스프링 프레임워크(Spring Framework)는 자바 플랫폼을 위한 오픈소스 애플리케이션 프레임워크로서 간단히 스프링(Spring)이라고도 불린다. 동적인 웹 사이트를 개발하기 위한 여러 가지 서비스를 제공하고 있다.

대한민국 공공기관의 웹 서비스 개발 시 사용을 권장하고 있는 전자정부 표준프레임워크의 기반 기술로서 쓰이고 있다.

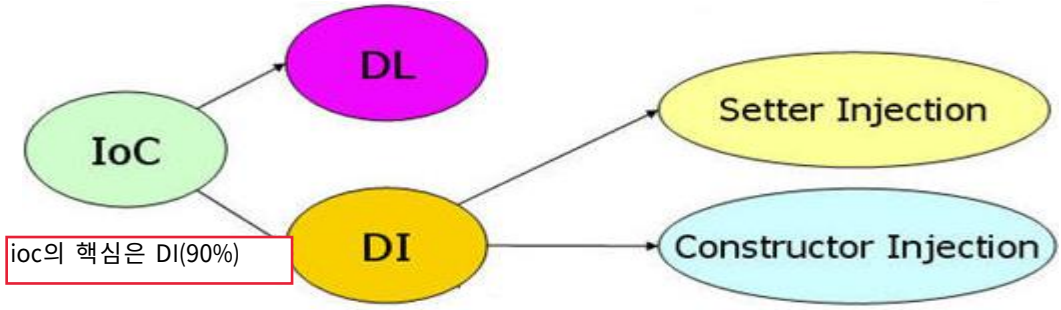
스프링은 다양한 서비스를 제공하는 프레임워크이지만 여기서는 DI와 MVC만 학습한다.

Spring IoC	Spring MVC

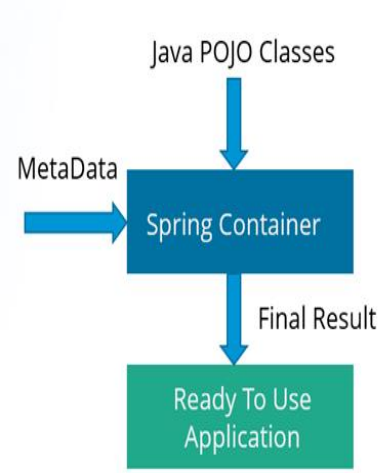
[Spring IoC]

프로그램에서 필요한 객체 생성을 Spring FW에서 하고 객체를 필요로 하는 곳에 주입하는 것과 객체를 찾을 때 제공하는 것 모두 Spring FW이 대신 처리한다. Spring FW에 의해 관리되는 Java 객체를 "bean" 이라고 부르며 이 일을 담당하는 Spring FW의 구성요서를 IoC 컨테이너라고 한다.

Spring DI는 객체간의 결합도를 느슨하게 하는 스프링의 핵심 기술이다.



ioc의 핵심은 DI(90%)



- Spring DI 컨테이너 초기화

```
ApplicationContext context
    = new ClassPathXmlApplicationContext("빈 설정 파일");
```

- DL의 예

```
타입명 bean=(타입명)context.getBean("빈이름");
```

- DI의 예

1. Construction Injection : 생성자를 통해서 객체 바인딩(의존관계를 연결)
2. Setter Injection : setter메서드를 이용해서 객체 바인딩(의존관계를 연결)

setter메서드에 넣어주는것

[DI 설정]

- XML 설정

설정 정보를 변경할 때는 XML만 변경하면 된다.
많은 프레임워크의 라이브러리가 XML 스키마를 이용한 설정의 편리함을 지원한다.
프로젝트에 규모에 따라서 XML 문서의 내용이 많아지게 된다.
코드를 실행해야 설정 정보의 오류를 확인 할 수 있다.

- ANNOTATION 설정

소스안에 정해진 ANNOTATION 들을 사용한다.

Spring 컨테이너가 어떻게 객체를 만들것인지 알려주려면
XML / 에노테이션 으로 하는 방법 이렇게 두가지가 있다.

[XML 설정]

<bean> : Spring DI 컨테이너가 관리할 Bean 객체를 설정
- id : 주입 받을 곳에서 호출할 이름 설정
- class : 주입할 객체의 클래스
- factory-method : Singleton패턴으로 작성된 객체의 factory 메소드 호출 시

Constructor를 이용한 객체간의 관계 또는 값을 주입

- 객체 또는 값을 생성자를 통해 주입 받는다. 한번에 여러 개 설정 가능하다.
- <constructor-arg> : <bean>의 하위태그로서
  다른 bean 객체 또는 값을 생성자를 통해 주입하도록 설정한다.

