"스프링 프레임워크"는 자바 기반의 애플리케이션 프레임워크로 엔터프라이즈급 애플리케이션을 개발하기 위한 다양한 기능을 제공한다.

> 애플리케이션 개발에 필요한 기반을 제공해서
> 개발자가 비즈니스 로직 구현에만 집중할 수 있게 한다.

스프링의 특징과 구조

1. 제어역전(IoC)

일반적인 자바 개발에서 사용되는 코드 :

```
@Controller
public class NoDIController {
	private MyService service = new MyServiceImpl();

	@GetMapping("/xxx")
	public String getHello() {
		return service.getHello();
	}
}
```

사용하려는 객체를 선언하고 해당 객체의 의존성을 생성한 후 객체에서 제공하는 기능을 사용하는 코드이다. 객체를 생성하고 사용하는 일련의 작업을 개발자가 직접 제어한다.

제어역전을 특징으로 하는 스프링은 기존 자바 개발 방식과 다르게 동작한다. IoC를 적용한 환경에서는 사용할 객체를 직접 생성하지 않고 객체의 생명주기 관리를 '스프링 컨테이너'에 위임한다. 객체의 관리를 컨테이너에 맡겨 제어권이 넘어간 것을 제어 역전이라고 부른다. 제어 역전을 통해 의존성주입(DI), 관점지향프로그래밍(AOP) 등이 가능해진다. 이런 제어역전을 통해 제어권을 컨테이너에 넘김으로써 개발자는 비즈니스 로직을 작성하는데 더 집중할 수 있다.

2. 의존성 주입(DI)

의존성 주입이란, 제어 역전의 방법 중 하나로써 사용할 객체를 직접 생성하지 않고 외부 컨테이너가 생성한 객체를 주입받아 사용하는 방식이다.

- 생성자를 통한 의존성 주입
- 필드 객체 선언을 통한 의존성 주입
- setter 메서드를 통한 의존성 주입

스프링에서는 @Autowired라는 어노테이션을 통해 의존성을 주입할 수 있는데, 스프링 4.3이후 버전에서는 생성자를 통한 의존성 주입에서는 어노테이션의 생략이 가능하다.

의존성 주입으로 변경한 코드(constructor injection) :

```
@Controller
public class DIController {
    MyService myService;

    @Autowired
    public DIController(MyService myService) {
        this.myService = myService;
    }
}
```

필드 객체 선언을 통한 의존성 주입(field injection) :

```
@Controller
public class FieldInjectionController {
    @Autowired
    private MyService myService;
}
```

setter 메서드를 통한 의존성 주입(setter injection) :

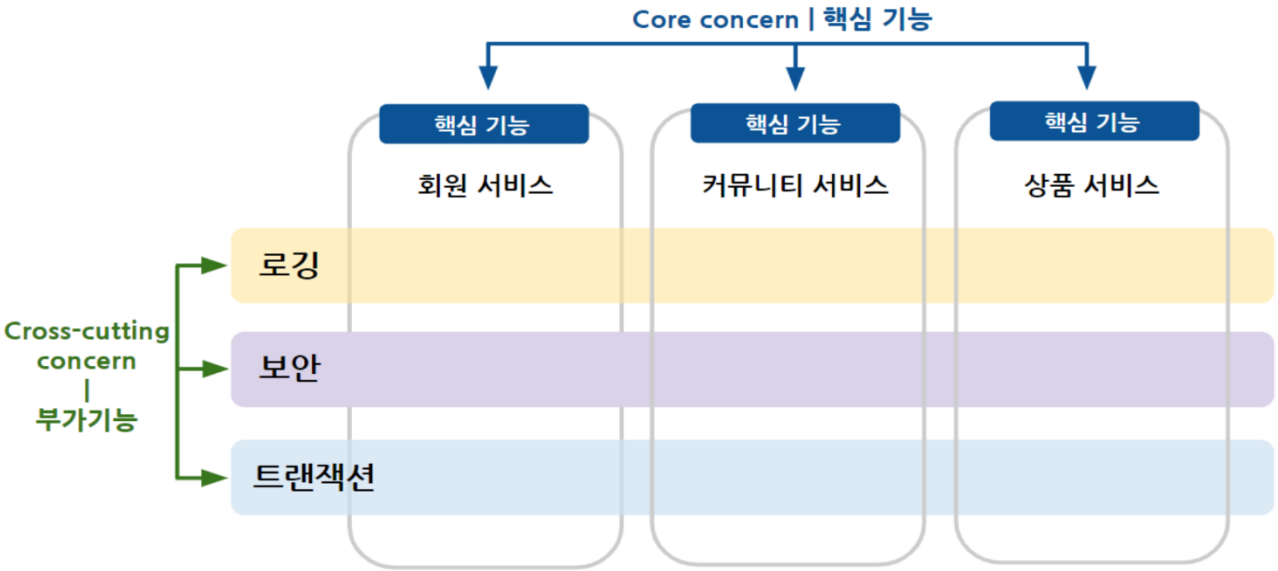
```
@Controller
public class SetterInjectionController {
    MyService myService;
    @Autowired
    public void setMyService(MyService myService) {
        this.myService = myService;
    }
}
```

3. 관점 지향 프로그래밍(AOP-Aspect Oriented Programming)

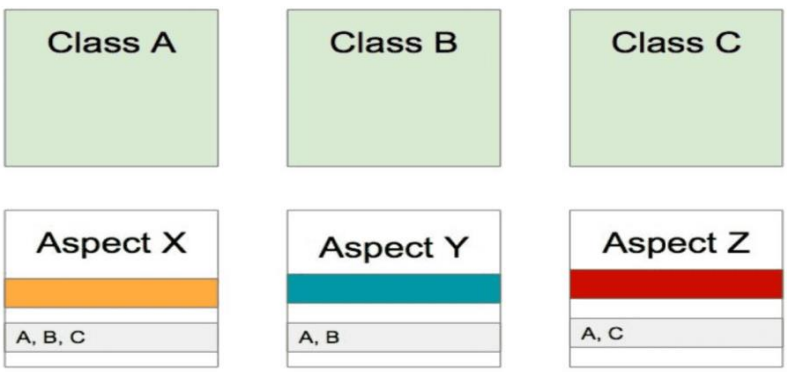
관점 - 어떤 기능을 구현할 때 그 기능을 핵심 기능과 부가 기능으로 구분한 각각의 기능

핵심 기능	부가 기능
비즈니스 로직이 처리하려는 목적 기능	여러 비즈니스 로직 사이에서 공통적이고 반복적으로 필요한 기능
회원서비스, 커뮤니티서비스, 상품서비스	로깅, 보안, 트랜잭션

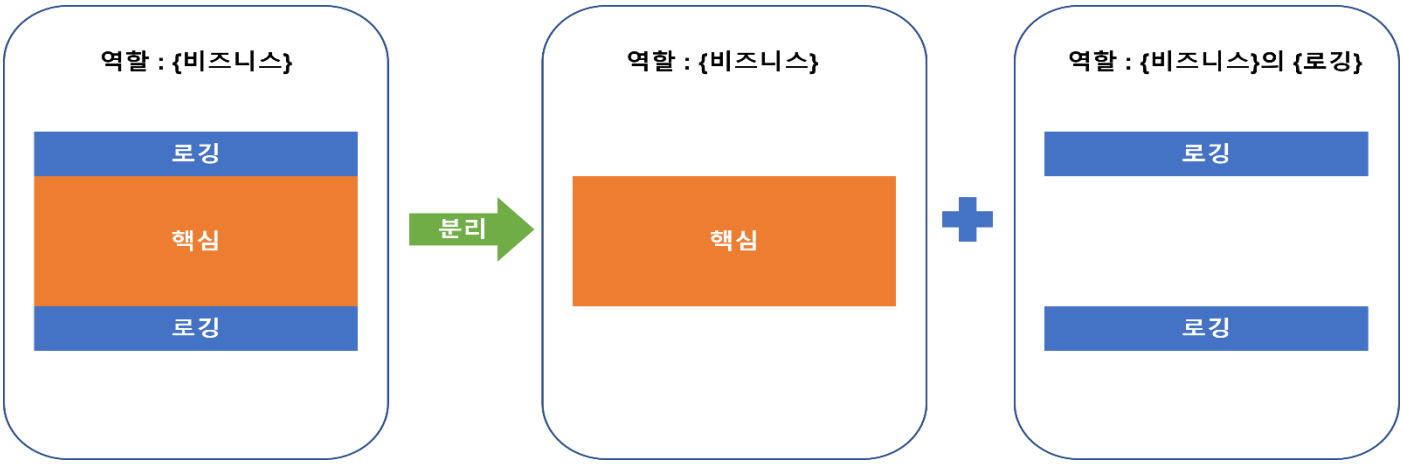
AOP는 어떤 기능을 구현할 때 각각을 하나의 관점으로 보며 그 관점을 기준으로 묶어서 개발하는 방식을 의미한다. 즉, 핵심 기능과 부가 기능을 나눠서 개발하는 것이다.



