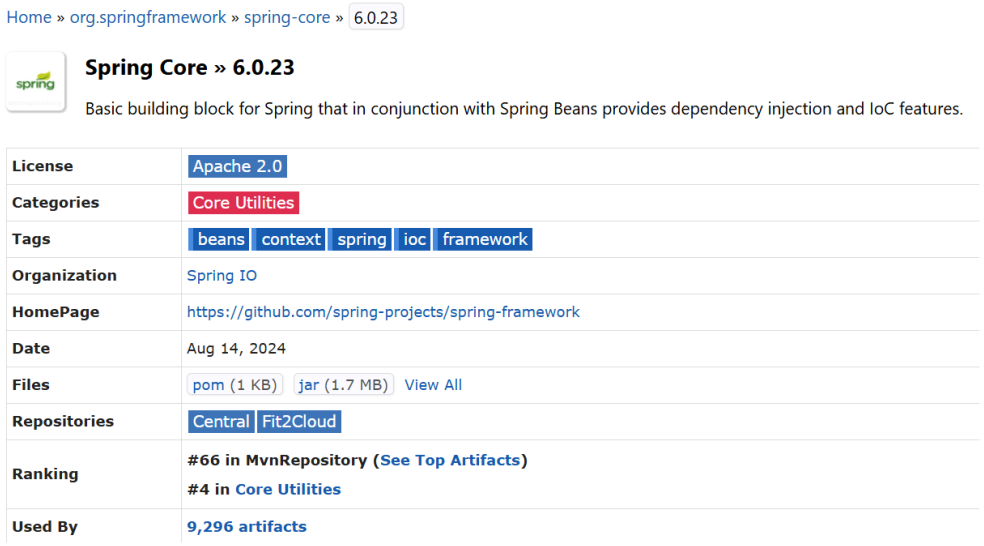
4장. 스프링과 스프링 WEB MVC

Spring 프레임워크 라이브러리 추가

- 경량 프레임워크들은 대부분 jar파일의 형태로 구성
- ‘spring – core’ 라이브러리 추가

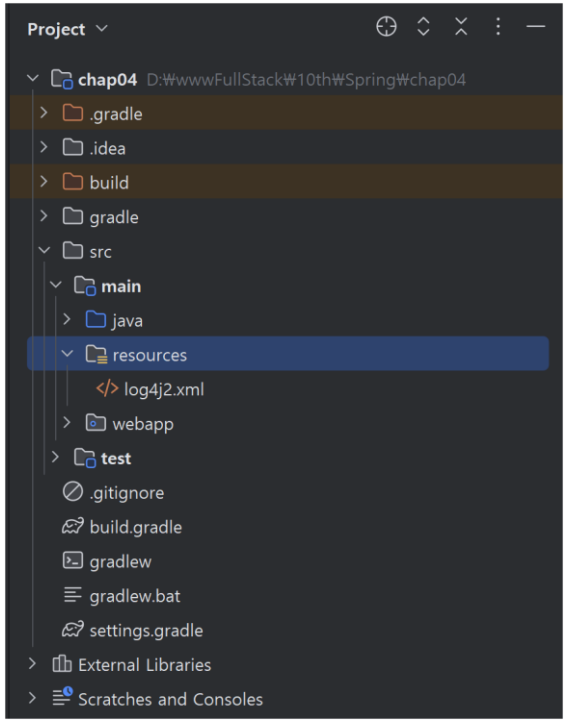
```
dependencies {
    compileOnly('jakarta.servlet:jakarta.servlet-api:6.0.0’)

    testImplementation("org.junit.jupiter:junit-jupiter-api:${junitVersion}")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:${junitVersion}")

    // https://mvnrepository.com/artifact/org.springframework/spring-core
    implementation 'org.springframework:spring-core:6.0.23’
    // https://mvnrepository.com/artifact/org.springframework/spring-context
    implementation 'org.springframework:spring-context:6.0.23’
    // https://mvnrepository.com/artifact/org.springframework/spring-test
    testImplementation 'org.springframework:spring-test:6.0.23'
}
```



Lombok / Log4j2/ JSTL추가

```
implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0'
implementation 'org.eclipse.jetty:glassfish-jstl:11.0.18'

implementation 'org.projectlombok:lombok:1.18.38'
annotationProcessor 'org.projectlombok:lombok:1.18.38'
testImplementation 'org.projectlombok:lombok:1.18.38'
testAnnotationProcessor 'org.projectlombok:lombok:1.18.38'

implementation 'org.apache.logging.log4j:log4j-api:2.24.3'
implementation 'org.apache.logging.log4j:log4j-core:2.24.3'
implementation 'org.apache.logging.log4j:log4j-slf4j2-impl:2.24.3'
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- https://logging.apache.org/log4j/2.x/manual/configuration.html -->
<configuration status="INFO">
    <Appenders>
        <Console name="CONSOLE" target="SYSTEM_OUT">
            <PatternLayout charset="UTF-8" pattern="%d{hh:mm:ss.SSS} [%t] %-5level %logger{36} -%msg%n" />
        </Console>
    </Appenders>
    <loggers>
        <logger name="net.fullstack10" level="INFO" additivity="false">
            <Appender-ref ref="CONSOLE" />
        </logger>
        <root level="INFO" additivity="false">
            <Appender-ref ref="CONSOLE" />
        </root>
    </loggers>
</configuration>
```



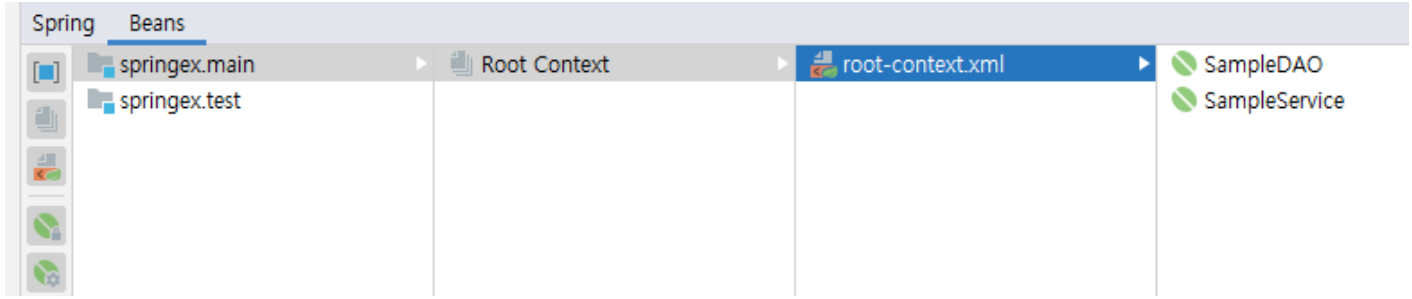
스프링 환경 설정

- 스프링 프레임워크 설정 방법
  - XML 설정 이용
    - webapp > WEB-INF > config > root-context.xml
  - 자바 설정 이용
  - 설정 파일과 어노테이션 사용

root-context.xml

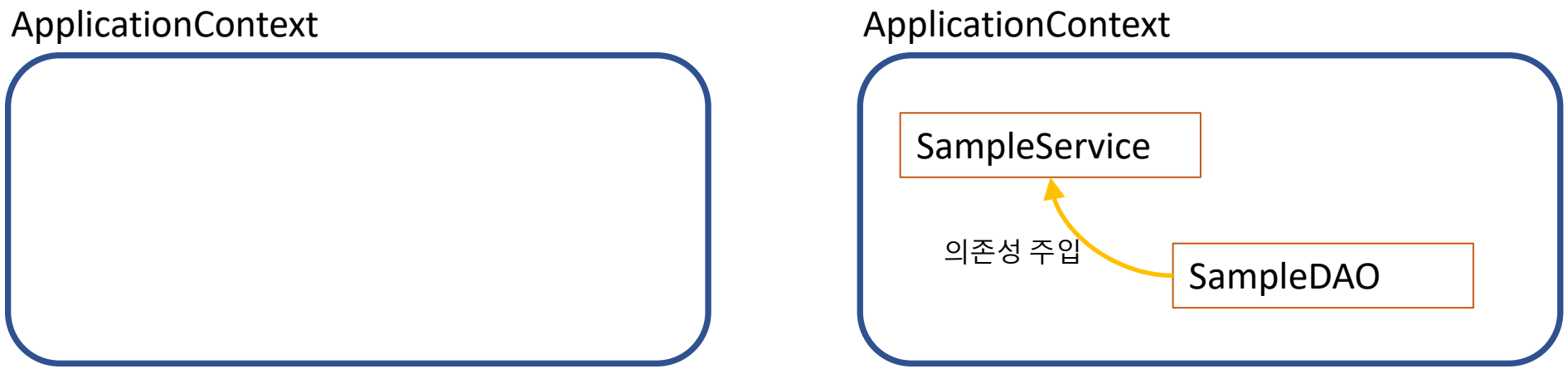
- 일반적으로 스프링 프레임워크 이용시 사용하는 기본 설정 파일
- 주로 POJO에 대한 설정
- 별도의 라이브러리들을 활용하는 경우에는 별도의 파일을 추가하는 방식을 이용

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean class="org.zerock.springex.sample.SampleDAO"></bean>

   <bean class="org.zerock.springex.sample.SampleService"></bean>


</beans>
```



ApplicationContext와 빈(Bean)

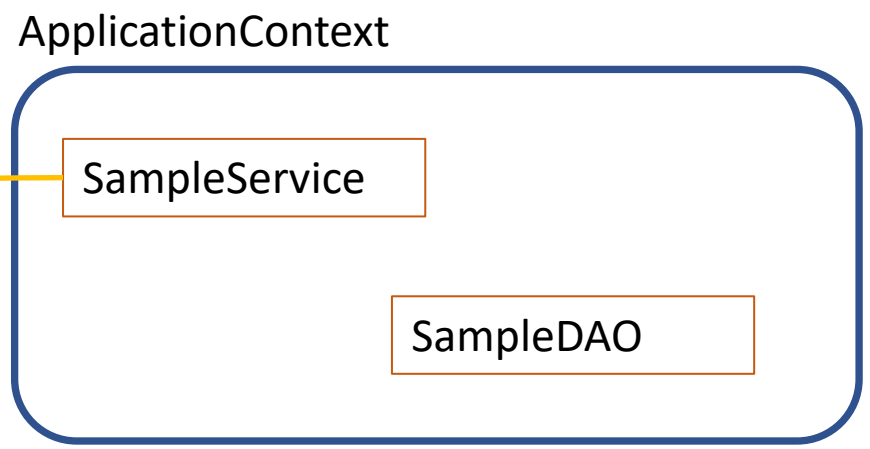
- 스프링이 빈들을 관리하는 공간 – ApplicationContext
- root-context.xml을 읽어서 해당 클래스들을 인스턴스화 시켜서 ApplicationContext 내부에서 관리



@Autowired의 의미와 필드 주입

- @Autowired 가 있는 필드의 경우 해당 타입의 객체가 스프링의 컨텍스트내 존재한다면 실행시 주입된다.