[하위태그 이용]
- <ref bean="bean name"/> => 객체를 주입 시
- <value>값</value> => 문자(String), Primitive data 주입 시
  type 속성 : 값을 1차로 String으로 처리한다. 값의 타입을 명시해야 하는 경우
  사용 예) <value type="int">10</value>

[속성 이용]
- ref="bean 이름"
- value="값"

setter 메소드를 통한 객체간의 관계 또는 값을 입

- setter를 통해서는 하나의 값만 설정할 수 있다.
- <property> : <bean>의 하위태그로 다른 bean 객체 또는 값을 property를 통해 주입하도록 설정
- name 속성 : 객체 또는 값을 주입할 property 이름을 설정한다.(setter의 이름)

설정방법
1) <ref>, <value>와 같은 하위태그를 이용하여 설정
2) 속성을 이용해 설정

[ANNOTATION 설정]

@Component

클래스에 선언하며 해당 클래스를 자동으로 bean으로 등록한다.
bean의 이름은 해당클래스명(첫글자는 소문자)이 사용된다.
범위는 디폴트로 singletonoi며 @Scope를 사용하여 지정할 수 있다.
소스안에 작성된 어노테이션이 적용되려면 다음과 같은 태그들이 설정파일에 정의되어 있어야 한다.

<context:annotation-config> - @Autowired 만 사용했을 때
<context:component-scan base-package="xxx" /> - 모든 어노테이션

이것으로 하면 모든 어노테이션을 할 수 있다.

@Scope

스프링은 기본적으로 빈의 범위를 "singleton" 으로 설정한다. "singleton" 이 아닌 다른 범위를 지정하고 싶다면 @Scope 어노테이션을 이용하여 범위를 지정한다.
설정 : prototype, singleton, request, session, globalSession

```
@Component
@Scope(value="prototype")
public class Worker {
        :
}
```

@Autowired

오토 와이어링 어노테이션은 Spring에서 의존관계를 자동으로 설정할때 사용된다. 이 어노테이션은 생성자, 필드, 메서드 세곳에 적용이 가능하며, 타입을 이용한 프로퍼티 자동 설정기능을 제공한다. 즉, 해당 타입의 빈 객체가 없으면 예외를 발생시킨다.
프로퍼티 설정 메서드(ex: setXXX()) 형식이 아닌 일반메서드에도 적용가능하다. 프로퍼티설정이 필수가 아닐경우 @Autowired(required=false)로 선언한다.(기본값은 true) byType으로 의존관계를 자동으로 설정할 경우 같은 타입의 빈이 2개 이상 존재하게 되면 예외가 발생하는데, Autowired도 이러한 문제가 발생한다. 이럴 때 @Qualifier를 사용하면 동일한 타입의 빈 중 특정 빈을 사용하도록 하여 문제를 해결할 수 있다.

```
@Autowired
@Qualifier("test")
private Test test;
```

@Autowired : 스프링 전용, type으로 찾는다.
객체의 이름 지정하려면 @Qualifier 애노테이션을 추가로 사용한다.

@Qualifier

@Autowired 어노테이션과 함께 사용된다. 빈의 타입이 아닌 이름으로 주입하려는 경우 사용된다.

@Resource

자바 6버전 및 JEE5 버전에 추가된 것으로 어플리케이션에서 필요로 하는 자원을 자동 연결할 때 사용 한다. 스프

다른 프레임웍에서도 사용하게 했다.

링 2.5 부터 지원하는 어노테이션으로 스프링에서는 의존하는 빈 객체를 전달할 때 사용하다. @Autowired 와 동일한 기능을 하며 @Autowired와 차이점은 @Autowired는 타입으로(by type), @Resource는 이름으로(by name)으로 연결시켜준다는 것이다. 설정 파일에서 <context:annotation-config> 태그를 사용해야 인식하며 name속성에 자동으로 연결될 빈객체의 이름을 입력한다.

```
@Resource(name="testDao")
```

@Inject

JSR-330 표준 Annotation으로 Spring 3 부터 지원하는 Annotation이다. 특정 Framework에 종속되지 않은 어플리케이션을 구성하기 위해서는 @Inject를 사용할 것을 권장한다. @Inject를 사용하기 위해서는 JSR-330 라이브러리인 javax.inject-x.x.x.jar 파일이 추가되어야 한다.

@Autowired, @Resource, @Inject 비교

@Autowired, @Resource, @Inject를 사용할 수 있는 위치는 다음과 같이 약간의 차이가 있으므로 필요에 따라 적절히 사용하면 된다.

