Spring Framework (Java)

Getting Started

- JDK 설치 (oracle.com)
- Eclipse 설치(eclipse.org), STS(Spring Tool Suite) 설치 ([Help]→[Eclipse Marketplace]→Spring Tool Suite(STS) for Eclipse 설치)
    - eclipse.ini 파일; –vmargs 옵션 위에 -vm 옵션 추가 및 javaw의 정확한 위치 지정
- Tomcat 설치(tomcat.apache.org) 및 eclipse 연동
- 데이터베이스 구축, h2 (https://h2database.com)

New Project

- 프로젝트 생성
    - [File] → [New] → [Spring Legacy Project] 또는 [Other] 선택 후 생성 창에서 [Spring] → [Spring Legacy Project] 선택
    - Project Name 입력, Templates 선택 'Spring MVC Project' → com.reth.wiki 같은 형식의 패키지 명 입력
- 프로젝트 설정
    - 프로젝트의 컨텍스트 메뉴 [Properties] 선택 → 왼쪽 트리메뉴 [Project Facetes] 선택 Java 버전 선택 수정
    - 오른쪽 [Runtimes] 탭에서 Tomcat 버전 선택 → Apply → OK
- 확인
    - [Java Build Path] → [Libraries] 탭, 경로 확인

Basic File Structures

- 프로젝트
    - src/main/webapp/WEB-INF/web.xml; Servlet 컨테이너 설정
    - pom.xml

처리 영역	프레임워크	설명
Presentation	Struts	Struts 프레임워크는 UI Layer에 중점을 두고 개발된 MVC(Model View Controller) 프레임워크이다.
	Spring	Struts와 동일하게 MVC 아키텍처를 제공하는 UI Layer 프레임워크이다. 하지만 Struts 처럼 독립된 프레임워크는 아니고 Spring 프레임워크에 포함되어 있다.
Business	Spring (IoC, AOP)	Spring은 컨테이너 성격을 가지는 프레임워크이다. Springj의 IoC와 AOP 모듈을 이용하여 Spring 컨테이너에서 동작하는 엔터프라이즈 비즈니스 컴포넌트를 개발할 수 있다.

처리 영역	프레임워크	설명
Persistence	Hibernate or JPA	Hibernate는 완벽한 ORM(Object Relation Mapping) 프레임워크이다. ORM 프레임워크는 SQL 명령어를 프레임워크가 자체적으로 생성하여 DB 연동을 처리한다. JPA는 Hibernate를 비롯한 모든 ORM의 공통 인터페이스를 제공하는 자바 표준 API이다.
	Ibatis or Mybatis	Ibatis 프레임워크는 개발자가 작성한 SQL 명령어와 자바 객체(VO 혹은 DTO)를 매핑해주는 기능을 제공하며, 기존에 사용하던 SQL 명령어를 재사용하여 개발하는 차세대 프로젝트에 유용하게 적용할 수 있다. Mybatis는 Ibatis에서 파생된 프로임워크로서 기본 개념과 문법은 거의 같다.

- POJO(Plain Old Java Object);
- IoC(Inversion of Control); 제어의 역행. 느슨한 결합, 낮은 결합도
- AOP(Aspect Oriented Programming); 관점지향 프로그래밍. 공통 로직을 분리하여 높은 응집도의 개발 지원
- polymorphism 다형성
- Design Patterns; Factory Pattern

- 스프링 설정 파일 생성; 프로젝트의 src/main/resources 소스 폴더 선택 → 컨텍스트 메뉴 [New] → [Other] → 'Spring' 폴더의 'Spring Bean Configuration File' 선택 <Next> → File name에 'applicationContext' 입력 <Finish>

```
  ...
  <bean id="{id}" class="{class path}"/>
  ...
```

```
...
import org.springframework.context.support.AbstractAppicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

...
public class {class}
{

  public static void main(String[] args)
  {
    // 1. Spring 컨테이너 구동
    Abstract ApplicationContext factory = new
GenericXmlApplicationContext("applicationContext.xml");
    // 2. Spring 컨테이너로부터 필요한 객체를 요청(Lookup).
    TV tv = (TV)factory.getBean("tv");
  }
}
...
```

구현 클래스	기능
GenericXmlApplicationContext	파일 시스템이나 클래스 경로에 있는 XML 설정 파일을 로딩하여 구동하는 컨테이너이다.
XmlWebApplicationContext	웹 기방의 스프링 애플리케이션을 개발할 때 사용하는 컨테이너이다.

```
<?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">
</beans>
```

- beans 루트 엘리먼트
  - import 엘리먼트
  - bean 엘리먼트
    - init-method 속성
    - destryo-method 속성
    - lazy-init 속성; boolean
    - scope 속성; singleton, prototype
- 의존성 관리
  - Dependency Lookup
  - Dependency Injection
    - Constructor Injection 생성자 인젝션
    - Setter Injection
    - p 네임스페이스 사용; p:변수명-ref="참조할 객체의 이름이나 아이디" p:변수명="설정할 값"
  - 컬렉션 객체 설정
    - list, set, map, properties

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ...>

 <!-- 생성자 인젝션 -->
  <bean id="tv" class="polymorphism.SamsungTV">
    <constructor-arg ref="sony"></constructor-arg> <!-- 아래 bean의 id sony와 매칭 -->
    <constructor-arg value="2700000"></constructor-arg>
  </bean>
  <bean id="sony" class="polymorphism.SonySpeaker"></bean>
  <bean id="apple" class="polymorphism.AppleSpeaker"></bean>
  <!-- Setter Injection -->
  <bean id="tv" class="polymorphism.SamsungTV">
    <property name="speaker" ref="apple"></property>
    <property name="price" value="2700000"></property>
```

```
  </bean>
  <!-- p namespace -->
  <bean id ="tv" class="polymorphism.SamsungTV" p:speaker-ref="sony"
p:price="2700000"/>
```

```
  <bean ...>
    <property name="addressList">
      <list>
        <value>something 1</value>
        <value>something 2</value>
      </list>
      <set value-type="java.lang.String">
        <value>something 1</value>
        <value>something 2</value>
        <value>something 2</value>
      <set>
      <map>
        <entry>
          <key><value>key value 1</value></key>
          <value>value 1</value>
        </entry>
        <entry>
          <key><value>key value 2</value></key>
          <value>value 2</value>
        </entry>
      </map>
      <props>
        <prop key="key1">value 1</prop>
        <prop key="key2">value 2</prop>
      </props>
    </property>
```

- 이클립스에서 생성자 추가; 편집창에서 해당 클래스를 연 상태에서 <Alt> + <Shift> + <S> → [Generate Constructor using Fields] 선택
- 이클립스에서 인터페이스 자동 생성; 편집창에서 해당 클래스를 연 상태에서 <Alt> + <Shift> + <T> → [Extract Interface]
- 이클립스에서 Setter 메소드 자동 생성; <Alt> + <Shift> + <s> → [Generate Getters and Setters]
- 이클립스에서 p 네이스페이스 추가; 스프링 설정 파일에서 [Namespace] 탭 선택 → p 네임스페이스 선택

- 인터페이스 생성 → implements 클래스 작성

Annotation 기반 설정

- Context 네임스페이스 추가

    - 스프링 설정 파일 [Namespaces] 탭 선택 'context' 체크
- 컴포넌트 스캔(component-scan) 설정
    - <context:component-scan base-package="…"></context:component-scan>
- @Component("id"); 기본 생성자 필요, 클래스 선언 바로 앞에.

- 의존성 주입 설정

어노테이션	설명
@Autowired	주로 변수 위에 설정하여 해당 타입의 객체를 찾아서 자동으로 할당한다. org.springframework.beans.factory.annotation.Autowired
@Qualifier	특정 객체의 이름을 이용하여 의존성 주입할 때 사용한다. org.springframework.beans.factory.annotation.Qualifier
@Inject	@Autowired와 동일한 기능을 제공한다. javax.annotation.Resource
@Resource	@Autowired와 @qualifier의 기능을 결합한 어노테이션이다. javax.inject.Inject

- @Autowired; 변수 앞에
- @Qualifier("id"); 의존성 주입 대상이 두 개 이상일 때 @Autowired와 함께
- @Resource(name="id");

- 어노테이션과 XML 설정 병행하여 사용

어노테이션	위치	의미
@Service	XXXXServiceImpl	비즈니스 로직을 처리하는 Service 클래스
@Repository	XXXDAO	데이터베이스 연동을 처리하는 DAO 클래스
@Controller	XXXController	사용자 요청을 제어하는 Controller 클래스

- XxxVO, XxxDAO, XxxService, XxxServiceImpl
- VO(Value Object) DTO(Data Transfer Object) 작성 데이터 전달 목적
- DAO(Data Access Object) 데이터 베이스 연동
    - 드라이버 다운로드; pom.xml <dependencies> 태그 안에 추가
    - JDBC Utility 클래스; Connection 획득 및 해제 작업
    - 클래스 작성; @Repository("boardDAO"), insert, update, delete, get, list 메서드 작성
- Service 인터페이스 작성; DAO 클래스에서 <Alt> + <Shift> + <T>
- Service 구현 클래스 작성;
    - @Service("boardService")
    - @Autowired로 DAO 연결
- 실행; 컨테이너 구동 → BoardServiceImpl 객체 Lookup → VO 객체 → insert/update/list,… → 컨테이너 종료

Spring AOP

- IoC → 결합도, AOP(Aspect Oriented Programming) → 응집도
- 관심 분리 Separation of Concerns; 횡단 관심 Crosscutting Concerns, 핵심 관심 Core Concerns

- AOP 라이브러리 추가; pom.xml → Maven Dependencies 확인 → 스프링 설정파일 applicationContext.xml에서 [Namespaces] 탭 → aop 네임스페이스 추가 → applicationContext.xml 에 <bean> 등록 → aop 관련 설정 추가

```
...
  <aop:config>
    <aop:pointcut id="allPointCut" expression="execution(*
com.reth.biz..*Impl.*(..))"/>
    <aop:aspect ref="{bean id}">
      <aop:before pointcut-ref="allPointCut" method="{method}"/>
    </aop:aspect>
  </aop:config>
...
```

- 조인포인트 Joinpoint; 클라이언트가 호출하는 모든 비즈니스 메소드
- 포인트컷 Pointcut; 필터링된 조인포인트

- expression 속성; {리턴타입} {패키지경로}{클래스명}.{메소드명 및 매개변수}
  - {*} {com.multicampus.biz..}{*Impl}.{*(..)}
  - {*} {com.multicampus.biz..}{*Impl}.{get*(..)}
- 어드바이스 advice; 횡단 관심에 해당하는 공통 기능의 코드, 독립된 클래스의 메소드로 작성.
  - 어드바이스의 동작시점; before, after, after-returning, after-throwing, around
- 위빙 Weaving; 포인트컷으로 지정한 핵심 관심 메소드가 호출될 때, 어드바이스에 해당하는 횡단 관심 메소드가 삽입되는 과정
  - 처리 방식; 컴파일타임 Compiletime 위빙, 로딩타임 Loadingtime 위빙, 런타임 Runtime 위빙. 스프링에서는 런타임 위빙 방식만 지원.
- Aspect는 Pointcut과 advise의 결합으로서, 어떤 포인트컷 메소드에 대해서 어떤 어드바이스 메소드를 실행할지 결정

- AOP 엘리먼트
  - <aop:config>; AOP 설정에서 루트 엘리먼트, 여러 번 사용가능, 하위에는 <aop:pointcut>, <aop:aspect> 엘리먼트가 올 수 있다
  - <aop:pointcut>; 포인트컷 지정을 위해 사용, <aop:config>나 <aop:aspect>의 자식 엘리먼트로 사용 가능, 여러 개 정의 가능, 유일한 아이디를 할당하여 애스팩트를 설정할 때 포인트컷을 참조하는 용도로 사용
  - <aop:aspect>; 핵심 관심에 해당하는 포인트컷 메소드와 횡단 관심에 해당하는 어드바이스 메소드를 결합하기 위해 사용
  - <aop:advisor>; 포인트컷과 어드바이스를결합한다는 점에서 애스팩트와 같은 기능, 하지만 트랜잭션 설정 같은 몇몇 특수한 경우는 애스팩트가 아닌 어드바이저를 사용해야 한다.

- 포인트컷 표현식
  - 리턴타입 지정; 기본적인 방법은 *
  - 패키지 지정; 패키지 경로 지정할 때는 *, .. 캐릭터 이용
  - 클래스 지정; *, +
  - 메소드 지정; 메소드 지정할 때는 주로 *, 매개변수를 지정할 때는 ..
  - 매개변수 지정; .., *, 또는 정확한 타입 지정

리턴타입 지정	
표현식	**설명**
*	모든 리턴타입 허용
void	리턴타입이 void인 메소드 선택
!void	리턴타입이 void가 아닌 메소드 선택

리턴타입 지정	
표현식	**설명**
패키지 지정	
표현식	**설명**
com.reth.biz	정확하게 com.reth.biz 패키지만 선택
com.reth.biz..	com.reth.biz 패키지로 시작하는 모든 패키지 선택
com.reth..impl	com.reth 패키지로 시작하면서 마지막 패키지 이름이 impl로 끝나는 패키지 선택
클래스 지정	
표현식	**설명**
BoardServiceImpl	정확하게 BoardServiceImpl 클래스만 선택
*Impl	클래스 이름이 Impl로 끝나는 클래스만 선택
BoardService+	클래스 이름 뒤에 '+'가 붙으면 해당 클래스로부터 파생된 모든 자식 클래스 선택, 인터페이스 뒤에 '+'가 붙으면 해당 인터페이스를 구현한 모든 클래스 선택
메소드 지정	
표현식	**설명**
*(..)	가장 기본 설정으로 모든 메소드 선택
get*(..)	메소드 이름이 get으로 시작하는 모든 메소드 선택
매개변수 지정	
표현식	**설명**
(..)	가장 기본 설정으로서 '..'은 매개변수의 개수와 타입에 제약이 없음을 의미
(*)	반드시 1개의 매개변수를 가지는 메소드만 선택
(com.reth.user.UserVO)	매개변수로 UserVO를 가지는 메소드만 선택. 이때 클래스의 패키지 경로가 반드시 포함되어야 함
(!com.reth.user.userVO)	매개변수로 UserVO를 가지지 않는 메소드만 선택
(Integer, ..)	한 개 이상의 매개변수를 가지되, 천 번째 매개변수의 타입이 Integer인 메소드만 선택
(Interger, *)	반드시 두 개의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만 선택

- 어드바이스 동작 시점

동작 시점	설명
Before	비즈니스 메소드 실행 전 동작
After	- After Returning: 비즈니스 메소드가 성공적으로 리턴되면 동작 - After Throwing: 비즈니스 메소드 실행 중 예외가 발생하면 동작(마치 try-catch 블록에서 catch 블록에 해당) - After: 비즈니스 메소드가 실행된 수, 무조건 실행(try~catch~finally 블록에서 finally 블록에 해당)
Around	Around는 메소드 호출 자체를 가로채 비즈니스 실행 전후에 처리할 로직을 삽입할 수 있음

- JoinPoint 메소드; 비즈니스 메소드 이름, 메소드가 속한 클래스와 패키지 정보를 알아야 할 때 사용

메소드	설명
Signature getSignature()	클라이언트가 호출한 메소드의 시그니처(리턴타입, 이름, 매개변수) 정보가 저장된 Signature 객체 리턴
Object getTarget()	클라이언트가 호출한 비즈니스 메소드를 포함하는 비즈니스 객체 리턴
Object[] getArgs()	클라이언트가 메소드를 호출할 때 넘겨준 인자 목록을 Object 배열로 리턴

- Signature가 제공하는 메소드

메소드명	설명
String getName()	클라이언트가 호출한 메소드 이름 리턴
String toLongString()	클라이언트가 호출한 메소드의 리턴타입, 이름, 매개변수를 패키지 경로까지 포함하여 리턴
String toShortString()	클라이언트가 호출한 메소드 시그니처를 축약한 문자열로 리턴

어노테이션 기반 AOP

- 어노테이션 기반 AOP 사용을 위한 스프링 설정
  - 스프링 설정파일에 <aop:aspectj-autoproxy> 엘리먼트 선언

Annotation 설정	@Service public class LogAdvice{}
XML 설정	<bean id="log" class="com.reth.biz.common.LogAdvice"></bean>

- 포인트컷 설정; 메소드 앞에 @PointCut("execution(* com.reth.biz..*Impl.*(..))") 와 같이 설정
- 어드바이스 설정; 메소드 앞에 @Before("allPointcut()") 과 같이 설정

어노테이션	설명
@Before	비즈니스 메소드 실행 전에 동작
@AfterReturning	비즈니스 메소드가 성공적으로 리턴되면 동작
@AfterThrowing	비즈니스 메소드 실행 중 예외가 발생하면 동작(마치 try~catch 블록에서 catch 블록에 해당)
@After	비즈니스 메소드가 실행된 후, 무조건 실행(try~catch~finally 블록에서 finally 블록에 해당)
@Around	호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음

- 애스팩트 설정; @Service 어노테이션과 클래스 선언 사이에 @Aspect 어노테이션을 삽입하여 설정, 포인트컷 + 어드바이스 설정이 되어 있어야 함
- 외부 Pointcut 참조; 어노테이션 설정으로 변경하고부터는 어드바이스 클래스마다 포인트컷 설정이 포함되면서, 비슷하거나 같은 포인트컷이 반복 선언되는 문제 → 포인트컷을 외부에 독립된 클래스에 따로 설정
  - 사용할 모든 포인트컷을 PointcutCommon 클래스에 등록