```java
@Log4j2
@ExtendWith(SpringExtension.class)
@ContextConfiguration(locations="file:src/main/webapp/WEB-INF/root-context.xml")
public class SampleTests {

    @Autowired
    private SampleService sampleService;

    @Test
    public void testService1() {
        log.info(sampleService);
        Assertions.assertNotNull(sampleService);
    }
}
```



<context:component-scan>

- 패키지를 지정해서 해당 패키지내 클래스의 인스턴스들을 스프링의 빈으로 등록하기 위해서 사용
- 특정 어노테이션을 이용해서 스프링의 빈으로 관리될 객체를 표시
  - @Controller
  - @Service
  - @Repository
  - @Component

생성자 주입 방식

- 주입받는 타입을 final로 선언하고 생성자를 통해서 의존성 주입
- lombok의 @RequiredArgsConstructor 를 통해서 생성자 자동 생성

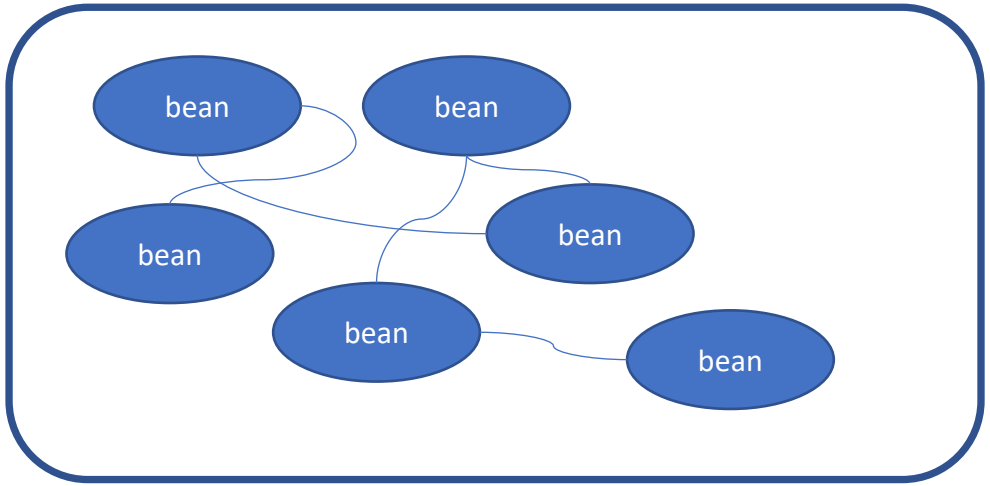
```
package net.fullstack10.springmvc.sample;

import lombok.RequiredArgsConstructor;
import lombok.ToString;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
@ToString
@RequiredArgsConstructor
public class SampleService {

    private final SampleDAO sampleDAO;

}
```

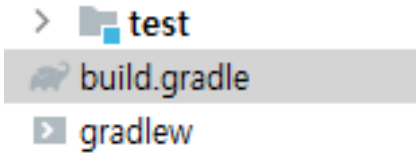
```
package net.fullstack10.springmvc.sample;

import org.springframework.stereotype.Repository;

@Repository
public class SampleDAO {
}
```

웹 프로젝트를 위한 스프링 준비

- 스프링의 ApplicationContext는 여러 빈들을 관리
- Web의 경우 web.xml을 이용해서
- 리스너를 통해서 ApplicationContext등록

ApplicationContext

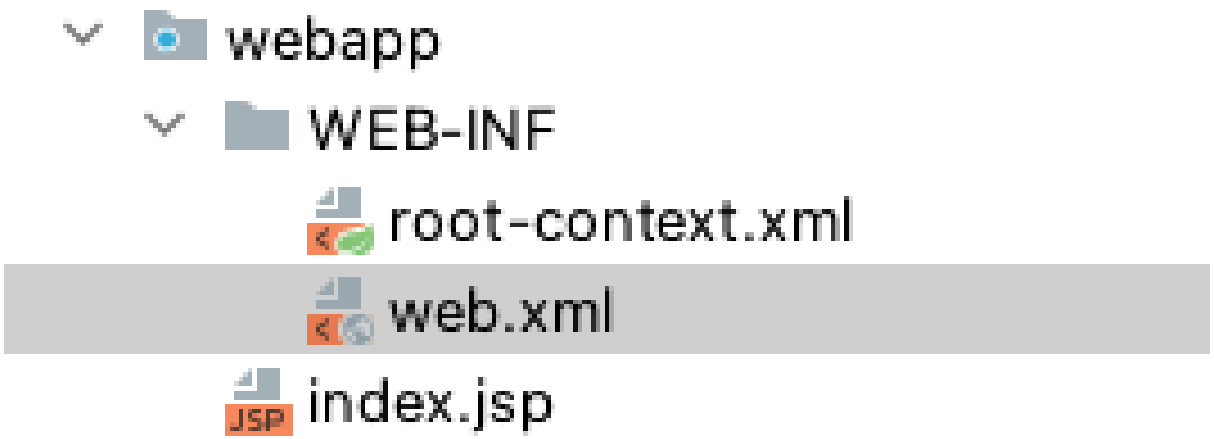





```
// https://mvnrepository.com/artifact/org.springframework/spring-core
implementation 'org.springframework:spring-core:6.1.0'
implementation 'org.springframework:spring-context:6.1.0'
implementation 'org.springframework:spring-test:6.1.0'
implementation 'org.springframework:spring-webmvc:6.1.0'
```

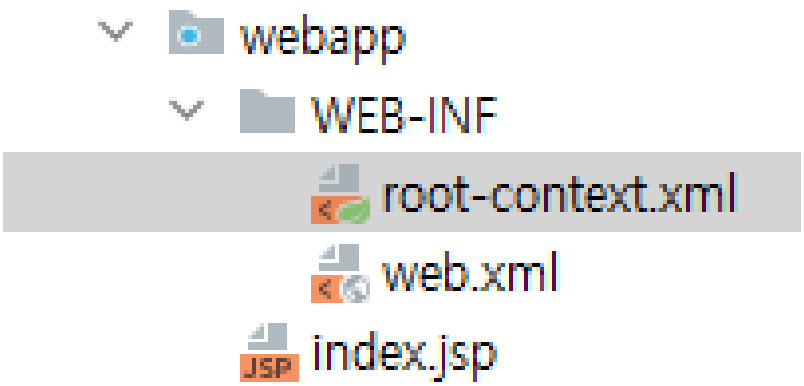
web.xml의 설정

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

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



DataSource 구성하기

```
dependencies {
    compileOnly('javax.servlet:javax.servlet-api:4.0.1')

    ...생략...
    // https://mvnrepository.com/artifact/org.mariadb.jdbc/mariadb-java-client
    implementation 'org.mariadb.jdbc:mariadb-java-client:3.2.0'


    // https://mvnrepository.com/artifact/com.zaxxer/HikariCP
    implementation 'com.zaxxer:HikariCP:6.0.0'
}
```




```
<bean id="hikariConfig" class="com.zaxxer.hikari.HikariConfig">
  <property name="driverClassName" value="org.mariadb.jdbc.Driver"></property>
  <property name="jdbcUrl" value="jdbc:mariadb://localhost:3306/webdb"></property>
  <property name="username" value="webuser"></property>
  <property name="password" value="webuser"></property>
  <property name="dataSourceProperties">
    <props>
      <prop key="cachePrepStmts">true</prop>
      <prop key="prepStmtCacheSize">250</prop>
      <prop key="prepStmtCacheSqlLimit">2048</prop>
    </props>
  </property>
</bean>

<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource"
    destroy-method="close">
  <constructor-arg ref="hikariConfig" />
</bean>
```

DataSource 구성하기

```java
@Log4j2
@ExtendWith(SpringExtension.class)
@ContextConfiguration(locations="file:src/main/webapp/WEB-INF/root-context.xml")
public class SampleTests {

  @Autowired
  private SampleService sampleService;

  @Autowired
  private DataSource dataSource;


  @Test
  public void testService1() {
    log.info(sampleService);
    Assertions.assertNotNull(sampleService);
  }

  @Test
  public void testConnection() throws Exception{

    Connection connection = dataSource.getConnection();
    log.info(connection);
    Assertions.assertNotNull(connection);
    connection.close();