@Autowired : 멤버변수, setter 메서드, 생성자, 일반 메서드에 적용 가능
@Resource : 멤버변수, setter 메서드에 적용가능
@Inject : 멤버변수, setter 메서드, 생성자, 일반 메서드에 적용 가능

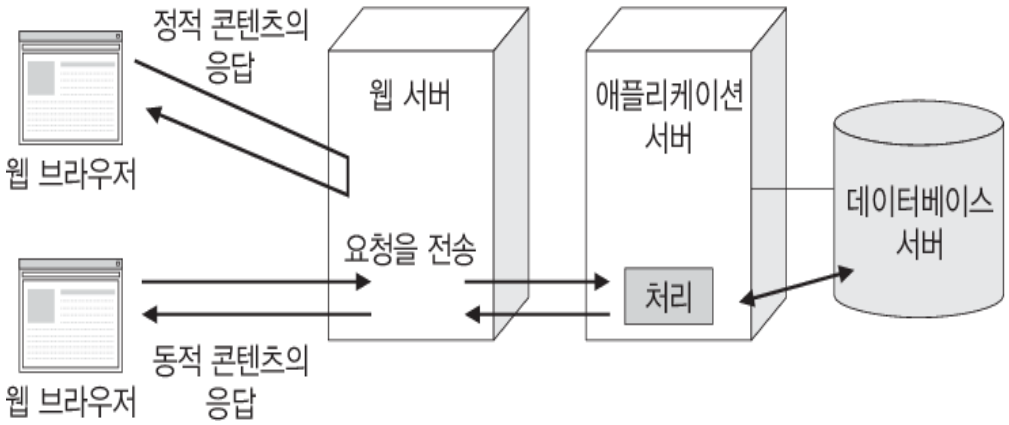
@Autowired, @Resource, @Inject를 멤버변수에 직접 정의하는 경우 별도 setter 메서드는 정의하지 않아도 된다.

표준

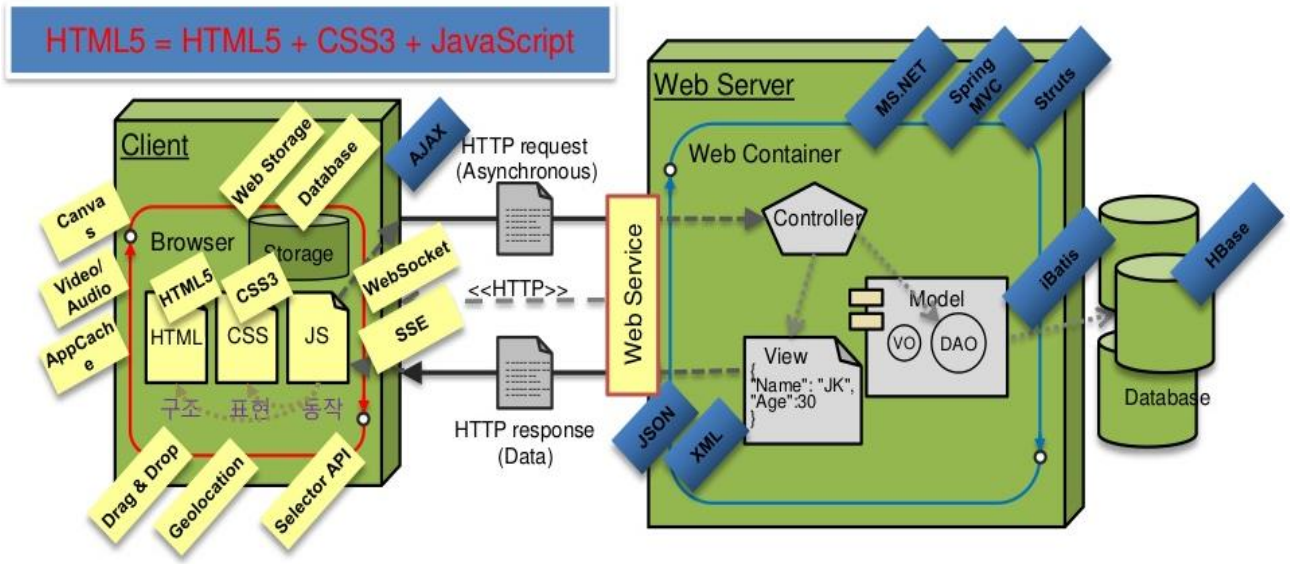
	@Autowired	@Inject	@Resource
범용	스프링 전용	자바에서 지원	자바에서 지원
연결방식	타입에 맞춰서 연결	타입에 맞춰서 연결	이름으로 연결

[웹 애플리케이션이란]

복수의 사용자가 인터넷을 통해 DB에 접근하고 안전하게 정보를 읽고 쓸 수 있게 지원하는 애플리케이션이다.