```
package com.reth.biz.common;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class PointcutCommon
{
  @Pointcut("execution(* com.reth.biz..*Impl.*(..))")
  public void allPointcut() {}
  @Pointcut("execution(* com.reth.biz..*Impl.get*(..))")
  public void getPointcut() {}
}
```

```
...
  @Before("PointcutCommon.allPointcut()")
  public void beforeLog(JoinPoint jp)
  {
    ...
  }
...
```

Spring JDBC

- JdbcTemplate; JDBC 기반의 DB 연동, 템플릿 메소드 패턴(복잡하고 반복되는 알고리즘을 캡슐화해서 재사용하는 패턴) 적용

스프링 JDBC 설정

- 라이브러리 추가; pom.xml 파일에 DBCP 관련 <dependency> 설정 추가 → DBCP 라이브러리 정상 등록 확인

```
...
  <!-- DBCP -->
  <dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>Commons-dbcp</artifactId>
    <version>1.4</version>
  </dependency>
...
```

- DataSource 설정; JdbcTemplate 객체가 사용할 DataSource를 <bean> 등록. 트랜젝션 처리나 Mybatis 연동, JPA 연동에서도 DataSource가 사용되므로 매우 중요한 설정

```
...
  <!-- DataSource 설정-->
  <bean id="dataSource" calss="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="org.h2.Driver" />
    <property name="url" value="jdbc:h2:tcp://localhost/~/test" />
    <property name="username" value="sa" />
    <property name="password" value="" />
  </bean>
...
```

- 프로퍼티 파일을 활용한 DataSource 설정; PropertyPlaceholderConfigurer를 이용, 외부의 프로퍼티 파일을 참조하여 DataSource를 설정
    - src/main/resource 소스 폴더에 config 폴더 생성 → config 폴더에 database.properties 파일 작성
    - 스프링 설정 <context:property-placeholder> 엘리먼트 사용

```
jdbc.driver=org.h2.Driver
jdbc.url=jdbc:h2:tcp://localhost/~/test
jdbc.username=sa
jdbc.password=
```

```
...
  <!-- DataSource 설정-->
  <context:property-placeholder
location="classpath:config/database.properties" />
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
  </bean>
...
```

JdbcTemplate 메소드

- SQL 구문에 ?로 값 대입; ?의 갯수만큼 값들을 차례로 나열하거나, ?의 수만큼 값들을 세팅하여 배역 객체로 전달
- update(); INSERT, UPDATE, DELETE 구문 처리.
- queryForInt(); SELECT 구문으로 검색된 정수값을 리턴 받음.
- queryForObject(); SELECT 구문의 실행 결과를 특정 자바 객체(Value Object)로 매핑하여 리턴 받음. 검색 결과가 없거나 검색 결과가 두 개 이상이면 예외(IncorrectResultSizeDataAccessException)를 발생.
    - 중요! 검색 결과를 자바 객체(Value Object)로 매핑할 RowMapper 객체를 반드시 지정해야 한다.

```
package com.reth.biz.board.impl;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.reth.biz.board.BoardVO;
```

```
import com.springframework.jdbc.core.RowMapper;

public class BoardRowMapper implements RowMapper<BoardVO>
{
  public BoardVO mapRow(ResultSet rs, int rowNum) throws SQLException
  {
    BoardVO board = new BoardVO();
    board.setSeq(rs.getInt("SEQ"));
    board.setTitle(rs.getString("TITLE"));
    board.setWriter(rs.getString("WRITER"));
    board.setContent(rs.getString("CONTENT"));
    board.setRegData(rs.getDate("REGDATE"));
    board.setCnt(rs.getInt("CNT"));
    return board;
  }
}
```

- query(); SELECT 문의 결과가 목록일 때 사용, 검색 결과를 VO 객체에 매핑하려면 RowMapper 객체 사용

DAO 클래스 구현

- JdbcDaoSupport 클래스 상속

```

...
@Repository
public class BoardDAOSpring extends JdbcDaoSupport
{
...
  @Autowired
  public void setSuperDataSource(DataSource dataSource)
  {
    super.setDataSource(dataSource);
  }
...
}

...
```

- JdbcTemplate 클래스 <bean> 등록, 의존성 주입; <bean> 등록 → DAO 클래스에서 @Autowired 어노테이션 이용, 의존성 주입 → 서비스구현 *ServiceImpl 클래스가 DAO 객체를 이용하여 DB 연동 처리하도록 수정

```

...
  <!-- Spring JDBC 설정 -->
```

```
  <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
  </bean>
...
```

```
...
@Repository
public class BoardDAOSpring
{
  @Autowired
  private JdbcTemplate jdbcTemplate;

  ...
}
...
```

```
...
@Service("boardService")
public class BoardServiceImpl implements BoardService
{
  @Autowired
  private BoardDAOSpring boardDAO;
  public void insertBoard(BoardVO vo)
  {
    boardDAO.insertBoard(vo);
  }
  ...
}
...
```

트랜잭션 처리

- XML 기반의 AOP 설정만 사용 가능, 어노테이션 불가.
- 애스팩트 설정; <aop:advisor> 사용

- 트랜잭션 네임스페이스 등록; 트랜젝션 관련 네임스페이스 추가 [Namespace] 탭 선택하고 tx 네임스페이스 추가 → <beans> 루트 앨리먼트에 추가로 설정된 것 확인
- 트랜잭션 관리자 등록; 모든 트랜잭션 관리자는 PlatformTransactionManager 인터페이스를 구현한 클래스. commit(), rollback()
    - JDBC 기반; DataSourceTransactionManager 클래스 이용
    - JPA를 이용한 DAO 구현; JPATransactionManager를 등록

```
...
<!-- DataSource 설정-->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
...
</bean>

<!-- Transaction 설정-->
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>
...
```

- 트랜잭션 어드바이스 설정; <tx:advice> 앨리먼트 사용.

```
...
<!-- Transaction 설정-->
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true" />
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
...
```

<tx:method> 속성	
속성	**의미**
name	트랜잭션이 적용될 메소드 이름 지정
read-only	읽기 전용 여부 지정(기본값 false)
no-rollback-for	트랜잭션을 롤백하지 않을 예외 지정
rollback-for	트랜잭션을 롤백할 예외 지정

- AOP 설정을 통한 트랜잭션 적용; txPointcut으로 지정한 메소드가 호출될 때, txAdvice로 등록한 어드바이스가 동작하여 트랜잭션을 관리하도록 설정

```
...
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true" />
    <tx:method name="*" />
  </tx:attributes>
```

```
</tx:advice>

<aop:config>
  <aop:pointcut id="txPointcut" expression="execution(*
com.reth.biz..*(..))" />
  <aop:advisor pointcut-ref="txPointcut" advice-ref="txAdvice" />
</aop:config>
...
```

- 트랜잭션 설정의 동작
  1. Client에서 insertBoard() 호출
  2. Container의 BoardServiceImpl에서 해당 비즈니스 로직 수행
  3. txAdvice에 의해 After 어드바시으가 동작
  4. txManager의 commit() 혹은 rollback() 메소드를 호출

Model 1 아키텍처

- Client ↔ Container(JSP as a Controller + View, JavaBeans as a Model) ↔ DBMS
- JSP 파일이 가장 중요 역할 수행; Controller + View 기능 처리 → 역할 구분이 명확하지 않음, JSP 파일에 대한 디버깅과 유지보수에 많은 어려움
- Model 1 구조의 단점 보완 → Model 2 == MVC; Model, View, Controller

> **Forward와 Redirect 차이**
>
> - 포워드 방식은 RequestDispatcher를 이용하여 응답으로 사용할 JSP 화면으로 넘겨서, 포워드된 화면이 클라이언에 전송되는 방식. 한 번의 요청과 응답으로 처리. 실행속도는 빠르지만 클라이언트 브라우저에서 URL이 바뀌지 않아 응답이 어디에서 들어왔는지 확인 불가
> - 리다이렉트는 요청된 JSP에서 일단 브라우저로 응답 메시지를 보냈다가 다시 서버로 재요청하는 형식. 포워드 방식과 달리 일단 응답이 브라우저로 들어간 다음, 재요청하는 방식. 응답이 들어온 파일로 브라우저의 URL이 변경, 두 번의 요청과 응답으로 처리되므로 실행 속도는 포워드 방식보다 느림.

Model 2 아키텍처

- Model 1 아키텍처는 엔터프라이즈 시스템에 부적합
- Client ↔ Container(Servlet as a Controller, JSP as a View, JavaBeans as a Model) ↔ DBMS
- Controller 구현
  1. 서블릿 생성 및 등록; 프로젝트 탐색창 src/main/java 컨텍스트 메뉴 → [New] → [Servlet] → Java Package에 com....view.controller, Class name에 DispatcherServlet 입력 → [Next] → Name에 action, URL mappings에 /action을 더블 클리하여 Pattern을 *.do로 설정 → [Finish] ⇒ DispatcherServlet 클래스가 만들어지는 순간 WEB-INF/web.xml 파일에 서블릿 관련 설정이 자동으로 추가됨. <description>, <display-name> 의미없는 설정이므로 제거

2. Controller 서블릿 구현; doGet(), doPost(), process(). doPost()에 한글 처리 인코딩 추가 request.setCharacterEncoding(“”); process()에서 분기 처리

MVC 프레임워크

- Controller를 서블릿 클래스 하나로 구현하는 것은 DispatcherServlet 클래스가 복잡하게 구현되어 복잡도 증가 → 프레임워크의 Controller를 사용
- Container
  - DispatcherServlet → HandlerMapping
  - DispatcherServlet → Controller
  - DispatcherServlet ← String ← Controller
  - DispatcherServlet → ViewResolver

클래스	기능
DispatcherServlet	유일한 서블릿 클래스로서 모든 클라이언트의 요청을 가장 먼저 처리하는 Front Controller
HandlerMapping	클라이언트의 요청을 처리할 Controller 매핑
Controller	실질적인 클라이언트의 요청 처리
ViewResolver	Controller가 리턴한 View 이름으로 실행될 JSP 경로 완성

- MVC 프레임워크 구현
  1. Controller 인터페이스 작성
  2. Controller 구현체
  3. HandlerMapping 클래스 작성
  4. ViewResolver 클라스 작성
  5. DispatcherServlet 수정

> **EL과 JSTL**
>
> - EL(Expression Language); JSP 2.0에서 추가된 스크립트 언어, 기존의 표현식(Expression)을 대체하는 표현 언어.
> - <%= session.getAttribute(“userName”) %> → ${username} 과 같이 표현
> - JSTL(JSP Standard Tag Library); Scriptlet에서 if, for, switch 등과 같은 자바 코드를 사용해야 할 때, 자바 코드을을 태그 형태로 사용할 수 있도록 지원
> - http://www.tutorialpoint.com/jsp/

Spring MVC 구조

1. Client → (request) → DispatcherServlet
2. DispatcherServlet → HandlerMapping
3. DispatcherServlet → Controller
4. DispatcherServlet ← ModelAndView ← Controller
5. DispatcherServlet → ViewResolver
6. DispatcherServlet → View

1. 클라이언트로부터의 모든 “.do” 요청을 DispatcherServlet이 받는다
2. DispatcherServlet은 HandlerMapping을 통해 요청을 치리할 Controller를 검색.
3. DispatcherServlet은 검색된 Controller를 실행하여 클라이언트의 요청을 처리.
4. Controller는 비즈니스 로직의 수행 결과로 얻어낸 Model 정보와 Model을 보여줄 View 정보를 ModelAndView 객체에 저장하여 리턴
5. DispatcherServlet은 ModelAndView로부터 View 정보를 추출하고, ViewResolver를 이용하여 응답으로 사용할 View를 얻어냄.
6. DispatcherServlet은 ViewResolver를 통해 찾아낸 View를 실행하여 응답 전송

1. DispatcherServlet 등록 및 스프링 컨테이너 구동
    1. DispatcherServlet 등록; WEB-INF/web.xml
    2. 스프링 컨테이너 구동; DispatcherServlet.java
    3. 스프링 설정 파일 등록; 프로젝트 탐색창 WEB-INF 컨텍스트 메뉴 → [New] → [Other] → Spring 폴더에서 Spring Bean configuration file 선택 → [Next] → File name에 action-servlet.xml 파일 명입력 → [Finish] → 생성된 파일의 이름과 경로 확인
2. 스프링 설정 파일 변경; WEB-INF 폴더 아래 config 폴더 생성 → 위에 생성한 파일 action-servlet.xml 파일 이동 → presentation-layer.xml로 이름 변경 → web.xml 파일 열고 DispatcherServlet 클래스 등록한 곳에 <init-param> 성정 추가

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/config/presentation-layer.xml</param-value>
  </init-param>
</servlet>
```

```
public class DispatcherServlet extends HttpServlet
{
  private String contextConfigiLocation;
  public void init(ServletConfig config) throws ServletException
  {
    contextConfigLocation = config.getInitParameter("contextConfigLocation");
    new XmlWebApplicationContext(contextConfigLocation);
  }
}
```