AOP를 적용하면?



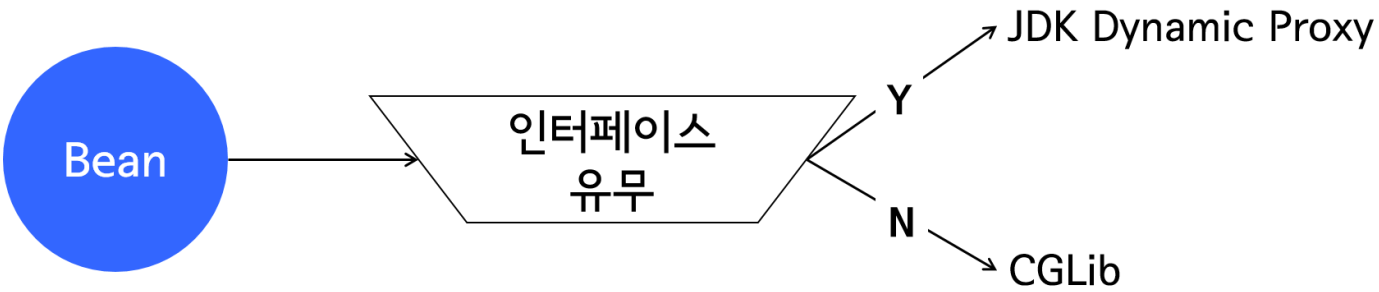
여러 비즈니스 로직에 반복되는 부가 기능을 하나의 공통 로직으로 처리하도록 모듈화 하여 삽입하는 방식을 AOP라고 한다.



[AOP를 구현하는 세가지 방법]

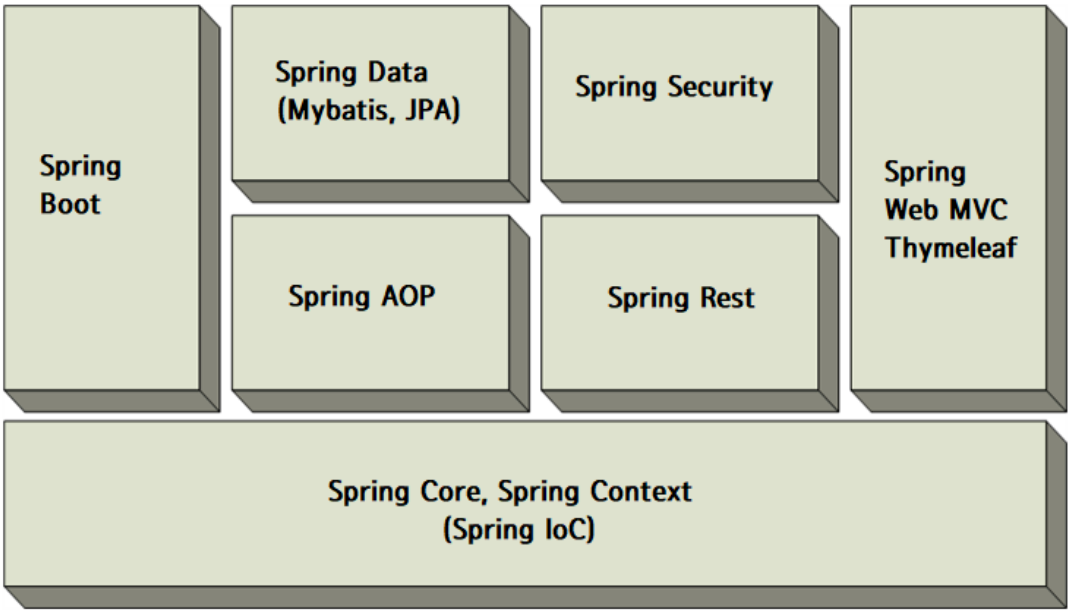
- 컴파일 과정에 삽입
- 바이트코드를 메모리에 로드하는 과정에서 삽입(LTW)
- 프록시 패턴을 이용

이 중에 스프링은 프록시 패턴과 LTW 를 지원한다.



4. 스프링 프레임워크의 모듈

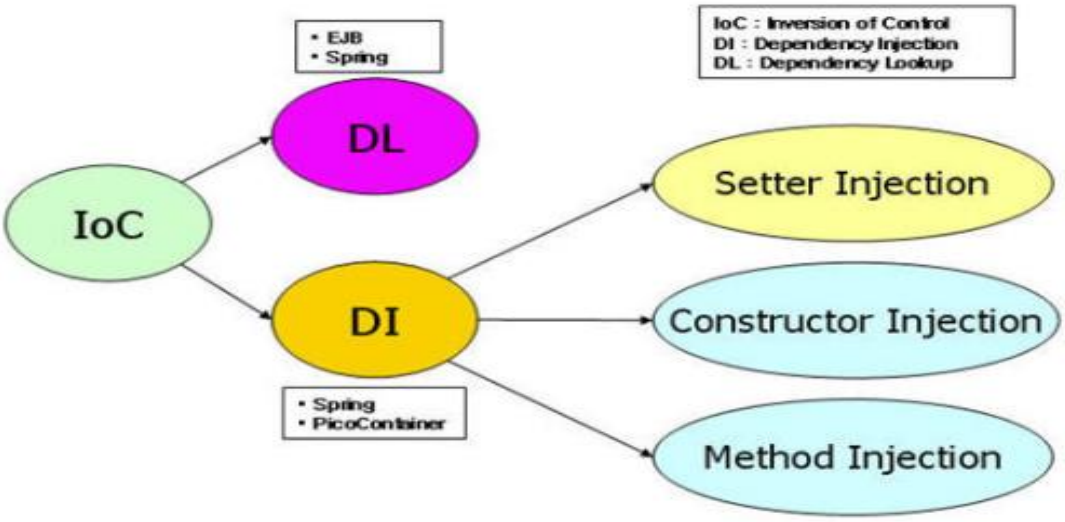
스프링 프레임워크는 기능별로 구분된 약 23개 정도의 모듈로 구성된다.



Spring IoC와 DI

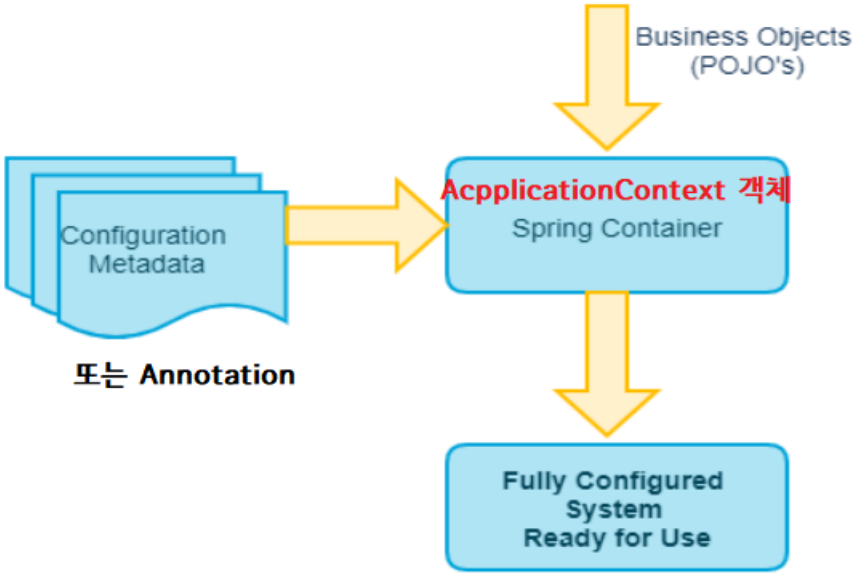
프로그램에서 필요한 객체의 생성을 Spring FW에서 하고 객체를 필요로 하는 곳에 주입하는 것 그리고 객체를 찾을 때 제공하는 것 모두 Spring FW이 대신 처리한다. Spring FW에 의해 관리되는 Java 객체를 "bean" 이라고 부르며 이 일을 담당하는 Spring FW의 구성요소를 IoC 컨테이너 또는 스프링 컨테이너라고 한다.

