IoC (Inversion of Control)

객체가 의존 하는 다른 객체를 생성하여 사용하는 것이 아닌, 의존 객체를 주입받아 사용하는 방법.

의존 관계 주입(Dependency Injection)이라고도 한다.

내가 라이브러리를 가져다 쓰는 것이 아닌, 라이브러리가 내 모듈을 가져다 쓰게 한다.

Spring의 IoC Container

- 주요 인터페이스
  - 최상위 : BeanFactory
  - ApplicationContext
- 애플리케이션 컴포넌트의 중앙 저장소
- Bean 설정 코드로부터 Bean 정의를 읽어들이고, Bean을 구성 및 생성하여 제공한다.

Bean

- 스프링 IoC 컨테이너가 관리하는 객체
- 예전엔 xml 방식으로 표기했었음. 현재는 Annotation을 사용하여 표기
- 자바빈과 헷갈릴 수 있으니 주의하자. 스프링에서의 빈은 자바빈이 아니다.
- @Service @Controller @Repository 등
- IoC 컨테이너로 생성되는 빈은 기본적으로 singleton 으로 등록된다.
- Bean의 Scope
  - 싱글톤 : 1개만 만들어서 사용
  - 프로토타입 : 매번 다른 객체
- Bean으로 등록하면 좋은 이유
  - 의존성 관리가 용이
- 단위 테스트(TDD)가 편함 (가짜 객체를 넣어 사용하는 등)
  - 라이프사이클 인터페이스 사용 가능

단위 테스트 예)

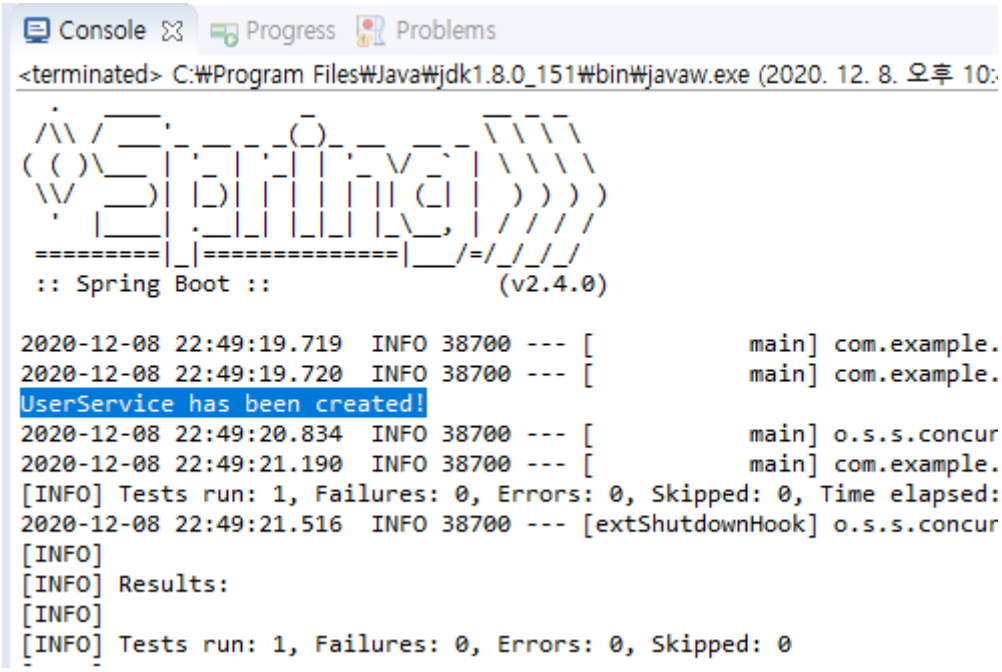
```
  package com.example.demo.user;

  import static org.assertj.core.api.Assertions.assertThat;
  import static org.mockito.Mockito.mock;
  import static org.mockito.Mockito.when;

  import org.junit.jupiter.api.Test;

  public class UserServiceTest {
    /*
     * UserService 를 테스트한다고 가정.
     * UserService는 UserRepository를 의존하고 있음
     * 따라서 UserRepository 가 구현되어있지 않다면, UserService는 단위 테스트를 진행하지 못함 (의존성의 단점)
```

```
     * UserRepository를 new하지 않는 대신 Mocking Bean으로 등록해두고 가상의 시나리오를 지정해주면
     * UserRepository가 구현되어있지 않더라도 단위테스트를 진행할 수 있다.
     */

    @Test
    public void saveUser() {
        UserRepository userRepository = mock(UserRepository.class); // 가짜 빈 생성
        User user = new User();
        user.setName("Hong");
        System.out.println("Before setRegdate : " + user.getRegdate());

        when(userRepository.insert(user)).thenReturn(user);
        // insert()가 호출되면 user를 return하라.
        // (실제 insert()는 null 을 return 하도록 되어있다.
        // 따라서 insert()를 그대로 두고 테스트를 진행한다면 하단에서 assertionError가 날 것이다.)

        UserService userService = new UserService(userRepository);

        User resUser = userService.saveUser(user);

        System.out.println("After setRegdate : " + resUser.getRegdate());

        assertThat(user).isNotNull(); // user 가 not null 인지 체크한 후, 아니라면 assert 실행
        assertThat(user.getName()).isEqualTo("Hong");
        assertThat(resUser).isNotNull();

    }

  }
```