  }

}
```

```
INFO [org.springframework.test.context.support.DefaultTestContextBootstrapper] Loaded default TestExecutionListener class name
INFO [org.springframework.test.context.support.DefaultTestContextBootstrapper] Using TestExecutionListeners: [org.springframew
INFO [com.zaxxer.hikari.HikariDataSource] HikariPool-1 - Starting...
INFO [com.zaxxer.hikari.pool.HikariPool] HikariPool-1 - Added connection org.mariadb.jdbc.Connection@63411512
INFO [com.zaxxer.hikari.HikariDataSource] HikariPool-1 - Start completed.
INFO [org.zerock.springex.sample.SampleTests] HikariProxyConnection@1118998513 wrapping org.mariadb.jdbc.Connection@63411512
```

MyBatis와 스프링 연동

- ‘Sql Mapping Framework’ - SQL의 처리를 객체와 매핑해서 처리
- JDBC를 이용해서 PreparedStatement/ResultSet에 대한 객체 처리를 자동으로 수행
- Connection등의 JDBC자원들에 대한 자동 close( )
- SQL은 별도의 XML등을 이용해서 분리

MyBatis와 스프링 연동

- MyBatis 의 특징
  - 간결한 코드의 처리
    - SQL Mapper 라이브러리
    - 개발 코드를 줄여 개발의 속도 향상
    - 코드 제작 없이도 JDBC 처리 가능
  - SQL 문의 분리 운영
    - XML 혹은 애노테이션 방식으로 SQL 문을 별도로 처리
    - 두 가지 방식을 혼합해서 사용하는 것도 가능
    - 간단한 SQL은 직접 애노테이션을 이용
    - 복잡하고 많은 양의 SQL은 XML 이용

MyBatis와 스프링 연동

- MyBatis 의 특징
  - Spring 과의 연동으로 자동화된 처리
    - MyBatis-Spring 라이브러리 이용
    - 직접 SQL 문의 호출 없이 처리 가능
    - MyBatis는 단독으로 사용하는 것보다 스프링과 연계하는 것이 코드의 양을 줄일 수 있음
  - 동적 SQL을 이용한 제어 기능
    - 간단한 제어문, 루프 등 처리 가능
    - SQL 관련 처리를 Java 코드에서 분리시킬 수 있음

MyBatis와 스프링 연동

- My⊢atis 의 특징
  - Sql Mapping Framework
    - SQL 실행 결과를 객체지향으로 매핑해 줌
    - SQL 파일 별도 처리 가능
  - MyBatis 는 단독으로 실행 가능한 독립적인 프레임워크
  - 스프링 프레임워크는 MyBatis 와 연동 쉽게할 수 있는 라이브러리, API 제공
  - 인터페이스, 어노테이션만으로도 처리 가능
  - @Autowired(required = false) → 해당 객체를 주입 받지 못해도 예외 발생 안함
  - 인텔리제이에서는 @Service, @Repository 등 빈으로 등록된 경우 아니면 경고 발생 시킴
  - MyBatis 와 스프링 연동하고 매퍼 인터페이스 활용 방식 사용

    → 스프링에서 자동으로 객체를 생성되는 방식 사용

MyBatis와 스프링 연동

- 장점
  - PreparedStatement/ResultSet 의 처리 편리
  - 파라미터나 ResultSet 의 getXXX() 를 MyBatis 가 처리하므로 코드의 양을 줄일 수 있음
  - Connection/PreparedStatement/ResultSet 의 close() 를 자동으로 처리
  - SQL 의 분리
  - 별도의 파일이나 어노테이션 이용하여 SQL 선언
  - 파일 : 별도로 분리하여 운영 가능
  - 인터페이스만으로 개발 가능
- 단점
  - 스프링에서 자동 생성된 객체 이용하기 때문에 개발자가 직접 코드 수정할 수 없음

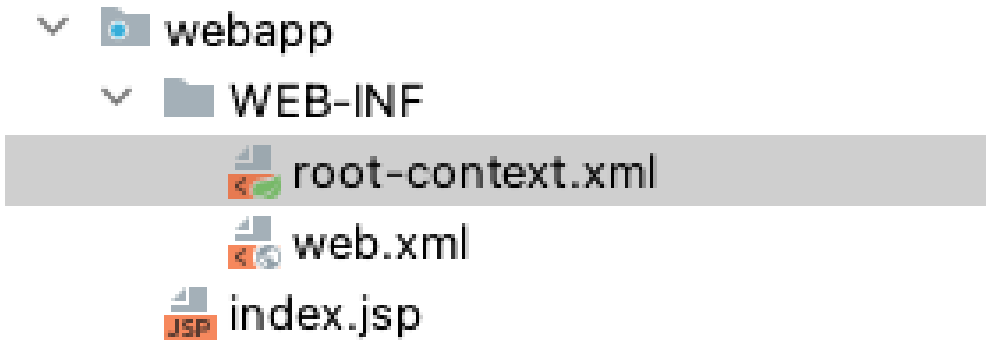
MyBatis와 스프링의 연동 방식

- MyBatis는 단독으로 실행이 가능한 프레임워크지만 mybatis-spring 라이브러리를 이용하면 쉽게 통합해서 사용 가능
- 과거에는 주로 별도의 DAO(Data Access Object)를 구성하고 여기서 MyBatis의 SqlSession을 이용하는 방식
- 최근에는 MyBatis는 인터페이스를 이용하고 실제 코드는 자동으로 생성되는 방식 – Mapper인터페이스와 XML
- 필요한 라이브러리
  - 스프링 관련: spring-jdbc, spring-tx
  - MyBatis 관련: mybatis, mybatis-spring

```
// 스프링 Mybatis 관련 추가
implementation 'org.springframework:spring-jdbc:6.1.0'
implementation 'org.springframework:spring-tx:6.1.0'
// https://mvnrepository.com/artifact/org.mybatis/mybatis
implementation 'org.mybatis:mybatis:3.5.19' //--> 스프링5.x 의 버전이 다르므로 확인하여 설치할 것
// https://mvnrepository.com/artifact/org.mybatis/mybatis-spring
implementation 'org.mybatis:mybatis-spring:3.0.4' //--> mybatis 버전과 일치하지 않으므로 주의
```

MyBatis의 SqlSessionFactory설정

- MyBatis에서 실제 SQL의 처리는 SqlSesssionFactory 에서 생성하는 SqlSession을 통해서 수행됨




```
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource"
    destroy-method="close">
  <constructor-arg ref="hikariConfig" />
</bean>


<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
</bean>
```

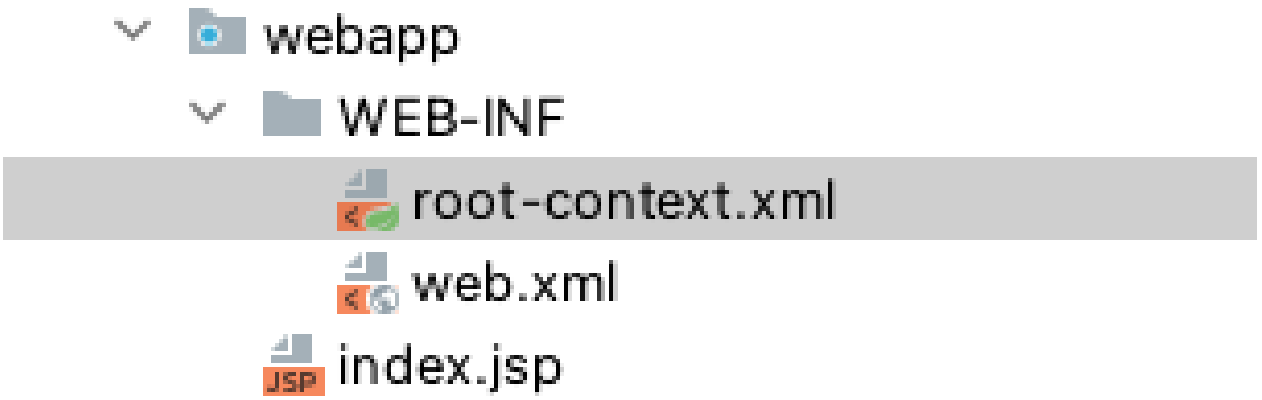
Mapper인터페이스 활용하기

- MyBatis를 통해서 수행해야 하는 기능을 매퍼 인터페이스로 작성
- 어노테이션 혹은 XML로 SQL 작성
- 스프링의 설정에서

```
package net.fullstack10.springmvc.mapper;

import org.apache.ibatis.annotations.Select;

public interface TimeMapper {

    @Select("select now()")
    String getTime();
}
```

root-context.xml 설정

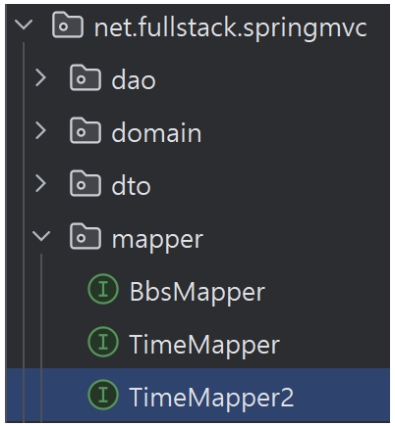
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    https://www.springframework.org/schema/context/spring-context.xsd
    http://mybatis.org/schema/mybatis-spring http://mybatis.org/schema/mybatis-spring.xsd">


  <context:component-scan base-package="org.zerock.springex.sample"></context:component-scan>

  <bean id="hikariConfig" class="com.zaxxer.hikari.HikariConfig">
    <property name="driverClassName" value="org.mariadb.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mariadb://localhost:3306/webdb"></property>
    <property name="username" value="webuser"></property>
    <property name="password" value="webuser"></property>
    <property name="dataSourceProperties">
      <props>
        <prop key="cachePrepStmts">true</prop>
        <prop key="prepStmtCacheSize">250</prop>
        <prop key="prepStmtCacheSqlLimit">2048</prop>
      </props>
    </property>
  </bean>

  <bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource"
      destroy-method="close">
    <constructor-arg ref="hikariConfig" />
  </bean>


  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <mybatis:scan base-package="org.zerock.springex.mapper"></mybatis:scan>

</beans>
```



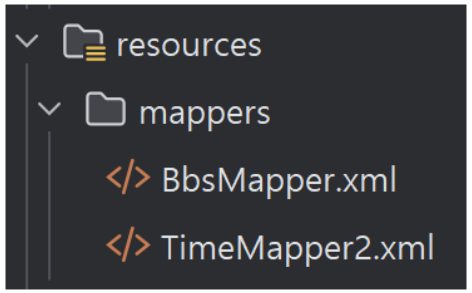
XML로 SQL분리하기

- SQL이 길고 복잡한 경우 XML을 이용해서 SQL을 분리
- XML을 이용하는 방식
  - 매퍼인터페이스에 메소드를 선언
  - XML파일을 작성하고 메서드의 이름과 네임스페이스를 작성
  - <select>, <insert>.. 을 이용할때 id 속성값은 메서드의 이름으로 지정

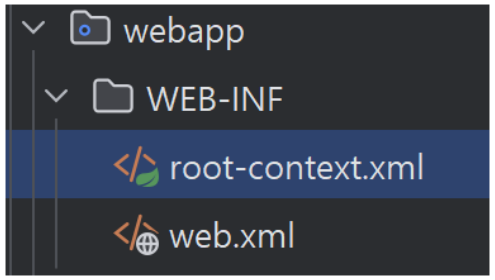
XML로 SQL분리하기




```
package net.fullstack.springmvc.mapper;

public interface TimeMapper2 {
    String getNow();
}
```




```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="net.fullstack.springmvc.mapper.TimeMapper2">
    <select id="getNow" resultType="string">
        select now()
    </select>
</mapper>
```