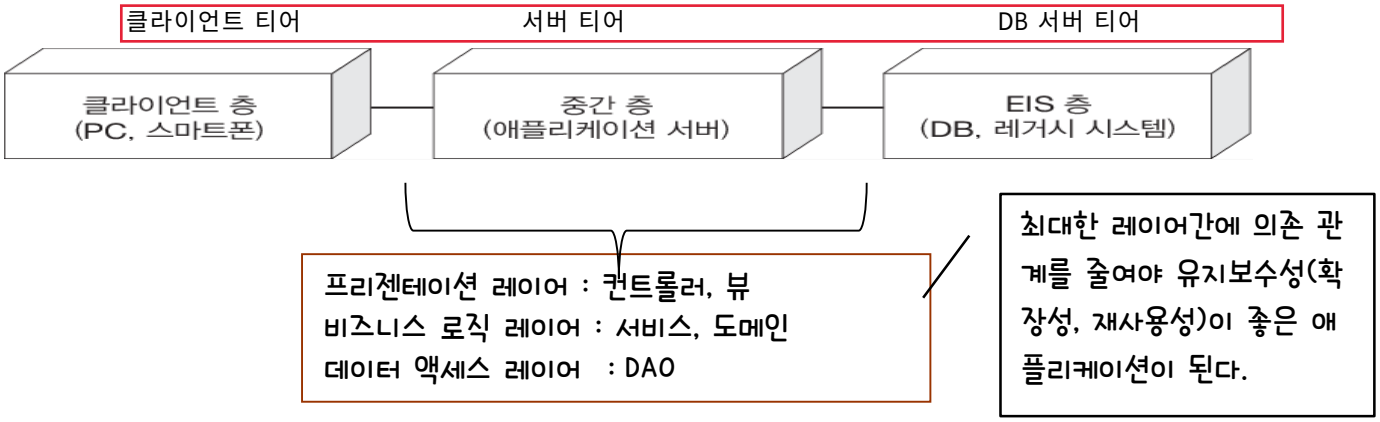


성공적인 시스템을 개발하는데 가장 중요한 요소 중 하나가 어플리케이션의 전체적인 구조이다. 다음은 최근에 많이 선택되고 있는 웹 어플리케이션 아키텍처를 소개하는 그림이다.



[웹 어플리케이션 구조]

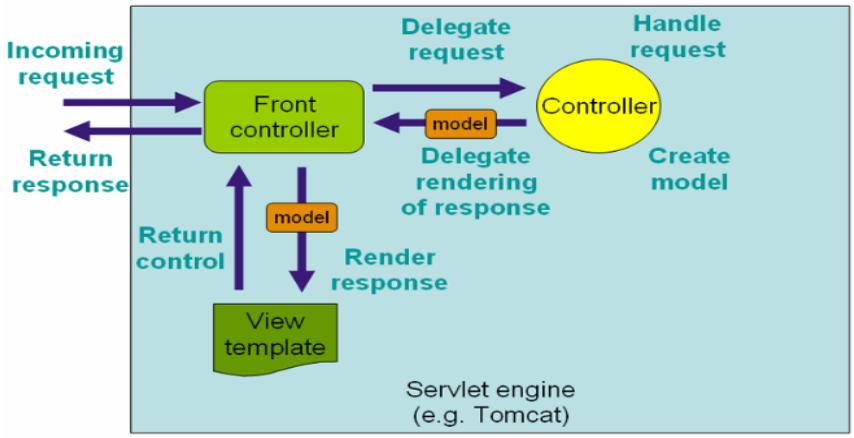
티어 : 어플리케이션의 구조를 물리적으로 나눈 것
레이어 : 어플리케이션의 구조를 논리적으로 나눈 것