Spring DI는 객체간의 결합도를 느슨하게 하는 스프링의 핵심 기술이다.



- Spring IoC 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 : setter 메서드를 이용해서 객체 바인딩(의존관계를 연결)
3. method Injection : 어노테이션을 정의한 메서드를 이용해서 객체 바인딩(의존관계를 연결)
4. field Injection : 어노테이션을 정의한 메서드를 이용해서 객체 바인딩(의존관계를 연결)

[IoC 설정]

(1) XML 설정

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

(2) ANNOTATION 설정

소스안에 정해진 ANNOTATION 들을 사용한다.

[XML 설정]

<bean> 태그 :

Spring IoC 컨테이너가 관리할 Bean 객체(자바 클래스) 설정
id : 주입 받을 곳에서 호출할 이름
class : 주입할 객체의 클래스명
factory-method : 객체 생성시 사용될 factory 메서드
scope : Bean 객체의 유효 범위 설정(singleton, prototype 등)

<constructor-arg> 태그 :

<bean>의 하위태그로 다른 bean 객체 또는 값을 생성자를 통해 주입하도록 설정

<ref bean="bean name"/> => 객체를 주입 시
<value>값</value> => 문자(String), Primitive data 주입 시
type 속성 : 값의 타입을 명시해야 하는 경우

[속성 이용]
- ref="bean 이름"
- value="값"

<property> 태그 :

<bean>의 하위태그로 다른 bean 객체 또는 값을 setter 메서드를 통해 주입하도록 설정
- name 속성 : 객체 또는 값을 주입할 property 이름을 설정(setter의 이름)
<ref bean="bean name"/> => 객체를 주입 시
<value>값</value> => 문자(String), Primitive data 주입 시
type 속성 : 값의 타입을 명시해야 하는 경우

[속성 이용]
- ref="bean 이름"
- value="값"

[ANNOTATION 설정]

@Component

클래스에 선언하며 해당 클래스를 bean 객체로 등록한다.
bean의 이름은 해당 클래스명(첫 글자는 소문자로 변경해서)이 사용된다.
범위는 디폴트로 singleton이며 @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

@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

@Auowired 어노테이션은 Spring에서 의존관계를 자동으로 설정할 때 사용된다. 이 어노테이션은 생성자, 필드, 메서드 세 곳에 적용이 가능하며, 타입을 이용한 프로퍼티 자동 설정기능을 제공한다. 즉, 해당 타입의 빈 객체가 없으면 예외를 발생시킨다. 프로퍼티 설정 메서드(ex: setXXX()) 형식이 아닌 일반 메서드에도 적용 가능하다.

같은 타입의 빈이 2개 이상 존재하게 되면 예외가 발생하는데, Autowired도 이러한 문제가 발생한다. 이런 경우에는 @Qualifier를 사용하면 동일한 타입의 빈 중 특정 빈을 사용하도록 하여 문제를 해결할 수 있다. 설정이 필수가 아닐경우 @Autowired(required=false)로 선언한다.(기본값은 true)

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

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

@Qualifier

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

@Resource

자바 6 버전 및 JEE 5 버전에 추가된 것으로 어플리케이션에서 필요로 하는 자원을 자동 연결할 때 사용 한다. 스프링 2.5 부터 지원하는 어노테이션으로 스프링에서는 의존하는 Bean 객체를 전달할 때 사용하다. @Autowired 와 동일한 기능이며 @Autowired 와의 차이점은 @Autowired는 타입으로(by type), @Resource는 이름으로(by name)으로 연결시켜준다는 것이다. 설정 파일에서 <context:annotation-config> 태그를 사용해야 인식하며 name 속성에 자동으로 연결될 Bean 객체의 이름을 입력한다.

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

@Inject

JSR-330 표준 어노테이션으로 Spring 3 부터 지원한다. 특정 Framework에 종속되지 않은 어플리케이션을 구성하고자 하면 @Inject를 사용할 것을 권장한다. @Inject를 사용하기 위해서는 JSR-330 라이브러리인 javax.inject-x.x.x.jar 파일이 추가되어야 한다.

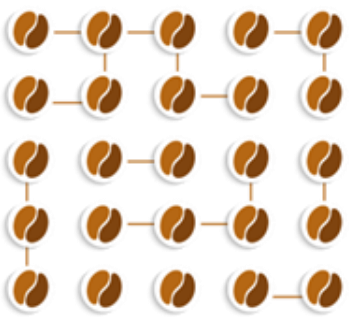
@Autowired, @Resource, @Inject 비교

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

@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 정의하는 경우 별도의 setter 메서드를 정의하지 않아도 된다.

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



스프링 부트(Spring Boot)란?

스프링은 기존 기술의 복잡성을 크게 줄인 프레임워크이지만, 그럼에도 불구하고 스프링을 사용하기 위해서는 초기에 여러 가지의 사항들을 설정해주어야 한다. 스프링 부트는 스프링으로 애플리케이션을 만들 때 필요한 초기 설정을 간편하게 처리해주는 별도의 프레임워크이다.

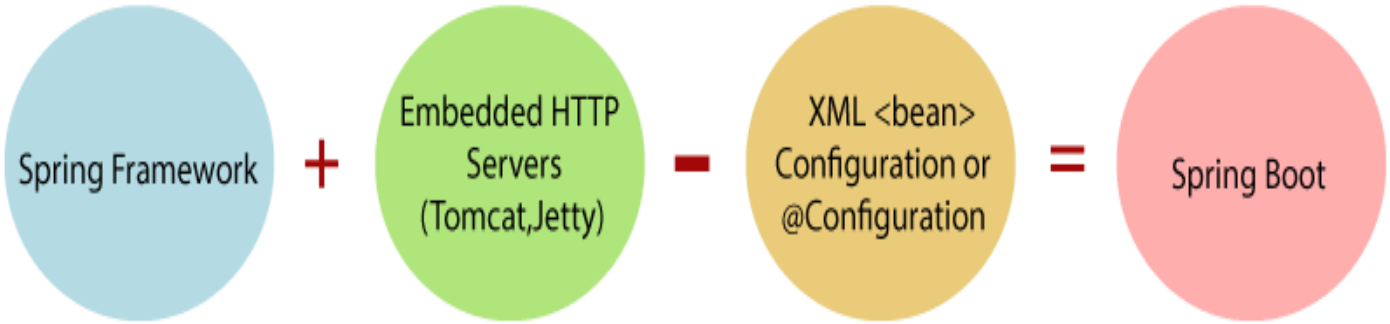
스프링 부트는 스프링 프레임워크에서의 기존 개발 방식의 문제와 한계를 극복하기 위해 다양한 기능을 제공한다. 스프링은 기능이 많은 만큼 설정이 복잡하다는 단점이 있는데 수많은 모듈을 설치하다보면 문제가 발생하기 마련이다. 이 문제점을 해결해 준 것이 바로 스프링부트이다.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jc
p.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-config/applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config/dispatcher-servlet.xml</param-va
lue>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <filter>
        <filter-name>CORSFilter</filter-name>
        <filter-class>com.codestates.filter.CORSFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>CORSFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

```
spring.h2.console.enabled=true
spring.h2.console.path=/console
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
```