1. 인코딩설정; web.xml 파일에 CharacterEncodingFilter 클래스를 필터로 등록

```
...
  <filter>
    <filter-name>characterEncoding</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>EUC-KR</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>characterEncoding</filter-name>
    <url-pattern>*.do</url-pattern>
  </filter-mapping>
...
```

Spring MVC 적용

- 기존의 com....view.controller의 패키지 삭제
- Controller의 handleRequest의 리턴타입 String → ModelAndView

1. Controller 구현; *Controller.java
2. HandlerMapping 등록; presentation-layer.xml에 HandlerMapping과 Controller를 <bean> 등록

```
<?xml version="1.0" encoding="UTF-8" ?>
....
  <!-- HandlerMapping 등록-->
  <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/login.do">login</prop>
      </props>
    </property>
  </bean>
  <!-- Controller 등록-->
  <bean id="login" class="com.reth.view.user.LoginController"></bean>
</beans>
....
```

- ViewResolver 활용; JSP를 View로 사용하는 경우에는 InternalResourceViewResolver 사용
  1. /WEB-INF/board/ 생성, getBoardList.jsp, getBoard.jsp 파일 이동
  2. Controller 수정

```
....
```

```
  <!-- ViewResolver 등록-->
  <bean id="viewResolver"
class="org.springframework.web.servlet.viewInternalResourceResolver">
    <property name="prefix" value="/WEB-INF/board/"></property>
    <property name="suffix" value=".jsp"></property>
  </bean>
....
```

어노테이션 기반 MVC

- 어노테이션 관련 설정; <beans> 루트 엘리먼트에 context 네임스페이스 추가. HandlerMapping, Controller, ViewResolver 클래스에 대한 <bean> 등록 삭제하고 <context:component-scan> 엘리먼트로 대체

```
<?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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">
  <context:component-scan base-package="com.reth.view">
  </context:component-scan>
</beans>
```

- 모든 Controller 클래스가 스캔 범위에 포함되도록 <context:componet-scan> 엘리먼트의 base-package 속성에 Controller 클래스들이 있는 가장 상위 패키지인 'com.reth.view'를 등록
- 또한, 어노테이션 활용에 집중하기 위해, 스프링 설정 파일에 등록한 ViewResolver 설정을 삭제. jsp 파일의 위치도 원래대로 src/main/webapp 폴더 밑으로

@Controller

- @Component를 상속한 @Controller는 @Controller가 붙은 클래스의 객체를 메모리에 생성하는 기능 제공. 단순히 객체를 생성하는 것에 그치지 않고, DispatcherServlet이 인식하는 Controller로 만들어 준다.
- 만일 @Controller를 사용하지 않는다면, 모든 컨트롤러 클래스는 반드시 스프링에서 제공하는 Controller 인터페이스를 구현해야 한다. 그리고 handleRequest() 메소드를 반드시 재정의하여 DispatcherServlet이 모든 Controller의 handleRequest() 메소드를 호출할 수 있도록 해야한다.

```
import org.springframework.stereotype.Controller;
```

```
@Controller
public class InsertBoardController
{
}
```

> **import 정리**
>
> 만약 이미 다른 Controller가 import에 등록되었다면, 클래스 위에 @Controller를 설정할 때 자동완성이 다음처럼 이상하게 입력될 수 있다.
>
> ```
> import org.springframework.web.servlet.mvc.Controller;
> @org.springframework.stereotype.Controller
> public class InsertBoardController implements Controller
> {
>  public ModelAndView handleRequest(HttpServletRequest request,
> HttpServletResponse response)
>  {
>  }
> }
> ```

이때는 import된 Controller를 제거하고 다시 @Controller를 자동완성하면 깔끔하게 처리된다. 또는 이클립스의 <Ctrl>+<Shift>+<O> 단축키를 눌러 import를 정리해도 된다.

@RequestMapping

- @RequestMapping을 이용하여 HandlerMapping 설정을 대체

```
....
import org.springframework.web.bind.annotation.RequestMapping;
....
  @RequestMapping(value="/insertBoard.do")
  public void insertBoard(HttpServletRequest request)
  {
  }
....
```

클라이언트 요청 처리

- 서블릿 객체의 service() 메소드가 호출되는 과정
  - Client → HTTP요청 → Servlet Container

1. HTTP요청; Start line, Message Header, Message Body;
    - 서블릿 컨테이너는 클라이언트의 HTTP 요청이 서버에 전달되는 순간
2. HttpServletRequest
    - HttpServletRequest 객체를 생성하고 HTTP 프로토콜에 설정된 모든 정보를 추출하여 HttpServletRequest 객체에 저장
3. DispatcherServlet의 service 메서드의 HttpServletRequest 인자
    - 그리고 이 HttpServletRequest 객체를 service() 메소드를 호출할 때, 인자로 전달

- Controller 객체의 메소드가 호출되는 과정
    - Client → Spring Container

1. 객체생성; 매개변수에 해당하는 VO 객체 생성
2. 사용자 입력 값 전달; 사용자가 입력한 파라미터 값들을 추출하여 VO 객체에 저장. 이 때, VO 클래스의 setter 메소드들이 호출된다.
3. 사용자가 요청한 액션에 해당 메소드를 호출할 때, 사용자 입력값들이 설정된 VO 객체가 인자로 전달된다.

> **리다이렉트로 넘기기**
>
> Controller 메소드가 실행되고 View 경로를 리턴하면 기본이 포워딩 방식이므로 url은 변경되지 않는다. 따라서 포워딩이 아닌 리다이렉팅을 원할 때는 “redirect:“라는 접두사를 붙여야 한다.

- @RequestMapping(value=”/login.do”, method=RequestMethod.GET)
    - value; url 위치
    - method; RequestMethod.GET, RequestMethod.POST

- JSP에서 Command 사용; ${…} 구문

- @ModelAttribute; Command 객체의 이름을 변경, View(JSP)에서 사용할 데이터를 설정하는 용도
- @RequestParam; 파라미터 정보 추출
- @SessionAttributes;

프레젠테이션 레이어와 비즈니스 레이어 통합

- Spring MVC 기반; request → Servlet(DispatcherServlet ←LOADING- presentation-layer.xml) → Spring Container(Controller -use→ DAO)
- Controller는 DAO 객체를 직접 이용해서는 안되며, 반드시 비즈니스 컴포넌트를 이용.
- 비즈니스 컴포넌트; VO, DAO, Service Interface, Service 구현 클래스
    1. DAO 클래스 교체
    2. AOP 설정 적용
    3. 비즈니스 컴포넌트 의존성 주입

- 2-Layered 아키텍처
    - Presentation Layer(MVC); presentation-layer.xml -LOADING→ DispatcherServlet (→ Controller & -Data→ View(JSP))
    - Business Layer; Controller → ServiceImpl(←implements- Service Interface) → use DAO & use VO

- ContextLoaderListner; web.xml 파일에 <context-param> 설정 추가

파일 업로드

1. 파일 업로드 입력 화면
2. Command 객체 수정; VO 클래스에 org.springframework.web.multipart.MultipartFile 타입의 변수 추가
3. FileUpload 라이브러리 추가; pom.xml 에 dependency 추가
4. MultipartResolver 설정; CommonsMultipartResolver 를 <bean> 등록
5. 파일 업로드 처리

- 예외처리
  - 어노테이션 기반 예외처리; @ControllerAdvice, @ExceptionHandler 컨트롤의 메소드 수행 중 발생하는 예회를 일괄적으로 처리. presentation-layer.xml에 예외 처리 관련 어노테이션 사용을 위한 설정 추가
  - XML 기반 예외처리; presentation-layer.xml 파일에 SimpleMappingExceptionResolver 클래스를 <bean> 등록

다국어 처리

- 메시지 파일 작성; Java Resources/sr/main/resources/message**“).access(“hasRole('USER')”)
  - antMatchers(”/login“).permitAll()
  - public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
  - withUser(“firstuser”).password(“password1”)
  - .roles(“USER”, “ADMIN”)
- 로그아웃

스프링 부트

- 스프링 부트의 기본 목표
  - 스프링 기반 프로젝트로 빠르게 구축
  - 의견을 가져라. 아니면 일반적인 사용법을 기본으로 한다. 기본값과의 차이를 처리하는 설정 옵션은 제공
  - 다양한 비기능적 특징을 제공
  - 코드 생성과 많은 XML 설정의 사용을 피해라
- 스프링 부투의 비기능적 특징
  - 광범위한 프레임워크, 서버와 스펙의 버전 관리 및 설정에 관한 기본 처리
  - 애플리케이션 시큐리티를 위한 기본 옵션
  - 확장 옵션이 있는 기본 애플리케이션 매트릭스
  - 상태 체크로 기본 애플리케이션 모니터링
  - 외부 설정의 여러 옵션

- 스프링 부트로 애플리케이션 구축
  1. pox.xml 파일에 spring-boot-starter-parent 구성
  2. 요구되는 스타터 프로젝트로 pom.xml 파일 구성
  3. 애플리케이션을 실행할 수 있도록 spring-boot-maven-plugin 구성

4. 첫 번째 스프링 부트 launch 클래스 생성

- 스프링 이니셜라이저 시작 SPRING INITIALIZER
  1. 빌드 도구(maven / gradle) 선택
  2. 사용할 스프링 부트 버전 선택
  3. 컴포넌트의 그룹 ID 및 아티팩트 ID 구성
  4. 프로젝트에 원하는 스타터(의존 관계) 선택.
  5. 컴포넌트 패키지 방법 선택(JAR/WAR)
  6. 사용하려는 자바 버전 선택
  7. 사용하려는 JVM 언어 선택

- 프로젝트 구조
  - pom.xml
  - src
    - main
      - java
        - com
          - reth
            - spring
              - FirstSpringInitializrApplication.java
      - resources
        - application.properties
        - static
        - templates
    - test
      - java
        - com
          - reth
            - spring
              - FirstSpringInitializrApplicationTests.java

- 자동설정
  - @ConditionalOnClass { Servlet.class, DispatcherServlet.class, WebMvcConfigurerAdapter.class }: 언급된 클래스 중 하나라도 클래스 패스에 있으면 자동 설정이 사용된다. 웹 스타터 프로젝트를 추가할 때 언급된 모든 클래스의 의존 관계를 가져온다. 그러면 자동 설정이 활성화된다.
  - @ConditionalOnMissingBean(WebMvcConfigurationSupport.class): 자동 설정은 애플리케이션이 WebMvcConfigurationSupport.class 클래스의 빈을 명시적으로 선언하지 않은 경우에만 활성화된다.
  - @AutoConfigurerOrder(Ordered.HIGHEST_PRECEDENCE + 10): 특정 자동 설정의 우선순위를 지정한다.
  - @ConditionalOnBean(ViewResolver.class): ViewResolver.class가 클래스 패스에 있으면 빈을 생성한다.
  - @ConditionalOnMissingBean(name = “viewResolver”, value = ContentNegotiatingViewResolver.class): viewResolver 이름과 ContentNegotiatingViewResolver.class 유형의 명시적으로 선언된 빈이 없으면 이 빈을 생성한다.
  - 나머지 메소드는 뷰 리졸버에서 구성된다

