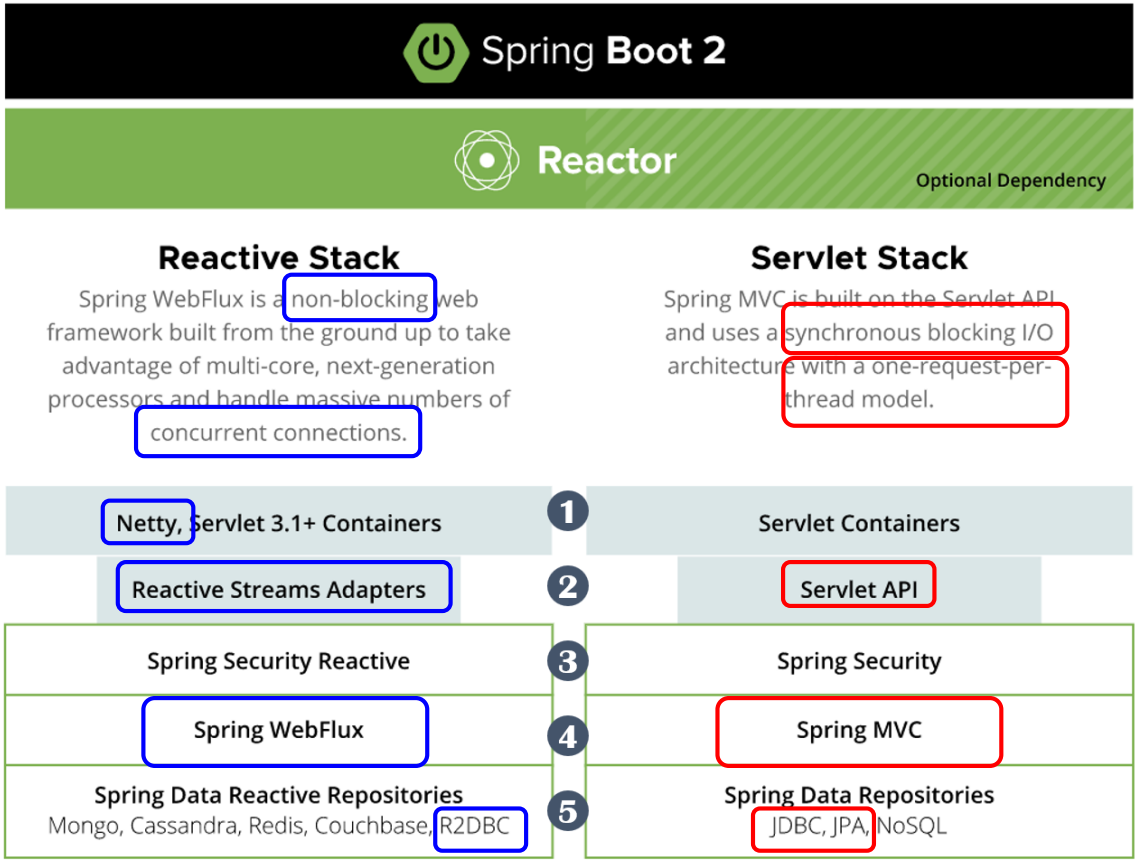
# Spring MVC, Form

558280-1
2025년 봄학기
4/2/2025
박경신

# Spring Boot 3 and Spring Framework 6

# Spring MVC or WebFlux

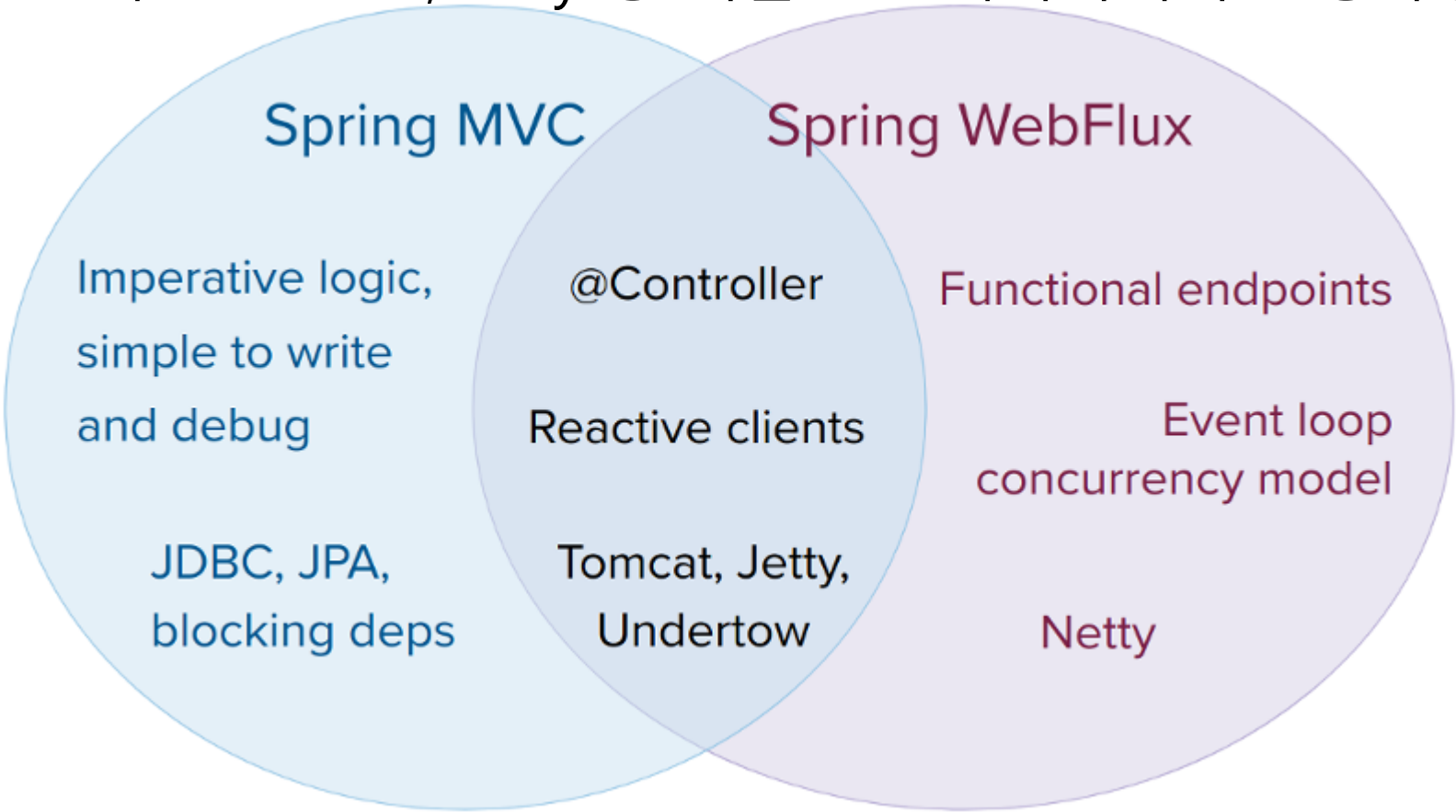
- Spring WebFlux는 Spring 5 (Spring Boot 2)부터 새롭게 추가된 Reactive Stack Web Framework (2017)




https://spring.io/reactive

# Spring MVC or WebFlux

- 공통점은 @Controller, @RestController, Web Client 사용
  그리고 Tomcat, Jetty 등 서블릿 컨테이너에서 실행가능



