp 페이지

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<h2>책 추천 결과</h2>
<%
String res = (String)request.getAttribute("result");
%>
좋은 책 : <%=res%>
```



# 예제

- <http://localhost:9090/mymvc/tips/book.do>
  - 실제 웹 페이지 주소가 아니라 요청을 전달할 명령어를 URI로 사용한 것.
  - 실제로 사용자의 요청을 받는 것은 컨트롤러인 DispatcherServlet

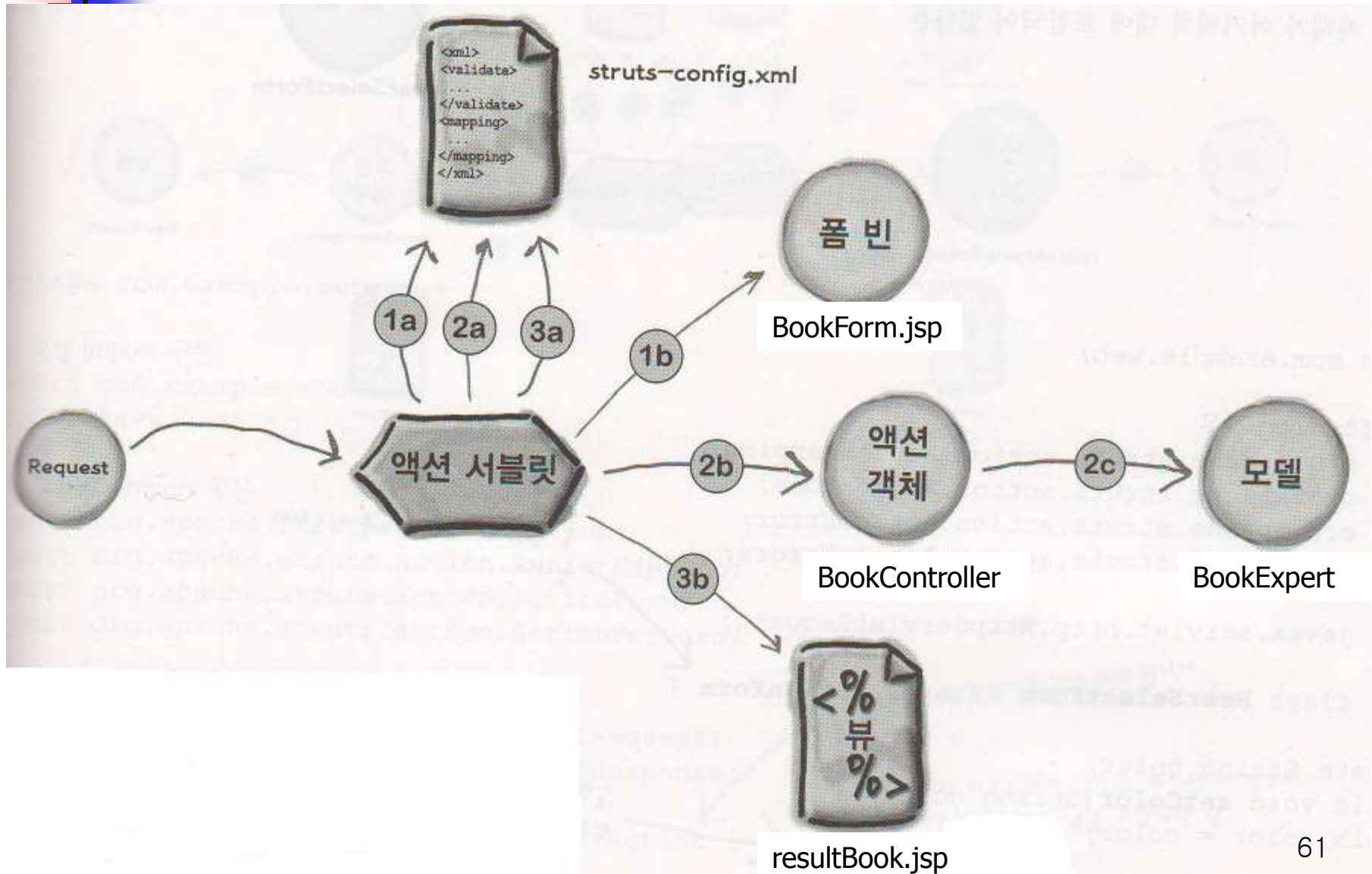
```
http://localhost:9090/mymvc/tips/bookForm2.jsp
```