- 애플리케이션 구성 외부화

    - application.properties 시작
    - application.properties를 통한 프레임워크 사용자 정의
        - 로깅 구성
        - 임베디드 서버 구성 사용자 정의
        - 스프링 MVC 설정
        - 스프링 스타터 시큐리티 구성
        - 데이터 소스, JDBC와 JPA 사용자 정의
        - 기타 구성 옵션
    - 애플리케이션별로 사용자 정의 속성 정의
        - 구성 속성을 통한 타입세이프 구성관리
    - 다른 환경에 프로파일 생성
        - 액티브 프로파일을 기반으로 동적 빈 구성
    - 애플리케이션 구성을 제공하기 위한 기타 옵션
    - YAML 구성

- 임베디드 서버
    - 전통적인 자바 애플리케이션 배포; APP. WAR, 톰캣, 자바
    - 임베디드 서버; 자바, ((임베디드 톰캣), APP. WAR)
    - 제티 및 언더토우 임베디드 서버
    - JAR를 사용하는 대신 기존 WAR 파일 빌드
- 개발자 도구를 사용해 생산성 향상
    - 브라우저에서 실시간 리로드 활성화
- 애플리케이션 모니터링에 스프링 부트 액추에이터 사용
    - HAL 브라우저를 사용해 액추에이터 엔드포인트 찾기
    - 애플리케이션 구성 속성
    - 환경 세부 정보
    - 애플리케이션 상태 모니터링
    - 매핑 정보 얻기
    - 빈 구성으로 디버깅
    - 중요한 매트릭스 탐색
    - 자동 설정의 디버그 정보 얻기
    - 디버깅

스프링 프레임워크 심화

- 스프링과 AspectJ로 AOP 살펴보기
    - 크로스 컷팅 및 AOP 탐색; 로깅, 시큐리티, 성능 추적
    - 중요한 AOP 용어
        - 어드바이스 advice
        - 포인트컷 PointCut
        - 포인트 컷 + 어드바이스 ⇒ 애스팩트 aspect
        - 위빙 weaving
            - 컴파일 타임 위빙 Compile-Time Weaving; 입력은 소스코드고 출력은 위빙이 있는 컴파일된 클래스
            - 바이너리 위빙 Binary Weaving; 코드가 컴파일된 후에 수행. 입력은 컴파일된 클래스 파일 또는 jar 파일이고 출력은 위빙이 있는 컴파일된 클래스나 jar 파일이다.
            - 런타임 위빙 Runtime Weaving; 위빙은 클래스가 JVM에 로드되기 직전에 수행된다.

        - AOP 프레임워크; AspectJ → 컴파일 타임 위빙
        - AOP 실습; spring-boot-starter-aop → 비즈니스 시나리오 설정 → 포인트컷 식별 → aspect 정의; 인터셉션 포인트
            - @Before; 메소드를 실행하기 전
            - @After; 메소드 실행 후. 메소드가 예외를 발생시키더라도 실행
            - @AfterReturning; 메소드가 성공적으로 실행된 후
            - @AfterThrowing; 메소드 호출 후 예외가 발생했다
            - @Around; 메소드 실행을 완전히 제어
            - 사용자 정의 AOP 어노테이션 정의;