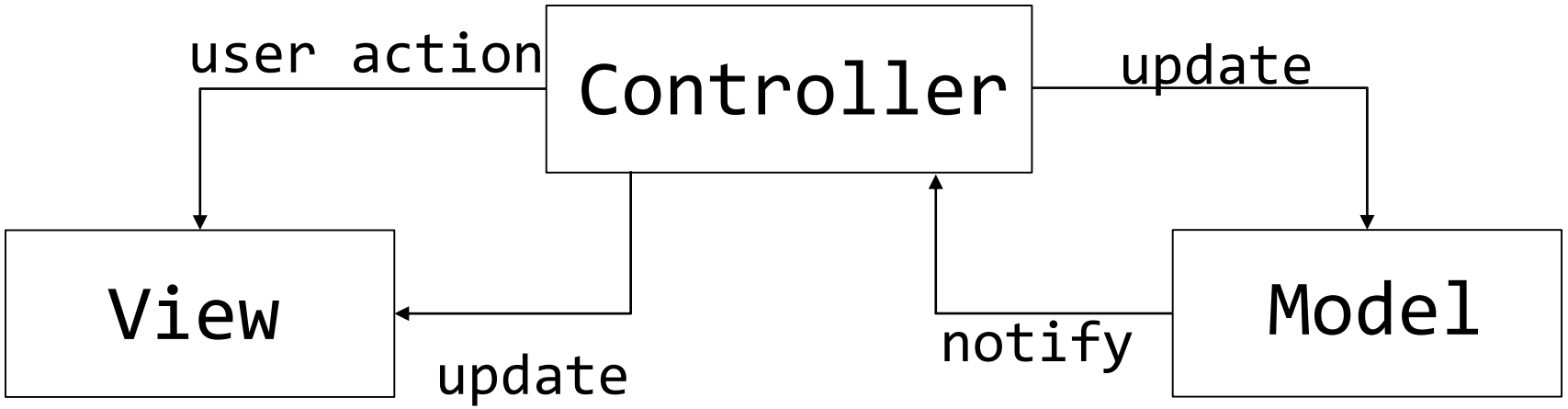

https://docs.spring.io/spring-framework/reference/web/webflux/new-framework.html#webflux-framework-choice

# MVC(Model-View-Controller) Pattern

- MVC 디자인 패턴은 애플리케이션을 비즈니스 로직(Controller)과 데이터(Model), 표현 부분 (View)의 세 가지 역할로 구분한 방법.
- MVC는 원래 Smalltalk 언어에서 유래되었지만, 현재 GUI applications 과 web frameworks에서 널리 사용되고 있음.
- MVC는 J2EE 디자인 패턴에서 가장 많이 사용되는 패턴.

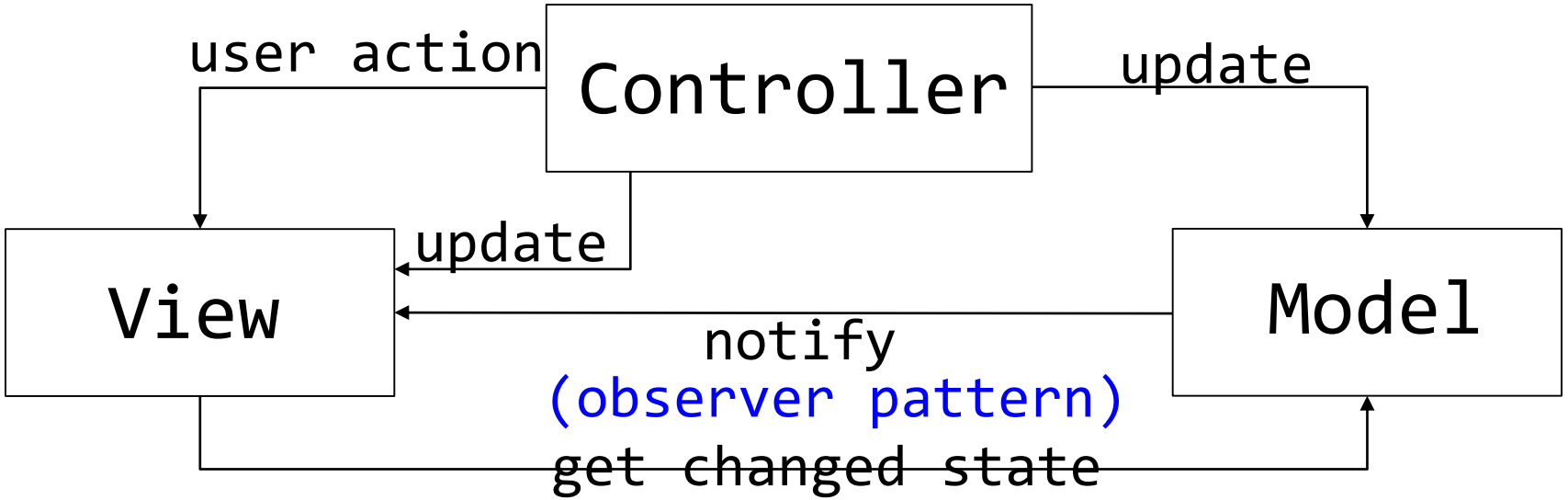
# MVC Pattern

- MVC 패턴은 Model과 View의 종속성을 제거함
  - View에 상관없이 모델이 변경될 수 있음
  - Model에 상관없이 View가 변경 가능함
- Passive Model
  - Controller만이 Model을 조작함



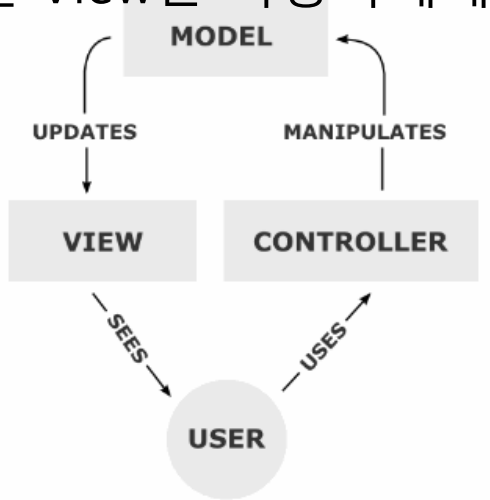
# MVC Pattern

- Active Model
  - Model은 Subject 인터페이스를 제공하고 View에 Observer로 등록
  - View는 (요청 후) Model로 부터 데이터를 수신하고 이를 화면에 업데이트



# Web MVC Pattern

- https://opentutorials.org/course/697/3828 생활코딩에서 인용
  1. 사용자가 웹사이트에 접속 (uses)
  2. Controller는 사용자가 요청한 웹페이지를 서비스 하기 위해서 Model을 호출 (manipulates)
  3. Model은 데이터베이스나 파일과 같은 데이터 소스를 제어한 후 그 결과를 반환
  4. Controller는 Model이 반환한 결과를 View에 반영 (updates)
  5. 데이터가 반영된 View는 사용자에게 보여짐 (sees)



# Controller

- Model과 View 사이에서 메인 비즈니스 로직을 수행
- 사용자로부터 입력에 대한 응답으로 Model 및 View를 업데이트하는 로직을 포함
- 비즈니스 로직이 복잡하면 BO(Business Object) 클래스를 두어 처리
- 사용자의 요청은 모두 컨트롤러를 통해 진행함
- 컨트롤러로 들어온 요청은 어떻게 처리할지 결정하여 모델로 요청을 전달함