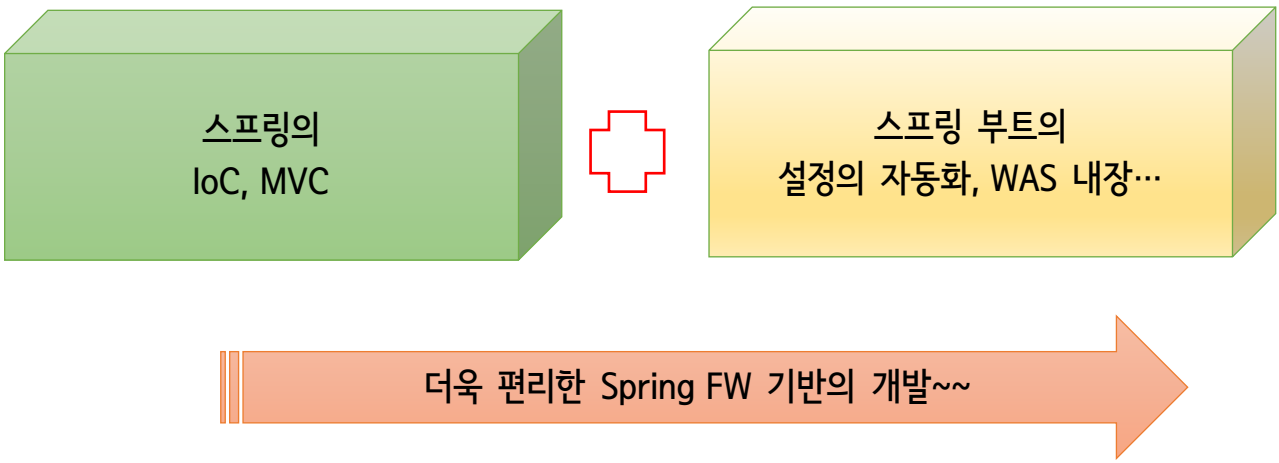
스프링 부트를 사용하면 초기 설정을 간편하게 할 수 있는 것 외에도 몇 가지 장점이 있다. 기존에는 배포를 할 때에 별도의 외장 웹 서버를 설치하고, 프로젝트를 War 파일로 빌드하여 배포를 진행했는데, 이러한 방식은 처리 속도가 느리며 번거롭다는 단점을 가진다. 반면, 스프링 부트는 자체적인 웹 서버를 내장하고 있어, 빠르고 간편하게 배포를 진행할 수 있다. 또한, 스프링 부트를 사용하면 독립적으로 실행 가능한 Jar 파일로 프로젝트를 빌드할 수 있어, 클라우드 서비스 및 도커와 같은 가상화 환경에 빠르게 배포할 수 있다.



Spring
Java 기반의 애플리케이션 개발을 위한 오픈소스 프레임워크

Spring MVC
웹 애플리케이션 개발에 있어 MVC 패턴을 적용할 수 있도록 Spring에서 제공하는 프레임워크

Spring Boot
Spring 설정들을 자동화하는 Spring 기반의 프레임워크



[스프링 부트의 특징]

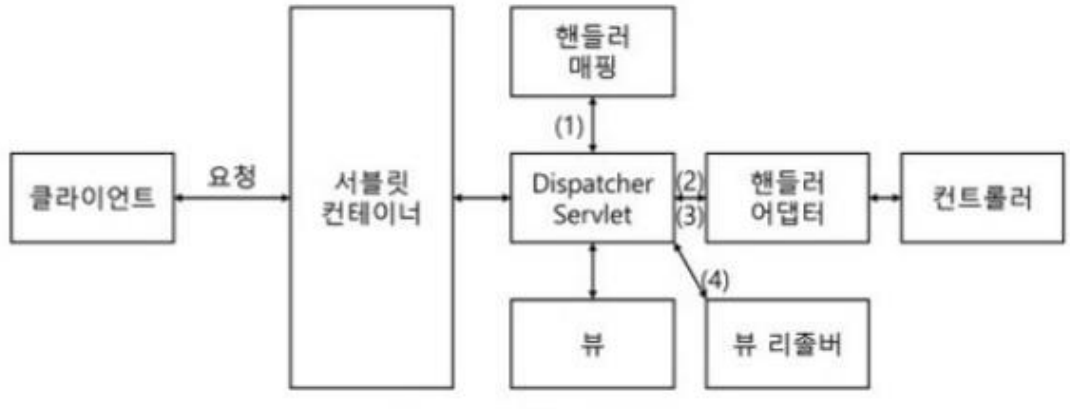
1. 의존성 관리
스프링에서는 개발에 필요한 각 모듈의 의존성을 직접 설정하지만 스프링부트는 'spring0boot-starter'라는 것을 통해 의존성을 제공해주고 서로 호환되는 버전의 모듈 조합을 제공한다.

2. 자동 설정
스프링 부트는 애플리케이션에 추가된 라이브러리를 실행하는 데 필요한 환경설정을 자동으로 해준다.

3. 내장 WAS
스프링부트의 각 웹 애플리케이션에는 내장 Web Application Server가 존재한다.
가장 기본이 되는 의존성인 'spring-boot-starter-web'의 경우 '톰캣'을 내장한다.
스프링부트의 자동 설정 기능을 통해 특별한 설정 없이도 톰캣을 실행할 수 있으며 필요에 따라서는 다른 웹서버로도 대체할 수 있다.

4. 모니터링
스프링 부트에는 스프링 부트 액추에이터라는 자체 모니터링ㅍ 도구가 있어서 서비스 운영에서 필요한 요소들을 모니터링 가능하다.

[스프링 부트의 동작 방식]
스프링 부트에서 spring-boot-starter-web 모듈을 사용하면 기본적으로 톰캣을 사용하는 스프링 MVC 구조를 기반으로 동작한다.



1. 서블릿
클라이언트의 요청을 처리하고 결과를 반환하는 자바 웹 프로그래밍 기술, 서블릿 컨테이너에서 서블릿 인스턴스를 생성하고 관리하는 역할을 하며 톰갯은 WAS의 역할과 서블릿 컨테이너의 역할을 수행하는 대표적인 컨테이너이다.

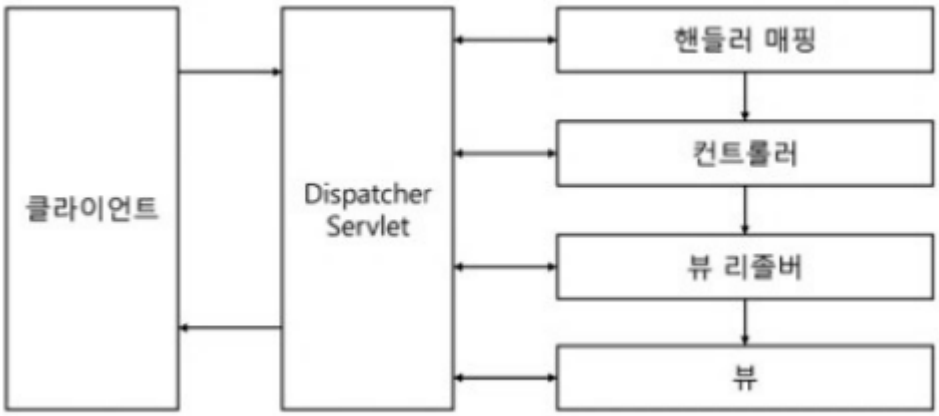
- 서블릿 컨테이너의 특징
    서블릿 객체를 생성, 초기화, 호출, 종료하는 생명주기를 관리
    서블릿 객체는 싱글톤 패턴으로 관리
    멀티 스레딩 지원

스프링에서는 DispathcerServlet이 서블릿의 역할을 수행한다.