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogEverything {
}
```

```
@LogEverything
public void placeOrder(int value) {
```

- 스프링으로 작업 스케줄링하기
    - @Scheduled로 작업 스케줄링 하기
    - @Async를 사용해 비동기식으로 작업 실행하기 → @Async 메소드에서 값 반환하기
    - 작업 실행자 executor
- 스프링 프레임워크를 이용한 스크립팅
    - JSR 233 - 자바TM 플랫폼용 스크립팅
        - 자바로 자바스크립트 코드 실행하기
        - 자바에서 그루비 코드 실행하기
    - 스프링 엔터프라이즈 애플리케이션에서 그루비 코드 실항하기; 그루비 코드를 스프링 컨텍스트 XML로 인라이닝하기

스프링 부트의 REST API

- SOAP (Simple object Access Protocol)를 사용하는 XML 기반 → REST
- REST (Representational State Transfer)
    - 서버 Server; 서비스 공급자. 클라이언트에게 서비스를 제공
    - 클라이언트 Client; 서비스 소비자. 브라우저 또는 다른 시스템일 수 있다.
    - 리소스 Resource; 모든 정보는 리소스가 될수 있다(사람, 이미지, 비디오 또는 판매하는 제품 등)
    - 리프리젠테이션 Representation; 리소스를 표현하는 구체적인 방법. 예를 들어 JSON, XML 나 HTML을 통해 제품 리소스를 나타낼 수 있다. 클라이언트마다 리소스의 다른 표현을 요청할 수 있다.
    - REST 제약 조건

        - 클라이언트-서버 Client-server; 서버(서비스 공급자)와 클라이언트(서비스 소비자)가 있어야 한다. 새로운 기술이 등장함에 따라 서버와 클라이언트의 느슨한 결합과 독립적 진화를 가능하게 한다
        - 무상태 Stateless; 각 서비스는 무상태여야 한다. 후속 요청은 일시적으로 저장되는 이전 요청의 일부 데이터에 의존하면 안 된다. 메시지는 스스로 설명할 수 있어야 한다.
        - 통일된 인터페이스 Uniform interface; 각 리소스에는 리소스 식별자가 있다. 웹 서비스의 경우, URI(Uniform Resource Identifier) 예제인 /users/Jack/todos/1을 사용한다. 여기서 URI Jack은 사용자 이름이고 1은 검색하려는 todo의 ID다.
        - 캐시 가능 Cacheable; 서비스 응답은 캐시가 가능해야 한다. 각 응답은 캐시 가능한지 여부를 표시해야 한다.
        - 레이어 시스템 Layered system; 서비스 소비자는 서비스 공급자와 지접 연결돼서는 안 된다. 요청을 캐시할 수 있으므로 클라이언트는 중간 레이어에서 캐시된 응답을 가져올 수 있다.
        - 리프리젠테이션을 통한 리소스 연산; 리소스는 여러 리프리젠테이션을 가질 수 있다. 리프리젠테이션이 포함된 메시지로 리소스를 수정할 수 있어야 한다.
        - HATEOAS; RESTful 애플리케이션의 소비자는 하나의 고정 서비스 URL만을 알아야 한다. 모든 후속 리소스는 리소스 표현에 포함된 링크에서 검색할 수 있어야 한다.

- 첫 번째 REST API 설계
    - REST API 연산을 위한 요청 메소드 및 URI 결정

HTTP 요청 메소드	
HTTP 요청 메소드	**오퍼레이션**
GET	읽기 - 리소스의 세부 정보 검색
POST	생성 - 새로운 아이템 또는 리소스 생성
PUT	업데이트/교체
PATCH	리소스 일부 업데이트/수정
DELETE	삭제

- 스프링 부트로 Hello World API 만들기
    - Rest Controller
        - @RestController; @RestController 어노테이션은 @ResponseBody와 @Controller 어노테이션의 조합을 제공. 일반적으로 REST 컨트롤러를 생성하는 데 사용
        - @GetMapping("welcome"); @GetMapping은 @RequestMapping(method = RequestMethod.GET)의 바로가기이다. 어노테이션이 있는 메소드는 welcome URI의 GET 요청을 처리한다
    - REST API 작성; 요청 실행
    - name 경로 변수로 welcome 메시지 만들기
        - @GetMapping("/welcome-with-parameter/name/{name}")

- todo REST API 만들기; 빈과 서비스 설정 → todo 리스트 검색 → 특정 Todo의 세부사항 검색 → Todo 추가 → Todo 업데이트 → Todo 삭제
    - 주어진 사용자의 todo 리스트 검색하기
    - 특정 todo의 세부사항 검색하기; @GetMapping(...)
    - todo 만들기; @PostMapping(...)
    - todo 삭제하기; @DeleteMapping(...)
    - todo 세부 정보 업데이트 하기; @PutMapping(...)

- 포스트맨 사용
    - GET

    - POST
    - PUT
    - DELETE

- REST API의 예외 처리 구현
    - 스프링 부트의 디폴트 예외처리
    - 예외 응답 커스터마이징
        - 예외 응답 구조 정의

시나리오와 응답 상태 매핑	
상태	**응답 상태**
요청 본문이 API 스펙을 충족하지 않는다. 세부사항이 충분하지 않거나 유효성 검사 오류가 있다.	400 BAD REQUEST
인증 또는 권한 부여에 실패했다.	401 UNAUTHORIZED
사용자는 제한을 초과하는 등 다양한 요인으로 인해 작업을 수행할 수 없다.	403 FORBIDDEN
리소스가 없다.	404 NOT FOUND
지원되지 않는 작업(예: GET만 허용되는 리소스에서 POST를 시도하는 경우)	405 METHOD NOT ALLOWED
서버에서 오류가 발생했다. 이상적으로는 이런 일이 발생하지 않아야 한다. 소비자는 문제를 해결할 수 없다.	500 INTERNAL SERVER ERROR

- HATEOAS; Hypermedia as the Engine of Application State, REST 애플리케이션 아키텍처의 제약 조건 중 하나
    - 응답으로 HATEOAS 링크 보내기; 스프링 부트 스타터 HATEOAS; spring-boot-starter-hateoas
- REST API의 유효성 검사 구현
    1. 컨트롤러 메소드에서 유효성 검사 사용
    2. 빈에 유효성 검사 추가
    - 컨트롤러 메소드에서 유효성 검사 활성화하기
    - 빈에서 유효성 검사 정의하기
        - @NotNull: 사용자 필드가 비어 있지 않은지 확인
        - @Size(min = 9, message = “Enter atleast 10 Characters.”): desc 필드의 문자수가 9자 이상인지 확인
        - @AssertFalse, @AssertTrue: 부울 Boolean 요소이면 어노테이션이 달린 요소를 확인
        - @AssertFalse: false를 확인
        - @Assert: true를 확인
        - @Future: 어노테이션이 달린 요소는 미래의 날짜여야 한다
        - @Past: 어노테이션이 달린 요소는 과거의 날짜여야 한다.
        - @Max: 어노테이션이 달린 요소는 값이 지정된 최대 값보다 작거나 같은 숫자여야 한다
        - @Min: 어노테이션이 달린 요소는 지정된 최솟값보다 크거나 같은 숫자여야 한다.
        - @NotNull: 어노테이션이 달린 요소는 null일 수 없다.
        - @Pattern: 어노테이션이 달린 {@code CharSequence} 요소는 지정된 정규 표현식과 일치해야 한다. 정규 표현식은 자바 정규 표현식 규칙을 따른다.
        - @Size: 어노테이션이 달린 요소 크기는 지정된 범위에 있어야 한다.
- OpenAPI 스펙을 사용한 REST 서비스 문서화
    - 스웨거 스펙 생성
        - @Configuration: 스프링 설정 파일을 정의
        - @EnableSwagger2: 스웨거 지원을 위한 어노테이션

        - Docket: 스웨거 스프링 MVC 프레임워크를 사용해 스웨거 문서 생성을 구성하는 간단한 빌더 클래스
        - new Docket(DocumentationType.SWAGGER_2): 스웨거2를 사용할 스웨거 버전으로 구성
        - .apis(RequestHandlerSelectors.any()) .paths(PathSelectors.any()): 문서의 모든 API와 경로를 포함한다
    - 스웨거 스펙 살펴보기; 스웨거 UI를 사용해 스웨거 문서 탐색하기, 어노테이션을 사용한 커스텀 스웨거 문서
        - @ApiOperation(value = “Retrieve all todos for a user by passing in his name”): 서비스 요약으로 문서에 생성된다.
        - notes = “A list of matching todos is returned. Current pagination is not supported.”: 서비스를 설명한 내용으로 문서에 생성된다
        - produce = “application/json”: 서비스 문서의 produce 섹션을 커스텀한다
        - @Api: 클래스를 스웨어 리소스로 표시
        - @ApiModel: 스웨거 모델의 추가 정보를 제공
        - @ApiModelProperty: 모델 속성의 데이터를 추가하고 조작
        - @ApiOperation: 특정 경로의 오퍼레이션이나 HTTP 메소드를 설명
        - @ApiParam: 오퍼레이션 파라미터에 추가 메타 데이터를 추가
        - @ApiResponse: 오퍼레이션 응답 예를 설명
        - @ApiResponses: 여러 ApiResponse 객체 리스트를 허용하는 래퍼
        - @Authorization: 리소스나 오퍼레이션에 사용되는 권한 부여 체계를 선언
        - @AurhorizationScope: OAuth2 인증 범위를 설명
        - @ResponseHeader: 응답의 일부로 제공될 수 있는 헤더를 나타낸다
        - @SwaggerDefinition: 생성된 스웨거 정의에 추가할 정의 레벨 속성
        - @Info: 스웨거 정의의 일반 메타 데이터
        - @Contact: 스웨거 정의를 위해 연락할 사람을 설명하는 속성
        - @License: 스웨거 정의의 라이센스를 설명하는 속성
- REST API 국제화 구현
    - LocaleResolver와 메시지 소스 추가
        - sessionLocaleResolver.setDefaultLocale(Locale.US): 기본 지역으로 Locale.US를 설정
        - messasgeSource.setBasenames(“message”): 메시지 소스의 기본 이름을 message로 설정. fr 지역(프랑스)에 있다면 message_fr.properties의 메시지를 사용. message_fr.properties에서 메시지를 사용할 수 없으면 기본 message.properties에서 검색
        - messageSource.setUseCodeAsDefaultMessage(true): 메시지를 찾지 못하면 코드는 기본 메시지를 반환
        - @RequestHeader(value = “Accept-Language”, required = false) Locale locale: 지역은 요청 헤어 Accept-Language에서 선택된다. 필수는 아니다. 지역을 지정하지 않으면 기본 지역이 사용된다.
        - messageSource.getMessage(“welcome.message”, null, locale): 따라서 messageSource는 컨트롤러에 자동 연결된다. 주어진 지역에 따라 welcome 메시지를 받는다
- REST API 캐싱 구현
    - caching-spring-boot-starter-cache를 위한 스타터 프로젝트
    - 애플리케이션의 캐싱 활성화; @EnableCaching
    - 데이터 캐싱; @Cacheable(”..“)
        - @CachePut; 데이터를 캐시에 명시적으로 추가하는 데 사용
        - @CacheEvict; 캐시에서 오래된 데이터를 제거하는 데 사용
        - @Caching; 여러 개의 중첩된 @Cacheable, @CachePut, @CacheEvict 어노테이션을 같

은 메소드에서 사용할 수 있다.
    - JSR-107 캐싱 어노테이션
        - @CacheResult; @Cacheable과 비슷하다
        - @CahceRemove; @CacheEvict와 유사하다. @CahceRemove는 예외가 발생하면 조건부 제거를 지원한다.
        - @CacheRemoveAll; @CacheEvict(allEntries=true)와 유사하게 캐시에서 모든 항목을 제거하는 데 사용
    - 캐싱 공급자의 자동 감지 순서; JCache(JSR-107)(EhCache 3, Hazelcast, Infinispan 등) → EhCache 2.x → 헤이젤캐스트 Hazelcast → 인피니스팬 Infinispan → 카우치베이스 Couchbase → 레디스 Redis → 카페인 Caffeine → 구아바 Guava → 심플 Simple
- 클라우드에 스프링 부트 애플리케이션 배포; 클라우드 파운드리 Cloud Foundry, 헤로쿠 Heroku, 오픈시프트 OpenShift, 아마존 웹 서비스 AWS

스프링 부트의 REST API 단위 테스트

- 스프링 부트를 이용한 REST API 단위 테스트
    - 스프링 부트 스타터 테스트에 의존 관계 추가
        - JUnit: 가장 널리 사용되는 자바 단위 테스트 프레임워크는 단위 테스트 작성을 위한 기본 구조를 제공한다
        - 스프링 테스트(spring-test): 스프링 컨텍스트를 시작하기 위해 단위 테스트를 작성하는 기능을 제공한다. REST API를 단위 테스트하는 데 사용할 수 있는 모크 MVC 프레임워크도 제공한다
        - 모키토 코어(mockito-core): 모크로 단위 테스트를 작성하는 데 도움이 된다
        - AssertJ 코어(assertj-core): 간단하고 읽기 쉬운 어설션을 작성할 수 있다. 예를 들면 assertThat(nubersList).hasSize(15).contains(1, 2, 3).allMatch(x → x < 100)이다
        - Hamcrest 코어(hamcrest-core): 훌륭한 어설션을 작성하는 또 다른 프레임워크다. 예를 들면 assertThat(“XYZ”, containsString(“XY”))이다
        - JSON Path(json-path): XML용 XPath와 유사하게 JSONPath로 JSON 응답의 요소를 식별하고 단위 테스트를 할 수 있다
        - JSON Assert(JSONAssert): JSON Assert를 사용하면 예상되는 JSON을 지정하고 이를 사용해 JSON 응답의 특정요소를 체크할 수 있다.
    - BasicController API 단위 테스트
        - 기본 단위 테스트 설정
            - @RunWith(SpringRunner.class): SpringRunner는 SpringJUnit4ClassRunner 클래스의 바로가기다. 단위 테스트를 위한 간단한 스프링 컨텍스트를 시작한다
            - @WebMvcTest(BasicController.class): 어노테이션을 SpringRunner와 함께 사용해 스프링 MVC 컨트롤러에 간단한 테스트를 작성할 수 있다. 스프링 MVC 관련 어노테이션으로 어노테이션이 달린 빈만 로드한다. 예제에서는 BasicController 클래스를 테스트하면서 웹 MVC 테스트를 컨텍스트를 시작한다
            - @Autowired private MockMvc mvc: 요청하는 데 사용할 수 있는 MockMvc 빈을 자동 연결한다
            - secure = false: 단위 테스트에서 시큐리티를 사용하고 싶지 않을 수도 있다. WebMvcTest 어노테이션의 secure = false 매개변수는 단위 테스트에 스프링 시큐리티를 사용하지 않는다
        - 문자열을 반환하는 Hello World API의 단위 테스트 작성하기
            - mvc.perform(MockMvcRequestBuilders.get(”/welcome“).accept(MediaType.AP

