servlet 는 http servlet 상속

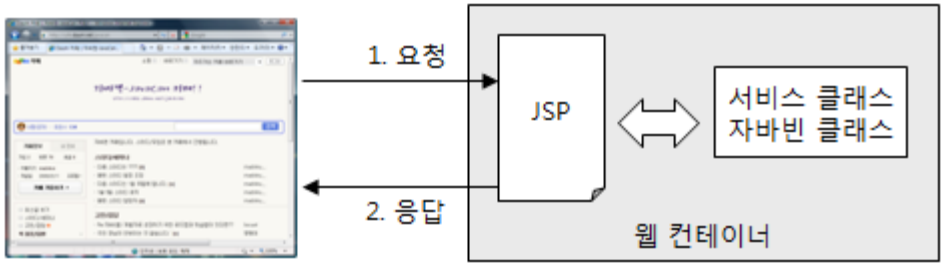
컨트롤러 클래스는
비즈니스로직을
담는 클래스.

14장 MVC

model → processing (servlet)
view → (JSP)
controller

모델 1 구조

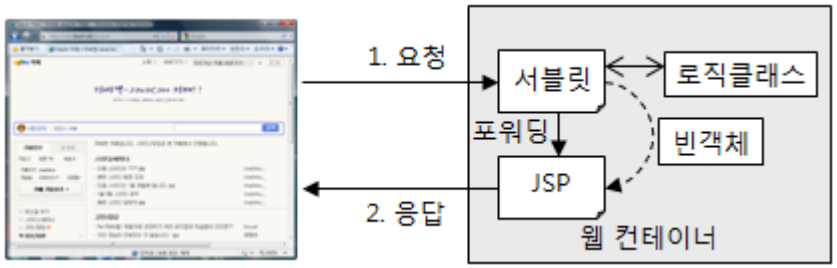
- JSP를 이용한 단순한 모델



- JSP에서 요청 처리 및 뷰 생성 처리
  - 구현이 쉬움
  - 요청 처리 및 뷰 생성 코드가 뒤섞여 코드가 복잡함

모델 2 구조