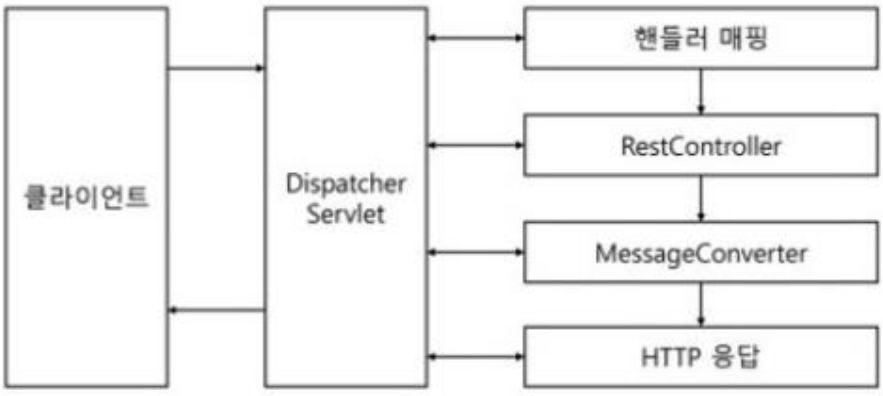
2. 동작구조

DispatcherServlet으로 요청이 들어오면 DispatcherServlet은 핸들러 매핑을 통해 요청 URI에 매핑된 컨트롤러를 탐색한다. 핸들러 어댑터로 컨트롤러를 호출하고 핸들러 어댑터에 컨트롤러의 응답이 돌아오면 ModelAndView로 응답을 가공해 반환한다. 뷰 형식으로 리턴하는 컨트롤러를 사용할 때는 View Resolver를 통해 View를 받아 리턴한다.

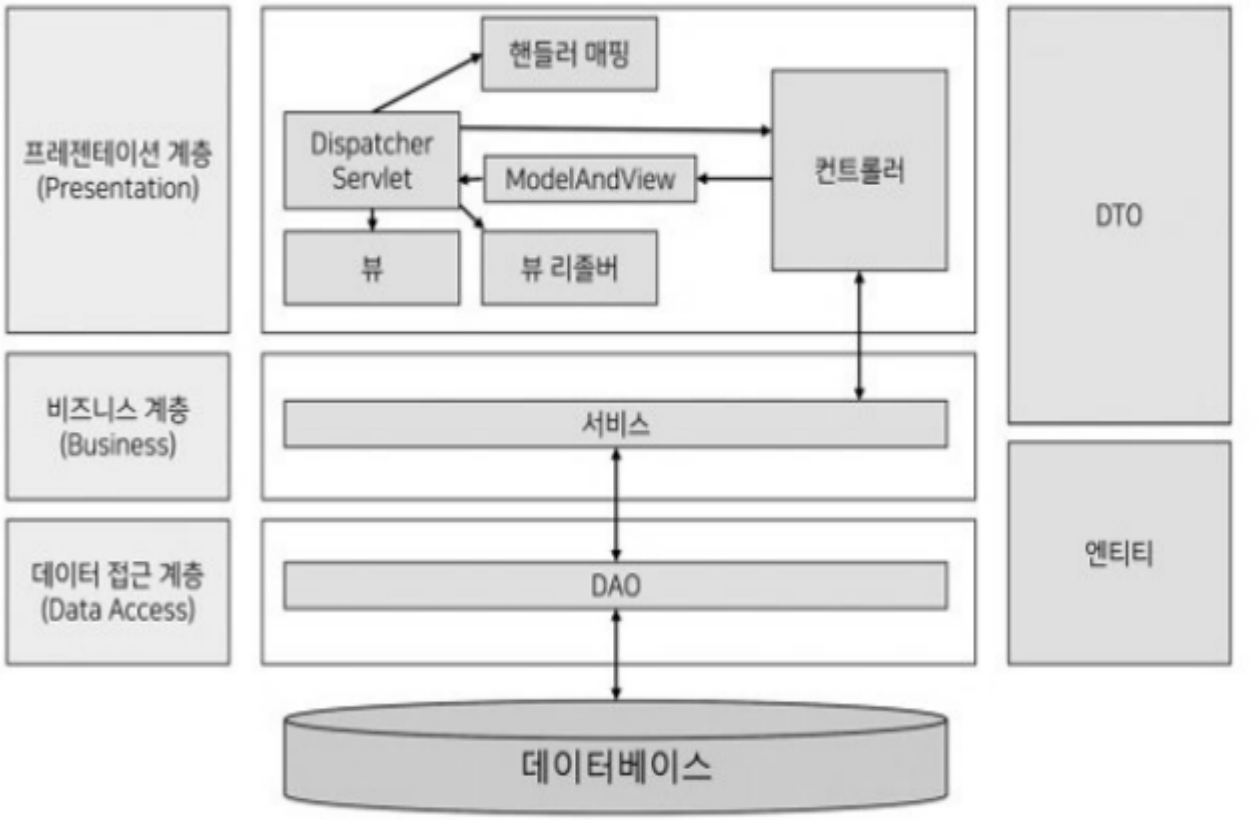
View를 사용하는 DispathcerServlet의 동작 방식



@RestController를 사용하는 DispathcerServlet의 동작 방식



[스프링의 레이어드 아키텍처]



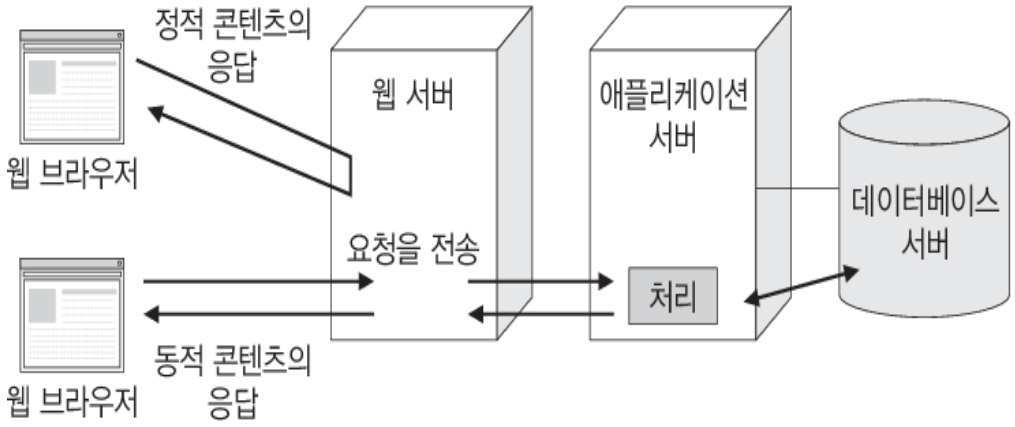
- 프레젠터이션 계층
UI 계층이라고도 불리며 클라이언트로부터 데이터와 함께 요청을 받고 처리 결과를 응답으로 전달한다.

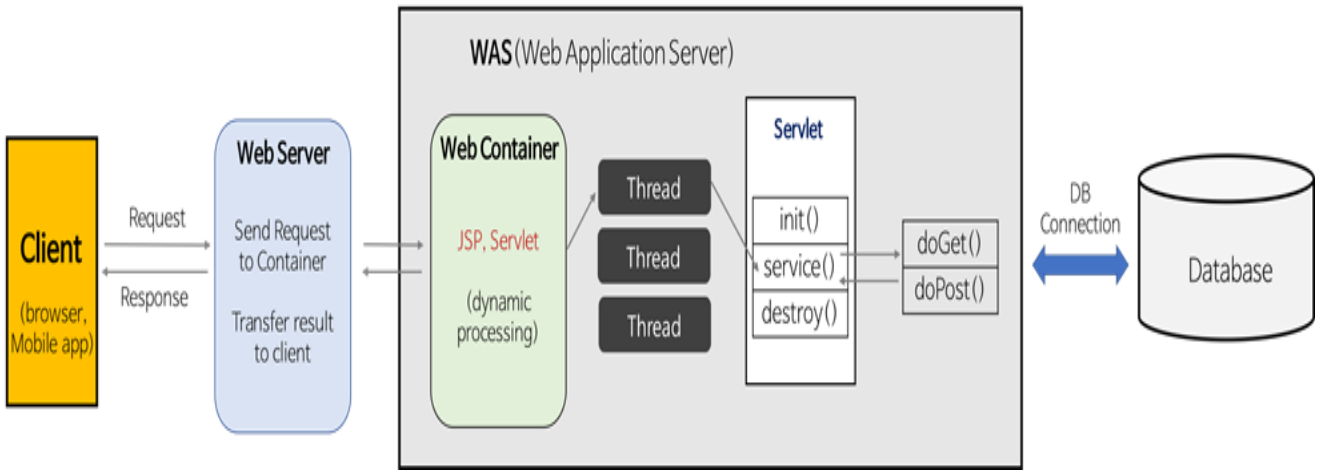
- 비즈니스 계층
서비스 계층이라도 불리며 핵심 비즈니스 로직을 구현하는 영역이다.
트랜잭션 처리나 유효성 검사 등의 작업도 수행한다.