PLICATION_JSON)): Accept 헤더 값, application/json을 사용해 /welcome의 요청을 수행한다
- andExpect(status().isOK()): 응답 상태는 200(success)다
- andExpect(content().string(equalTo("Hello World"))): 응답 내용이 "Hello World"와 같을 것으로 예상한다

- JSON을 반환하는 Hello World API의 단위 테스트 작성하기
- 경로 매개변수를 사용해 Hello World API의 단위 테스트 작성하기

- TodoController API의 단위 테스트
  - TodoController API의 단위 테스트 설정
  - 모든 todo를 검색하기 위한 단위 테스트 작성 - GET 메소드
  - 특정 todo를 검색하기 위한 단위 테스트 작성 - GET 메소드
  - todo 생성을 위한 단위 테스트 작성 - POST 메소드
  - 유효성 검사 오류가 있는 todo를 생성하기 위한 단위 테스트 작성
  - todo 업데이트를 위한 단위 테스트 작성 - PUT 메소드
  - todo 삭제를 위한 단위 테스트 장성 - DELETE 메소드

- 스프링 부트와 REST API 통합 테스트
  - BasicController의 통합 테스트 작성
    - MOCK: 모크 웹 환경에서 컨트롤러를 단위 테스트한다. 앞에서 단위 테스트에서 한 것과 유사하다
    - RANDOM_PORT: 사용 가능한 랜덤 포트의 임베디드 서버에서 다른 레이어를 포함한 전체 웹 컨텍스트를 시작한다
    - DEFINED_PORT: 포트 번호를 하드 코딩한다는 점을 제외하고 RANDOM_PORT와 유사하다
    - NONE: 웹 컨텍스트 없이 스프링 컨텍스트를 로드한다
    - 문자열을 반환하는 Hello World API의 통합 테스트 작성 → JSON을 반환하는 Hello World API의 통합 테스트 작성 → 경로 매개변수를 사용해 Hello World API의 통합 테스트 작성
  - TodoController API의 통합 테스트
    - Todo API 통합 테스트 설정
    - 모든 todo를 검색하기 위한 통합 테스트 작성 - GET 메소드
    - todo를 만들기 위한 통합 테스트 작성 - POST 메소드
    - todo 업데이트를 위한 통합 테스트 작성 - PUT 메소드
    - todo 삭제를 위한 통합 테스트 작성 - DELETE 메소드

- 단위 및 통합 테스트 모범 사례
  - 단순한 디자인의 4가지 원칙 4 Principles of Simple Design
    1. 모든 테스트로 실행한다: 단위 테스트는 소스코드와 함께 지속적으로 작성되며 통합된다
    2. 중복 포함을 최소화한다: 코드에는 가능한 한 중복을 최소화한다
    3. 프로그래머의 의도를 표현한다: 코드는 이해하기 쉬워야 한다. 테스트에서 중요한 값을 명확하게 강조해 쉽게 읽을 수 있게 하고, 테스트에 좋은 메소드 이름을 지정한다. Hamcrest Matchers, AssertJ 및 JSON Path와 같은 프레임워크를 사용하 훌륭한 어설션을 작성해보자
    4. 클래스 및 메소드 수를 최소화한다: 관련된 각 요소(메소드, 클래스, 패키지, 구성요소 및 애플리케이션)는 가능한 한 최소로 한다. 테스트당 조건을 하나만 작성해 단위 테스트를 작게 유지할 수 있다. 또한 단위 테스트의 범위도 가능한 작게 유지한다(일반적으로 최대의 메소드는 메소드 그룹이다)
  - 단위 테스트와 관련된 모범 사례
    - 프로덕션 코드에 문제가 있는 경우에만 테스트가 실패한다: 데이터베이스의 데이터 변경과 같은 외부 의존 관계로 인해 테스트가 실패하면 개발팀은 테스트에 신뢰를 잃게 된

다. 일정 시간 지나면 테스트가 부패하고 팀은 테스트 실패를 무시하기 시작한다
    - 테스트는 프로덕션 코드와 관련된 모든 중요한 문제를 찾아야 한다: 예외를 포함한 모든 가능 시나리오에 단위 테스트를 시도하고 작성해보자
    - 테스트는 빠르게 실행돼야 한다: 테스트는 빠르게 실행해야 한다. 개발자는 테스트를 자주 실행하는 경향이 있다. 지속적인 통합 빌드가 빠르게 실행되고 개발자는 빠른 피드백을 받을 수 있다. 단위 테스트에서 스프링 컨텍스트를 시작하지 않고 모크로 단위 테스트를 실행하는 것을 선호한다
    - 테스트는 가능한 한 자주 실행한다: 지속적으로 통합 테스트를 실행해보자. 코드가 버전 컨트롤에 들어가자마자 빌드가 트리거되고 테스트가 실행된다. 이를 통해 팀이 작성하는 훌륭한 단위 테스트를 최대한 활용할 수 있다.

스프링 시큐리티를 활용한 REST API

- 스프링 시큐리티를 이용한 시큐리티 REST API
  - REST API 시큐리티
    - 인증authentication - 유효한 사용자인가?
    - 권한부여authorization - 유효한 사용자가 작업을 수행할 수 있는가?
  - REST API 시큐리티 구현
    - REST API 요정은 URL, 요청 데이터와 사용자 자격 증명 정보를 제공받이 실행된다.
    - 필터가 인증을 확인한다(유효한 사용자?). 일반적으로 사용자 ID/암호 조합 또는 토큰이 자격 증명으로 사용된다. 그후 필터는 권한을 검사한다(사용자에게 올바른 권한이 있는가?)
    - REST API 요청이 실행된다.
  - 스프링 시큐리티 추가하기
    - 스프링 시큐리티 스타터 추가
    - 스프링 부트 스타터 시큐리티 의존관계
    - 스프링 부트 스타터 시큐리티 자동 설정
    - 기본 인증 자격 증명으로 통합 테스트 업데이트
    - 시큐리티를 비활성화하기 위해 단위 테스트 업데이트
- 스프링 시큐리티
  - 로그 검토
  - 스프링 시큐리티 필터
    - UsernamePasswordAuthenticationFilter: 사용자 자격 증명을 사용해 인증한다. 요청이 POST고 사용자 자격 증명이 있는 경우 실행된다.
    - BasicAuthenticationFilter: 기본 인증한다. 요청에 기본 인증 요청 헤더가 있는 경우 실행된다.
    - AnonymousAuthenticationFilter: 요청에 인증 자격 증명이 없으면 익명 사용자가 생성된다. 일반적으로 익명 사용자는 인증이 필요 없는 API인 퍼블릭 API의 요청을 실행할 수 있다.
    - ExceptionTranslationFilter: 추가 시큐리티를 제공하지 않는다. 인증 예외를 적절한 HTTP 응답으로 변환한다.
    - FilterSecurityInterceptor: 인증을 결정한다
    - HeaderWriterFilter: 시큐리티 헤더를 응답에 기록한다(X-FrameOptions, X-XSS-Protection 및 X-Content-Type-Options).
    - CsrfFilter: CSRF 보호를 확인한다.
  - 스프링 시큐리티의 인증

        - 인증 매니저
            - ProviderManager
            - AuthenticationProvider
            - UserDetailsService 구현하기
            - UserDetailsManager로 사용자 관리하기
    - 스프링 시큐리티의 인증 확장 포인트
        - UserDetailsService의 커스텀 구현 제공
        - 글로벌 인증 매니저 구성을 위한 웹 시큐리티 구성 프로그램 어댑터 확장
        - 웹 시큐리티 구성 어댑터를 사용한 시큐리티 구성
    - 스프링 시큐리티의 권한
        - access decision manager를 사용한 권한 부여
    - 스프링 시큐리티의 인증 확장 포인트
        - 웹 시큐리티 구성 어댑터를 사용한 HTTP 시큐리티 구성
    - 서비스 메소드에 시큐리티 어노테이션 제공
    - 서비스 메소드에 JSR-250 어노테이션 제공
    - 스프링 시큐리티 pre 어노테이션과 post 어노테이션
    - 스프링 시큐리티를 이용한 시큐리티 모범 사례 구현
        - CSRF 공격을 방지한다
        - 세션 고정 문제로부터 보호한다
        - 응답에 헤더를 자동으로 추가해 시큐리티를 강화한다
        - 요청이 HTTP인 경우 엄격한 HTTP 전송 시큐리티를 지정한다
            - 캐시 컨트롤 헤더를 추가한다
            - cross-site protection 헤더를 추가한다
            - X-Frame-Options 헤더를 추가해 클릭 재킹을 방지한다

- OAuth2 인증
    - OAuth는 다양한 웹 지원 애플리케이션과 서비스 사이의 권한 및 인증 정보를 교환하기 위해 플로우를 제공하는 프로토콜
    - 일반적인 OAuth2 교환에서 중요한 플레이어
        - 리소스 소유자: 리소스 소유자는 Todo API의 사용자인 데이터 소유자다. 리소스 소유자는 API로 제공되는 정보 중 타사 애플리케이션에 제공할 수 있는 정보의 양을 결정한다
        - 리소스 서버: Todo API에 액세스가 제공되는 실제 리소스다
        - 클라이언트: Todo API 애플리케이션과 통합하려는 타사 애플리케이션이다.
        - 인증서버: OAuth 서비스를 제공하는 서버다
        - 클라이언트 자격 증명: 각 타사 애플리케이이션에는 자신을 OAuth 사버에 식별하는 자격 증명이 있다. 이를 클라이언트 자격 증명이라고 한다
    - OAuth2 상위 플로우
        - 권한 부여 플로우
        - 리소스 액세스 플로우
    - OAuth2 서버 생성
        - 인증 서버 설정: REST API에 사용자 자격 증명 설정 → 타사 클라이언트 자격 증명으로 인증 서버 설정 액세스 토큰 얻기 → 리소스 서버 설정 → 액세스 토큰을 사용해 요청 실행 → 통합 테스트 업데이트
            - access_token: 클라이언트 애플리케이션은 액세스 토큰을 사용해 추가 API 호출을 인증할 수 있다. 그러나 액세스 토큰은 일반적으로 매우 짧은 기간 안에 만료된다.
            - refresh_token: 클라이언트 애플리케이션은 refresh_token을 사용해 인증 서버에 새로운 요청을 제출해 새로운 access_token을 얻을 수 있다.

- JWT를 이용한 인증
  - JWT; JWT 토큰은 사용자 세부사항, 사용자 권한 부여 및 몇 가지 커스텀 애플리케이션 특정 세부사항을 포함하는 암호화된 토큰이다.
    - JWT 페이로드 payload;
      - sub: 주제 - JWT 토큰에는 어떤 세부 정보가 포함될까
      - iat: 당시 발행 - JWT 토큰은 언제 작성됐을까
      - name: 이름
      - admin: 사용자는 매니저인가?
      - custom: 커스텀 값
    - JOSE 헤더; JSON 객체 서명 및 암하화의 줄임말
      - HS256: SHA-256과 함께 HMAC 사용
      - HS512: SHA-512와 함께 HMAC 사용
      - RS256: SHA-256과 함께 RSASSA-PKCS1-v1_5 사용
      - RS512: SHA-512와 함께 RSASSA-PKCS1-v1_5 사용
    - JWT 서명 및 토큰 생성
    - REST API 인증을 위한 JWT 사용
  - OAuth2와 함께 JWT 사용

리액트 및 스프링 부트가 포함된 풀스택 앱

- 풀스택 아키텍처
- 리액트 React; 오픈소스 자바스크립트 프레임워크
- 프론트 엔드 애플리케이션의 컴포넌트; Header, Footer, 메뉴, 페이지1, 페이지2,…
- JSX
- JSX와 컴포넌트 결합
  - Header 컴포넌트 만들기
  - Footer 컴포넌트 만들기
  - Todo 컴포넌트 작성하기
- 리액트 애플리케이션 빌드
  - create-react-app
- 비주얼 스튜디오 코드 IDE로 리액트 애플리케이션 가져오기
  - 리액트 프레임워크 초기화; package.json
  - Todo 컴포넌트 생성
  - 기본 Todo 관리 기능 추가
    - 유효성 검사
    - axios 프레임워크로 API에서 todo 로드하기
  - RESTful API를 호출하는 todo 추가하기
  - 인증
    - 기본 인증
    - JWT 토큰 기반 인증

스프링 데이터

- 다양한 데이터 레파지토리
  - RDBMS
  - NoSQL

        - 비정형 데이터; 데이터에 관한 특정 구조가 없다
        - 대용량; 일반적으로 기존의 데이터베이스(예: 로그 스트림, 페이스북 게시물 및 트윗)에서 처리할 수 있는 것보다 용량이 많다
        - 쉽게 확장 가능; 일반적으로 수평 및 수직으로 확장할 수 있는 옵션을 제공한다
- 관계형 데이터베이스와 통신
    - 마이바티스 myBatis(기존 아이바티스 iBatis); 마이바티스는 매개변수를 설정하고 결과를 검색하기 위해 코드를 수동으로 작성할 필요가 없다. 자바 POJO를 데이터베이스에 매핑하는 간단한 XML이나 어노테이션 기반 구성을 제공한다.
    - 하이버네이트 Hibernate; 하이버네이트는 ORM(오브젝트/관계형 매핑) 프레임워크다. ORM 프레임워크는 오브젝트를 관계형 데이터베이스의 테이블에 매핑하는 데 유용하다. 하이버네이트의 가장 큰 장점은 개발자가 직접 쿼리를 작성할 필요가 없다는 것이다. 객체와 테이블 간의 관계가 매핑되면 하이버네이트는 매핑을 사용해 쿼리를 생성하고 데이터를 채우거나 검색한다.

- 스프링 데이터
    - 기능
        - 다양한 레파지토리를 통해 여러 데이터 레파지토리와 쉽게 통합
        - 레파지토리 메소드 이름을 기반으로 쿼리를 구문 분석하고 구성하는 기능
        - 기본 CRUD 기능 제공
        - 사용자가 생성한 후 마지막으로 변경된 경우와 같은 감사의 기본 지원
        - 스프링과 강력한 통합
        - 스프링 데이터 레스트 Spring Data Rest를 통해 REST 컨트롤러를 노출시키는 스프링 MVC와 통합
    - 중요 스프링 데이터 모듈
        - 스프링 데이터 커먼즈 Spring Data Commons: 모든 스프링 데이터 모듈에 공통 개념을 정의한다(레파지토리 및 쿼리 메소드)
        - 스프링 데이터 JPA Spring Data JPA: JPA 레파지토리와 쉽게 통합한다
        - 스프링 데이터 몽고DB Spring Data MongoDB: 문서 기반 데이터 레파지토리인 몽고DB와 쉽게 통합한다
        - 스프링 데이터 REST Spring Data REST: 스프링 데이터 레파지토리를 최소한의 코드로 구성된 REST 서비스로 노출하는 기능을 제공한다
        - 아파치 카산드라용 스프링 데이터 Spring Data for Apache Cassandra: 카산드라와 쉬운 통합을 제공한다
        - 아파치 하둡용 스프링 Spring for Apache Hadoop: 하둡과 쉬운 통합을 제공한다
    - 스프링 데이터 커먼즈; 레파지토리 인터페이스, CrudRepository 인터페이스, PagingAndSortingRepository 인터페이스

- 스프링 데이터 JPA를 사용해 관계형 데이터베이스에 연결
    - 스프링 데이터 JPA 예제
        - 스타터 데이터 JPA로 새 프로젝트 만들기
        - 엔티티 정의하기
            - @Entity; 해당 클래스가 엔티티임을 지정하는 어노테이션
            - @Id는 ID가 엔티티의 기본 키임을 지정
            - @GeneratedValue(strategy = GenerationType.AUTO): GeneratedValue 어노테이션은 기본 키 생성 방법을 지정하는 데 사용된다. 예에서는 GenerationType.AUTO 전략을 사용하고 있다. 퍼시스턴스 공급자가 올바른 전략을 선택하기를 원한다는 것을 나타낸다
            - @ManyToOne(fetch - FetchType.LAZY)는 User와 Todo 사이의 다대일 관계를 나타낸다. 관계의 한쪽에서 @MayToOne 관계가 사용된다. FetchType.Lazy는 데이

터를 느리가 가져올 수 있음을 나타낸다
- @JoinColumn(name = “userid”): JoinColumn 어노테이션은 외래 키 열의 이름을 지정한다
- SpringBootApplication 클래스 생성하기
- 일부 데이터 채우기
- 간단한 레파지토리 만들기
- 단위 테스트
  - @DataJpaTest: @DataJpaTest 어노테이션은 일반적으로 JPA 레파지토리 단위 테스트에서 SpringRunner와 함께 사용된다. @DataJpaTest 어노테이션은 JPA 관련 자동 설정만 활성화한다. 테스트는 기본적으로 인메모리 데이터베이스를 사용한다.
  - @RunWith(SpringRunner.class): SpringRunner는 SpringJUnit4ClassRunner의 간단한 별칭이다. 스프링 컨텍스트를 시작한다