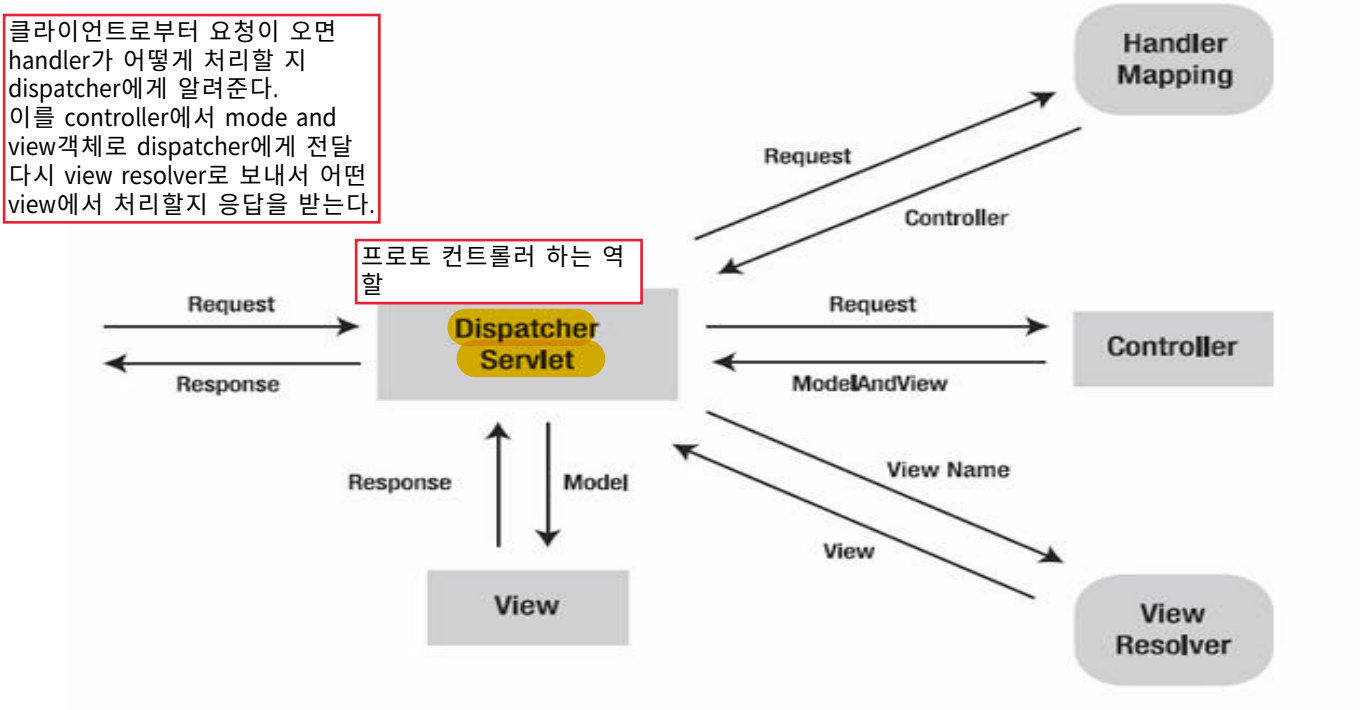
[스프링 MVC 처리 흐름]

스프링 MVC는 프론트 컨트롤러 패턴을 적용한다. 프론트 컨트롤러 패턴이란, 하나의 핸들러 객체를 통해서 요청을 할당하고, 일관된 처리를 작성할 수 있게 하는 개발 패턴이다.

컨트롤러가 두개로 나뉘었다.
front-controller, controller로 나뉘어져 있다. 클라이언트 요청을 제일 먼저 받는것이 front



클라이언트로부터 요청이 오면 handler가 어떻게 처리할 지 dispatcher에게 알려준다.
이를 controller에서 mode and view객체로 dispatcher에게 전달
다시 view resolver로 보내서 어떤 view에서 처리할지 응답을 받는다.



브라우저로부터 받은 요청은 스프링 MVC가 제공하는 DispatcherServlet클래스가 모두 관리한다.

web.xml파일에 다음과 같이 설정 내용을 추가한다.

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
<!-- 한글 처리 -->
<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>
</filter>
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

위와 같이 설정을 추가하는 경우 스프링 MVC용 설정파일의 이름은 "springmvc-servlet.xml"이 된다.

[Spring MVC 구현에서 사용되는 다양한 애노테이션]

@Controller

spring MVC의 Controller 클래스 선언을 단순화시켜준다. 스프링 컨트롤러, 서블릿을 상속할 필요가 없으며, @Controller로 등록된 클래스 파일에 대한 bean을 자동으로 생성해준다.

Controller로 사용하고자 하는 클래스에 @Controller 지정해 주면 component-scan으로 자동 등록된다.

<context:component-scan base-package="패키지명"/>

※ 컨트롤러 메서드의 파라미터 타입

HttpServletRequest 쿼리문자열 직접 추출하려면	
, HttpServletResponse, HttpSession	Servlet API
java.util.Locale	현재 요청에 대한 Locale
InputStream, Reader	요청 컨텐츠에 직접 접근할 때 사용
OutputStream, Writer	응답 컨텐츠를 생성할 때 사용
@PathVariable 어노테이션 적용 파라미터	URI 템플릿 변수에 접근할 때 사용
@RequestParam 어노테이션 적용 파라미터	HTTP 요청 파라미터를 매핑
@RequestHeader 어노테이션 적용 파라미터	HTTP 요청 헤더를 매핑
@CookieValue 어노테이션 적용 파라미터	HTTP 쿠키 매핑
@RequestBody 어노테이션 적용 파라미터	HTTP 요청의 몸체 내용에 접근할 때 사용, HttpMessage Converter를 이용 HTTP 요청 데이터를 해당 타입으로 변환한다.
Map, Model, ModelMap	뷰에 전달할 모델 데이터를 설정할 때 사용
커맨드 객체	HTTP 요청 파라미터를 저장한 객체. 기본

	적으로 클래스 이름을 모델명으로 사용. @ModelAttribute 어노테이션을 사용 하여 모델명을 설정할 수 있다.
Errors, BindingResult	HTTP 요청 파라미터를 커맨드 객체에 저장 한 결과. 커맨드 객체를 위한 파라미터 바로 다음에 위치
SessionStatus	폼 처리를 완료 했음을 처리하기 위해 사용. @SessionAttribute 어노테이션을 명시한 session 속성을 제거하도록 이벤트를 발생시킨다.