라이프사이클 예) @PostConstruct 인터페이스

```
package com.example.demo.user;

import java.util.Date;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Service;

@Service
public class UserService {

    @PostConstruct
    public void postContsruct() {
        System.out.println("UserService has been created!");
    }
}
```

결과)



ApplicationContext 인터페이스

- BeanFactory를 상속 받음
- BeanFactory가 가지고 있는 IoC 컨테이너로서의 기능 포함하여 다음 기능을 가짐
    - ApplicationEventPublisher, BeanFactory, EnvironmentCapable, HierarchicalBeanFactory, ListableBeanFactory, MessageSource, ResourceLoader, ResourcePatternResolver

Maven의 IoC 활용하기

1. 디펜던시 기본 구성

pom.xml

```
...
<!-- 의존성을 관리하는 부분 -->
<dependencies>
    <!-- 의존성 추가 -->
    <dependency>
        <groupId>그룹명</groupId>
        <artifactId>아티팩트명</artifactId>
        <version>버전정보</version>
    </dependency>
    <!-- 의존성 추가 -->
    <dependency>
        <groupId>그룹명</groupId>
        <artifactId>아티팩트명</artifactId>
        <version>버전정보</version>
    </dependency>
</dependencies>
```

```
...
```

groupId

- 모든 프로젝트 내부에서 사용할 고유 이름
- package 명명 규칙과 동일한 컨벤션을 따른다.
- 하위 그룹은 지정하지 않아도 된다.

예) `com.my.package`

artifactId

- 버전 정보를 생략한 jar 파일의 이름
- 소문자로만 작성한다.
- 써드 파티 jar 파일이라면, 할당된 이름을 사용한다.

예) `org.apache.maven`, `org.apache.maven.plugins`, `org.apache.maven.reporting`

version

- 숫자와 점으로 이루어진 일반적인 버전 형태를 사용한다(`1.0`, `1.1`, `1.0.1`, ...).
- SNAPSHOT(nightly) 빌드 날짜를 버전으로 사용하지 않도록 한다.
- 써드 파티 아티팩트라면, (좀 이상하게 보일 수 있어도) 그들의 버전 넘버를 이어받아 사용하도록 한다.
- 자동 버전 관리가 적용되는 의존성이라면 생략 가능하다.

  하지만 버전을 써주는 것을 추천한다.

예) `2.0`, `2.0.1`, `1.3.1`

출처 : https://maven.apache.org/guides/mini/guide-naming-conventions.html

2. 디펜던시 추가하기

1. 메이븐 저장소를 사용하는 방법

https://mvnrepository.com/

자주 사용할 곳이니 북마크 해두자!

2. 부모 프로젝트의 의존성 버전 오버라이드

`<parent>`가 제공하는 의존성들이 있다. 왠만한 필수 의존성은 ``'가 가지고 있는데, 만약 이들의 버전을 변경하고자 한다면 이를 오버라이딩 할 수 있다.

예) 스프링 버전과 JDK 를 수정하고 싶다면?

```
[pom.xml]
...
<properties>
    <spring.versions>5.0.6.RELEASE</spring.versions>
    <java.version>1.8</java.version>
</properties>
```

`<parent>`의 pom.xml에 들어가 parent에 정의된 `<properties>`, `<dependencies>`, `<plugins>`를 확인해보자. 오버라이딩하고 싶은 의존성이 있다면 그 부분을 그대로 복붙하여 버전만 변경해주면 된다.