```
<form name="frm1" method="post"  
action="<%=request.getContextPath() %>/tips/book.do">
```

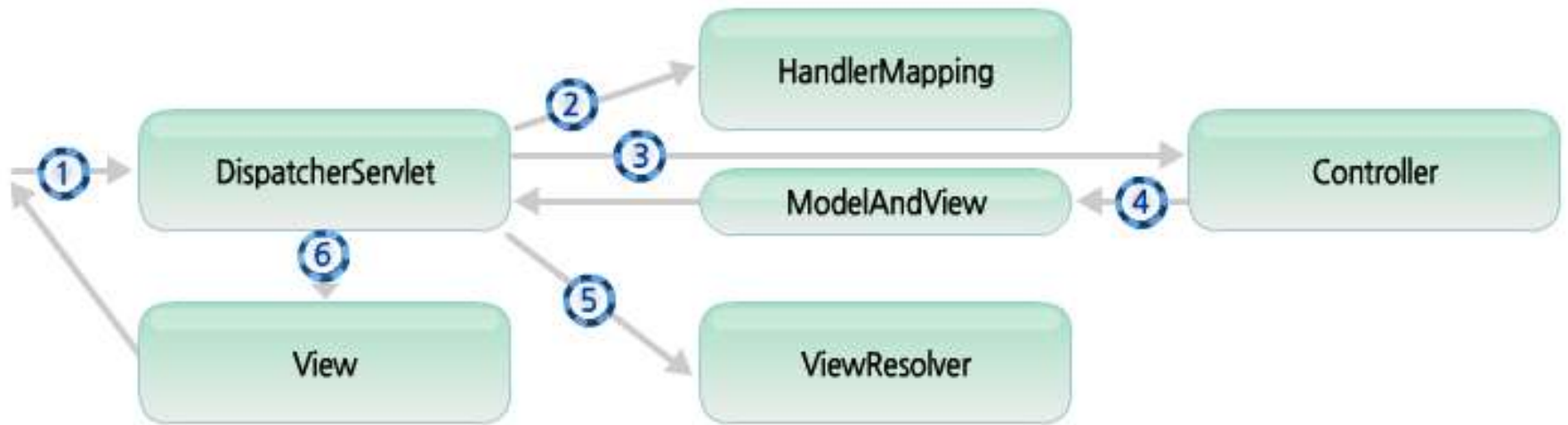
```
com.tips.controller.BookController2
```

```
/tips/bookResult.jsp 로 포워딩
```

# 스트럿츠로 재구성한 구조



# Spring MVC의 처리 흐름



- DispatcherServlet을 시작으로 정의된 HandlerMapping 에 의해 Controller를 호출하고, ViewResolver에 의해 View를 찾아 화면을 보여주는 구조임



# DispatcherServlet2

---

```
package com.controller;

public class DispatcherServlet2 extends HttpServlet {
    String configFilePath;
    Properties prop;
    FileInputStream fis;

    //명령어와 처리클래스가 매핑되어 있는 properties 파일은 Command.properties파일
    public void init(ServletConfig config) {
        String configFile = config.getInitParameter("configFile");//web.xml에서 configFile에 해당하는
        init-param 의 값을 읽어옴
        configFilePath = config.getServletContext().getRealPath(configFile); //실제 경로 구하기

        prop = new Properties();//명령어와 처리클래스의 매핑정보를 저장할 Properties객체 생성(key,
        value 쌍)
        try {
            fis = new FileInputStream(configFilePath); //Command.properties파일의 내용을 읽어옴
            prop.load(fis);//Command.properties파일의 정보를 Properties객체에 저장
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e){
            e.printStackTrace();
        } finally {
            if (fis != null) try { fis.close(); } catch (IOException ex) {}
        }
    }
}
```

```

public void doGet(//get방식의 서비스 메소드
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    requestProcess(request, response);
}

public void doPost(//post방식의 서비스 메소드
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    requestProcess(request, response);
}

