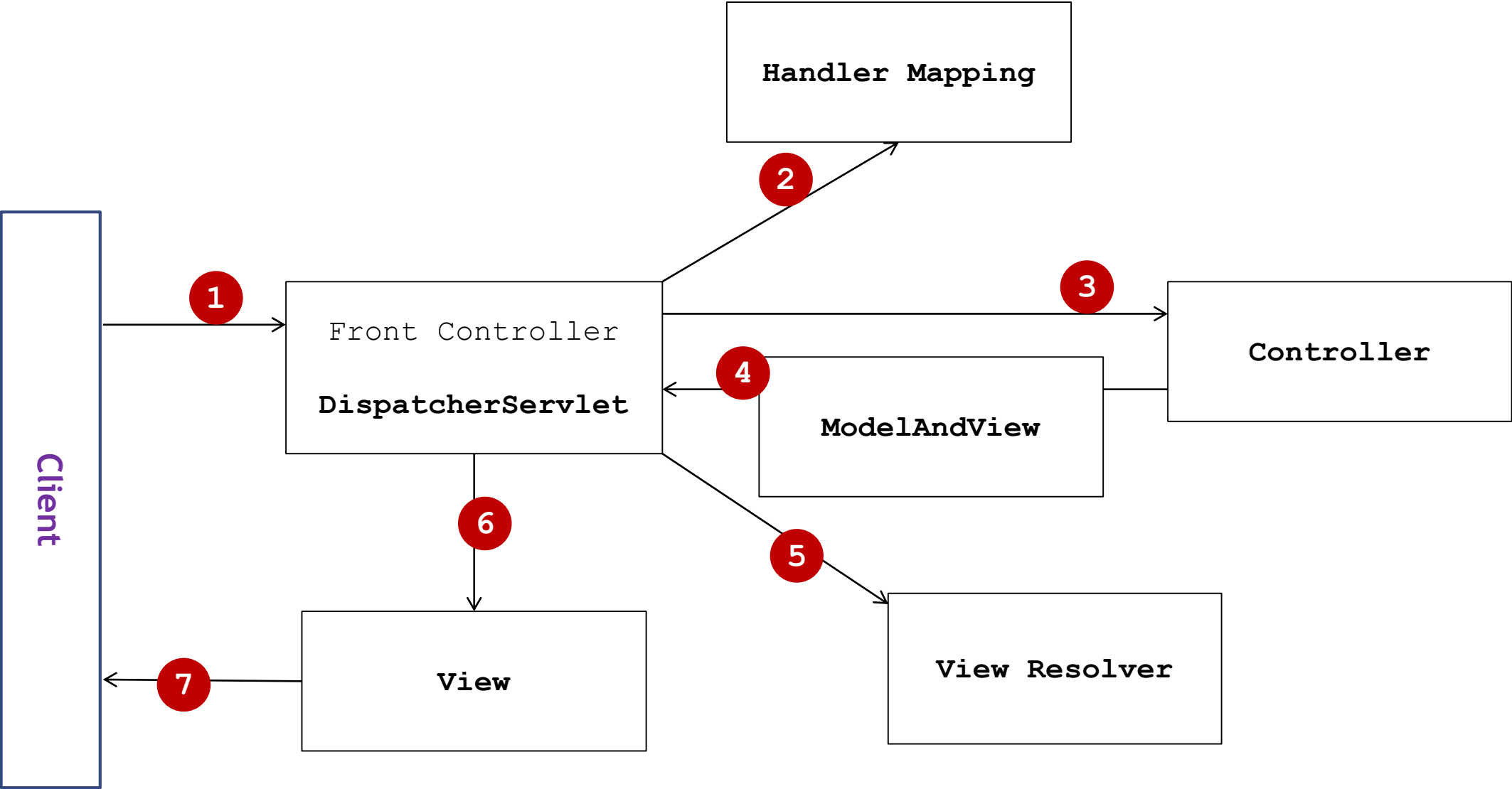


Spring 조금 더 들여다보기

DispatcherServlet과 MVC



DispatcherServlet과 MVC

▶ 사용자 요청의 처리

- 1) 사용자의 요청을 **DispatcherServlet**이 받는다
- 2) 요청을 처리해야 하는 컨트롤러를 찾기 위해 **HandlerMapping**에 질의
HandlerMapping은 컨트롤러 객체에 매핑되어 있는 **URL**을 찾아낸다
- 3) **DispatcherServlet**은 찾은 컨트롤러에게 요청을 전달하고 **Controller**는 서비스 계층의 **Interface**를 호출하여 적절한 비즈니스 로직을 수행한다.
- 4) 컨트롤러는 비즈니스 로직의 수행 결과로 받아낸 도메인 모델 객체와 함께 뷰 이름을 **ModelAndView** 객체에 저장하여 반환한다
- 5) **DispatcherServlet**은 응답할 **View**를 찾기 위해 **ViewResolver**에 질의한다
- 6) **DispatcherServlet**은 찾아낸 **View** 객체에 요청을 전달한다