※ 컨트롤러 메서드의 리턴 타입

ModelAndView	뷰 정보 및 모델 정보를 담고 있는 ModelAndView 객체
Model	뷰에 전달할 객체 정보를 담고 있는 Model을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다. (RequestToViewNameTranslator를 통해 뷰 결정)
Map, ModelMap	뷰에 전달할 객체 정보를 담고 있는 Map 혹은 ModelMap을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다. (RequestToViewNameTranslator를 통해 뷰 결정)
String	뷰 이름을 리턴한다.
View 객체	View 객체를 직접 리턴. 해당 View 객체를 이용해서 뷰를 생성한다.
void	메서드가 ServletResponse나 HttpServletResponse 타입의 파라미터를 갖는 경우 메서드가 직접 응답을 처리한다고 가정한다. 그렇지 않을 경우 요청 URL로부터 결정된 뷰를 보여준다. (RequestToViewNameTranslator를 통해 뷰 결정)
@ResponseBody 어노테이션 적용	
	메서드에서 @ResponseBody 어노테이션이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다. HttpMessageConverter를 이용해서 객체를 HTTP 응답 스트림으로 변환한다.

@Service

@Service를 적용한 Class는 비지니스 로직이 들어가는 Service로 등록이 된다.

Controller에 있는 @Autowired는 @Service("xxxService")에 등록된 xxxService와 변수명이 같아야 하며
Service에 있는 @Autowired는 @Repository("xxxDao")에 등록된 xxDao와 변수명이 같아야 한다.

```
@Service(“myHelloService")
  public class HelloServiceImpl implements HelloService {
     @Autowired
     private HelloDao helloDao;
```

```
    public void hello() {
      System.out.println("HelloServiceImpl :: hello()");
      helloDao.selectHello();
    }
}
```

@RequestMapping

RequestMapping 어노테이션은 url을 class 또는 method와 mapping 시켜주는 역활을 한다.
이 어노테이션을 사용하면 Controller등록을 위한 url bean 설정을 생략 할 수 있다.
class에 하나의 url mapping을 할 경우, class위에 @RequestMapping("/url")을 지정하며, GET 또는 POST 방식 등의 옵션을 줄 수 있다.
해당되는 method가 실행된 후, return 페이지가 따로 정의되어 있지 않으면 RequestMapping("/url")에서 설정된 url로 다시 돌아간다.

[상세 속성 정보]

value : "value='/getMovie.do'"와 같은 형식의 매핑 URL 값이다. 디폴트 속성이기 때문에 value만 정의하는 경우에는 'value='은 생략할 수 있다.

method : GET, POST, HEAD 등으로 표현되는 HTTP Request method에 따라 requestMapping을 할 수 있다. 'method=RequestMethod.GET' 형식으로 사용한다. method 값을 정의하지 않는 경우 모든 HTTP Request method에 대해서 처리한다. value 값은 클래스 선언에 정의한 @RequestMapping의 value 값을 상속받는다.

params : HTTP Request로 들어오는 파라미터 표현이다. 'params={"param1=a", "param2", "!myParam"}' 로 다양하게 표현 가능하다.

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {  }

@RequestMapping("/hello.do")
public ModelAndView hello(){  }

@RequestMapping(value="/select1.do", method=RequestMethod.GET)
public String select() {   }

@RequestMapping(value="/insert1.do", method=RequestMethod.POST)
public String insert() {   }
```

@RequestParam

요청 파라미터(Query 문자열)를 메서드의 매게변수로 1:1 대응해서 받는 것이 @RequestParam 이다.

```
public String hello(@RequestParam("name") String name,
                @RequestParam(value="pageNo", required=false) String pageNo){    }


public ModelAndView seachInternal(
          @RequestParam("query") String query,
          @RequestParam("p") int pageNumber){    }

public String getAllBoards(@RequestParam(value="currentPage", required=false,
                                      defaultValue="1") int currentPage,Model model){    }


public String hello(String bookName,  int bookPrice){    }


public String check(@RequestHeader("User-Agent")String clientInfo) {
```

@ModelAttribute

요청 파라미터를 메서드의 매게변수로 1:1 대응해서 받는 것이 @RequestParam 이고, 도메인 오브젝트나 DTO 또는 VO의 프로퍼티에 요청 파라미터를 바인딩해서 한 번에 받으면 @ModelAttribute 이다. 하나의 오브젝트에 클라이언트의 요청 정보를 담아서 한 번에 전달되는 것이기 때문에 이를 커맨드 패턴에서 말하는 커맨드 오브젝트라고 부르기도 한다.