//사용자의 요청을 분석해서 해당 작업을 처리
public void requestProcess(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");
    String view = null;
    Controller controller=null;
    try {
        //PrintWriter out = response.getWriter();
        //out.print("command.properties 파일 경로 : "+configFilePath + "<br>");

        //클라인트측에서 전송한 URI를 반환해 줌
        String command = request.getRequestURI(); // =>/tips/book.do
        if (command.indexOf(request.getContextPath()) == 0) {
            command = command.substring(request.getContextPath().length());
        }
    }
}

```



```
String className = prop.getProperty(command);  
Class commandClass = Class.forName(className);//해당 문자열을 클래스로 만든다.  
controller = (Controller)commandClass.newInstance();//해당클래스의 객체를 생성,  
//=>BookController
```

```
view = controller.requestProcess(request, response);
```

```
// 결과 페이지 처리
```

```
// 1) 요청(request)을 유지, 결과가 있는 경우(값을 저장) - forward
```

```
// 2) 요청(request)을 끊고 새롭게 접속, 단순 링크 - sendRedirect (js: location.href)
```

```
if(controller.isRedirect()){
```

```
    response.sendRedirect(view); //결과 페이지 Redirect
```

```
}else{
```

```
    RequestDispatcher dispatcher=request.getRequestDispatcher(view);
```

```
    dispatcher.forward(request, response); //결과 페이지 forward
```

```
}
```

```
} catch(Throwable e) {
```

```
    throw new ServletException(e);
```

```
}
```

```
}
```

```
}//class
```



# Controller.java

---

//요청 파라미터로 명령어를 전달하는 방식의 슈퍼 인터페이스

```
public interface Controller {  
    public String requestProcess(HttpServletRequest request,  
                                HttpServletResponse response)throws Throwable;
```

//요청(Request) 유지 여부

```
    public boolean isRedirect();
```

```
}
```



# BookController

---

```
public class BookController implements Controller{
    public String requestProcess(HttpServletRequest request,
        HttpServletResponse response)throws Throwable {

        //request.setCharacterEncoding("utf-8");
        String lang = request.getParameter("lang");

        BookExpert book = new BookExpert();
        String result = book.getTip(lang);

        request.setAttribute("result", result);

        return "/tips/bookResult.jsp";

    }

    public boolean isRedirect() {
        return false; //forward
    }
}
```



# Command.properties

---

```
/tips/book.do=com.tips.controller.BookController  
/tips/travel.do=com.tips.controller.TravelController
```

controller 클래스 매핑



# BookFormController2

---

```
package com.tips.controller;

public class BookFormController2 implements Controller{

    @Override
    public String requestProcess(HttpServletRequest request,
                                HttpServletResponse response) throws Throwable {

        //String viewPage = "/mymvc/tips/bookForm2.jsp";
        String viewPage = request.getContextPath()+"/tips/bookForm2.jsp";

        return viewPage;
    }

    @Override
    public boolean isRedirect() {
        return true; //redirect
    }

}

} //class
```



# Command.properties

```
/tips/bookForm.do=com.tips.controller.BookFormController2  
/tips/book.do=com.tips.controller.BookController2  
/tips/travel.do=com.tips.controller.TravelController2
```

```
http://localhost:9090/mymvc/tips/bookForm.do
```

```
com.tips.controller.BookFormController2
```

```
/mymvc/tips/bookForm2.jsp 로 forward
```

```
<form name="frm1" method="post"  
action="<%=request.getContextPath() %>/tips/book.do">
```

```
com.tips.controller.BookController2
```

```
/tips/bookResult.jsp 로 포워딩
```



# 실습

---

- 점심메뉴 추천하기
- [1] 사용자 입력 받는 Form 화면 - jsp
  - menuForm.jsp
- [2] 직원 컨트롤러(명령어 처리 클래스) 만들기
  - MenuController.java <- Controller 구현
  - 실제적인 작업을 함
- [3] 모델 만들기 - 로직 처리
  - MenuExpert.java <- Expert 구현
- [4] 결과 처리하는 View 페이지
  - menuResult.jsp
- 매핑 파일에 추가
  - /tips/menu.do => MenuController