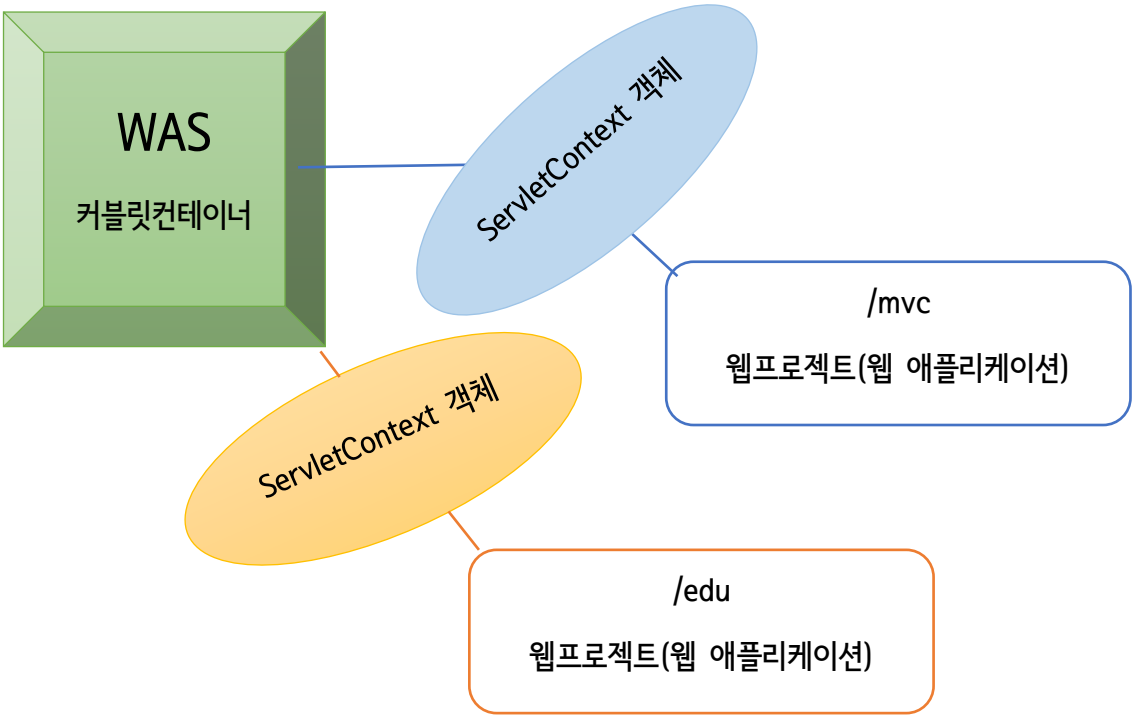
- 데이터접근계층
영속(Persistance)계층이라고도 불리며 데이터베이스에 접근해야하는 작업을 수행한다.
Spring Data JPA에서는 DAO의 역할을 Repository가 수행한다.

[웹 애플리케이션이란]

웹 애플리케이션(Web Application)은 클라이언트(사용자)와 서버 사이에 HTTP 프로토콜을 이용하여 데이터를 주고 받으면서 동작하는 소프트웨어 프로그램이다.

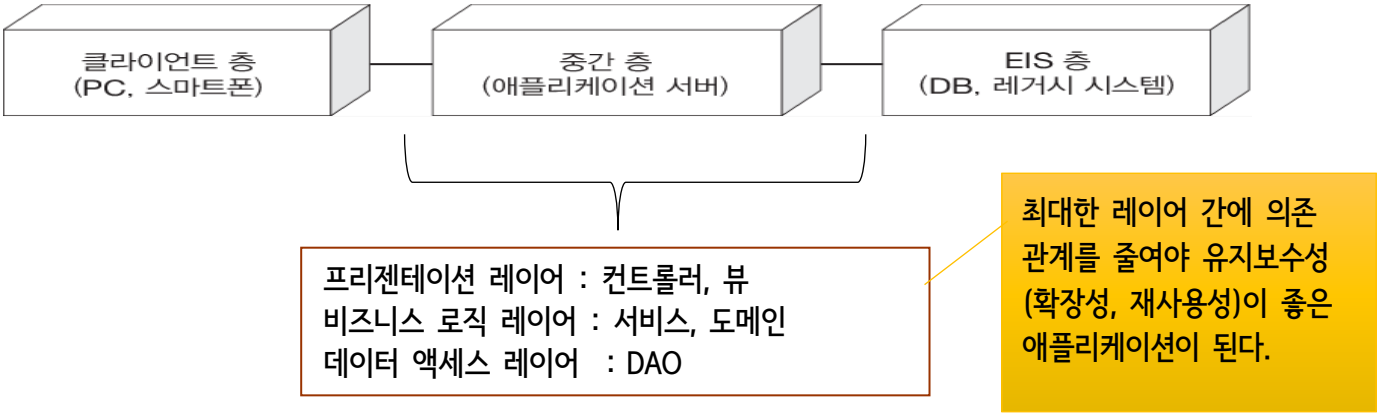






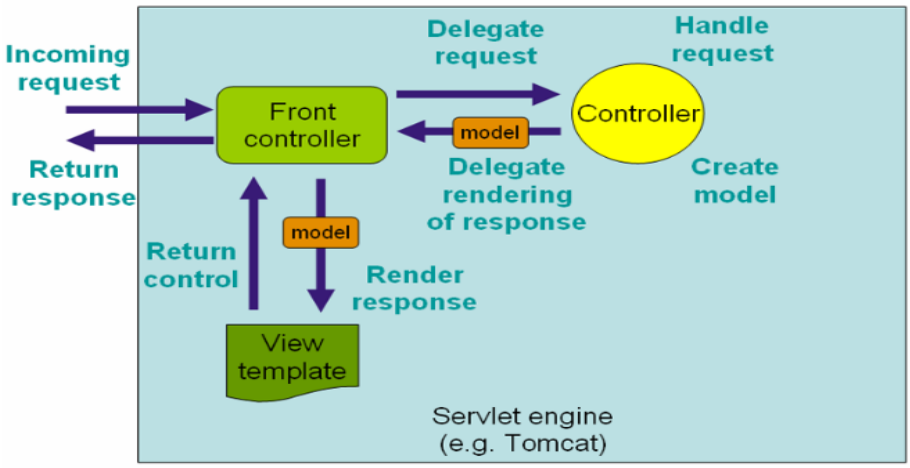
[웹 어플리케이션 구조]