  - @Autowired TodoRepository todoRepository: 테스트에 사용될 TodoRepository를 오토와이어링한다.
  - assertEquals(3, todoRepository.count()): 반환된 카운트가 3인지 확인한다. data.sql에 3개의 todo를 삽입했다는 것을 기억하자.
- CrudRepository 인터페이스를 확장하는 레파지토리 생성하기
  - 단위 테스트를 사용한 테스트
- PagingAndSortingRepository 인터페이스를 확장하는 레파지토리 생성하기
  - 단위 테스트를 이용한 탐색하기
- 커스텀 쿼리 메소드 작성하기
- 커스텀 JPQL 쿼리 작성하기
  - 명명된 매개변수 사용
  - 명명된 쿼리 사용
  - 네이티브 SQL 쿼리 실행하기
- 거래 관리 시작
  - 스프링 @Transactional 어노테이션
  - 트랜잭션을 위한 스프링 부트 자동 설정

- 스프링 데이터를 이용한 몽고DB와의 상호 작용
  - 스프링 데이터 몽고DB의 의존 관계 설정
  - Person 엔티티 생성
  - Person 레파지토리 생성
  - 좋은지 확인하기 위해 단위 테스트 작성

- 스프링 데이터 REST를 사용해 REST API 생성하기
  - 중요 기능
    - 스프링 데이터 레파지토리를 중심으로 REST API를 노출한다
    - 페이지 매김 기능과 필터링을 지원한다
    - 스프링 데이터 레파지토리의 쿼리 메소드를 이해하고 이를 검색 리소스로 노출한다
    - 지원되는 프레임워크 중에는 JPA, 몽고DB, 카산드라가 있다
    - 리소스를 커스텀하는 옵션이 기본적으로 표시된다
  - GET 메소드
  - POST 메소드
  - 검색 리소스

마이크로 서비스

- 애플리케이션 개발 목표; 애플리케이션을 개발할 때 필요한 공통 목표를 이해해야 마이크로서비스 아키텍처로 이동하는 이유를 이해하는 데 도움
- 빠른 애플리케이션 구축 - 속도
  - 기술과 비즈니스 환경의 중요한 변화
    - 새로운 프로그래밍 언어;
      - 고 Go,
      - 스칼라 Scala,
      - 클로저 Closure,...
    - 새로운 프로그래밍 패러다임;
      - 함수형 프로그래밍,
      - 리액티브 프로그래밍
    - 새로운 프레임워크
    - 새로운 도구
      - 개발
      - 코드 품질
      - 자동화 테스트
      - 배포
      - 컨테이너화
    - 새로운 프로세스 및 사례
      - 애자일 Agile
      - 테스트 중심 개발
      - 행동 중심 개발
      - 지속적인 통합
      - 지속적인 배포
      - 데브옵스 DevOps
    - 새로운 장치 및 기회
      - 모바일
      - 클라우드

- 신뢰할 수 있는 애플리케이션 구축 - 안전
  - 신뢰성 - 애플리케이션이 예상대로 작동할까?
  - 가용성 - 애플리케이션을 항상 사용할 수 있을까?
  - 안전성 - 애플리케이션은 안전한가?
  - 성능 - 애프리케이션이 충분히 빠른가?
  - 높은 복원력 - 애플리케이션이 실패에 잘 반응할까?
  - 확장성 - 애플리케이션 로딩이 급격히 증가할 때 무엇을 지원해야 할까?

- 모놀리식 애플리케이션 문제
  - 모놀리식 애플리케이션 특징
    - 큰 크기: 대부분의 모놀리식 애플리케이션에는 코드 베이스가 많다
    - 대규모 팀: 팀 규모는 20명에서 300명까지 다양하다
    - 같은 일을 하는 여러 가지 방법: 팀 규모가 큰 탓에 의사소통에 차이가 있다. 그 결과 애플리케이션의 다른 부분에서 같은 문제의 솔루션이 여러 개 생성된다.
    - 자동화 테스트 부족: 대부분 애플리케이션에는 단위 테스트가 거의 없으며 통합 테스트가 완벽하지 않다. 애플리케이션은 수동 테스트에 크게 의존한다
  - 릴리스 업데이트의 문제 - 긴 릴리스 주기
  - 확장의 어려움

    - 새로운 기술에 적응의 어려움
    - 새로운 방법론 적용의 어려움
    - 현대적인 개발 사례 적용의 어려움; TDD Test-driven development, BDD behavior-driven development

- 마이크로서비스 시작
    - “많은 조직에서는 세밀한 마이크로서비스 아케텍처를 채택함으로써 소프트웨어를 좀더 빠르게 제공하고 새로운 기술을 도입할 수 있다는 것을 발견했다” - 샘 뉴먼, '마이크로서비스 구축'
    - “마이크로서비스는 함께 작동하는 소규모 자율 서비스다.” - 샘 뉴먼, 소트웍스
    - “제한된 컨텍스트를 사용해 느슨하게 결합된 서비스 지향 아키텍처다.” - 아드리안 콕크로프트, 배터리 벤처
    - 마이크로서비스 아키텍처의 큰 그림
    - 마이크로서비스 특성; 높은 코드 품질, 스몰, 경량, 메시지 기반, 무상태, 싱글 비즈니스 능력, 독립 팀, 독립 배포, 자동 빌드 릴리스, 이벤트 주도
        - 작고 가벼운 마이크로서비스
        - 메시지 기반 커뮤니케이션과의 상호 운용성
        - 용량 할당 마이크로서비스
        - 독립적으로 배포 가능한 마이크로서비스
        - 무상태 마이크로서비스
        - 완전 자동화된 빌드 및 릴리스 프로세스
        - 이벤트 주도 아키텍처 준수
            - 접근법
                - 순차적 접근법
                - 이벤트 주도 접근법
        - 마이크로서비스 개발 및 지원을 위한 독립 팀 - 데브옵스
    - 마이크로서비스 아키텍처의 장점
        - 출시 시간 단축
        - 기술 진화에 빠른 적응
        - 확장성
        - 현재 개발 방법론과의 호환성
    - 마이크로서비스 문제
        - 자동화의 필요성 증가
        - 서브 시스템의 경계 정의
        - 가시성 및 모니터링 필요성 증가
        - 내결함성 증가
        - 마이크로서비스 간 일관성 유지
            - 표준화된 공유 기능 구축(엔터프라이즈 레벨)
                - 하드웨어: 어떤 하드웨어를 사용하는가? 클라우드를 사용하는가?
                - 코드 관리: 어떤 버전 관리 시스템을 사용하는가? 분기 및 커밋 코드의 관행은 무엇인가?
                - 구축 및 배포: 어떻게 구축할까? 배포를 자동화하려면 어떤 도구를 사용하는가?
                - 데이터 레파지토리: 어떤 종류의 데이터 레파지토리를 사용하는가?
                - 서비스 오케스트레이션 orchestration: 서비스를 어떻게 조율하는가? 아떤 종류의 메시지 브로커를 사용하는가?
                - 시큐리티 및 ID: 사용자 및 서비스를 인증하고 승인하는 방법은 무엇인가?
                - 시스템 가시성 및 모니터링: 서비스를 어떻게 모니터링하는가? 시스템 전체에서 결함 격리를 어떻게 제공하는가?

        - 운영팀의 필요성 증가

- 클라우드 네이티브 애플리케이션
    - 12 팩터 앱 Twelve-Factor
        - 하나의 코드 베이스 유지
        - 명시적 의존 관계 선언
        - 환경 설정
        - 모든 의존 관계는 백엔드 서비스로 취급된다
        - 빌드, 릴리스, 실행 단계를 명확하게 분리한다
            - 빌드: 코드에서 실행 가능한 번들(EAR, WAR, JAR)과 여러 환경에 배포할 수 있는 의존 관계를 만든다
            - 릴리스: 실행 가능한 번들을 특정 환경 설정과 결합해 배포한다
            - 실행: 특정 릴리스를 사용해 실행 환경에서 앱을 실행한다.
        - 애플리케이션은 상태를 저장하지 않는다 - 무상태
        - 모든 서비스는 포트 바인딩으로 노출된다
        - 수평 확장 가능성 - 동시성
        - 각 애플리케이션 인스턴스는 일회용이다
        - 환경 평가 - 모든 환경이 같아야 한다
        - 모든 로그를 이벤트 스트림을로 취급한다
        - 관리자 프로세스에 구별이 없다

- 마이크로서비스를 위한 스프링 프로젝트
    - 스프링 부트
    - 스프링 클라우드
        - 구성관리: '12 팩터 앱'에서 설명한 것처럼 구성 관리는 클라우드 애플리케이션 개발의 중요한 부분이다. 스프링 클라우드느 마이크로서비스를 위한 중앙 집중식 구성 관리 솔루션인 스프링 클라우드 컨피그를 제공한다
        - 서비스 디스커버리: 서비스 디스커버리는 서비스 간 느슨한 연결을 촉진한다. 스프링 클라우드는 유레카, 주키퍼, 콘솔과 같이 인기있는 서비스 디스커버리 옵션과 통합된다
        - 서킷 브레이커: 클라우드 네이티브 애플리케이션은 내결함성이 있어야 하고, 서비스를 정상적으로 지원하지 못하는 문제를 처리할 수 있어야 한다. 서킷 브레이커는 장애 발생 시 최소한의 기본 서비스를 제공하는 데 중요한 역할을 한다. 스프링 클라우드는 넷플릭스 히스트릭스 내결함성 라이브러리와의 통합을 제공한다
        - API 게이트웨이: API 게이트웨이는 중앙 집중식 집계, 라우팅, 캐싱 서비스를 제공한다. 스프링 클라우는 API 게이트웨이 라이브러리인 넷플릭스 주울Zuul과의 통합을 제공한다
    - 중요 스프링 클라우드 하위 프로젝트
        - 스프링 클라우드 넷플릭스 Spring Cloud Netflix: 넷플릭스는 마이크로서비스 아케텍처의 초기 채택자 중 하나다. 많은 내부 넷플릭스 프로젝트가 스프링 클라우드 넷플릭스의 범위의 오픈소스로 제공됐다. 유레카, 히스트릭스 Hystrix와 주울을 예로 들 수 있다
        - 스프링 클라우드 컨피그 Spring Cloud Config: 다양한 환경의 여러 마이크로서비스에서 중앙 집중식 외부 구성이 가능하다
        - 스프링 클라우드 버스 Spring Cloud Bus: 간단한 메시지 브로커와 마이크로서비스 통합을 보다 쉽게 구축할 수 있다
        - 스프링 클라우드 슬러시 Spring Cloud Sleuth: 제프킨 Zipkin과 함께 분산 추적 솔루션을 제공한다
        - 스프링 클라우드 데이터 플로우 Spring Cloud Data Flow: 마이크로서비스 애플리케이션을 중심으로 오케이스트레이션을 구축하는 기능을 제공한다. DSL, GUI, REST API를 제공한다

        - 스프링 클라우드 스트림 Spring Cloud Stream: 스프링 기반(및 스프링 부트 기반) 애플리케이션을 아파치 카프카나 RabbitMQ와 같은 메시지 브로커와 통합하기 위해 간단한 선언적 프레임워크를 제공한다
    - 스프링 클라우드 넷플릭스
        - 유레카; 마이크로서비스를 위한 서비스 등록과 검색 기능을 제공한다
        - 히스트릭스; 서킷 브레이커를 통해 내결함성 마이크로서비스를 구축하는 기능으로 대시보드를 제공한다
        - 페인 Feign: 선언적 REST 클라이언트를 사용하면 JAX-RS와 스프링 MVC로 생성된 서비스를 쉽게 호출할 수 있다
        - 리본 Ribbon: 클라이언트 측 로드 밸런싱 기능을 제공한다
        - 주울: 라우팅, 필터링, 인증, 시큐리티와 같은 일반적인 API 게이트웨이 기능을 제공한다. 사용자 지정 규칙과 필터를 사용해 확장할 수 있다

스프링 부트 + 스프링 클라우드 => 마이크로서비스

- 마이크로서비스
    - 마이크로서비스A ↔ 서비스 소비자 마이크로서비스

- 마이크로서비스A 설정
    1. 스프링 이니셜라이저로 마이크로서비스A 초기화하기
    2. 마이크로서비스A에서 랜덤 목록 서비스 만들기
- 서비스 소비자 마이크로서비스 구축하기
    - 마이크로서비스A에서 랜덤 목록 서비스를 소비하는 메소드 만들기
    - 서비스 소비자 마이크로서비스 테스트
    - 다른 마이크로서비스에 사용되는 포트 표준화

- 스프링 부트와 스프링 클라우드 권장 버전 사용

- 중앙 집중식 마이크로서비스 구성
    - 문제 기술
        - 데이터베이스 구성: 데이터베이스에 연결하는 데 필요한 세부 정보
        - 메시지 브로커 구성: AMQP나 유사한 리소스에 연결하는 데 필요한 구성
        - 외부 서비스 구성: 마이크로서비스에 필요한 기타 서비스
        - 마이크로서비스 구성; 마이크로서비스의 비즈니스 로직과 관련된 일반적인 구성
    - 솔루션
    - 옵션
    - 스프링 클라우드 컨피그
        - 스프링 클라우드 컨피그 서버: 버전 관리 레파지토리(GIT 또는 Subversion)에 의해 백업된 중앙 집중식 구성 노출을 지원한다.
        - 스프링 클라우드 컨피그 클라이언트: 애플리케이션이 스프링 클라우드 컨피그 서버에 연결하도록 지원한다
        - 스프링 클라우드 컨피그 서버 구현
        - 스프링 클라우드 컨피그 서버 설정
    - 애플리케이션 구성에서 메시지를 반환하기 위해 마이크로서비스A에 서비스 생성하기
        - 스프링 클라우드 컨피그 서버를 로컬 깃 레파지토리에 연결
        - 마이크로서비스A용 개발 환경 특정 구성 생성
        - 마이크로서비스A를 스프링 클라우드 컨피그 클라이언트로 만들기

- 이벤트 중심 접근법의 개요
  - JMS API를 이용한 스프링 JMS(Java Messaging Service)
  - AQMP(Advanced Message Queuing Protocol)

- 스프링 클라우드 버스
  - 스프링 클라우드 버스의 필요성
  - 스프링 클라우드 버스를 이용한 설정 변경 전파
  - 스프링 클라우드 버스 구현

- 선언적 REST 클라이언트 - 페인
- 마이크로서비스를 위한 로드 밸런싱 구현
  - 클라이언트-사이드 로드 밸런싱을 위한 립본
    - 서비스 소비자 마이크로서비스에서 립본 구현

- 네임 서버의 필요성
  - 마이크로서비스 URL 하드 코딩의 한계

- 네임 서버 작동
  1. 등록: 모든 마이크로서비스(다른 마이크로서비스와 각 인스턴스)는 각 마이크로서비스가 시작될 때 네임 서버에 자신을 등록한다
  2. 다른 마이크로서비스의 위치 찾기: 서비스 소비자가 특정 마이크로서비스의 위치를 얻으려고 할 때 네임 서버를 요청한다. 서비스 소비자는 마이크로서비스 ID로 네임 서버를 찾을 때마다 해당 마이크로서비스의 인스턴스 리스트를 가져온다
- 스프링 클라우드에서 지원하는 네임 서버 옵션
  - 유레카 명명 서비스 구현
    - 유레카 서버 설정
    - 유레카에 마이크로서비스 등록
    - 서비스 소비자 마이크로서비스와 유레카 연결

- API 게이트웨이
  - 중요 크로스 컷팅 문제
    - 인증, 권한 부여, 시큐리티: 마이크로서비스 소비자가 자신이 주장하는 사람인지 어떻게 확인할까? 소비자가 마이크로서비스에 올바르게 액세스할 수 있도록 하려면 어떻게 하야 할까?
    - 비율 제한: 소비자를 위한 다양한 종류의 API 계획과 각 계획에 대한 다양한 제한(마이크로서비스 호출 수)이 있을 수 있다. 특정 소비자의 제한을 어떻게 적용할까?
    - 동적 라우팅: 특정 상황(예: 마이크로서비스가 다운된 경우)에는 동적 라우팅이 필요할 수 있다.
    - 서비스 통합: 모바일용 UI 요구사항은 데스크톱과 다르다. 일부 마이크로서비스 아키텍처에는 특정 장치에 맞는 서비스 집합이 있다.
    - 내결함성: 단일 마이크로서비스에서 장애가 발생해도 전체 시스템이 중단되지 않도록 하려면 어떻게 해야 할까?
  - API 게이트웨이가 일반적으로 제공하는 마이크로서비스 기능
    - 인증 및 시큐리티
    - 속도 제한
    - 통찰력 및 모니터링
    - 동적 라우팅 및 정적 응답 처리
    - 로드 차단
    - 여러 서비스의 응답 집계
  - 주울을 이용한 API 게이트웨이 구현
    - 새로운 주울 API 게이트웨이 서버 설정

        - 모든 요청을 기록하기 위한 주울 커스텀 필터 구성
        - 주울을 통한 마이크로서비스 호출
        - 주울 API 게이트웨이를 사용하도록 서비스 소비자 구성

- 분산 추적
    - 스프링 클라우드 슬루스 Sleuth와 집킨 Zipkin 구현
        - 마이크로서비스 컴포넌터를 스프링 클라우드 슬루스와 통합
        - 집킨 분산 추적 서버 설정
        - 집킨과 마이크로서비스 구성요소 통합

- 히스트릭스로 내결함성 구현
    - 서비스 소비자 마이크로서비스에 히스트릭스 통합

리액티브 프로그래밍

- 리액티브 선언 reactive manifesto; https://www.reactivemanifesto.org

> "우리는 시스템 아키텍처에 일관된 접근 방식이 필요하고 필수적인 측면이 이미 개별적으로 인식돼 있다고 생각한다. 반응성, 회복력, 탄력성, 메시지 기반 시스템을 원하며, 이러한 시스템을 리액티브 시스템이라고 한다. 리액티브 방식으로 구축된 시스템은 보다 유연하고 느슨하게 연결되며 확장 가능하므로 개발하거나 변경이 쉽다. 이들은 장애에 훨씬 더 관대해 장애가 발생하면 큰 장애를 일으키지 않고 간결하게 대처한다. 리액티브 시스템은 반응이 뛰어나 사용자에게 효과적인 대화식 피드백을 제공한다.