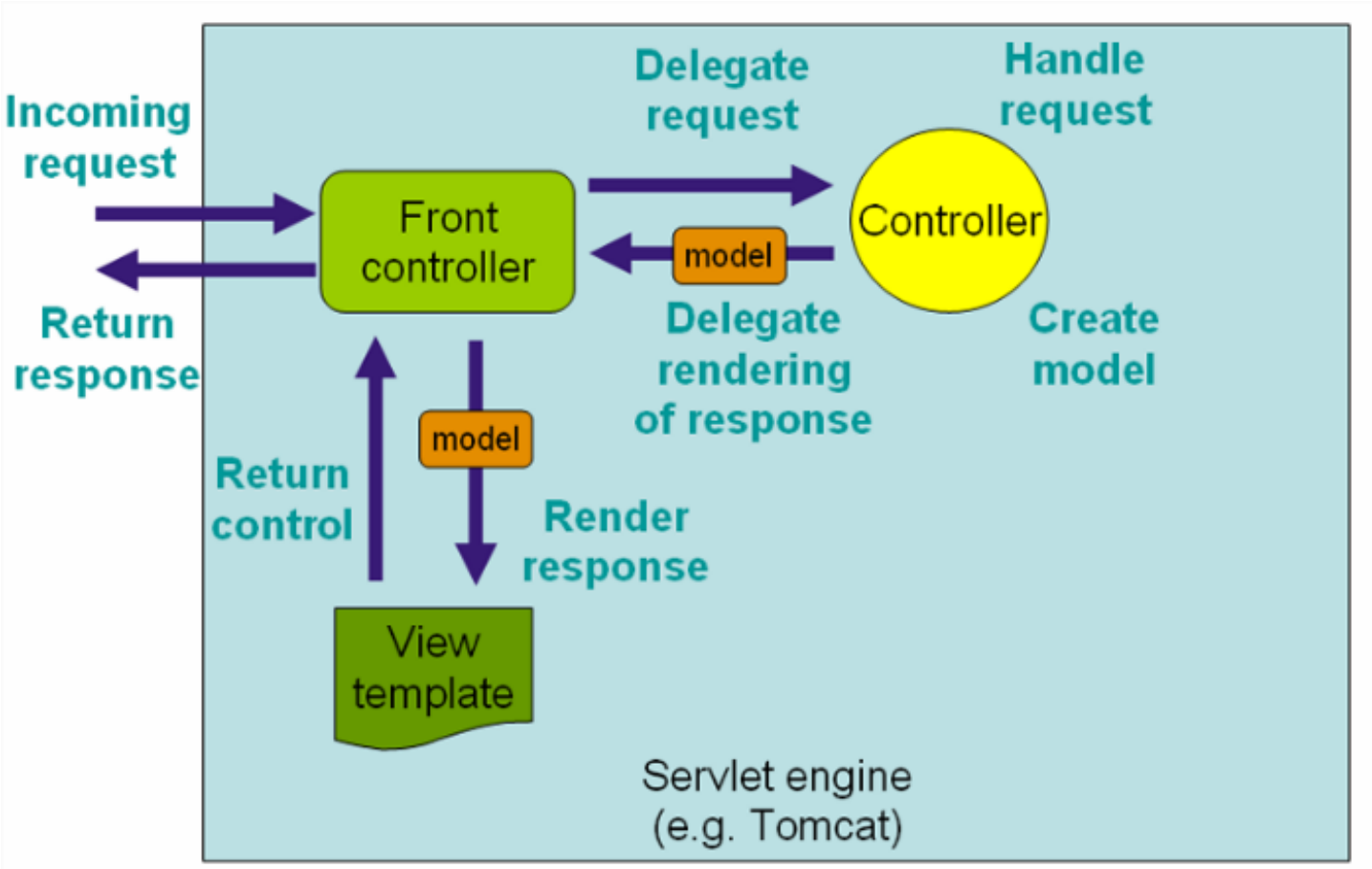



```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mapperLocations" value="classpath:/mappers/**/*.xml"></property>
</bean>
```

스프링 Web MVC 기초

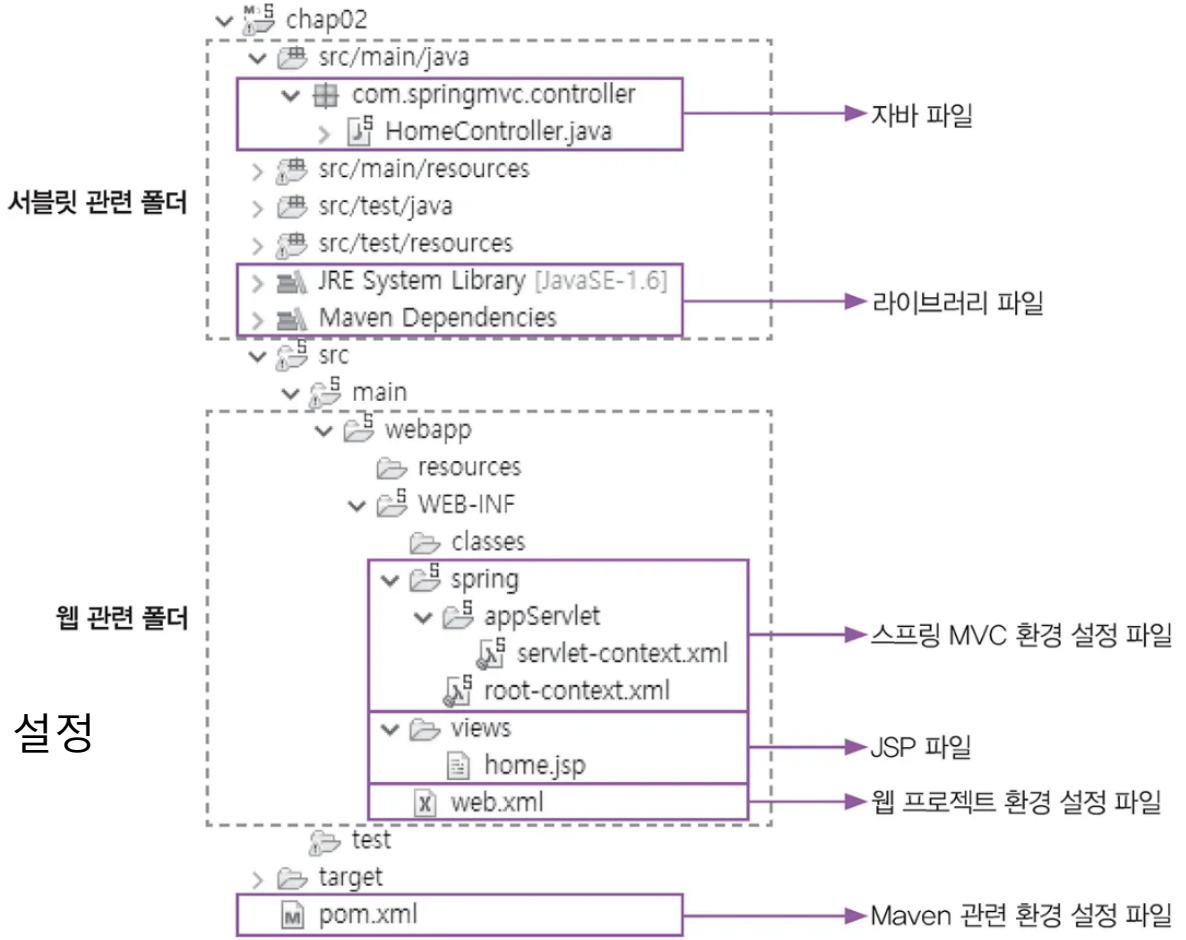
스프링 Web MVC의 특징

- 기존의 MVC구조에서 추가적으로 Front-Controller패턴 적용
- 어노테이션의 적극적인 활용
- 파라미터나 리턴타입에 대한 자유로운 형식
- 추상화된 api들의 제공



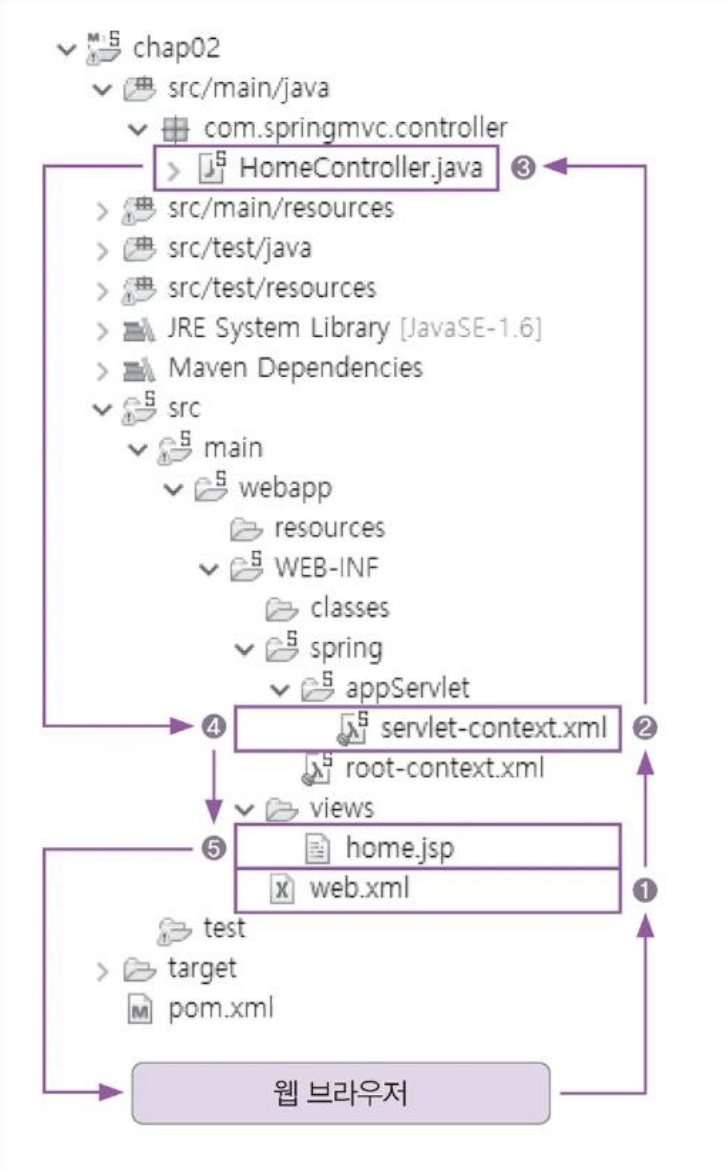
스프링 Web MVC의 특징

- 프로젝트 구조
  - 서블릿 관련 폴더
    - src/main/java
  - 웹 관련 폴더
    - src/main/webapp
    - resources : 이미지, 자바스크립트, css 등
    - WEB-INF : 웹과 관련된 파일, 보안상 외부에서 접근 불가
    - view : JSP 파일
    - web.xml : 웹 프로젝트 설정 파일, 리스너, 서블릿필터 등 설정