# Model

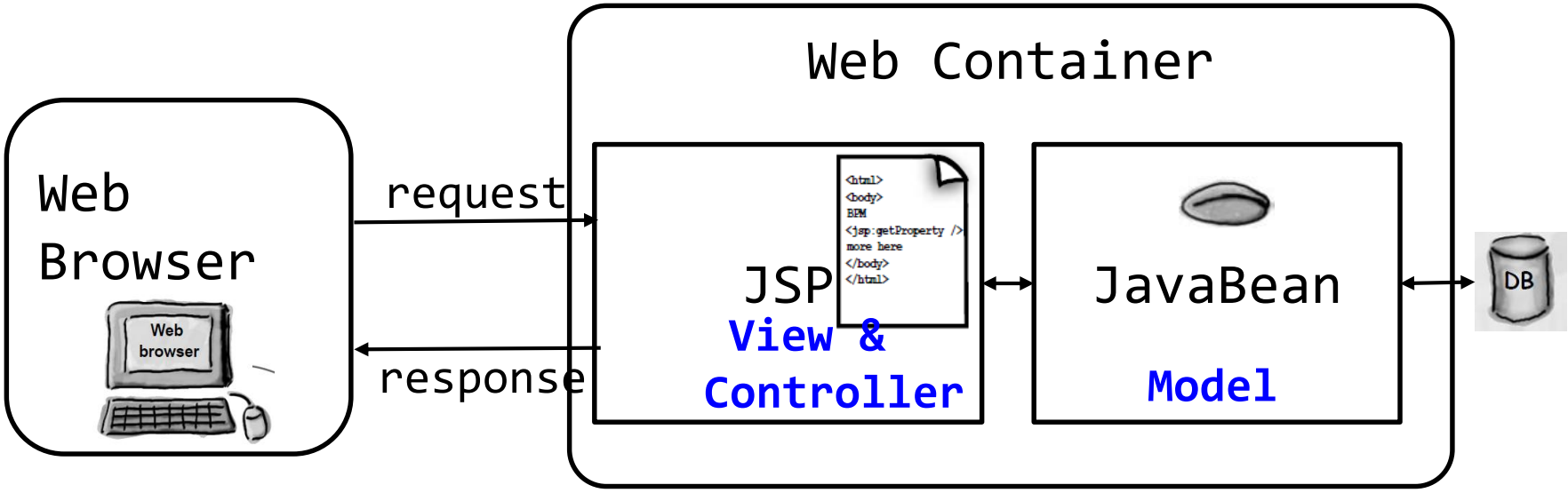
- 데이터를 처리하는 부분 담당
- 데이터베이스와 연동을 위한 DAO(Data Access Object)과 데이터의 구조를 표현하는 DTO(Data Transfer Object)/VO(Value Object)로 구성됨
- 빈즈, 자바 클래스로 구현

# View

- 사용자에게 보여지는 화면 자체의 영역을 담당
- 사용자 인터페이스(User Interface) 요소들이 여기에 포함되며, 데이터를 각 요소에 배치함
- View에서는 별도의 데이터를 보관하지 않음