Annotation을 이용한 Mapping

@RequestMapping

: Handler 매핑

▶ 메서드 단독 매핑

```
public class UserController {  
    @RequestMapping("/hello")  
    public String hello( ... ) {  
        // ...  
    }  
  
    @RequestMapping("/main")  
    public String main( ... ) {  
        // ...  
    }  
}
```

▶ 메서드 선언부 앞쪽에 @RequestMapping 어노테이션을 이용, 매칭할 URL을 기술한다

@RequestMapping

: Handler 매핑

▶ 타입 + 메서드 매핑

```
@RequestMapping( "/user" )
public class UserController {
    @RequestMapping( "/add" )
    public String add( ... ) { }

    @RequestMapping( "/delete" )
    public String delete( ... ) { }
}

@RequestMapping( "/user/add" )
public class UserController {
    @RequestMapping( method = RequestMethod.GET )
    public String form( .... ) { }

    @RequestMapping( method = RequestMethod.POST )
    public String submit( ... ) { }
}
```

@RequestMapping

: Handler 매핑

▶ 타입 단독 매핑

```
@RequestMapping( "/user/*" )  
public class UserController {  
    @RequestMapping  
    public String add( ... ) {  
  
    }  
  
    @RequestMapping  
    public String edit( ... ) {  
  
    }  
}
```

▶ /user/add, /user/edit 등으로 접근 가능

@RequestParam

: 파라미터 매핑

▶ 기본 사용법

- ▶ http 요청 파라미터를 메서드 파라미터에 넣어주는 어노테이션

```
public String view(@RequestParam("id") int id,  
    @RequestParam("name") String name) {  
    //...  
}
```

- ▶ RequestParam을 기본 사용법으로 사용했다면 반드시 파라미터가 넘어와야 한다.
없으면 HTTP 404 - Bad Request를 반환한다
- ▶ 보통은 기본값을 더 추가하여 다음과 같이 파라미터를 매핑

```
public String view(@RequestParam(value="id", required=false, defaultValue="-1") int id,  
    @RequestParam(value="name", required=false, defaultValue="Spring") String name) {  
    //...  
}
```


@PathVariable

: URL Path 기반 파라미터 매핑

▶ 사용법

▶ URL에 쿼리 스트링 대신 URL Path로 풀어 쓰는 방식

▶ 예) /board/view?no=10 -> /board/view/10

```
@RequestMapping("/board/view/{no}")  
public String view(@PathVariable("no") int no) {  
    //...  
}
```

@ModelAttribute

▶ 사용법

- ▶ 요청 파라미터를 객체에 담아 전달

```
public class UserVo {  
    long no;  
    String name;  
    String password;  
    //...  
}  
  
@RequestMapping(value="/user/join", method=RequestMethod.GET)  
public String join(@ModelAttribute UserVo userVo) {  
    userService.join(userVo);  
    //...  
}
```

핸들러 메서드의 파라미터

▶ 다양한 파라미터

- HttpServletRequest, HttpServletResponse
- HttpSession
- Writer

▶ Model 타입 파라미터

- 모델 정보를 담을 수 있는 오브젝트가 전달

```
public String hello(ModelMap model) {  
    User user = new User(1, "Spring");  
    model.addAttribute("user", user);  
    //...  
}
```

Application Context

Web Application Context

- ▶ Spring은 web.xml 서블릿 매핑 설정의 <servlet-name>에 '-servlet.xml'을 붙인 이름의 파일을 WEB-INF에서 찾아 컨테이너에 **Bean**을 생성하고 초기화함

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

- ▶ 서블릿 매핑의 기본 파일명을 무시하고 특정 파일명을 지정하고자 한다면 DispatcherServlet의 초기화 파라미터로 부여할 수 있다

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring-context.xml</param-value>
  </init-param>
</servlet>
```

Web Application Context

: Annotation 방식

- ▶ <spring-name/>'-servlet.xml' 설정파일

```
<context:annotation-config />  
<context:component-scan base-package="com.example.hellospring.controller" />
```

- ▶ Controller 빈을 등록하고 빈의 이름(URL)로 핸들러가 매핑
- ▶ @MVC 기반에서 빈의 생성은 어노테이션 기반의 컴포넌트 스캐닝을 통해 생성되고 메서드가 핸들러 매핑과 어댑터의 대상이 된다

Web Application Context

: Non Annotation 방식

- ▶ <spring-name/>'-servlet.xml' 설정파일
 - ▶ SimpleUrlHandlerMapping

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/member">memberController</prop>
    </props>
  </property>
</bean>

<bean id="memberController"
      class="com.example.hellospring.controller.MemberController" />
```

- ▶ 핸들러 어댑터의 대상이 객체이고 객체의 `handleRequest(HttpServletRequest req, HttpServletResponse res)` 메서드 하나만이 url 대상이 된다

Root Application Context

- ▶ 리스너를 등록해 두면 루트 컨텍스트가 생성되며 설정 파일은 디폴트로 /WEB-INF/applicationContext.xml이다 (in web.xml)