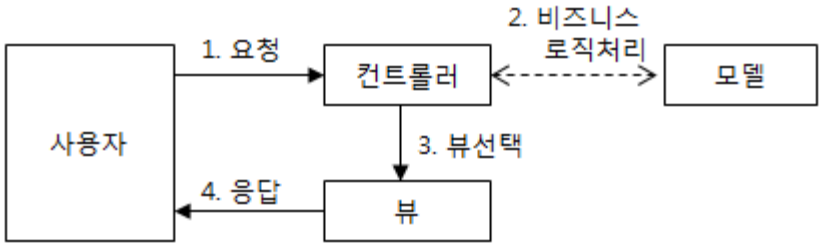
- 서블릿이 요청을 처리하고 JSP가 뷰를 생성



- 모든 요청을 단일 서블릿에서 처리
  - 요청 처리 후 결과를 보여줄 JSP로 이동

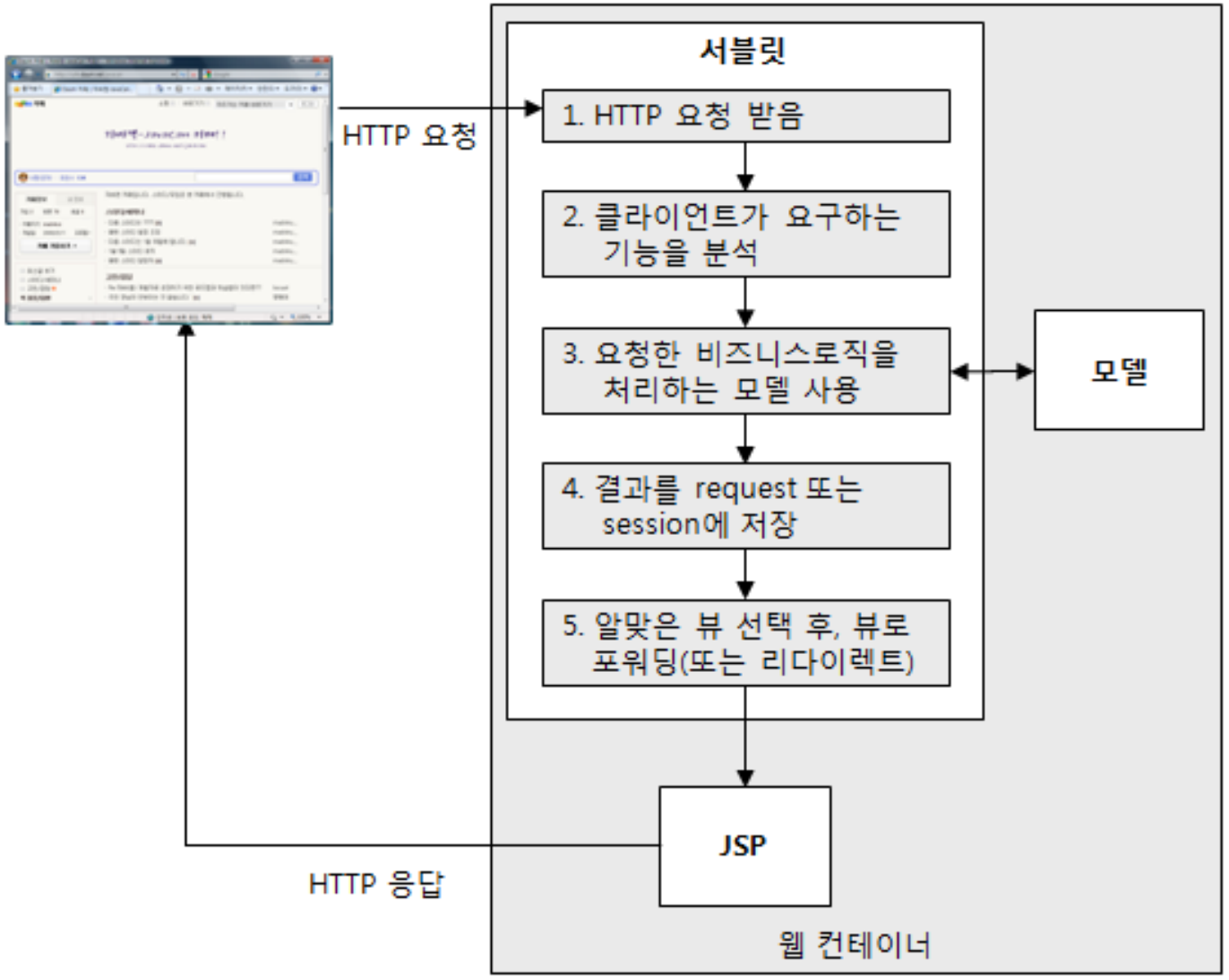
MVC 패턴

- Model-View-Controller
  - 모델 - 비즈니스 영역의 상태 정보를 처리한다.
  - 뷰 - 비즈니스 영역에 대한 프레젠테이션 뷰(즉, 사용자가 보게 될 결과 화면)를 담당한다.
  - 컨트롤러 - 사용자의 입력 및 흐름 제어를 담당한다.