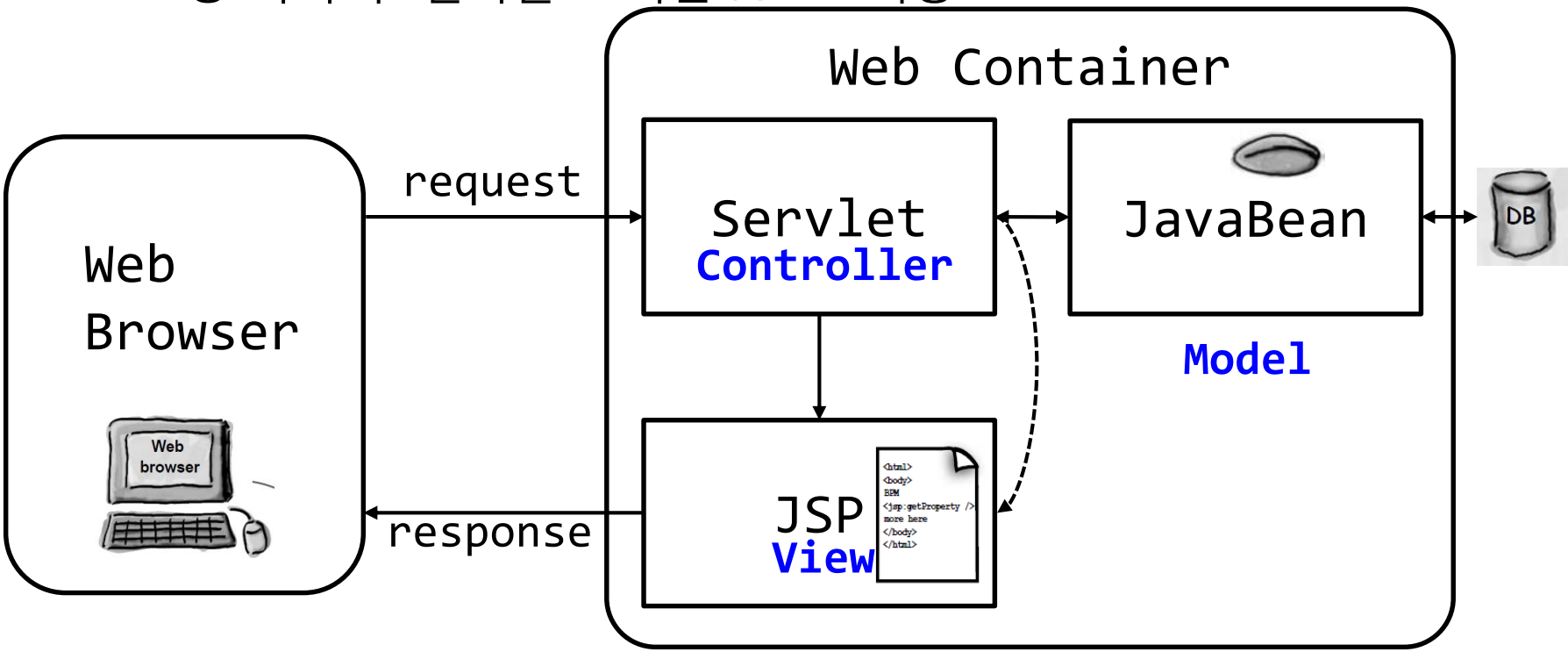
# Servlet 변천 - Model1

- Model1은 초기 JSP개발에 사용된 모델
- 일반적으로 사용되는 JSP개발 방법
- 유지보수의 어려움으로 인해 모델2가 권장됨



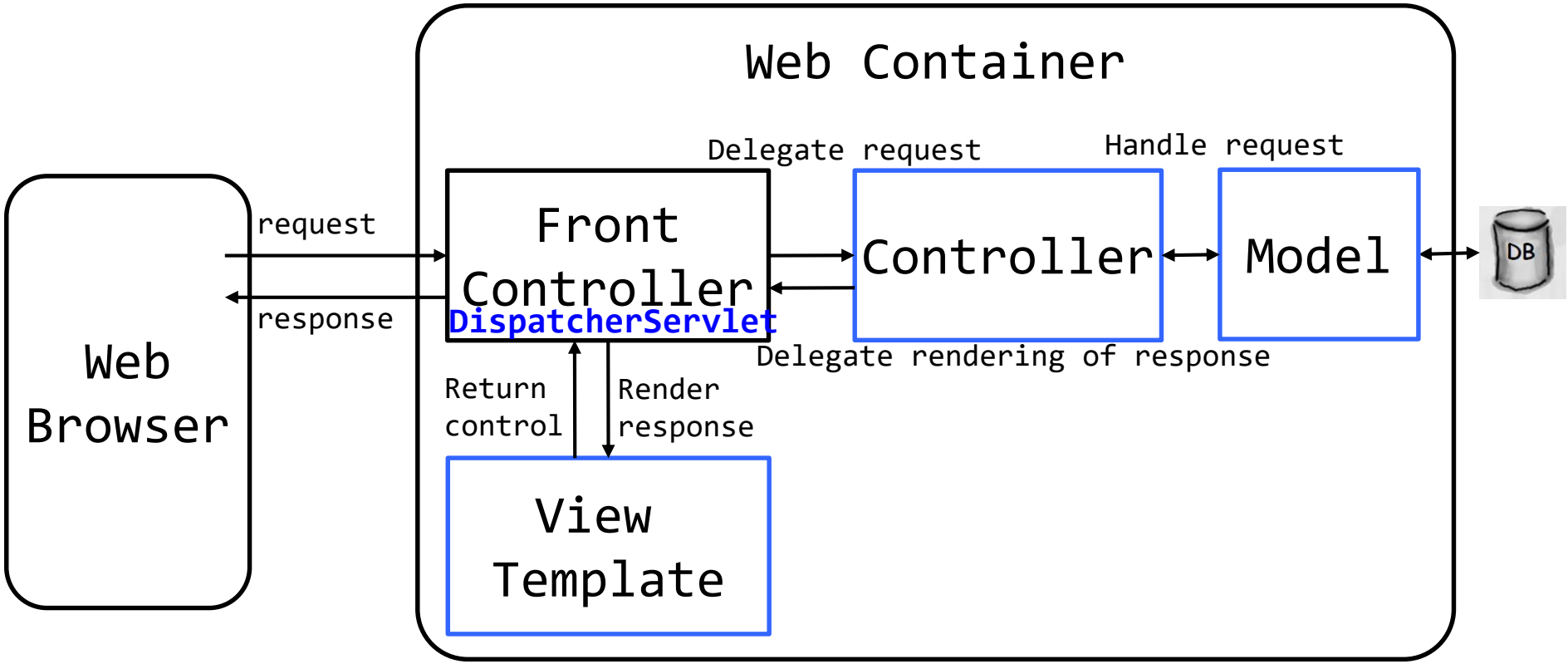
# Servlet 변천 - Model2

- Model2는 MVC 패턴을 Web에 적용한 방식
- 서블릿이 요청을 처리하고 JSP가 뷰를 생성
- 모든 요청을 단일 서블릿에서 처리
  - 요청 처리 후 결과를 보여줄 JSP로 이동



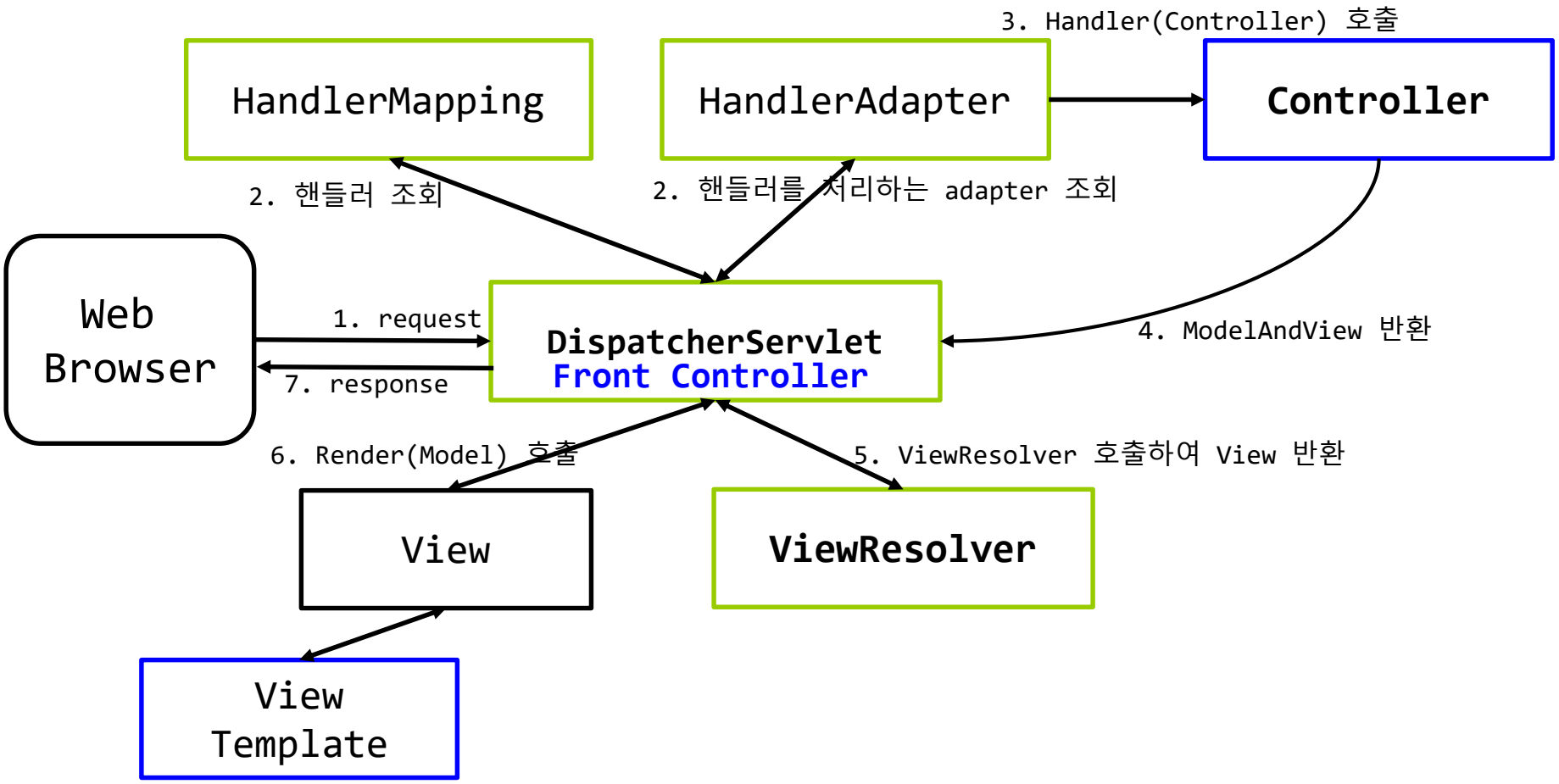
# Spring Web MVC

- Spring Web MVC
  - https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html