Gradle 의 IoC 활용하기

build.gradle 의 예

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}
```

Maven VS Gradle

공통점 : 둘 모두 의존성 관리를 위한 빌드툴이다.

다음 두개의 파일을 보면 둘 모두 같은 의존성을 가지고 있다.

pom.xml

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-
starter-web -->
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.3.7.RELEASE</version>
</dependency>
```

build.gradle

```
// https://mvnrepository.com/artifact/org.projectlombok/lombok
providedCompile group: 'org.projectlombok', name: 'lombok', version: '1.18.12'


// https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web
compile group: 'org.springframework.boot', name: 'spring-boot-starter-web', version: '2.3.7.RELEASE'
```

Maven의 Lifecycle

maven 에서는 미리 정의하고 있는 빌드 순서가 있으며 이 순서를 라이프사이클이라고 한다. 라이프 사이클의 가 빌드 단계를 Phase라고 하는데 이런 Phase들은 의존관계를 가지고 있다.

- Clean : 이전 빌드에서 생성된 파일들을 삭제하는 단계
- Validate : 프로젝트가 올바른지 확인학고 필요한 모든 정보를 사용할 수 있는 지 확인하는 단계
- Compile : 프로젝트의 소스코드를 컴파일하는 단계
- Test : 유닛(단위) 테스트를 수행하는 단계(테스트 실패시 빌드 실패로 처리, 스킵 가능)
- Package : 실제 컴파일된 소스 코드와 리소스들을 jar등의 배포를 위한 패키지로 만드는 단계
- Verify : 통합테스트 결과에 대한 검사를 실행하여 품질 기준을 충족하는지 확인하는 단계
- Install : 패키지를 로컬 저장소에 설치하는 단계
- Site : 프로젝트 문서를 생성하는 단계
- Deploy : 만들어진 Package를 원격 저장소에 release하는 단계

위에 9개의 라이프사이클 말고도 더 많은종류의 라이프사이클이 존재한다. 이를 크게 Clean, Build, site 세가지 라이프사이클로 나누고 있다. 각 단계를 모두 수행하는 것이 아니라 원하는 단계까지만 수행할 수도 있으며 test단계는 큰 프로젝트의 경우에 몇시간이 소요될 수도있으니 수행하지 않도록 스킵이 가능하다.

#Phase와 Goal

위에서 잠깐 언급했던 Phase는 Maven의 Build LifeCycle의 각각의 단계를 의미한다. 각각의 Phase는 의존관계를 가지고있어 해당 Phase가 수행되려면 이전 단계의 Phase가 모두 수행되어야 한다.

메이븐에서 제공되는 모든 기능은 플러그인 기반으로 동작하는데 메이븐은 라이플사이클에 포함되어있는 Phase마저도 플러그인을 통해 실질적인 작업이 수행된다. 즉 각각의 Phase는 어떤 일을 할지 정의하지 않고 어떤 플러그인의 Goal을 실행할지 설정한다.

메이븐에서는 하나의 플러그인에서 여러작업을 수행할 수 있도록 지원하며, 플러그인에서 실행할 수 있는 각각의 기능을 goal이라고 한다. 플러그인의 goal을 실행하는 방법은 다음과 같다.

- -mvn groupId:artifactId:version:goal(아래와 같이 생략가능)
- -mvn plugin:goal

POM - Project Object Model

pom은 Project Object Model 의 약자로 이름 그대로 Project Object Model의 정보를 담고있는 파일이다. 이 파일에서 주요하게 다루는 기능들은 다음과 같다.

- -프로젝트 정보 : 프로젝트의 이름, 개발자 목록, 라이센스 등
- -빌드 설정 : 소스, 리소스, 라이프 사이클별 실행한 플러그인(goal)등 빌드와 관련된 설정
- -빌드 환경 : 사용자 환경 별로 달라질 수 있는 프로파일 정보
- -POM연관 정보 : 의존 프로젝트(모듈), 상위 프로젝트, 포함하고 있는 하위 모듈 등

POM은 pom.xml파일을 말하며 Maven의 기능을 이용하기 위해 POM이 사용된다.

Gradle

지금까지 Maven을 알아보았다. 다음은 Gradle에 대해 알아볼 차례이다.

Gradle이란 기본적으로 빌드 배포 도구(Build Tool)이다. 안드로이드 앱을 만들때 필요한 공식 빌드시스템이기도 하며 JAVA, C/C++, Python 등을 지원한다.