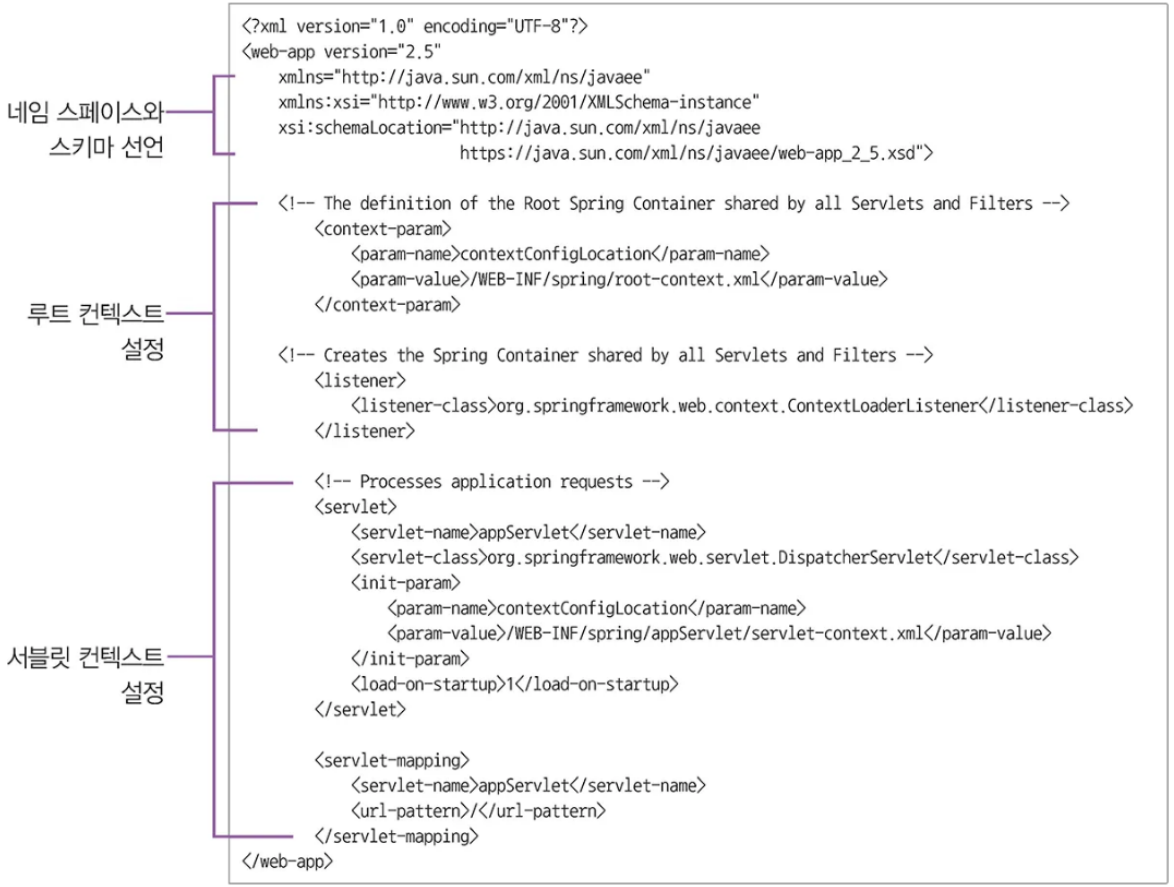
스프링 Web MVC의 특징

1. web.xml 파일
    1. 클라이언트의 요청 전달
    2. 디스패처 서블릿이 클라이언트의 요청 URL 제어
2. servlet-context.xml 파일
    1. 요청 URL을 처리하는 컨트롤러 검색
3. 컨트롤러
    1. HomeController는 클라이언트의 요청 처리
    2. 결과를 출력할 뷰를 디스패처 서블릿에 반환
4. 컨트롤러에서 보내온 뷰 이름을 토대로 처리할 뷰를 검색
5. 처리 결과가 포함된 뷰를 디스패처 서블릿에 반환하고 최종 결과를 출력



스프링 Web MVC의 특징

- web.xml
  - 웹 프로젝트의 배포 설명자/배치 기술서(deployment descriptor)
  - 웹 프로젝트가 배포되는데 이용되는 XML 형식의 자바 웹 어클리케이션 환경 설정 담당
  - 프로젝트 실행시 가장 먼저 읽어들이고 위에서부터 차례로 해석
  - <web-app> ~~~ </web-app> 으로 설정




```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        https://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml</param-value>
    </context-param>

    <!-- Creates the Spring Container shared by all Servlets and Filters -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Processes application requests -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

스프링 Web MVC의 특징

네임 스페이스와 스키마 선언

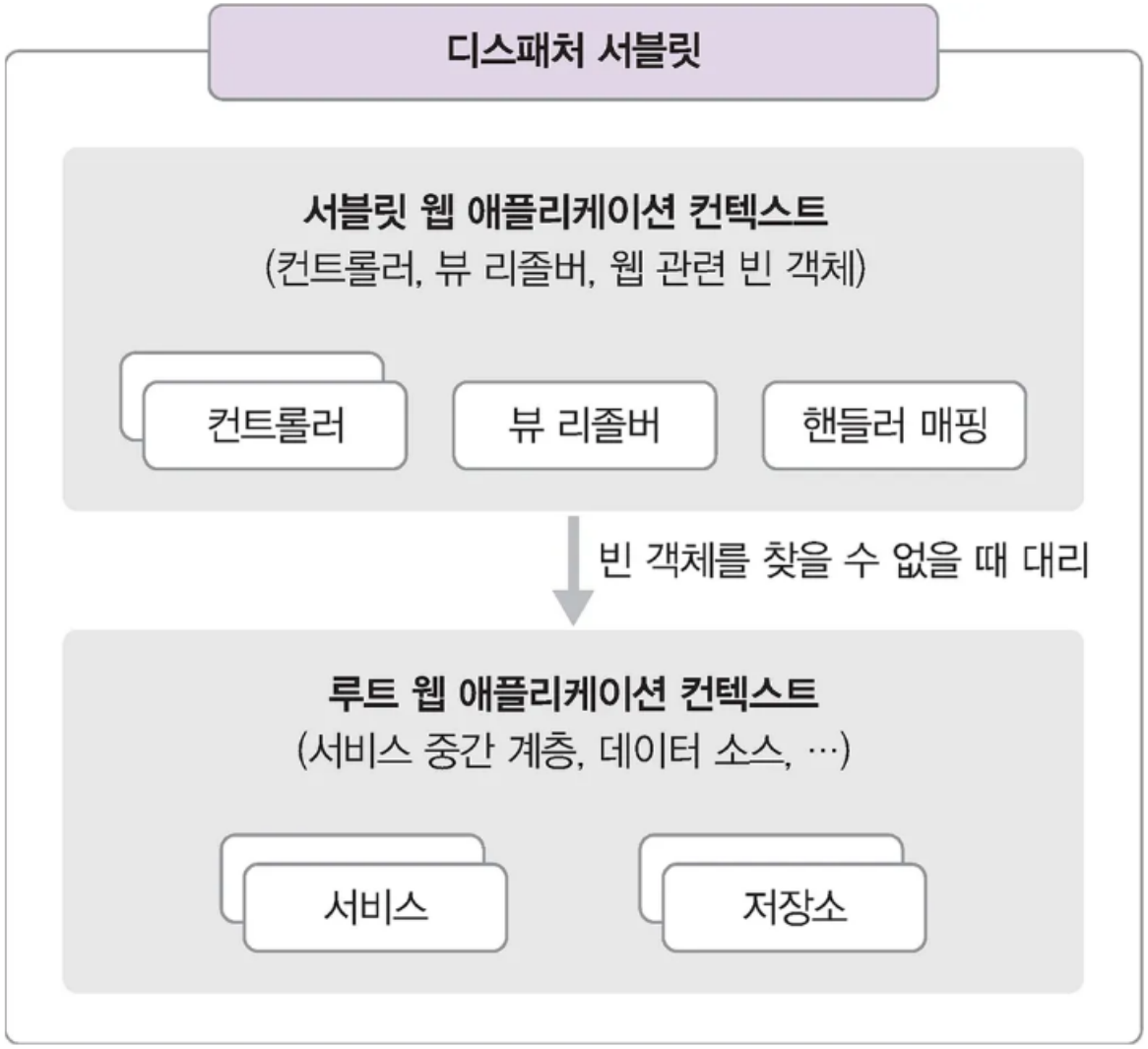
- 네임 스페이스는 코드에서 다른 요소와 충돌하지 않도록, 요소를 구별하는 데 사용
- 스키마는 코드의 구조와 요소, 속성의 관계를 정의
- 다양한 자료형을 사용할 수 있도록 정의된 문서 구조