```
   @RequestMapping(value="/user/add", method=RequestMethod.POST)
   public String add(@ModelAttribute User user) {
      :
   }
```

@ModelAttribute 가 해주는 기능이 한 가지가 더 있데, 그것은 컨트롤러가 리턴하는 모델에 파라미터로 전달한 오브젝트를 자동으로 추가해 주는 것이다. 이때 모델의 이름은 기본적으로 파라미터의 이름을 따른다.

```
   public String update(@ModelAttribute("currentUser") User user) {
       ...
   }
```

위와 같이 정의하면 update() 컨트롤러가 DispatcherServlet 에게 돌려주는 모델 맵에는 "currentUser" 라는 키로 User 오브젝트가 저장되어 있게 된다.

@PathVariable

url의 특정 부분을 변수화 하는 기능을 지원하는 어노테이션이다.

@RequestMapping에서는 변수를 {}로 감싸주고, 메서드의 파라미터에 @PathVariable 을 지정하여 메서드에서 파라미터로 활용한다.

```
@RestController
public class HomeController {
        @RequestMapping("/{name}")
        public String home(@PathVariable String name) {
              return "Hello, " + name;
       }
}
```

```
/board/list_controller/1/test/듀크
@RequestMapping(value="/board/list_controller/{currentPage}/test/{name}")
 public String getAllBoards(@PathVariable(value="currentPage") int currentPage,
                                  @PathVariable(value="name") String name, Model model){
                :
     return "view페이지";
 }
```

@RequestParam 과 @PathVariable 비교

- @RequestParam

"/board/list_controller?currentPage=1"형태의 요청을 처리

```
@RequestMapping(value="/board/list_controller")
 public String getAllBoards(@RequestParam(value="currentPage", required=false,
                                     defaultValue="1") int currentPage,Model model){
     model.addAttribute("list",boardService.selectAll(currentPage));
     return "board_list";
 }
```

- @PathVariable

"/board/list_controller/1"형태의 요청을 처리

```
 @RequestMapping(value="/board/list_controller/{currentPage}")
 public String getAllBoards(@PathVariable(value="currentPage") int
                                                                  currentPage,Model model){
     model.addAttribute("list",boardService.selectAll(currentPage));
```