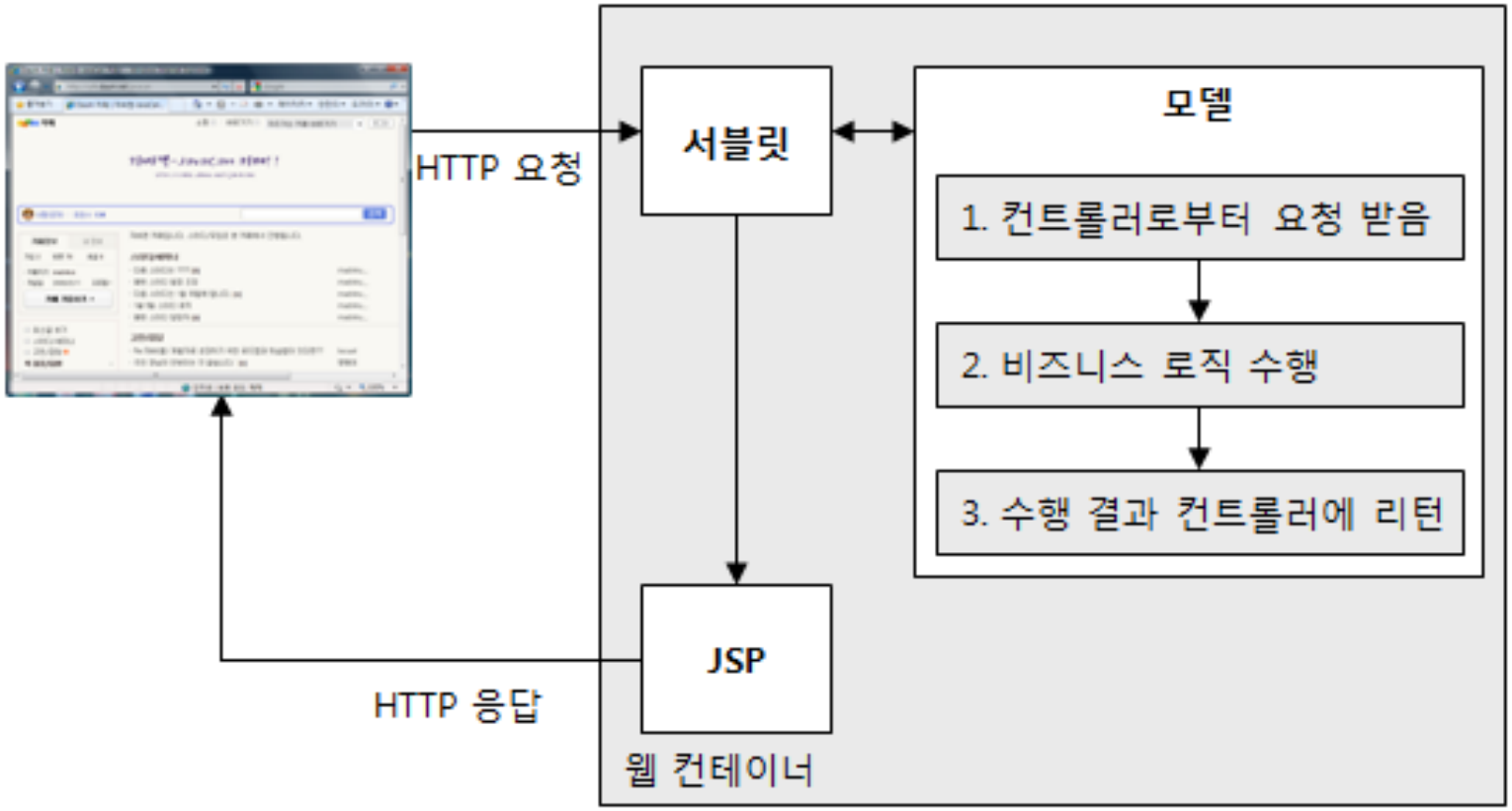


- 특징
  - 로직을 처리하는 모델과 결과 화면을 보여주는 뷰가 분리
  - 흐름 제어나 사용자의 처리 요청은 컨트롤러에 집중
- 모델 2 구조와 매핑 : 컨트롤러-서블릿, 뷰-JSP

MVC의 컨트롤러 : 서블릿



MVC의 모델 : 로직 수행 클래스



MVC 패턴 컨트롤러 기본 구현

```
public class ControllerServlet extends HttpServlet {
    // 1단계, HTTP 요청 받음
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
    throws IOException, ServletException {
        processRequest(request, response);
    }
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
    throws IOException, ServletException {
        processRequest(request, response);
    }
    private void processRequest(HttpServlet request,
                        HttpServletResponse response)
    throws IOException, ServletException {
        // 2단계, 요청 분석
        // request 객체로부터 사용자의 요청을 분석하는 코드
        ...

        // 3단계, 모델을 사용하여 요청한 기능을 수행한다.
        // 사용자에 요청에 따라 알맞은 코드
        // 4단계, request나 session에 처리 결과를 저장
        request.setAttribute("result", resultObject); // 이런 형태의 코드
        ...
        // 5단계, RequestDispatcher를 사용하여 알맞은 뷰로 포워딩
        RequestDispatcher dispatcher =
                    request.getRequestDispatcher("/view.jsp");
        dispatcher.forward(request, response);
    }
}
```