```
<?xml version="1.0" encoding="UTF-8"?> <web-app
version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee" ❶
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ❷
xsi:schemaLocation="http://java.sun.com/xml/ns/ja
vaee ❸ http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">
```

1. 기본 네임 스페이스 선언
    1. xmlns 속성은 기본 XML 스키마 네임 스페이스 명시
    2. 속성 값은 모든 스키마를 가지고 있음
2. 인스턴스 네임 스페이스 URI 선언
    1. xmlns:xsi 속성은 인스턴스 네임 스페이스 URI를 지정
    2. 속성 값은 대부분 XML 스키마로 사용되는 표준인 W3C(World Wide Web Consortium) XML 스키마
3. 참조하는 인스턴스 문서의 URL 선언
    1. xsi:schemaLocation 속성은 참조하고자 하는 인스턴스 문서의 URL을 지정
    2. 두 개의 속성 값은 공백으로 구분
    3. 첫 번째는 사용할 네임 스페이스(보통 기본 네임 스페이스와 동일)
    4. 두 번째는 참조할 스키마 파일 이름

스프링 Web MVC의 특징

- 스프링 MVC 환경 설정 파일
  - 빈 객체를 정의
  - root-context.xml
    - 모든 컨텍스트에서 공유
    - 뷰(JSP 웹 페이지)와 관련 없는 빈 객체를 설정
    - 서비스, 저장소, 데이터베이스, 로그 등 웹 애플리케이션의 비즈니스 로직을 위한 컨텍스트를 설정
  - servlet-context.xml
    - 서블릿 컨텍스트에서만 사용
    - 뷰(JSP 웹 페이지)와 관련 있는 빈 객체를 설정
    - 컨트롤러, MultipartResolver, Interceptor, URI와 관련 설정을 담는 클래스를 설정



스프링 Web MVC의 특징

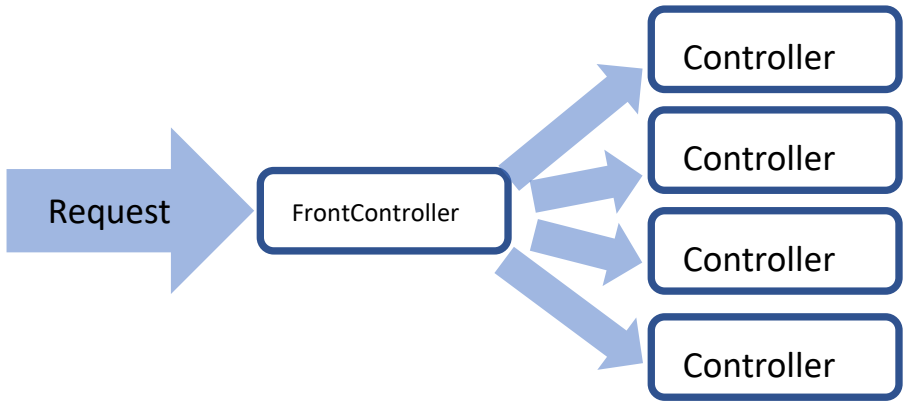
- **루트 컨텍스트 파일**
- root-context.xml
  - 다른 웹 컴포넌트들과 공유하는 자원을 선언하는 용도로 사용
  - 뷰와 관련되지 않은 Service, Repository(DAO), DB 등 객체를 정의
  - 스프링 MVC 프로젝트를 처음 생성하면 root-context.xml 파일에 내용이 없음
  - 공통 빈을 설정하는 곳으로 주로 뷰 지원을 제외한 빈 객체를 설정

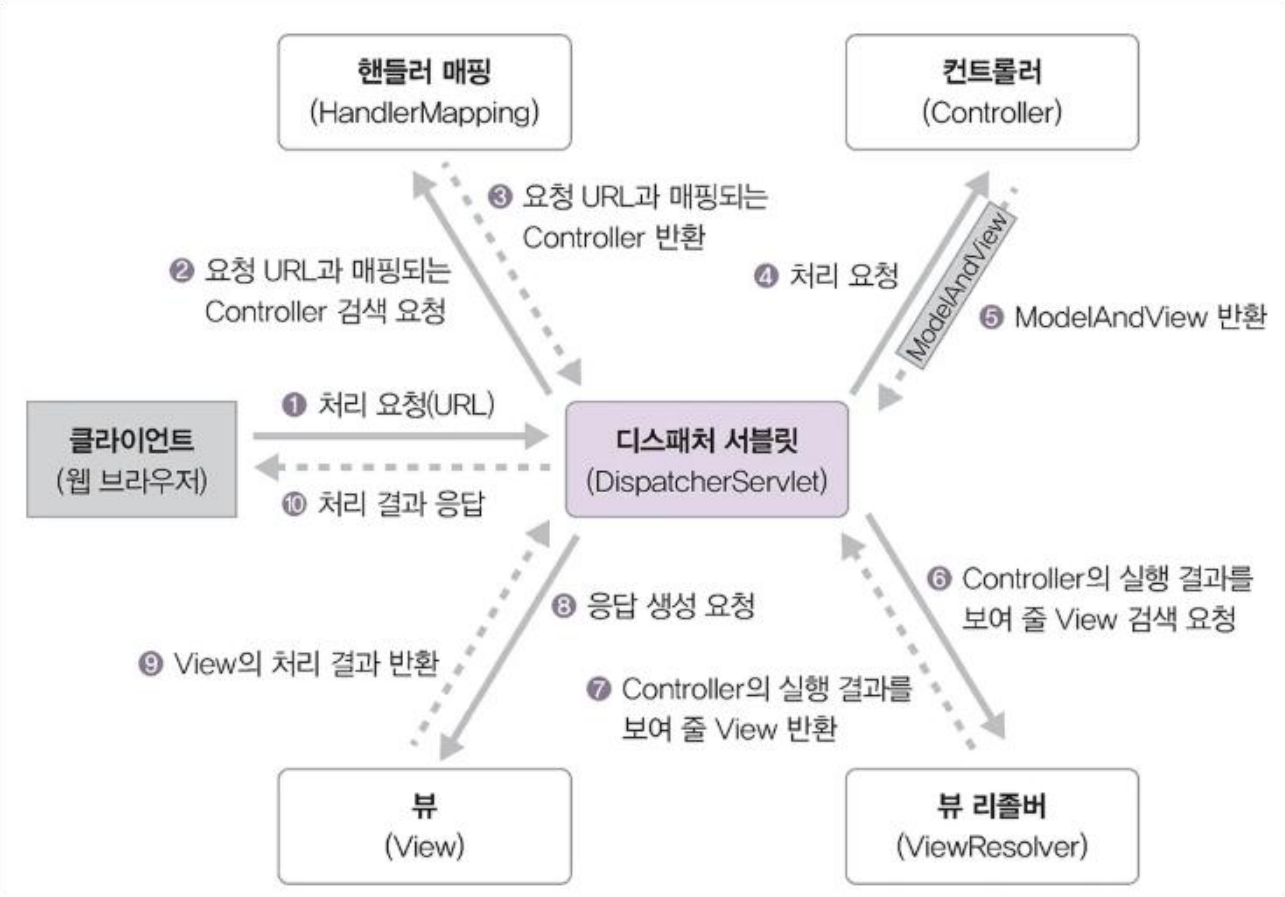
```
<?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- Root Context: defines shared resources visible to all other web components -->

</beans>
```

DispatcherServlet과 Front Controller

- Front Controller패턴은 모든 요청을 하나의 컨트롤러를 거치는 구조로 일관된 흐름을 작성하는데 도움이 됨

- 스프링 Web MVC에서는 DispatcherServlet이 Front Controller





servlet-context.xml

서블릿 컨텍스트 파일

- servlet-context.xml
- 웹 요청을 직접 처리할 컨트롤러의 매핑을 설정(HandlerMapping)
- 뷰를 어떻게 처리할지 설정(ViewResolver)

1. 컨트롤러 매핑 설정하기

- 요청 URL을 처리하는 컨트롤러에 매핑
- 요청 URL과 같은 컨트롤러의 @RequestMapping 애너테이션에 지정된 URL을 매핑
- <annotation-driven> 요소는 @Controller, @RequestMapping 같은 애너테이션을 사용할 때 필요한 빈 객체들을 자동으로 등록
- 핸들러 매핑과 핸들러 어댑터의 빈 객체도 대신 등록
- <annotation-driven> 요소를 사용하지 않으려면 핸들러 매핑과 핸들러 어댑터의 빈 객체를 등록해야 함

servlet-context.xml

서블릿 컨텍스트 파일

```
<!-- Enables the Spring MVC @Controller programming model --> <annotation-driven />


<!-- HandlerMapping -->
<beans:bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>


<!-- HandlerAdapter -->
<beans:bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>
```

servlet-context.xml

2. 정적 리소스 설정하기

- JS, CSS 이미지 등 리소스 파일 매핑 설정
- <resources> 요소는 서버에서 앞서 처리될 필요가 없는 정적 리소스 파일을 처리하는 역할
- <resources> 요소에 웹 애플리케이션의 물리적 경로 이름 설정
- 이 경로에 정적 리소스 파일들을 저장
- 웹 브라우저의 주소창에서 해당 리소스의 경로를 사용하여 직접 접속 가능

servlet-context.xml

2. 정적 리소스 설정하기

<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources in the ${webappRoot}/resources directory -->

<resources mapping="/resources/**" location="/resources/"/>

속성	설명
mapping	- 웹 요청 경로 패턴 설정 - 컨텍스트 경로를 제외한 나머지 부분의 경로와 매핑
location	- 웹 애플리케이션 내에서 실제 요청 경로의 패턴에 해당하는 자원 위치를 설정 - 위치가 여러 곳이면 각 위치를 쉼표로 구분
cache-period	- 웹 브라우저에 캐시 시간 관련 응답 헤더를 전송 - 초 단위로 캐시 시간을 지정 - 값이 0 이면 웹 브라우저가 캐시하지 않음 - 값을 설정하지 않으면 캐시 관련 응답 헤더를 전송하지 않음

servlet-context.xml

3. 뷰 매핑 설정하기

- 응답 결과를 보여 주려고 컨트롤러가 모델을 반환하고 디스패처 서블릿이 JSP 파일을 찾을 수 있게 설정
- 컨트롤러에서 설정한 뷰 이름으로 실제 사용할 뷰를 선택하는 뷰 리졸버 객체 설정
- 컨트롤러가 설정한 뷰 이름 앞뒤로 prefix 프로퍼티와 suffix 프로퍼티를 추가 → 실제로 사용될 뷰의 경로
- 컨트롤러에서 설정된 뷰 이름이 home → 뷰의 실제 경로 /WEB-INF/views/home.jsp

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
   <beans:property name="prefix" value="/WEB-INF/views/"/>
   <beans:property name="suffix" value=".jsp"/>
 </beans:bean>
```

servlet-context.xml

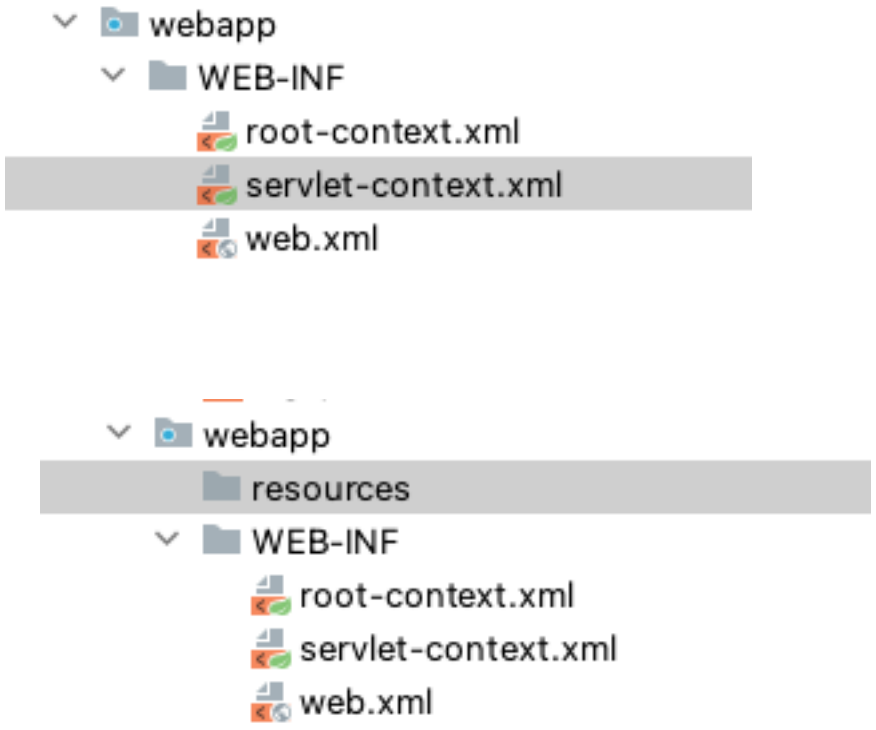
4. 자바 클래스의 빈 객체 설정하기

- 자바 클래스를 생성할 때 빈 객체를 자동으로 등록하기위한 설정
- <component-scan> 요소는 스프링 MVC에서 사용할 빈 객체를 등록하지 않아도 자동으로 인식할 수 있게 함
- <component-scan> 요소를 사용하지 않으면 @Controller가 선언된 컨트롤러를 빈 객체로 등록해야 함
- 의존 관계가 있는 자바 클래스가 있다면 <bean> 요소를 이용하여 빈 객체를 등록해야 함

```
<context:component-scan base-package=" net.fullstack10.springmvc.controller"/>
```