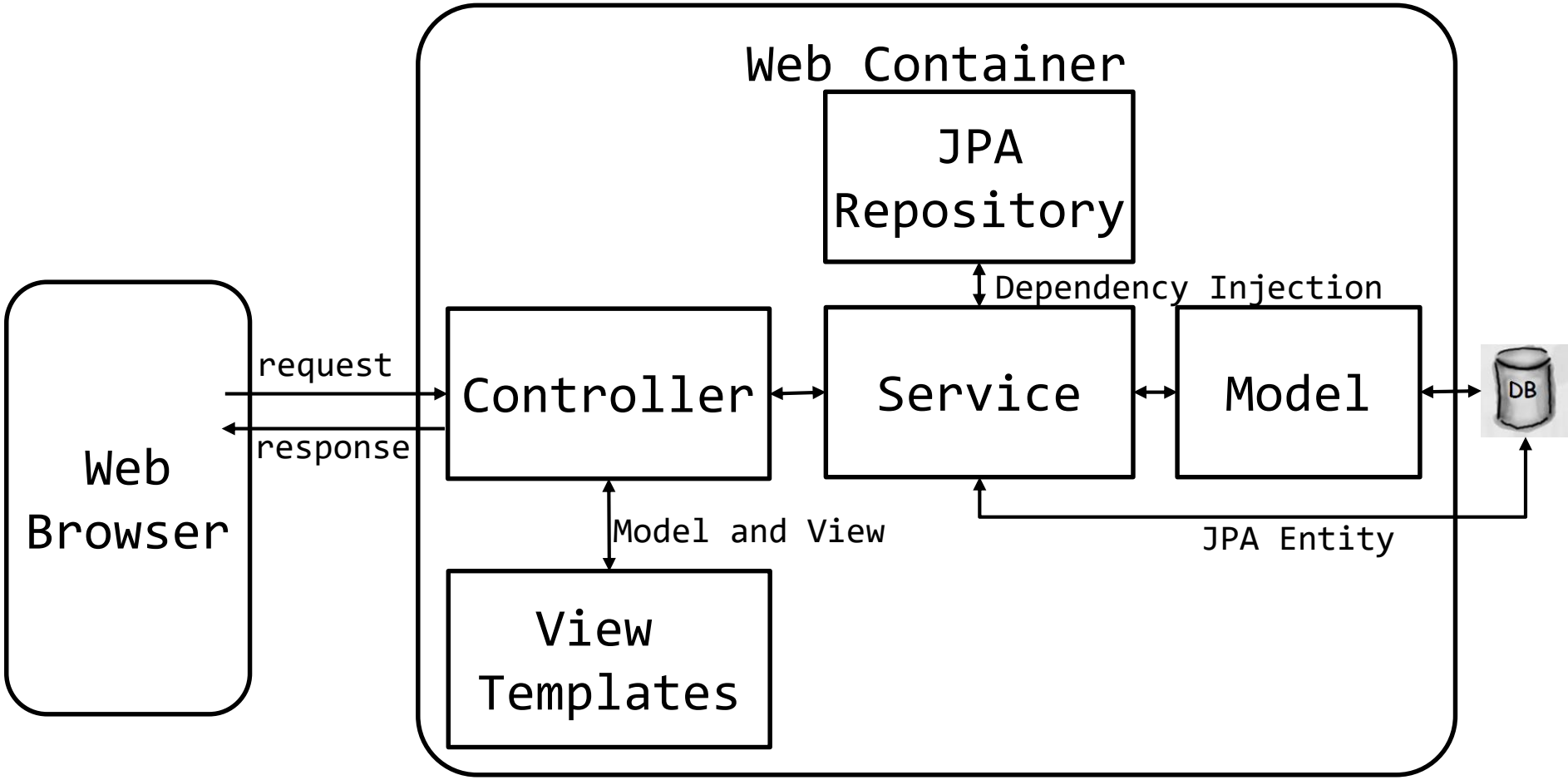
# Spring MVC

- Spring MVC 패턴은 Model2의 발전된 형태



# Spring Boot MVC

- Spring Boot MVC




https://www.researchgate.net/publication/341151097_Review_on_Spring_Boot_and_Spring_Webflux_for_Reactive_Web_Development Fig.2

# Spring Boot MVC

- Spring MVC에 대한 대부분 의존성은 spring-boot-starter를 **spring-boot-starter-web**으로 변경하여 충족됨.
- 무엇보다도 이로 인해 spring-webmvc 및 spring-boot-starter-tomcat에 대한 의존성이 발생함.
- pom.xml에 spring-boot-starter-web 의존성(dependency) 추가

```
<dependencies>
   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

# Spring Boot DispatcherServlet, HandlerMapping, HandlerAdapter

- DispatcherServlet는 스프링부트 자동설정 기능으로 자동 등록
- 스프링부트는 HandlerMapping, HandlerAdapter 자동 등록
  - HandlerMapping
    - 0 = **RequestMappingHandlerMapping** - 애노테이션 기반의 컨트롤러인 **@RequestMapping**에서 사용 (요청 매핑 정보를 관리하고, 요청이 왔을 때 이를 처리하는 Handler를 찾는 클래스)
    - 1 = BeanNameUrlHandlerMapping - 스프링 빈의 이름으로 핸들러를 찾음 (빈이름을 url로 사용한다는 매핑 전략)
  - HandlerAdapter
    - 0 = **RequestMappingHandlerAdapter** - 애노테이션 기반의 컨트롤러인 **@RequestMapping**에서 사용
    - 1 = HttpRequestHandlerAdapter - HttpRequestHandler 처리
    - 2 = SimpleControllerHandlerAdapter - Controller 인터페이스 (애노테이션 아니고, 과거에 사용)

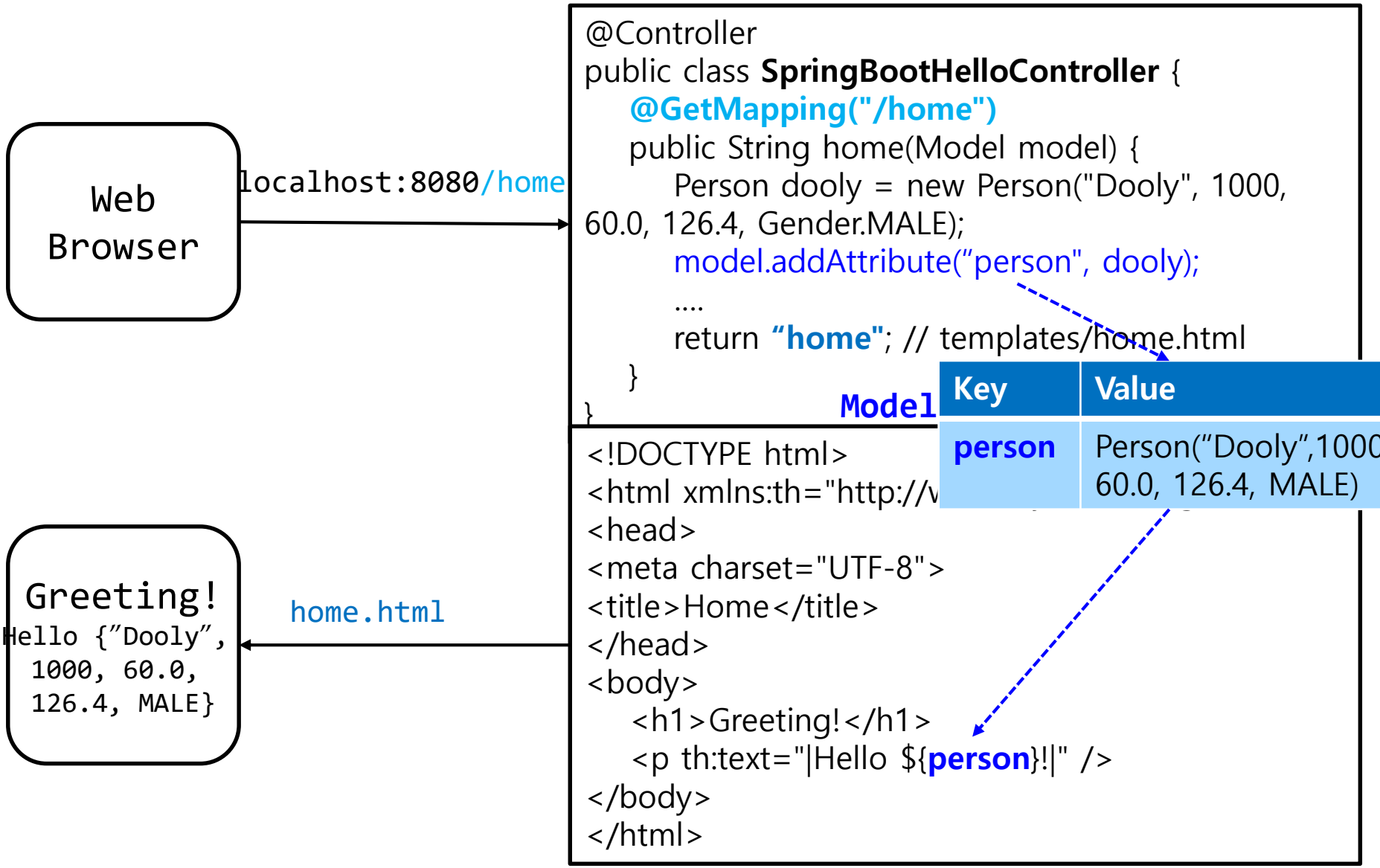
# Spring Boot ViewResolver

- 스프링부트에서는 InternalResourceViewResolver를 기본 ViewResolver로 사용
  - 스프링부트는 DispatcherServlet에서 initViewResolvers를 실행
  - 스프링부트에서 thymeleaf 사용
    - 스프링부트에서 **spring-starter-thymeleaf 의존성을 주입**하고 프로젝트 생성 시 만들어져 있는 **templates 폴더 안에 html** 만 만들어주면 됨
  - 스프링부트에서는 JSP 사용대신 템플렛 엔진을 권장
    - spring-starter-thymeleaf 의존성이 들어있으면 ViewResolver가 선언되어있어서 InternalResourceViewResolver를 ViewResolver로 등록하지 못해 jsp를 찾지 못함 (스프링부트는 가능하다면 jsp를 피하고 Thymeleaf와 같은 템플릿 엔진을 사용하라고 권장하고 있으므로 스프링부트에서 jsp 사용 설정은 여기에서는 생략함)

# Controller 구현