```
    return "board_list";
}
```

@RequestBody 와 @ReponseBody

웹 서비스와 REST 방식이 시스템을 구성하는 주요 요소로 자리 잡으면서 웹 시스템간에 XML이나 JSON 등의 형식으로 데이터를 주고 받는 경우가 증가하고 있다.
이에 따라 스프링 MVC도 클라이언트에서 전송한 XML 데이터나 JSON 또는 기타 데이터를 컨트롤러에서 DOM 객체나 자바 객체로 변환해서 받을 수있는 기능(수신)을 제공하고 있으며,
비슷하게 자바 객체를 XML이나 JSON 또는 기타 형식으로 변환해서 전송할 수 있는 기능(송신)을 제공하고 있다.

@RequestBody 어노테이션과 @ResponseBody 어노테이션은 각각 HTTP 요청 몸체를 자바 객체로 변환하고 자바 객체를 HTTP 응답 몸체로 변환하는 데 사용된다.
@RequestBody 어노테이션을 이용하면 HTTP 요청 몸체를 자바 객체로 전달받을 수 있다. 비슷하게
@ResponseBody 어노테이션을 이용하면 자바 객체를 HTTP 응답 몸체로 전송할 수 있다.

@RestController

@RestController 어노테이션은@Controller를 상속하여 @Controller + @ResponseBody
의 기능을 지원한다.
Restful 웹 서비스를 구현할 때 응답은 항상 응답바디(response body)에 보내져야 하는데
이를 위해 스프링4.0에서 특별히 @ResrController를 제공한다.
도메인객체를 Web Service로 노출 가능하며 각각의 @RequestMapping method에
@ResponseBody할 필요가 없어진다. 그러므로 Spring MVC에서 @ReponseBody를 이용하여
JSON or XML 포맷으로 데이터를 넘길 수 있다.

Spring Scheduling(TASK)

스프링에서는 특정 시간에 반복적으로 처리되는 코드를 스케줄링할 수 있다. 이 때 반복적으로 수행되는 코드를 Task 라고 한다.

[환경 설정]

springmvc-context.xml 파일에 스케줄링 기능과 관련된 태그를 추가한다.
<beans> 루트엘리먼트에 xmlns:task="http://www.springframework.org/schema/task"을 추가하고
xsi:schemaLocation 속성의 마지막 값으로
http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-3.0.xsd 을 추가한 다음 <task:annotation-driven/> 태그를 추가한다.

[Task 기능의 메서드 정의]
설정된 주기(스케줄링)에 맞춰서 호출되는 Task 메서드 앞에 @Scheduled 라는 애노테이션을 다음에 제시한 속성 중 하나를 정의하여 추가한다.

- cron : CronTab에서의 설정과 같이 cron="0/10 * * * * ?" 과 같은 설정이 가능하고
- fixedDelay : 이전에 실행된 Task의 종료시간으로 부터 정의된 시간만큼 지난 후 Task를 실행한다.
- fixedRate : 이전에 실행된 Task의 시작시간으로 부터 정의된 시간만큼 지난 후 Task를 실행한다.

Seconds	0 ~ 59
Minutes	0 ~ 59
Hours	0 ~ 23
Day of Month	1 ~ 31
Month	1 ~ 12
Day of Week	1 ~ 7 (1 => 일요일, 7=> 토요일 / MON,SUN...)
Years(optional)	1970 ~ 2099

*	모든수를 의미, Minutes 위치에 사용될 경우 매분마다 라는 뜻
?	Day of Month, Day of Week에만 사용 가능, 특별한 값이 없다는 뜻
-	기간을 설정, Hour 위치에 10 - 12 라고 쓰면 10, 11, 12dp 동작하라는 뜻
,	특정 시간을 설정. Day of Week 위치에 2, 4, 6 이라고 쓰면 월, 수, 금에만 동작하라는 뜻
/	증가를 표현, Seconds 위치에 0/15로 설정되어 있으면, 0초에 시작해서 15초 간격으로 동작 하라는 뜻
L	Day Of Month 에서만 사용하며, 마지막 날의 의미 Day of Month 에 L로 설정되어 있으면 그달 의 마지막날에 실행하라는 의미
W	Day of Month 에만 사용하며, 가장 가까운 평일을 의미. 15W로 설정되어 있고 15일이 토요일 이며, 가장 가까운 평일인 14일 금요일에 실행, 15일이 일요일이면 16일 월요일에 실행된다. 15일이 평일이면 그날 그대로 실행됨
LW	L과 W를 결합하여 사용, 그달의 마지막 평일의 의미
#	Day of Week에 사용, 6#3 의 경우 3번째 주 금요일에 실행된다.

```
package dao;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@Component
public class SpringSchedulerTest {
        @Scheduled(cron = "10 * * * * *")  // 매시간 10초
       //@Scheduled(cron = "0 0 7 * * 2") // 월요일 오전 7시
        //@Scheduled(fixedDelay = 5000) // 5초 간격으로 (5초에 한 번씩 : fixedRate)
        public void scheduleRun() {
          Calendar calendar = Calendar.getInstance();
          SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
          System.out.println("스케줄 실행 : " + dateFormat.format(calendar.getTime()));
        }
}
```

[오류 처리]

@ExceptionHandler와 @ControllerAdvice

@ExceptionHandler : 스프링 MVC에서는 에러나 예외를 처리하기 위한 특별한 방법을 제공하며 @ExceptionHandler 어노테이션을 이용하면 된다. 스프링 컨트롤러에서 정의한 메서드(@RequestMapping)에서 기술한 예외가 발생되면 자동으로 받아낼 수 있다. 이를 이용하여 컨트롤러에서 발생하는 예외를 View단인 JSP등으로 보내서 처리할 수 있다.

```
@Controller
public class MemberDetailController {
    @RequestMapping("/member/detail/{id}")
    public String detail(@PathVariable("id") Long memId, Model model) {
        Member member = memberDao.selectById(memId);
        if (member == null) {
            throw new MemberNotFoundException();
        }
        model.addAttribute("member", member);
        return "member/memberDetail";
    }
    @ExceptionHandler(TypeMismatchException.class)
    public String handleTypeMismatchException(TypeMismatchException ex) {
        return "member/invalidId";
    }
    @ExceptionHandler(MemberNotFoundException.class)
    public String handleNotFoundException() throws IOException {
        return "member/noMember";
    }
}
```

@ControllerAdvice : @ControllerAdvice는 스프링3.2 이상에서 사용가능하며 @Controller나 @RestController에서 발생하는 예외 등을 catch하는 기능을 가지고 있다. 클래스 위에 @ControllerAdvice를 붙이고 어떤 예외를 잡아낼 것인지 내부 메서드를 선언하여 메서드 상단에 @ExceptionHandler(예외클래스명.class) 와 같이 기술한다.

```
@ControllerAdivce(“edu.myspring")
public class CommonExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    private ModelAndView errorModelAndView(Exception ex) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("/error_common");
        modelAndView.addObject("exception", ex );
        return modelAndView;
    }
}
```