servlet-context.xml

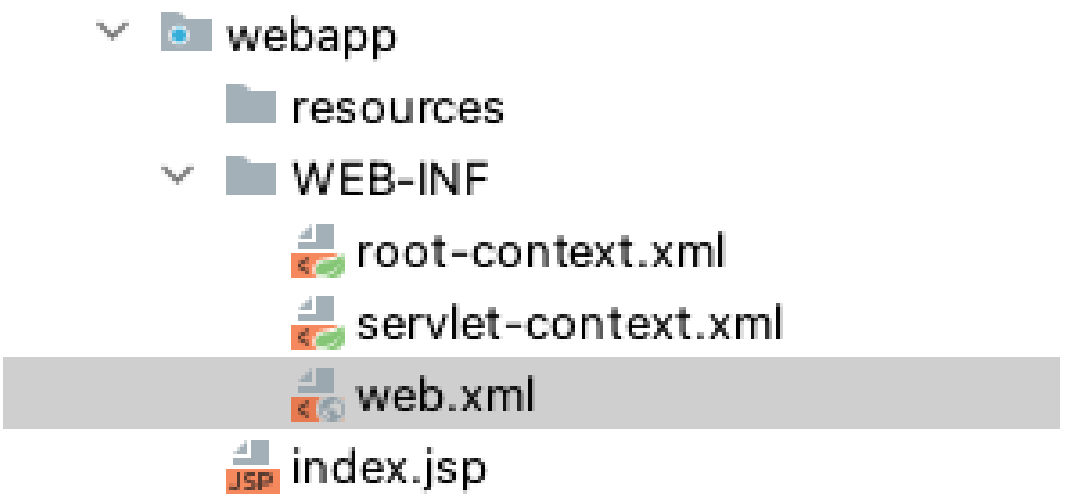
- spring-core와 달리 웹과 관련된 처리를 분리하기 위해서 작성하는 설정파일
- 반드시 구분할 필요는 없으나 일반적으로 계층별로 분리하는 경우가 많음




```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/mvc
    https://www.springframework.org/schema/mvc/spring-mvc.xsd">

  <mvc:annotation-driven></mvc:annotation-driven>

  <mvc:resources mapping="/resources/**" location="/resources/"></mvc:resources>


  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"></property>
    <property name="suffix" value=".jsp"></property>
  </bean>


</beans>
```

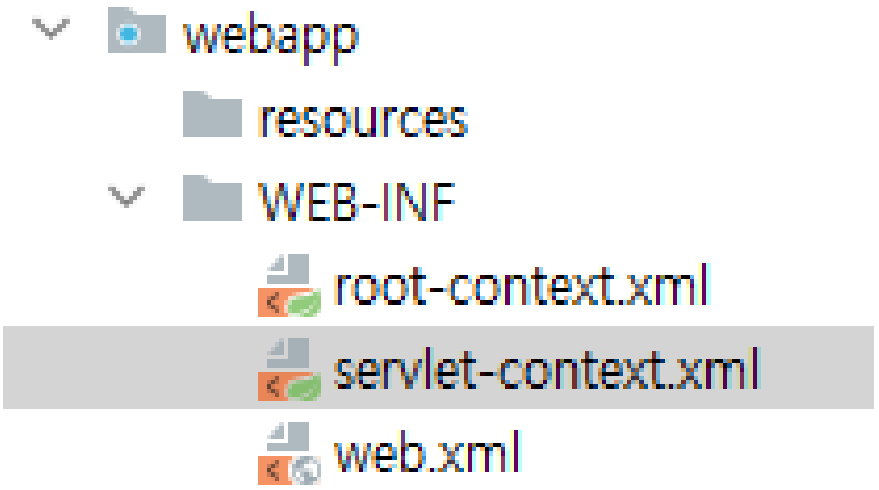
web.xml설정

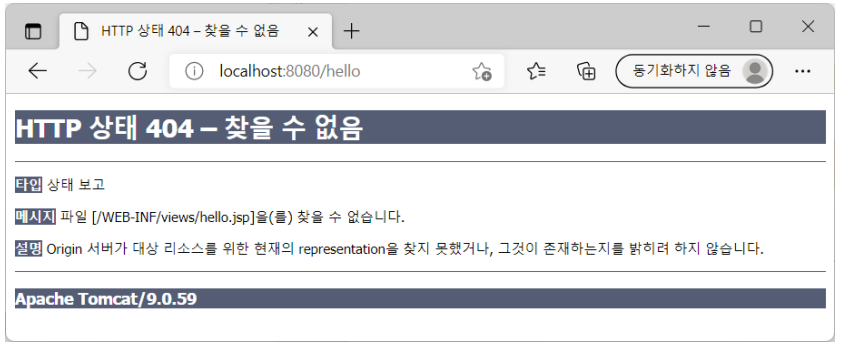
- 스프링 Web MVC에서 사용하는 DispatcherServlet을 web.xml에 설정




```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

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

servlet-context.xml의 component-scan






```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <mvc:annotation-driven></mvc:annotation-driven>
    <mvc:resources mapping="/resources/**" location="/resources/"></mvc:resources>


    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

    <context:component-scan base-package= " net.fullstack10.springmvc.controller"/>

</beans>
```

@RequestMapping과 파생 어노테이션들

- @RequestMapping은 경로를 처리하기 위한 용도로 사용
- 스프링에서 가장 많이 사용
- 특정 경로의 요청(Request) 지정하기 위해 사용
- 컨트롤러 클래스 선언부, 메소드에도 사용 가능
- 서블릿 중심 MVC 에서는 Servlet 상속 → doGet(), doPost() 메소드 오버라이드하여 사용
- 스프링 MVC 에서는 하나의 컨트롤러 이용하여 여러 경로의 호출 모두 처리 가능

@RequestMapping과 파생 어노테이션들

- method 속성
  - GET/POST 방식 처리
  - @RequestMapping(value="/register", method=RequestMethod.GET) ⇒ @GetMapping 과 동일
  - @RequestMapping(value="/register", method=RequestMethod.POST) ⇒ @PostMapping 과 동일
  - @RequestMapping(value="/register", method={RequestMethod.GET, RequestMethod.POST}) ⇒ 둘다 동시 처리
  - @RequestMapping(value="/register") ⇒ 둘다 동시 처리
- @GetMapping, @PostMapping
  - 스프링4 이후 추가
- 컨트롤러 클래스의 어노테이션이

  @RequestMapping("/todo") 이고,

  list() 메소드가 @RequestMapping("/list") 이므로 최종 경로 → /todo/list

@스프링 MVC 에서 주로 사용하는 어노테이션

- 스프링 MVC 에서 주로 사용하는 어노테이션
  - 컨트롤러 선언부에 사용하는 어노테이션
    - @Controller : 스프링의 빈(Bean) 으로 처리됨을 명시
    - @RestController : REST 방식의 컨트롤러임을 명시
    - @RequestMapping : 특정한 URL 패턴에 맞는 컨트롤러 명시
  - 메소드 선언부에 사용하는 어노테이션
    - @GetMapping/@PostMapping/@DeleteMapping/@PutMapping
      - HTTP 전송 방식(method) 에 따라 해당 메소드를 지정하는 경우 사용
      - @GetMapping, @PostMapping 이 주로 사용됨
      - @RequestMapping : GET/POST 를 모두 지원하는 경우 사용
      - @ResponseBody : REST 방식에서 사용, 문자열이나 JSON 데이터를 그대로 전송

@스프링 MVC 에서 주로 사용하는 어노테이션

- 스프링 MVC 에서 주로 사용하는 어노테이션
  - 메소드의 파라미터에 사용하는 어노테이션
    - @RequestParam : Request 에 있는 파라미터를 받아서 처리하는 경우 사용
    - @PathVariable : URL 경로의 일부를 변수로 처리하기 위해 사용
    - @ModelAttribute : 해당 파라미터는 반드시 Model 에 포함되어서 뷰(View) 로 전달 됨을 명시
      - 주로 기본자료형, Wrapper 클래스, 문자열에 사용
  - 기타
    - @SessionAttribute
    - @Valid
    - @RequestBody
      - REST 방식에서 문자열이나, JSON 데이터를 그대로 전송할 때 사용

파라미터의 자동 수집과 변환

- DTO나 VO등으로 자동으로 HttpServletRequest의 파라미터 수집
  - 기본자료형의 경우는 자동으로 형 변환처리가 가능
  - 객체자료형의 경우는 setXXX( )의 동작을 통해서 처리
  - 객체자료형의 경우 생성자가 없거나 파라미터가 없는 생성자가 필요
- @RequestParam: 파라미터의 이름을 지정하거나 기본값(defaultValue)를 지정할 수 있음
- @ModelAttribute를 이용해서 명시적으로 해당 파라미터를 view까지 전달하도록 구성할 수 있음

```
@GetMapping("/ex4")
public void ex4(Model model){

    log.info("----------------------");

    model.addAttribute("message", "Hello World");
}
```

파라미터의 자동 수집과 변환

- 특징
  - DTO 나 VO 등을 메소드의 파라미터로 설정하면, HttpServletRequest 의 파라미터를 자동으로 수집
  - 문자열, 숫자, 배열, 리스트, 첨부 파일 등도 가능
  - 동작 기준
    - 기본 자료형
      - 자동으로 형변환 처리 가능
    - 객체 자료형
      - setXXX() 동작 통해 처리
      - 객체 자료형은 생성자가 없거나 파라미터가 없는 생성자 필요(Java Beans)

파라미터의 자동 수집과 변환

- 단순(기본 자료형) 파라미터 수집
  - 자동으로 형변환 처리 가능
  - 스프링MVC 의 파라미터는 요청(Request)에 전달된 파라미터 이름 기준으로 동작
  - URL 의 query string 이 메서드의 매개변수로 자동 매핑
  - 파라미터 미전달 시에 문제 발생 가능 → @RequestParam 사용

```
→ WARN [org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver] Resolved
[org.springframework.web.method.annotation.MethodArgumentTypeMismatchException: Failed to convert value of type
'java.lang.String' to required type 'int'; nested exception is java.lang.NumberFormatException: For input string: ""]
```

  - HTTP 는 문자열로 데이터 전달
  - 컨트롤러는 문자열 기준으로 객체 처리 작업 진행
  - 날짜 관련 타입 처리 시 에러 발생

파라미터의 자동 수집과 변환

- 단순(기본 자료형) 파라미터 수집
  - @RequestParam 어노테이션 사용
    - defaultValue 속성으로 기본값 지정