```
@Controller
public class SpringBootHelloController {
    @RequestMapping(value="/home", method=RequestMethod.GET)
    public String home(Model model) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        Person dooly = new Person("Dooly", 1000, 60.0, 126.4, Gender.MALE);
        List<Person> people = new ArrayList<Person>();
        people.add(dooly);
        people.add(new Person("Heedong", 3, 40.0, 56.4, Gender.MALE));
        model.addAttribute("data", "Problem Solving Programming"); 모델추가
        model.addAttribute("serverTime", dateFormat.format(new Date()));
        model.addAttribute("person", dooly);
        model.addAttribute("people", people);
        return "home"; // templates/home.html 이란 view 이름 return
    }
}
```

# 실행 흐름


Web Browser
localhost:8080/home
@Controller
public class SpringBootHelloController {
@GetMapping("/home")
public String home(Model model) {
Person dooly = new Person("Dooly", 1000, 60.0, 126.4, Gender.MALE);
model.addAttribute("person", dooly);
....
return "home"; // templates/home.html
}
}
Model
Key
Value
person
Person("Dooly",1000, 60.0, 126.4, MALE)
<!DOCTYPE html>
<html xmlns:th="http://v
<head>
<meta charset="UTF-8">
<title>Home</title>
</head>
<body>
<h1>Greeting!</h1>
<p th:text="|Hello ${person}!|" />
</body>
</html>
home.html
Greeting!
Hello {"Dooly", 1000, 60.0, 126.4, MALE}

# @RequestMapping 으로 요청 매핑하기

- 요청 URL은 다양
  - /webmvc/members
  - /webmvc/cart/emptycart
- 각각의 유의미한 URL에 대해서 이를 처리할 메소드를 구현하여 연결하는 작업 -> 요청 매핑
  - 아래 예시는 사용자가 localhost:8080**/members** 연결 시 (GET메소드로) **list 메소드**를 실행하여 적절한 뷰로 연결

```
@RequestMapping(value = "/members", method = RequestMethod.GET) 경로 매핑
public String list(Model model) {
    List<Member> list = dao.list();
    model.addAttribute("memberList", list);   뷰에 넘겨줄 모델
    return "list";                templates/list.html 뷰 이름 리턴
}
```

# @RequestMapping 으로 요청 매핑하기

- @RequestMapping을 클래스에 @Controller와 함께 사용하기
  - 이 경우 컨트롤러 경로 **/webmvc/hello** 에 대해 **hello 메소드** 실행

```
@Controller
@RequestMapping(value = "/webmvc“)
public class SpringBootWebmvcController {
    @GetMapping("/hello")
    public String hello(Model model) {
        Person person = new Person("Dooly", 100, 40.4, 126.4, Gender.MALE);
        model.addAttribute("person", person);
        return “hello"; // templates/hello.html
    }
}
```

# @RequestMapping 으로 요청 매핑하기

- 요청 URL 경로 처리에 유의
  - 스프링부트 프로젝트 **application.properties** 에서 **server.servlet.context-path=/webmvc** 설정했고
  - **컨트롤러 클래스에 @RequestMapping(value="/main")** 매핑했고
  - 그리고, 이 컨트롤러의 한 메소드가 **@GetMapping**으로 **/person/list**를 처리한다면
  - 실제로는 **/webmvc/main/person/list**로 접근시에 이 메소드가 실행

# @RequestMapping 으로 요청 매핑하기

- application.properties 설정

```
server.servlet.context-path=/webmvc
```

- Controller 클래스에 요청 경로 처리
  - 컨트롤러 경로 **/webmvc/main/person/list** 에 대해 **list 메소드** 실행

```
@Controller
@RequestMapping(value="/main")
public class SpringBootWebmvcController {
    @GetMapping("/person/list") // localhost:8080/webmvc/main/person/list
    public String list(Model model) {
        List<Member> list = dao.list();
        model.addAttribute("list", list);
        return "list"; // templates/list.html
    }
}
```

# @RequestMapping 으로 요청 매핑하기

- HTTP 메소드 선택 기능
  - GET, POST, PUT, PATCH, DELETE, TRACE, OPTIONS
- 이 경로에 접근하는 사용자의 의도가 무엇인지에 따라 다른 동작 할당 가능
- 웹 브라우저에서 보내지는 요청은 **GET/POST**만 가능
  - GET : **주소창에 경로**를 입력/링크를 클릭 등
  - POST : **<form method = "post">** 등의 태그를 통해 값을 전달

# @GetMapping, @PostMapping

- 스프링 MVC는 별도 설정이 없으면 GET, POST 방식에 상관없이 @RequestMapping에 지정한 경로와 일치하는 요청을 처리
- 그 외에도 @GetMapping 또는 @PostMapping 사용 가능

server.servlet.context-path=/webmvc 존재한다면

```
@Controller
@RequestMapping(value = "/main")
public class HomeController {

// /main/register 경로로 들어오는 요청 중 POST 방식만 처리
@PostMapping("/register") // localhost:8080/webmvc/main/register
public String register(Model model) {
    ....
}
// /home 경로로 들어오는 요청 중 GET 방식만 처리
@GetMapping("/home") // localhost:8080/webmvc/main/home
public String home() {
    return "redirect:/main/register";
}
```

# Redirect 처리

- URL을 직접 입력하는 경우, redirect 사용
  - 웹어플리케이션을 기준으로 이동 경로를 생성
    - "redirect" 뒤의 문자열이 "/"로 시작하는 경우
    - 예를 들어, "**redirect:/main/register**"의 경우 웹 어플리케이션 경로와 합쳐져 "**/webmvc/main/register**"가 됨
  - 현재 경로를 기준으로 상대 경로를 이용
    - "/"로 시작하지 않을 경우
  - 절대 경로를 이용
    - 완전한 URL을 사용

# Redirect

- Redirect
  - 클라이언트가 처음 요청한 URL이 아닌, 다른 URL로 이동할 수 있도록 서버가 응답(Response)하는 방식
  - 클라이언트가 새로 페이지를 요청한 것과 같은 방식으로 페이지가 이동됨
  - 즉, Request, Response가 유지되지 않음 (새로 만들어짐)
  - “redirect:/new-page”와 같이 주면, new-page로 이동된 URL이 화면에 보임

# @PathVariable로 경로 변수 만들기

- localhost:8080/webmvc/main/members/
- 각 멤버에 대해 매번 다른 메소드를 만들 수는 없으니...
  - @RequestMapping("/members/**{memberId}**")
    public String detail**(@PathVariable("memberId")** int memberId, Model model)
    - 메소드 안에서 memberId에 "Park"으로 사용 가능
    - 한 경로에 **여러 PathVariable** 사용 가능
      - /members/**{memberId}**/orders/**{orderId}**
    - 타입은 알아서 적절하게 변환됨(문자열->숫자)

# Ant 패턴을 이용한 경로 표현

- Ant는 ?, *, ** 을 이용하여 경로 패턴을 명시
  - ? 1개 글자
  - * 0개 이상의 글자
  - ** 0개 이상의 디렉토리 경로
- m/category/files, m/category/sub/files, m/cat/sub/my/files 모두를 받은 경로를 만들려면?
  - "/m/**/files"
  - 실제 경로 값을 구하기 위해 request.getRequestUri()를 실행해야 함

@GetMapping("/members/?*.html") /members/로 시작하고 확장자가 .html로 끝나는 모든 경로

@PostMapping("/folders/**/files") /folders/로 시작하고 중간에 0개 이상 중간 경로가 존재하고 /files로 끝나는 모든 경로

@RequestMapping("/m/image?.html") /m/image로 시작하고 1글자가 사이에 위치하고 .html로 끝나는 모든 경로

# 처리 가능한 요청/응답 가능한 컨텐트 타입 제한

- 웹브라우저에서 input 태그를 이용해 폼을 전송할 때는 기본적으로 application/x-www-form-urlencoded 사용
- AJAX등의 등장에 따라 json/xml등을 전송하는 경우가 늘어남
- 요청이 json인 경우만 처리하는 매핑 :
  - @RequestMapping(value="...."... **consumes**="application/json")
  - consumes 속성은 요청헤더 Content-type을 제한하겠다는 의미
- 응답이 json인 경우만 처리하는 매핑 :
  - @RequestMapping(value="...."... **produces**="application/json")
  - produces 속성은 반환하는 Content-type을 정의

# Model을 통한 컨트롤러에서 뷰로 데이터 전달

- 컨트롤러는 뷰가 응답 화면을 구성하는데 필요한 데이터를 생성해서 Model을 이용하여 전달
  - RequestMapping이 적용된 메소드의 파라미터로 Model을 추가
  - Model addAttribute(String attrName, Object attrValue);
  - Model addAllAttributes(Map<String, ?> attributes);
  - boolean containsAttribute(String attrName);