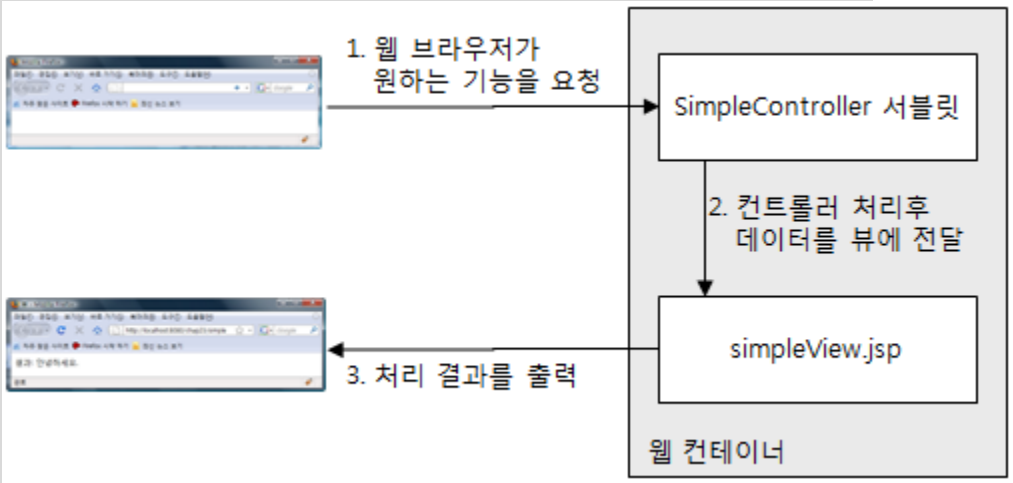
컨트롤러 구현 예

```
private void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws IOException, ServletException {
    // 2단계, 요청 파악
    // request 객체로부터 사용자의 요청을 파악하는 코드
    String type = request.getParameter("type");

    // 3단계, 요청한 기능을 수행한다.
    // 사용자에 요청에 따라 알맞은 코드
    Object resultObject = null;
    if (type == null || type.equals("greeting")) {
        resultObject = "안녕하세요.";
    } else if (type.equals("date")) {
        resultObject = new java.util.Date();
    } else {
        resultObject = "Invalid Type";
    }

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

    // 5단계, RequestDispatcher를 사용하여 알맞은 뷰로 포워딩
    RequestDispatcher dispatcher =
            request.getRequestDispatcher("/simpleView.jsp");
    dispatcher.forward(request, response);
}
```



클라이언트의 요청 명령을 구분하는 방식

- 컨트롤러가 알맞은 로직을 수행하려면 클라이언트가 어떤 기능을 요청하는지 구분할 수 있어야 함

- 요청 기능 구분 방식
  - 특정 이름의 파라미터에 명령어 정보를 전달
    - http://host/chap14/servlet/ControllerServlet?**cmd=BoardList**&...
  - 요청 URI를 명령어로 사용
    - http://host/chap14**/boardList**?...

모델 1 vs 모델 2

모델	장점	단점
모델 1	-배우기 쉬움 -자바 언어를 몰라도 구현 가능 -기능과 JSP의 직관적인 연결. 하나의 JSP가 하나의 기능과 연결	-로직 코드와 뷰 코드가 혼합되어 JSP 코드가 복잡해짐 -뷰 변경 시 논리코드의 빈번한 복사 즉, 유지보수 작업이 불편함
모델 2	-로직 코드와 뷰 코드의 분리에 따른 유지 보수의 편리함 -컨트롤러 서블릿에서 집중적인 작업 처리 가능.(권한/인증 등) -확장의 용이함	-자바 언어에 친숙하지 않으면 접근하기가 쉽지 않음 -작업량이 많음(커맨드클래스+뷰JSP)