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

- ▶ 서비스 계층, 데이터 액세스 계층을 포함하여 웹 환경과 직접 관련 없는 모든 빈은 여기에 등록한다

Root Application Context

- ▶ applicationContext.xml 만들기
 - ▶ WEB-INF에서 New > Others > Spring > Spring Bean Configuration
 - ▶ 파일명을 applicationContext.xml로 잡아주기
 - ▶ beans, context, mvc xsd 선택
 - ▶ applicationContext.xml의 설정 예

```
<context:annotation-config />
<context:component-scan base-package="com.example.hellospring">
  <context:include-filter type="annotation"
    expression="org.springframework.stereotype.Repository" />
  <context:include-filter type="annotation"
    expression="org.springframework.stereotype.Service" />
  <context:include-filter type="annotation"
    expression="org.springframework.stereotype.Component" />
</context:component-scan>
```

인코딩 필터 설정

- ▶ 한글 처리를 위해 다음과 같은 필터 설정을 사용 (in web.xml)

```
<!-- Encoding Filter -->
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>

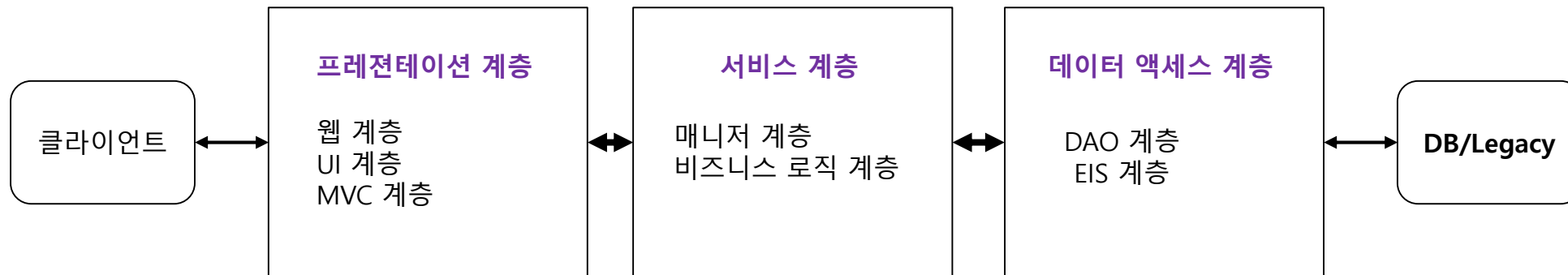
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Application Architecture

Application Architecture

: 3-tier layers



▶ 스프링에서는

- ▶ 3계층은 스프링을 사용하는 엔터프라이즈 애플리케이션에서 가장 많이 사용되는 구조
- ▶ 스프링 주요 모듈과 기술은 3계층 구조에 맞게 설계
- ▶ 논리적 개념이므로 상황과 조건에 따라 언제든지 달라질 수 있음

Application Architecture

: Example

[예제] myportal

▶ 비즈니스 분석(사용자 스토리 도출)

- 1) 사용자는 회원 가입을 한다
- 2) 사용자는 로그인을 한다
- 3) 사용자는 로그아웃을 한다
- 4) 방문자는 방명록에 글을 남긴다
- 5) 방문자는 방명록에 있는 자신의 글을 삭제한다
- 6) 로그인한 사용자는 자신의 정보를 수정한다
- 7) 로그인한 사용자는 자신의 게시판에 글을 작성한다
- 8) 로그인한 사용자는 자신의 글을 수정한다
- 9) 로그인한 사용자는 자신의 글을 삭제한다
- 10) 로그인한 사용자는 다른 사람의 글에 댓글을 달 수 있다
- 11) 사용자는 게시판 글의 목록을 볼 수 있다
- 12) 사용자는 게시판 글을 읽을 수 있다

Application Architecture

: Example

[예제] myportal

▶ 서비스의 정의

- 1) 사용자는 회원 가입을 한다 (join)
- 2) 사용자는 로그인을 한다 (login)
- 3) 사용자는 로그아웃을 한다 (logout)
- 4) 로그인한 사용자는 자신의 정보를 수정한다 (modifyInfo)
- 5) 방문자는 방명록에 글을 남긴다 (write)
- 6) 방문자는 방명록에 있는 자신의 글을 삭제한다 (remove)
- 7) 로그인한 사용자는 자신의 게시판에 글을 작성한다 (write)
- 8) 로그인한 사용자는 자신의 글을 수정한다 (modify)
- 9) 로그인한 사용자는 자신의 글을 삭제한다 (remove)
- 10) 로그인한 사용자는 다른 사람의 글에 댓글을 달 수 있다 (write)
- 11) 사용자는 게시판 글의 목록을 볼 수 있다 (list)
- 12) 사용자는 게시판 글을 읽을 수 있다 (view)



UserService

GuestbookService

BoardService

Application Architecture

: Example

[예제] myportal

▶ 서비스의 인터페이스 정의

```
public interface UserService {  
    void join();  
    void login();  
    void logout();  
    void modifyInfo();  
}
```

```
public interface GuestbookService {  
    void write();  
    void remove();  
}
```

```
public interface BoardService {  
    void write();  
    void remove();  
    void modify();  
    void list();  
    void view();  
}
```

Application Architecture

: Example