```
@RequestMapping(value = "/members/{id}", method = RequestMethod.GET)
public String detail(@PathVariable("id") int id, Model model) {
    Member member = dao.get(id);
    model.addAttribite("member" , member); // model에 데이터 추가
    return "detail"; // 뷰 이름을 리턴 detail.html
}
```

# ModelAndView를 통한 뷰선택과 모델 전달

- ModelAndView를 사용하여 뷰와 모델을 한번에 처리 가능
  - ModelAndView addObject(String name, Object object);
  - ModelAndView setViewName(String viewName);

```
@RequestMapping(value = "/members/add", method = RequestMethod.GET)
public ModelAndView add(ModelAndView mav) {
    Member member = new Member();
    mav.addObject("member", member); // model에 데이터 추가
    mav.setViewName("addForm"); // 뷰 이름을 지정
    return mav; // ModelAndView 리턴
}
```

# HTTP Request 처리하기

- Get/Post 전송된 요청 파라미터 값을 사용하기 위한 방법
  - **HttpServletRequest**를 직접 이용
  - **@RequestParam** 어노테이션을 사용

```
@RequestMapping(value = "/detail", method = RequestMethod.GET)
public ModelAndView detail(HttpServletRequest request) {
    String name = request.getParameter("name");
    ....
} // localhost:8080/detail?name=AJ

@RequestMapping(value = "/detail", method = RequestMethod.GET)
public ModelAndView detail(@RequestParam("name") String name) {
   // 스프링에서 지원하는 변환기에서 지원되는 모든 타입을 변환 가능
    ....
} // localhost:8080/detail?name=AJ

@RequestMapping(value = "/detail", method = RequestMethod.GET)
public ModelAndView detail(@RequestParam(value="id" defaultValue="0") int id,
@RequestParam("name") String name) {
....
} // localhost:8080/detail?id=123&name=AJ
```

# HTTP Request 처리하기

- 웹 페이지에서 서버로 다양한 값이 전달됨
  - GET방식/POST 방식
  - 로그인 데이터, 게시판에 글을 쓴 데이터, …

```
@RequestMapping("/detail", method = RequestMethod.GET)
public String detail(HttpServletRequest request, Model model) throws IOException {
    String id = request.getParameter("id");
    if (id == null)
        return REDIRECT_EVENT_LIST;
    Long eventId = null;
    try {
        eventId = Long.parseLong(id);
    } catch (NumberFormatException e) {
        return REDIRECT_EVENT_LIST;
    }
    Event event = getEvent(eventId);
    if (event == null)
        return REDIRECT_EVENT_LIST;
    model.addAttribute("event", event);
    return "detail";
} // localhost:8080/detail?id=123&name=AJ
```

/detail?id=123

id==123

<input name="id">123</input>

# HTTP Request 처리하기

- **@RequestParam**

```
@RequestMapping("/detail2")
public String detail2(@RequestParam("id") long eventId, Model
model) {
    Event event = getEvent(eventId);
    if (event == null)
        return REDIRECT_EVENT_LIST;
    model.addAttribute("event", event);
    return "detail";
} // localhost:8080/detail2?id=123
```

- RequestParam("id", required=false)를 이용해 id자리에 null을 넣을 수도 있음
  - defaultValue 속성을 이용해 null 대신 기본값도 사용 가능

# Command 객체를 이용한 폼 전송 처리

- 여러 값을 한꺼번에 parameter로 받는 경우 @RequestParam을 여러 번 사용해야 함

```
<form>
<input type="text" name="email">...
<input type="text name="name">....
<input type="text name="password">....
```

```
@PostMapping("/register/step3")
public String handleStep3(HttpServletRequest request) {
    String email = request.getParameter("email");
    String name = request.getParameter("name");
    String password = request.getParameter("password");
    String confirmPassword = request.getParameter("confirmPassword");
... // 파라미터의 개수가 훨씬 많아진다면 그만큼 일일히 다 값을 읽어야함
```

# Command 객체를 이용한 폼 전송 처리

- RegisterRequest와 같은 **커맨드 객체**(자바빈즈 규약 준수)를 만들고 컨트롤러 메소드의 parameter에 전달 가능
  - 모델에도 자동으로 포함되므로 매우 편리
  - "${regRequest.name}님 환영합니다" 와 같이 뷰에서 커맨드 객체 사용 가능

```
@PostMapping("/register/step3")
public String handleStep3(RegisterRequest regRequest){
      …
}
public class RegisterRequest {
    private String email, name, password;
    public void setEmail(String email){ this.email = email; }
    public String getEmail(){…} // getter/setter
    …
} // 요청 파라미터의 값을 전달 받을 수 있는 setter 를 포함하는 객체를
커맨드 객체로 사용
```

# Command 객체를 이용한 폼 전송 처리

- **@ModelAttribute** 어노테이션으로 커맨드 객체 폼 전송
  - 커맨드 객체에 접근할 때 사용할 속성 이름을 @ModelAttribute 어노테이션으로 사용해서 변경
  - <form> 태그를 사용하려면 커맨드 객체가 존재해야 함

```
@RequestMapping(value="/update", method=Request.POST)
public String update(@ModelAttribute("regRequest") RegisterRequest regRequest) {
    System.out.println(regRequest);
....
}
```

regRequest 객체를 모델에 넣어야 함

```
<!– editForm.html -->
<form th:action="@{/update}" method="post" th:object="${regRequest}">
<label>Email:</label><br><input type="text" th:field="*{email}" />
<label>Name:</label><br><input type="text" th:field ="*{name}" />
<label>Password:</label><br> <input type="text" th:field ="*{password}" />
...
</form>
```

# Command 객체를 이용한 폼 전송 처리

- editForm.html 뷰를 호출하는 컨트롤러 코드에 regRequest 커맨드 객체를 넣어줘야 함