```
@RequestMapping(value = "/view", method = RequestMethod.GET)
public void View(@RequestParam(name = "idx", defaultValue = "0") int idx) {

}
```

파라미터의 자동 수집과 변환

- 단순(기본 자료형) 파라미터 수집
  - Formatter 를 이용한 파라미터의 커스텀 처리
    - 날짜나 형 변환을 커스터마이징 해야 하는 경우 주로 사용
    - LocalDateFormatter 사용
      - controller 패키지에 formatter 패키지 작성
      - LocalDateFomatter 클래스 작성 → Formatter<LocalDate> 인터페이스 구현
      - servlet-context.xml 에 적용
        - FormattingConversionServiceFactoryBean 객체를 스프링의 빈(Bean) 으로 등록
        - 등록된 빈(Bean) 에 작성한 LocalDateFomatter 추가
      - conversion-service 등록 → <mvc:annotation-driven> 지정

```
package net.fullstack.springmvc.controller.formatter;

public class LocalDateFormatter implements Formatter<LocalDate> {
   @Override
   public LocalDate parse(String text, Locale locale) {
      return LocalDate.parse(text, DateTimeFormatter.ofPattern("yyyy-MM-
dd"));
   }

   @Override
   public String print(LocalDate object, Locale locale) {
      return DateTimeFormatter.ofPattern("yyyy-MM-dd").format(object);
   }
}
```

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
   <property name="prefix" value="/WEB-INF/views/"></property>
   <property name="suffix" value=".jsp"></property>
</bean>
<mvc:annotation-driven  conversion-service="conversionService" />
```

파라미터의 자동 수집과 변환

- 1) 객체 생성 → setXXX() 이용하여 처리
- 2) Lombok 이용시 @Setter, @Data 등 이용
- 3) 등록 순서
- /WEB-INF/views/bbs/regist.jsp 추가
- @GetMapping(“register”) 가 처리
- Submit 시 아래 @PostMapping(“register”) 가 처리
- DTO 패키지 추가
- DTO 클래스 추가, 필드 추가
- Controller 에 POST 방식 처리 메소드 추가
- @PostMapping(“/bbs/register”)
- DTO 객체 사용하도록 메서드 설정
- public void registerPost(DTO dto) { } → 자동으로 형변환 처리
- 내부에서 dto 처리

객체 자료형 파라미터 수집

1) Java Beans 형식으로 만들어진 클래스 타입은 자동으로 객체가 생성

객체 생성 → setXXX() 이용하여 처리

2) Lombok 이용시 @Setter, @Data 등 이용

3) 등록 순서

- /WEB-INF/views/bbs/regist.jsp 추가
  - @GetMapping("register") 가 처리
  - Submit 시 아래 @PostMapping("register") 가 처리
- DTO 패키지 추가
- DTO 클래스 추가, 필드 추가
- Controller 에 POST 방식 처리 메소드 추가
  - @PostMapping("/bbs/register")
  - DTO 객체 사용하도록 메서드 설정
    - public void registerPost(DTO dto) { } → 자동으로 형변환 처리
    - 내부에서 dto 처리

Model이라는 특별한 파라미터

- 스프링 MVC → 웹 MVC 와 동일한 방식
  - 모델 데이터를 JSP 까지 전달할 필요
  - 순수 서블릿 방식 → request.setAttribute() 로 데이터를 JSP 로 전달
  - 예전 서블릿에서 request.setAttribute( )로 처리했던 모델대신 사용
  - 스프링 MVC 방식 → Model 객체를 이용하여 처리
    - 초기 스프링 MVC → ModelAndView 객체 사용
    - 스프링 MVC3 이후 → Model(org.springframework.ui.Model) 파라미터 사용
- Model → addAttribute() 이용 → 뷰(View) 에 전달할 이름/값 객체 지정
  - 메소드의 파라미터에 Model을 선언하면 자동으로 객체 생성
  - Controller 의 메서드에 Model model 매개변수 추가
  - model.addAttribute(변수, 값) 설정
  - /WEB-INF/뷰경로/ → jsp 파일 추가
  - JSP 에서 EL : ${message}, JSTL : <c:out value="${message}"></c:out> 처럼 사용

Model이라는 특별한 파라미터

- Java Beans 와 @ModelAttribute
  - 스프링 MVC 컨트롤러는 파라미터로 getter/setter 를 이용하는 Java Beans 형식의 사용자 정의 클래스가 파라미터인 경우 → 자동으로 화면까지 객체를 전달

```
ex)
@GetMapping(“/test”)
public void test(TodoDTO todoDTO, Model model) { }
→ 별도 처리 없이 JSP 에서 ${todoDTO} 사용가능, 앞문자만 자동으로 소문자
```

  - @ModelAttribute() : 자동으로 생성된 변수 외에 지정하고 싶을 때 사용

```
ex)
@GetMapping(“/test”)
public void test(@ModelAttribute(“dto”) TodoDTO todoDTO, Model model){
}
```

RedirectAttributes와 리다이렉션

- PRG(Post Redirect Get) 패턴
  - POST 방식으로 처리 후에 Redirect 하여 GET 방식으로 특정 페이지로 이동하는 패턴
- RedirectAttributes
  - 스프링 MVC의 경우 반환타입이 문자열이고 redirect: 로 시작하는 경우 리다이렉트 처리
  - 리다이렉트시에 필요한 쿼리 스트링을 구성하기 위한 객체
  - 파라미터로 추가하면 자동으로 생성
  - addAttribute(키, 값) : 리다이렉트할 때 쿼리 스트링이 되는 값을 지정
    - 리다이렉트할 URL 에 쿼리 스트링으로 추가
  - addFlashAttribute(키, 값) : 일회용으로만 데이터를 전달하고 삭제되는 값을 지정
    - URL 에는 보이지 않고, JSP 에서 일회용으로 사용 가능
    - 페이지 refresh 하면 사라짐
  - redirect 방법
    - return “redirect:이동할 경로”;

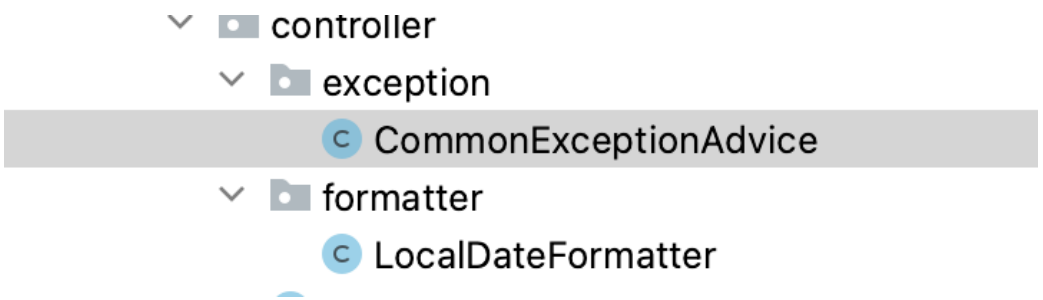
```
ex)
@GetMapping(“/test”)
public void test(RedirectAttributes redirectAttributes) {
  redirectAttributes.addAttribute(“키1”, “값1”);
  redirectAttributes.addFlashAttribute(“키2”, “값2”);
  return “redirect:/test2”;
}
```

다양한 리턴타입

- 상속이나 인터페이스와 달리 다양한 리턴 타입을 사용할 수 있다.
- void
  - 화면이 따로 있는 경우
  - 상황에 관계 없이 동일한 화면 출력시 사용
  - 컨트롤러의 @RequestMapping, @GetMapping 등 메소드에서 선언된 값을 그대로 뷰(View) 의 이름으로 사용
- 문자열
  - 화면이 따로 있는 경우
  - 상황에 따라 다른 화면 보여줄 경우 사용
  - 접두어 사용
    - redirect: → 리다이렉션 이용, 주로 이용
    - forward: → 브라우저의 URL 은 고정하고 내부적으로 다른 URL 로 처리하는 경우
- 객체
  - JSON 타입 리턴
- ResponseEntity
  - JSON 타입 리턴
- 배열
- 기본 자료형

스프링 MVC의 예외처리

- @ControllerAdvice 사용
  - 컨트롤러에서 발생하는 예외 처리 기능 제공
  - @ControllerAdvice 선언된 클래스 → 스프링의 빈(Bean)으로 처리됨
  - 해당 메서드에 @ExceptionHandler(예외처리클래스.class) 어노테이션 사용




```
package net.fullstack.springmvc.controller.exception;

import lombok.extern.log4j.Log4j2;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
@Log4j2
public class CommonExceptionAdvice {

   @ResponseBody
   @ExceptionHandler(NumberFormatException.class)
   public String exceptNumber(NumberFormatException numberFormatException){

      log.error("--------------------------------");
      log.error(numberFormatException.getMessage());

      return "NUMBER FORMAT EXCEPTION";
   }
}
```

범용적인 예외처리

- 예외 발생시 상위 타입인 Exception을 이용해서 예외 메시지를 구성
  - 디버깅 용도로 활용

```
@ResponseBody
@ExceptionHandler(Exception.class)
public String exceptCommon(Exception exception){

    log.error("-----------------------------------");
    log.error(exception.getMessage());

    StringBuffer buffer = new StringBuffer("<ul>");

    buffer.append("<li>" +exception.getMessage()+"</li>");

    Arrays.stream(exception.getStackTrace()).forEach(stackTraceElement -> {
        buffer.append("<li>"+stackTraceElement+"</li>");
    });
    buffer.append("</ul>");

    return buffer.toString();
}
```

404 에러 페이지 처리

1) 해당 메서드에 핸들러 및 Response 상태 추가

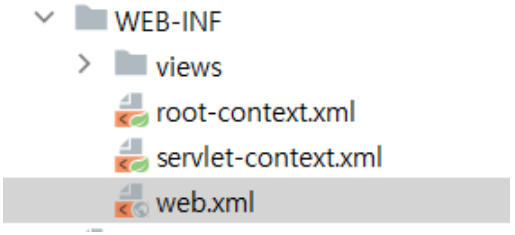
- @ExceptionHandler(NoHandlerFoundException.class)
- @ResponseStatus(HttpStatus.NOT_FOUND)
- public String 메서드() { return “404처리jsp파일”; }

```
@ExceptionHandler(NoHandlerFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public String notFound(){

    return "custom404";
}
```


```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
  <h1>Oops! 페이지를 찾을 수 없습니다!</h1>
</body>
</html>
```

2) web.xml 의 DispatcherServlet 설정 조정 후 프로젝트 재시작

- servlet 태그 내에 <init-param></init-param> 태그 추가




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

    <init-param>
        <param-name>throwExceptionIfNoHandlerFound</param-name>
        <param-value>true</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
</servlet>
```