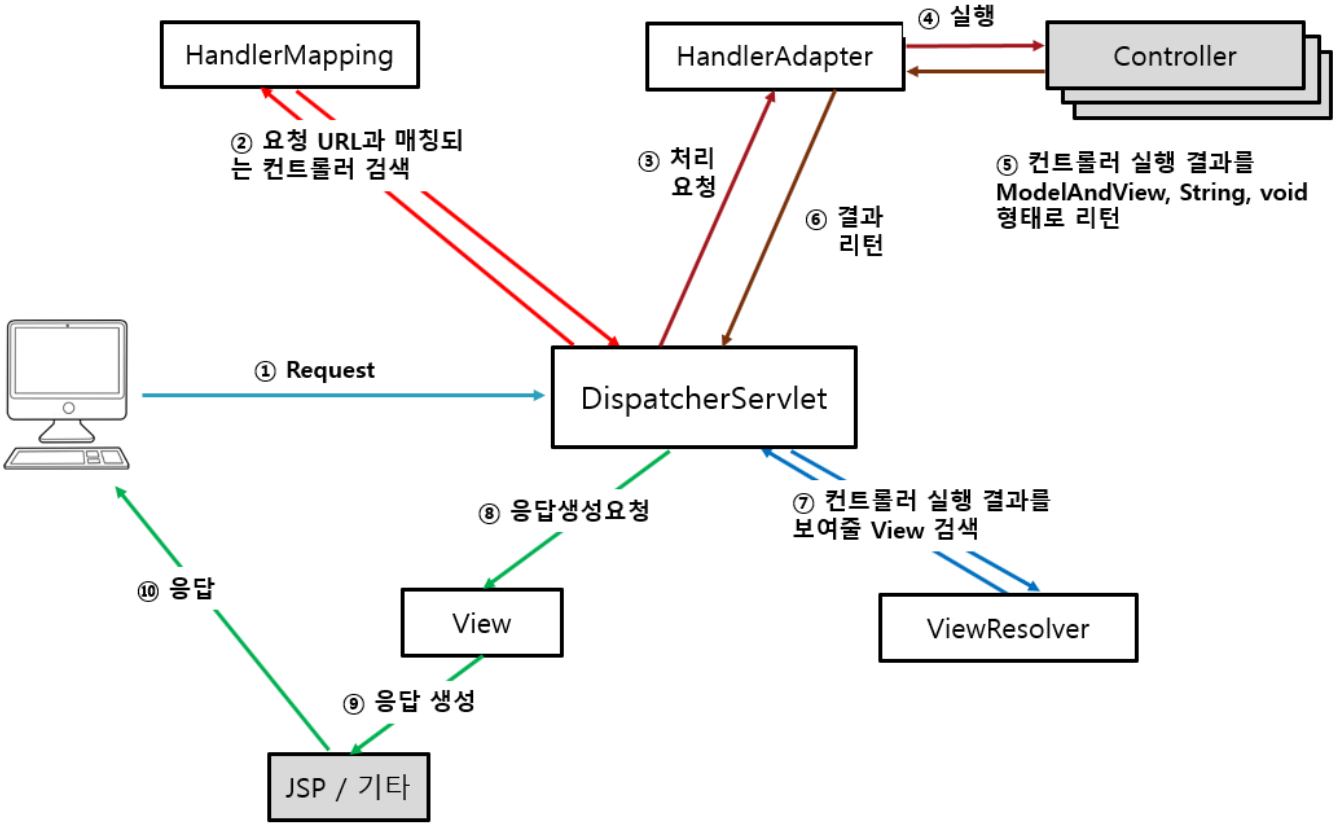
티어 : 어플리케이션의 구조를 물리적으로 나눈 것
레이어 : 어플리케이션의 구조를 논리적으로 나눈 것



[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.





1) DispatcherServlet
가장 앞서 요청을 받아들여 FrontController라고 불림
스프링 프레임워크의 중심이 되는 서블릿으로 클라이언트의 모든 요청을 받아 흐름을 제어
각 컨트롤러에 요청을 전달하고 컨트롤러가 반환한 결과값을 View에 전달해 응답
web.xml에 정의되어 있으며, 보통 servlet-context.xml 설정 파일을 읽어 컨테이너를 구동

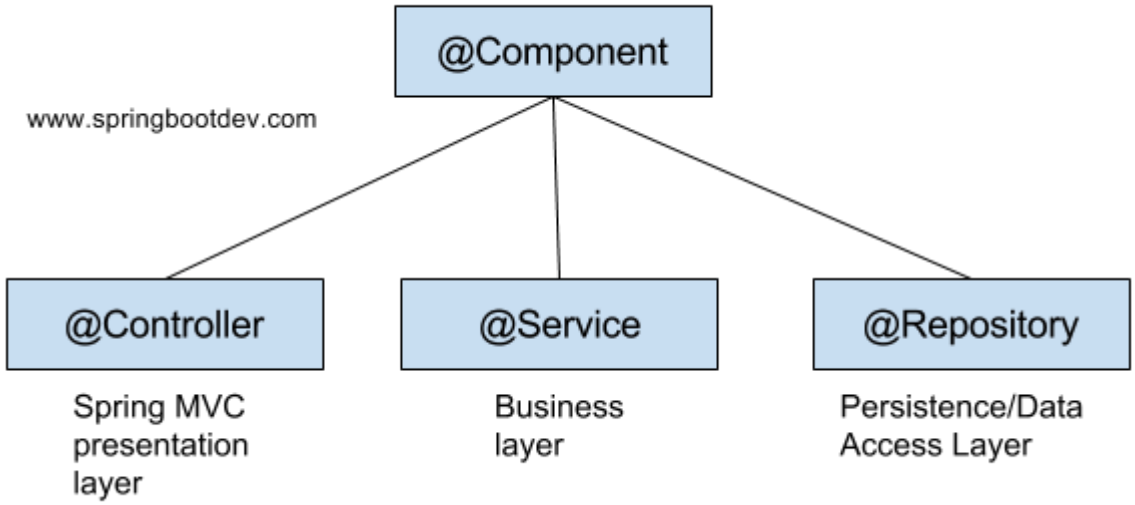
2) HandlerMapping
클라이언트의 요청 URL을 처리할 컨트롤러를 결정해 DispatcherServlet에 반환
@Controller 어노테이션이 적용된 객체의 @RequestMapping 값을 이용해 요청을 처리할 컨트롤러 탐색

3) HandlerAdapter
DispatcherServlet의 처리 요청을 변환해서 컨트롤러에게 전달
컨트롤러의 응답 결과를 DispatcherServlet이 요구하는 형식으로 변환

4) Contoller
실제 클라이언트의 요청을 처리한 뒤, 처리 결과를 void, String, ModelAndView 형태로 반환
GET, POST 방식 등 전송 방식에 대한 처리를 어노테이션으로 처리

5) ViewResolver
컨트롤러의 처리 결과를 보여줄 뷰를 결정

[Spring MVC 구현에서 사용되는 다양한 어노테이션]



@Controller

Spring MVC의 Controller 클래스 정의을 단순화시켜준다.
스프링의 컨트롤러는 상속 구문을 적용할 필요가 없으며, @Controller로 등록된 클래스 파일에 대한 bean을 자동으로 생성해준다. 컨트롤러로 사용하고자 하는 클래스에 @Controller 지정해 주면 component-scan 설정에 등록된 패키지 정보를 기반으로 하여 **컨트롤러 객체로 자동 등록**된다.

```
@SpringBootApplication
@ComponentScan(basePackages={"com.example.springedu","thymeleaf.exam"})
public class SpringeduApplication {
	public static void main(String[] args) {
		SpringApplication.run(SpringeduApplication.class, args);
	}
}
```

※ 컨트롤러 메서드의 파라미터 타입

HttpServletRequest , 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, HttpMessageConverter를 이용 HTTP 요청 파라미터를 해당 타입으로 변환한다.
Map, Model, ModelMap	뷰에 전달할 모델 데이터를 설정할 때 사용
커맨드 객체	HTTP 요청 파라미터를 저장한 객체. 기본적으로 클래스 이름을 모델명으로 사용. @ModelAttribute 어노테이션을 사용하여 모델명을 설정할 수 있다.
Errors, BindingResult	HTTP 요청 파라미터를 커맨드 객체에 저장한 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 뷰이름은 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 URL 문자열을 클래스 또는 메서드와 매핑시켜주는 역활을 한다. 클래스에 하나의 URL 매핑을 사용할 경우, 클래스 위에 @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 요청 매핑을 처리할 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.

```
@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() {    }
```

[Spring 4.3 에서 추가된 애노테이션]

```
@GetMapping("/test")
@PostMapping("/test")
@PutMapping("/test")
@DeleteMapping("/test")
@PatchMapping("/test")
```

@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

쿼리 문자열 또는 요청 파라미터를 메서드의 매개변수로 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 어노테이션은 HTTP request body를 전달 형식 그대로 또는 자바 객체로 변환하여 전달받는 데 사용된다.

```
String test2(@RequestBody String param)
PersonVO test3(@RequestBody PersonVO vo)
Map test4(@RequestBody Map<String,String> map)
```

@ResponseBody 어노테이션을 이용하면 자바 객체를 HTTP response body로 전송할 수 있다. 이 때는 view 를 거치지 않고 컨트롤러가 직접 응답하므로 응답 형식을 설정해야 한다.

```
@RequestMapping(value = "/body/json/{id}", produces = "application/json; charset=utf-8")
@RequestMapping(value = "/body/xml/{id}", produces = "text/xml; charset=utf-8")
```

@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.