```
@RequestMapping(value = "/edit", method = RequestMethod.GET)
public ModelAndView edit(HttpServletRequest request) {
    String name = request.getParameter("name");
    RegisterRequest regRequest = ... // find by name
    ModelAndView mav = new ModelAndView("editForm"); // 뷰 이름을 지정
    mav.addObject("regRequest", regRequest);
    return mav;
} // localhost:8080/edit?name=AJ
```

# Form 처리

- 폼(form)
  - 사용자가 웹 브라우저를 통해 입력된 모든 데이터를 한 번에 웹 서버로 전송하는 양식
  - 전송한 데이터는 웹 서버가 처리하고 처리 결과에 따라 다른 웹 페이지를 보여줌
  - 사용자와 웹 애플리케이션이 상호 작용하는 중요한 기술 중 하나임
  - 사용자가 어떤 내용을 원하는지, 사용자의 요구 사항이 무엇인지 파악할 때 가장 많이 사용하는 웹 애플리케이션의 필수적인 요소임

# Form 태그

- 폼(form)을 구성하는 태그 종류

| 태그 | 설명 |
| --- | --- |
| form | 폼을 정의하는 태그로 최상위 태그 |
| input | 사용자가 입력할 수 있는 태그 |
| select | 항목을 선택할 수 있는 태그 |
| textarea | 여러 줄을 입력할 수 있는 태그 |

# 주요 폼 태그

- <form> 태그
  - 커맨드 객체의 이름이 기본값인 “command”가 아니라면 modelAttribute 속성을 사용해 설정
  - 커맨드 객체를 이용해 이전에 입력한 값을 출력 가능
- <input> 태그
  - <input> 태그는 text 입력에 사용. **th:field** 속성을 사용해 연결할 커맨드 객체의 프로퍼티를 지정
- <select>, <options>, <option> 태그
  - <select> 태그는 선택 옵션을 제공할 때 사용. items 속성에 옵션목록(Array 또는 List)을 Model을 통해 전달하면 간단하게 생성

# Form 태그

- form 태그
  - 사용자가 다양한 정보를 입력하고 서로 전달할 때 사용하는 태그
  - 단독으로 쓰이지 않고 사용자가 다양한 정보를 입력하는 양식을 포함하는 최상위 태그
  - 속성을 이용하여 폼 데이터를 전송할 때 어디로 보낼지, 어떤 방식으로 보낼지 설정

```
<form attribute1="value1" [attribute2="value2" ...]>
    // 다양한 입력 양식 태그 <input>, <select>, <textarea>
</form>
```

# Form 태그

- form 태그의 속성
  - form 태그의 모든 속성은 필수가 아니라 선택적으로 사용

```
<form action="/nameAction" method="post">
    <input type="text" name="name" value="K">
</form>
```

| 속성 | 설명 |
|---|---|
| **action** | **폼 데이터를 받아 처리하는 웹페이지의 URL 설정** |
| **method** | **폼 데이터가 전송되는 HTTP 방식 설정** |
| name | 폼을 식별하기 위한 이름 설정 |
| target | 폼 처리 결과의 응답을 실행할 프레임 설정 |
| enctype | 폼을 전송하는 콘텐츠 MIME 유형 설정 |
| accept-charset | 폼 전송에 사용할 문자 인코딩 설정 |

# Form 태그의 Method

- Form 태그의 Method
  - 폼 데이터가 전송되는 HTTP 방식

| Method | 설명 |
| --- | --- |
| **GET** | 요청된 URI의 정보를 검색하여 응답 |
| **POST** | 요청된 자원을 생성/업데이트하기 위해 서버로 데이터를 전송하는데 사용. 리소스의 위치를 지정하지 않고 리소스를 생성하는 연산. |

  - DELETE, PUT, PATCH는 Form 태그의 Method로 사용 안함
    - 일반적인 관행은 hidden input field _method를 사용하여 보냄

# Form의 Input 태그

- Form의 input 태그
  - 사용자가 텍스트 입력이나 선택 등을 다양하게 할 수 있도록 공간을 만드는 태그
  - 종료 태그 없이 단독으로 사용할 수 있음
  - input 태그의 기본 속성
    - type은 text, button, radio, checkbox, password, hidden, file, reset, submit 등 다양한 입력
    - name은 입력 양식을 식별하는 이름 설정
    - value는 입력 양식의 초기값 설정

```
<input type="text" name="firstname" value="K">
```

# Form의 Input 태그 종류

- Form의 input type 종류
  - **<input type="button">** 버튼 선택 입력
  - **<input type="checkbox">** 체크박스 버튼 선택 입력
  - <input type="color">
  - <input type="date">
  - <input type="datetime-local">
  - <input type="email">
  - **<input type="file">** 파일 전송 입력
  - **<input type="hidden">** 보이지 않게 숨겨서 값을 전송
  - <input type="image">
  - <input type="month">
  - <input type="number">

# Form의 Input 태그 종류

- Form의 input type 종류
  - **<input type="password">** 암호 입력
  - **<input type="radio">** 라디오 버튼 선택 입력
  - <input type="range">
  - **<input type="reset">** 폼에 입력된 값을 모두 초기화
  - <input type="search">
  - **<input type="submit">** 폼에 입력된 값을 모두 서버에 전송
  - <input type="tel">
  - **<input type="text">** 한 줄 텍스트 입력
  - <input type="time">
  - <input type="url">
  - <input type="week">

# Form의 Select 태그

- Form의 select 태그
  - 여러 개의 항목이 나타나는 목록 상자에서 항목을 선택하는 태그
  - 시작 태그와 종료 태그가 있으며, 리스트 박스에 여러 항목을 추가 삽입하기 위해 반드시 option 태그를 포함해야 함.

```
<select name="itemselect">
    <option value="item1"> item1 </option>
    <option value="item2"> item2 </option>
    <option value="item3"> item3 </option>
    <option value="item4"> item4 </option>
</select>
```

# Form의 Select 태그

- select 태그의 속성
  - name은 입력 양식을 식별하는 이름 설정
  - size는 한 번에 표시할 항목의 개수를 설정
  - multiple은 다중 선택이 가능하도록 함. CTRL-key를 눌러 목록 상자의 항목을 다중 선택함
- option 태그의 속성
  - value는 항목의 값을 설정
  - selected는 해당 항목을 초기값으로 선택
  - disabled는 항목을 비활성화
- optgroup 태그의 속성
  - select 태그 내에 있는 option 태그들을 그룹화하는 데 사용

# Form의 Textarea 태그

- Form의 textarea 태그
  - **여러 줄의 텍스트를 입력**할 수 있는 태그
  - 기본 값은<textarea>와 </textarea> 태그 사이에 설정
  - 입력 폼 안에 사용된 태그와 띄어쓰기가 그대로 출력됨

```
<textarea name="이름" cols="너비" rows="높이">
    ... // 생략
</textarea>
```

| 속성 | 속성값 | 설명 |
|---|---|---|
| name | 텍스트 | 텍스트영역의 이름 설정 |
| **cols** | **숫자** | **입력할 텍스트영역의 너비(열 크기) 설정** |
| **rows** | **숫자** | **입력할 텍스트영역의 높이(행 크기) 설정** |
| wrap | off | 줄 바꿈 안함 |
| | soft | 엔터키를 누르지 않아도 끝에서 자동으로 행이 바뀜 |
| | hard | soft와 비슷하며 서버에 전송할 때 캐리지 리턴 문자를 전달 |

# Form 데이터 처리하기

- 요청 파라미터의 값 받기
  - request 내장 객체는 웹 브라우저가 서버로 보낸 요청에 대한 다양한 정보를 담고 있어 **getParameter()** 메소드를 이용하여 요청 파라미터의 값을 얻을 수 있음

```
String 변수 = request.getParameter("요청 파라미터 이름");
```

# Form 데이터 처리하기

- 요청 파라미터의 전체 값 받기
  - 요청 파라미터를 설정하지 않아도 모든 값을 전달받을 수 있음. 또한 텍스트 박스, 라디오 버튼, 드롭다운 박스와 같은 다양한 유형에 대해 한 번에 폼 데이터를 전달받을 수 있음.
  - 폼 데이터의 일괄 처리 메소드

| 메소드 | 설명 |
|---|---|
| **getParameterNames()** | 모든 입력 양식의 요청 파라미터 이름을 순서에 상관없이 Enumeration 형태로 전달받음 |
| hasMoreElements() | Enumeration 요소가 있으면 true를 반환하고 그렇지 않으면 false를 반환 |
| nextElement() | Enumeration 요소를 반환 |