빌드툴인 Ant Builder와 그루비 스크립트를 기반으로 구축되어 기존 Ant의 역할과 배포 스크립트의 기능을 모두 사용가능하다.

기본 메이븐의 경우 XML로 라이브러리를 정의하고 활용하도록 되어 있으나, Gradle의 경우 별도의 빌드 스크립트를 통하여 사용할 어플리케이션 버전, 라이브러리등의 항목을 설정 할 수 있다.

장점으로는 스크립트 언어로 구성되어 있기때문에, XML과 달리 변수선언, if, else, for등의 로직이 구현 가능하여 간결하게 구성 가능하다.

- 라이브러리 관리 : 메이븐 레파지토리를 동일하게 사용할 수 있어서 설정된 서버를 통하여 라이브러리를 다운로드 받아 모두 동일한 의존성을 가진 환경을 수정할 수 있다. 자신이 추가한 라이브러리도 레파지토리 서버에 올릴 수 있다.
- 프로젝트 관리 : 모든 프로젝트가 일관된 디렉토리 구조를 가지고 빌드 프로세스를 유지하도록 도와준다.
- 단위 테스트 시 의존성 관리 : junit 등을 사용하기 위해서 명시한다.

그래서 뭐가 좋은데?

Maven에는 Gadle과 비교문서가 없지만, Gradle에는 비교문서가 있다... 그만큼 Gradle이 자신있는 모습이다.

Gradle이 시기적으로 늦게 나온만큼 사용성, 성능 등 비교적 뛰어난 스펙을 가지고 있다.

Gradle이 Maven보다 좋은점

- Build라는 동적인 요소를 XML로 정의하기에는 어려운 부분이 많다.
    - 설정 내용이 길어지고 가독성 떨어짐
    - 의존관계가 복잡한 프로젝트 설정하기에는 부적절
    - 상속구조를 이용한 멀티 모듈 구현

    - 특정 설정을 소수의 모듈에서 공유하기 위해서는 부모 프로젝트를 생성하여 상속하게 해야함 (상속의 단점 생김)
- Gradle은 그루비를 사용하기 때문에, 동적인 빌드는 Groovy 스크립트로 플러그인을 호출하거나 직접 코드를 짜면 된다.
    - Configuration Injection 방식을 사용해서 공통 모듈을 상속해서 사용하는 단점을 커버했다.
    - 설정 주입시 프로젝트의 조건을 체크할 수 있어서 프로젝트별로 주입되는 설정을 다르게 할 수 있다.

> Gradle은 메이븐보다 최대 100배 빠르다.

결론

지금 시점에서 Gradle을 사용하지 않을 이유는 익숙함 뿐인 것 같다. Gradle이 출시되었을 때는 Maven이 지원하는 Scope를 지원하지 않았고 성능면에서도 앞설것이 없었다. Ant의 유연한 구조적 장점과 Maven의 편리한 의존성 관리 기능을 합쳐놓은 것만으로도 많은 인기를 얻었던 Gradle은 버전이 올라가며 성능이라는 장점까지 더해지면서 대세가 되었다.

물론 그동안 사용해왔던 Maven과 이제는 익숙해진 XML을 버리고 Gradle과 Groovy문법을 배우는 것은 적지않은 비용이 든다. 특히 협업을 하는 경우, 프로젝트 구성과 빌드만을 위해 모든 팀원이 Groovy 문법을 익혀야 한다는 사실은 Gradle을 사용하는데 큰 걸림돌이 된다.

실제로 여전히 Maven의 사용률은 Gradle을 앞서고 있으며, 구글 트랜드 지수도 Maven이 Gradle을 앞선다.

협업과 러닝커브를 고려하여 여전히 Maven을 사용하는 팀이 많고 부족함 없이 잘 사용하고 있지만, 빌드타임 비용문제로 이어질경우 Gradle을 사용해야 할 것 같다. 왜냐하면 Gradle이 Maven보다 최대 100배 빠르기 때문이다.

리드미컬하게 테스트를 진행하고 민첩한 지속적 배포를 생각하고 있다면 새로운 배움이 필요하더라도 Gradle을 사용해보자.

출처

https://gradle.org/
https://maven.apache.org/
http://egloos.zum.com/kwon37xi/v/4747016
https://bkim.tistory.com/13

https://hyojun123.github.io/2019/04/18/gradleAndMaven/