[스프링 MVC 에서 파일 업로드 구현하는 방법]

클라이언트에서 업로드 되는 파일은 여러 개의 파트로 구성되어 전달된다.

-> multipart/form-data

멀티 파트를 아규먼트로 받기 위해서는 컨트롤러 메서드의 매개변수 타입을 다음 세 가지 중 하나로 지정한다.

```
xxx(MultipartFile mfile)
xxx(MultipartFile 타입을 멤버변수로 정의한 VO클래스 vo)
xxx(MultipartFile[] 타입을 멤버변수로 정의한 VO클래스 vo) → 다중 파일일 때
xxx(MultipartRequest mreq)      → 다중 파일일 때
```

MultipartFile 의 주요 메소드

String getName()	파라미터의 이름을 리턴한다.
String getOriginalFilename()	업로드 한 파일의 실제!! 이름을 리턴한다.
boolean isEmpty()	업로드 한 파일이 존재하지 않으면 true를 리턴한다.
long getSize()	업로드 한 파일의 크기를 리턴한다.
byte[] getBytes() throws IOException	업로드 한 파일의 데이터를 byte 배열로 리턴다.
InputStream getInputStream()	InputStrem 객체을 리턴한다.
void transferTo(File dest)	업로드 한 파일 데이터를 지정한 파일에 저장한다.

[Spring Scheduling(TASK)]

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

[환경 설정]

(1) build.gradle 의 dependencies 블록에 스케쥴링 관련 API 에 대한 의존성 정보를 추가한다.

```
implementation 'org.springframework.boot:spring-boot-starter-quartz'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
```

(2) XXXApplication 클래스에 @EnableScheduling 을 추가한다.

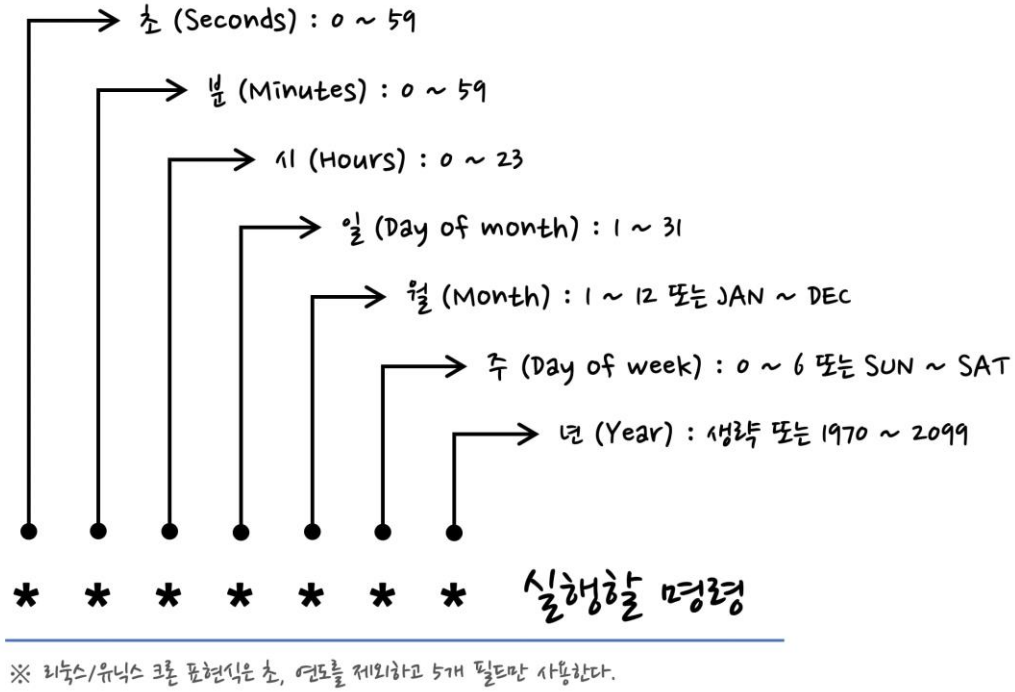
[Task 기능의 메서드 정의]

설정된 주기(스케쥴링)에 맞춰서 호출되는 Task 메서드 앞에 @Scheduled 라는 어노테이션을 다음에 제시한 속성중 하나를 정의하여 추가한다.

- cron : CronTab에서의 설정과 같이 cron = "10 30 12 * * 5" 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.

Cron 표현식의 각 시간단위는 아래의 범위로 구성된다.

초	분	시	일	월	요일	년도
0 ~ 59	0 ~ 59	0 ~ 23	1 ~ 31	1 ~ 12	0 ~ 6	*생략가능*



Cron 표현식 특수문자
* : 모든 값(매시, 매일, 매주처럼 사용한다.)
? : 특정 값이 아닌 어떤 값이든 상관 없음
- : 범위를 지정할 때
, : 여러 값을 지정할 때
/ : 증분값, 즉 초기값과 증가치를 설정할 때
L : 지정할 수 있는 범위의 마지막 값 표시
W : 가장 가까운 평일(weekday)을 설정할 때
: N번 째 특정 요일을 설정할 때

[예제]
매 10분마다
0 0/10 * * * *

매 3시간마다
0 0 0/3 * * *

2018년도 매일 14시 30분마다
0 30 14 * * * 2018

매일 10시 ~ 19시 사이에 10분 간격으로
0 0/10 10-19 * * *

매일 10시와 19시에만 10분 간격으로
0 0/10 10,19 * * *

매달 25일 01시 30분에
0 30 1 25 * *

매주 월, 금요일 10시와 19시 사이 10분마다
0 10 10-19 ? * MON,FRI

매달 마지막날 15시 30분
0 30 15 L * *

2017~2018년 매월의 마지막 토요일 오후 1시 20분
0 20 13 ? * 6L 2017-2018

```
@Component
public class SpringSchedulerTest {
    //@Scheduled(cron = "10 30 12 * * 5")// 초, 분, 시, 일, 월, 요일(0:일요일)
    //@Scheduled(fixedDelay = 5000) // 5초에 한 번씩
    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 제공하는데 @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 ExceptionLocalController {
    @Autowired
    FriendService ms;
    @RequestMapping("/exceptionTest")
    public String detail(int num, Model model) throws FriendNotFoundException {
        FriendVO vo = ms.get(num);
        if (vo == null) {
            throw new FriendNotFoundException();
        }
        model.addAttribute("friend", vo);
        return "friendView";
    }

    @ExceptionHandler(TypeMismatchException.class)
    public ModelAndView handleTypeMismatchException(TypeMismatchException ex) {
        System.out.println("TypeMismatchException 발생시 처리하는 핸들러가 오류 처리합니다.");
        ModelAndView mav = new ModelAndView();
        mav.addObject("msg", "타입을 맞춰주세용!!");
        mav.setViewName("errorPage");
        return mav;
    }

    @ExceptionHandler(FriendNotFoundException.class)
    public String handleNotFoundException() throws IOException {
        System.out.println("FriendNotFoundException 발생시 처리하는 핸들러가 오류 처리합니다.");
        return "noFriend";
    }

    @ExceptionHandler(IllegalStateException.class)
```

```
    public ModelAndView handleIllegalStateException() throws IOException {
        System.out.println("IllegalStateException 발생시 처리하는 핸들러가 오류 처리합니다.");
        ModelAndView mav = new ModelAndView();
        mav.addObject("msg", "num=숫자 형식의 쿼리를 전달하세요!!");
        mav.setViewName("errorPage");
        return mav;
    }
}
```

@ControllerAdvice : @ControllerAdvice는 스프링 3.2 이상에서 사용가능 하며 @Controller나 @RestController 에서 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.

```
@ControllerAdvice
public class CommonExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    private ModelAndView errorModelAndView(Exception ex) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("commonErrorPage");
        modelAndView.addObject("msg", "RuntimeException은 내가 다 잡는다!!!" );
        modelAndView.addObject("exceptionInfo", ex );
        return modelAndView;
   }
}
```