    - 수 초 단위의 응답 시간 → 밀리초 단위의 응답 시간
    - 여러 시간의 오프라인 유지 관리 → 100% 가용성
    - 소량의 데이터 → 데이터 볼륨이 기하 급수적으로 증가
- 리액티브 시스템의 특성
    - 반응성 Responsive; 시스템은 사용자에게 적시에 응답한다. 명확한 응답 시간 요구 사항이 설정되고 시스템은 모든 상황에서 이를 충족시킨다.
    - 회복력 Resilient; 분산 시스템은 여러 구성요소를 사용해 구축된다. 이러한 구성요소 중 하나에서 오류가 발생할 수 있다. 리액티브 시스템은 지역화된 공간(예: 각 구성요소)에서 오류가 발생하도록 설계해야 한다. 이렇게 하면 로컬 장애가 발생해도 전체 시스템이 다운되는 것을 방지할 수 있다.
    - 탄력성 Elastic; 리액티브 시스템은 다양한 부하에서 반응성을 유지한다. 과부하가 걸리면 로드가 다운될 때 시스템을 릴리스하는 동안 리소스를 추가할 수 있다. 탄력성은 범용 하드웨어와 소프트웨어를 사용해 달성된다.
    - 메시지 주도 Message drive; 리액티브 시스템은 메시지(또는 이벤트)에 의해 구동된다. 이렇게 하면 구성요소들 사이의 낮은 결합이 보장돼 시스템의 다른 여러 구성 요소를 독립적으로 확장할 수 있다. 비차단 통신을 사용하면 스레드가 더 짧은 시간 동안 활성화된다.
    - 이벤트에 반응 React to events; 리액티브 시스템은 메시지 전달을 기반으로 구축돼 이벤트에 빠르게 반응한다.
    - 로드에 반등 React to load; 리액티브 시스템은 다양한 로드에서 반응성을 유지한다. 로드가 많을수록 더 많은 리소스를 사용하고 로드가 낮을 때는 해제한다.
    - 장애 대응 React to failures; 리액티브 시스템은 장애를 정상적으로 처리할 수 있다. 리액티브 시스템의 구성요소는 장애를 지역화하기 위해 만들어졌다. 외부 구성은 구성요소의 가용성을 모니터링하고 필요할 때 구성요소를 복제할 수 있는 기능을 제공한다.

    - 사용자에게 반응 React to users; 리액티브 시스템은 사용자에게 반응한다. 소비자가 특정 이벤트에 가입하지 않았다면 추가 처리를 수행하는 데 시간을 낭비하지 않는다

- 리액티브 유스케이스 - 주가 페이지
    - 전통적인 접근 방식; 조사(poll)해서 주가가 변경됐는지 여부를 확인
    - 리액티브 접근 방식; 이벤트 구독, 이벤트 발생, 가입 취소
    - 전통적인 접근 방식과 리액티브 방식의 비교

- 자바에서 리액티브 프로그래밍 구현
    - 리액티브 스트림 reactive streams
    - 리액터 프레임워크 Reactor
    - 스프링 웹 플럭스 프레임워크 Spring WebFlux framworks

스프링 모범 사례

- 메이븐 표준 디렉토리 레이아웃
    - src/main/java: 모든 애플리케이션 관련 소스코드
    - src/main/resources: 모든 애플리케이션 관련 리소스 스프링 컨텍스트 파일, 등록 정보 파일, 로깅 구성 등
    - src/main/webapp: 웹 애플리케이션과 관련된 모든 리소스, 뷰 파일(JSP, 뷰 템플릿, 정적 콘텐츠 등)
    - src/test/java: 모든 단위 테스트 코드
    - src/test/resources: 단위 테스트와 관련된 모든 리소스

- 레이어 아키텍처를 사용한 애플리케이션 구축
    - 프리젠테이션 레이어: 마이크로서비스에서 프리젠테이션 레이어는 REST 컨트롤러가 있는 위치다. 일반적인 웹 애플리케이션에서 프리젠테이션 레이어에는 뷰 관련 콘텐츠(JSP, 템플릿 및 정족 콘텐츠)가 포함된다. 프리젠테이션 레이어는 서비스 레이어와 통신한다.
    - 서비스 레이어: 비즈니스 레이어의 껍데기 역할을 한다. 모바일, 웹 및 태블릿과 같은 다양한 뷰에는 다른 종류의 데이터가 필요할 수 있다. 서비스 레이어는 요구사항을 이해하고 프리젠테이션 레이어를 기반으로 올바른 데이터를 제공한다.
    - 비즈니스 레이어: 모든 비즈니스 로직이 있는 곳이다. 또 다른 모범 사례는 대부분의 비즈니스 로직을 도메인 모델에 적용하는 것이다. 비즈니스 레이어는 데이터 레이어와 통신해 데이터를 가져오고 그 위에 비즈니스 로직을 추가한다.
    - 퍼시스턴스 레이어: 데이터를 검색해 데이터베이스에 저장한다. 퍼시스턴스 레이어는 일반적으로 JPA 매핑 또는 JDBC 코드를 포함한다.

- 다른 레이어를 위한 별도의 컨텍스트 파일이 있다
    - application-context.xml
    - presentation-context.xml
    - services-context.xml
    - business-context.xml
    - persistence-context.xml
    - PresentationConfig.java
    - ServicesConfig.java
    - BusinessConfig
    - PersistenceConfig

- 중요한 레이어의 API와 impl 분리
- 예외 처리 모범 사례

    - 예외 타입
        - 체크된 예외: 서비스 메소드가 체크된 예외를 발생시키면 모든 소비자 메소드가 예외를 처리하거나 예외를 살생시켜야 한다.
        - 언체크된 예외: 소비자 메소드는 서비스 메소드에 의해 발생한 예외를 처리하거나 발생시킬 필요가 없다.
    - 예외 처리에 대한 스프링의 접근 방식
    - 권장 접근법

- 스프링 구성 간결하게 유지
- ComponentScan에서 basePackageClasses 속성 사용
- 스키마 참조에서 버전 번호를 사용하지 않는다
- 필수 의존 관계에 생성자 주입 선호
    - 빈의 의존 관계 종류
        - 필수 의존 관계: 빈에 사용하려는 의존 관계다. 의존 관계를 사용할 수 없을 때는 컨텍스트 로드되지 않는 편이 좋다.
        - 선택적 의존 관계: 이들은 선택적 의존 관계로 항상 이용할 수 있지는 않다. 컨텍스트를 사용할 수 없는 경우에도 컨텍스트를 로드하는 것이 좋다.
- 스프링 프로젝트의 의존 관계 버전 관리
- 단위 테스트 모범 사례
    - 비즈니스 레이어에 테스트 작성
    - 웹 레이어에 테스트 작성
    - 데이터 레이어에 테스트 작성
    - 다른 애플리케이션 개발 모범 사례; TDD
- 통합 테스트 모범 사례
    - 스프링 세션을 이용한 세션 관리
    - 레디스로 스프링 세션 구현
- 캐싱 모범 사례
    - spring-boot-starter-cache 의존 관계 추가
    - 캐싱 어노테이션 추가
- 로깅 모범 사례
    - 로그백 프레임워크 사용
    - log4j2
    - 프레임워크 독립 구성 제공

스프링에서 코틀린 개발

- 코틀린 https://github.com/jetbrains/kotlin
- 코틀린 대 자바
    - 변수 생성 및 타입 유추
    - 변수의 불변성
    - 타입 시스템
    - null 세이프
    - 크틀린의 함수 정의; fun 키워드
    - 배열
    - 코틀린 컬렉션
    - 언체크된 예외 처리
    - 빈에 데이터 클래스 사용하기

- 이클립스에서 코틀린 프로젝트 만들기
  - 코틀린 플러그인
  - 코틀린 프로젝트 생성
  - 코틀린 클래스 만들기
  - 코틀린 클래스 실행

- 코틀린을 사용해 스프링 부트 프로젝트 생성; 스프링 이니셜라이저 사용
  - 프로젝트 구조
    - src/main/kotlin: 모든 코틀린 소스코드가 있는 폴더다. 자바 프로젝트의 src/main/java와 유사하다
    - src/test/kotlin: 모든 코틀린 테스트 코드가 있는 폴더다. 자바 프로젝트의 src/test/java와 유사하다
    - 리소스 폴더는 일반적인 자바 프로젝트와 같다(src/main/resources 및 src/test/resources)
    - JRE 시스템 라이브러리 대신 코틀린 런타임 라이브러리가 실행 환경으로 사용된다
  - pom.xml 의존 관계 및 플러그인 추가
  - 스프링 부트 애플리케이션 클래스
  - 스프링 부트 애플리케이션 테스트 클래스

- 코틀린을 이용한 REST 서비스 구현
  - 문자열을 반환하는 간단한 메소드
    - 단위 테스트 작성
    - 통합 테스트 작성
  - 객체를 반환하는 간단한 REST 메소드
    - 요청 실행
    - 단위 테스트 작성
    - 통합 테스트 작성
  - path 변수를 가진 GET 메소드
    - 요청 실행
    - 단위 테스트 작성
    - 통합 테스트 작성

References

Design Patterns used in Spring Framework

- Proxy Pattern
- Singleton Pattern
- Factory Pattern
- Template Pattern
- MVC(Model-View-Controller) Pattern
- Front Controller Pattern
- View Helper Pattern
- Dependency Injection / Inversion of Control
- Service Locator
- Observer-Observable
- Context Object Pattern

SEE ALSO (Links)

- 스프링과 DAO, DTO, Repository, Entity
- SpringBoot2로 Rest api 만들기(5) - API 인터페이스 및 결과 데이터 구조 설계
- [스프링] XML을 이용한 쿼리 실행 (mapper 구현)
- Spring Tool Suite 4(STS) 기본 개념 및 단축키 등
- myBatis parameterType="String" 일 때 동적쿼리에서 사용
- [Spring] Spring Project 생성, xml 없이 java로 설정하기 (web.xml servlet-context.xml root-context.xml 제거)
- 쉽게 알아보는 서버 인증 2편(Access Token + Refresh Token)
- SPRING SECURITY + JWT 회원가입, 로그인 기능 구현
- [Spring] 보안에 대한 고민 : JWT 토큰?
- [toby의스프링] 1장 - 오브젝트와 의존 관계
- springboot로 Rest api 만들기(2) HelloWorld
- JPA H2 데이터베이스 및 서버 실행시 자동 Insert
- H2 DB 설정
- Spring Boot + MyBatis 설정 방법(HikariCP, H2)
- Swagger란?
- Swagger란
- [Spring]스프링 프레임워크 디자인 패턴
- Spring Filter, Interceptor, AOP 차이 및 정리
- 스프링 MVC - 구조 이해
- spring 빌드 & 실행
- 공통 처리의 구현 방법
- [기술면접] CS 기술면접 질문 - 백엔드 (8/8)
- [Spring] 스프링 인터셉터(Interceptor)란 ?
- 스프링 AOP 로그인 체크
- Spring MVC 동작원리 / 구성요소
- [Spring] Web on Reactive Stack 문서(1) - Spring WebFlux
- JUnit 5 공식 가이드 문서 정리
- Spring - AOP 개념 정리
- Spring MVC 흐름
- Today I Learned(TIL) Wiki
- [Spring] WebFlux의 개념 / Spring MVC와 간단비교
- [Spring] Web on Reactive Stack 문서(1) - Spring WebFlux
- 정의, IoC DI개념, Bean 라이프사이클
- [Spring] Bean LifeCycle 이란 무엇일까?
- [Spring] 스프링 의존성 주입(DI) 이란?
- [Spring] Bean은 어떻게 등록되는 것일까?
- Spring boot 설정 파일 yaml 사용법 (설정 파일을 읽어서 bean으로 필요할 때 사용하는 방법)
- [Spring Boot] 외부에서 설정 주입하기
- [Spring boot]yml configuration
- [Springboot] 설정 파일에서 값 가져오는 방법
- Spring Boot에서 properties 값 주입받기
- Accessing spring loaded properties in BeanDefinitionRegistryPostProcessor
- Spring dynamic beans
- 스프링 properties 변경 감지하여 동적으로 로딩하기!

- Spring에서 속성 파일 다시로드
- 스프링 프로퍼티 설정 주입3 - 실시간(run-time)으로 properties Relaod
- Spring AOP의 메커니즘과 Proxy Bean 생성
- [후펴파는 스프링] 스프링 컨테이너 계층 정리
- (요약) Spring Framework Overview
- Spring @Import, @Enable 어노테이션 활용
- @Enable + Registrar
- Spring Mybatis 다중 연결 맵퍼 스캔하기 : Multiple Datasource Mapper scan
- [AWS] Spring Boot에서 AWS S3와 연계한 파일 업로드처리
- [Spring/ AWS] Spring Boot S3를 이용하여 File 다운로드하기
- [AWS]Spring에서 S3 파일 업로드,다운로드, 삭제, 이름변경 방법 SeoDongEok/study
- 몰라도 되는 Spring - AOP Proxy
- [Spring] 스프링 기초(백기선 님의 스프링 프레임워크 핵심기술 강좌 공부)
- [Spring] 스프링 AOP with (프록시 패턴)
- [Spring] Spring의 AOP 프록시 구현 방법(JDK 동적 프록시, CGLib 프록시)과 @EnableAspectJAutoProxy의 proxyTargetClass - (3/3)
  [Spring] AOP(Aspect Oriented Programming, 관점 지향 프로그래밍)의 이해 - (1/3)
  [SpringBoot] AOP(Aspect Oriented Programming, 관점 지향 프로그래밍)의 개념 및 사용 방법 예제 코드 - (2/3)
- 스프링 AOP 개념 및 Proxy를 이용한 구동원리
- [Spring] Spring AOP - 원리편
  [Spring] Spring AOP - 기본 이론편
  [Spring] Spring AOP - 실전편
- MyBatis @Mapper 인터페이스는 어떻게 스프링 빈으로 와이어될 수 있을까?
- Spring Professional Study Notes
- Proxy 패턴, 그리고 Dynamic Proxy API

스프링 캐시

- [SPRING] 스프링 캐시 사용하기
- [Spring] 캐시(Cache) 추상화와 사용법(@Cacheable, @CachePut, @CacheEvict)
- Spring - Spring Cache 추상화(스프링 캐시 추상화),@Cacheable,@CacheEvict,@CachePut
- * [[https://velog.io/@qotndus43/Cache|스프링부트 Caching 도입하기(Redis, Ehcache)
- [SpringBoot] SpringBoot의 기본 Cache 사용하기
- [Spring Boot] Redis Cache, MySQL 이용한 간단한 API 제작
- Spring Boot + Redis + PostgreSQL Caching
- Spring Boot Cache 적용
- SpringBoot2로 Rest api 만들기(15) – Redis로 api 결과 캐싱(Caching)처리
- [Spring] 캐시 추상화 (Cache Abstraction) 알아보기
- Spring boot에서 Redis Cache 사용하기
- Spring boot & Redis Cache 서버 구축하기
- 스프링 캐시 인터셉터
- SpringBoot 구성 로딩 원리에 대한 포괄적 인 분석
- [SpringBoot] Ehcache를 사용한 Cache 사용
- Spring Boot Swagger 3.x 적용
- java-swagger-ui-3.0-사용하기
- SpringBoot에서 Ehcache 사용해보기

- 필터
- 인터셉터
- 어드바이저
- 이벤트

- Startups
  - configurations
  - logging
  - exception handlings
- Running
  - rest api
  - soap(xml)
  - web socket
  - socket
  - auth
  - views
  - data
  - files
  - users
  - i18n
  - cache
- Tests
  - unit test

From:
http://www.theta5912.net/ - **reth**

Permanent link:
http://www.theta5912.net/doku.php?id=public:computer:java_spring_framework&rev=1655794421

Last update: **2022/06/21 15:53**