

# chpt2

## Spring Boot에서 Java의 빌드를 자동화 시켜주는 Maven / Gradle

- JAVA로 이뤄진 프로젝트는 `.java` 파일에 정의된 java source code로 이뤄짐
- jdk : 자바 소스 코드를 자바 바이트 코드로 바꾸기 위한 javac 파일이 들어있는 상태

### > javac 란?

- JAVA COMPILER (컴파일 : 인간을 위한 소스코드를 기계가 알아들을 수 있게 기계어로 바꾸는 과정)
- 컴파일 다음 단계는 `*BUILD**` 라고한다
- 빌드란 실제로 어떤 기계어가 실행이 되어서 완성이 되는데까지 필요한 모든 과정을 `빌드` 라고 함

### <빌드의 과정>

1. javac를 통해서 java 언어를 컴파일 한다
2. java 소스코드는 java byte code로 변환되게 된다
3. 테스트 코드로 만든 bytecode는 그 단계에서 바로 실행을 한다
4. 실행 후 아무 문제 없다면 이를 실행 가능한 파일로 제작 (=패키징 과정)

=> - 하지만 javac 명령어만으로도 전체 프로젝트를 다루기는 어려움

==> 이를 보완하고자 Maven 과 Gradle 이라는 아이가 등장 (spring boot initializer)

- 자바 프로젝트 관리하기 위한 가장 대표적인 도구, 이들을 빌드를 자동화시켜주는 역할을 한다

---

## Maven

- JAVA를 위한 빌드 자동화 도구
- 사실 JAVA이외의 C#, Ruby 등의 다른 언어를 위해서도 사용하지만 거의 다 JAVA로만 쓰인다
- Project Object Model (POM)
- xml의 형태로 프로젝트 정의

- pom.xml 분석해 프로젝트 빌드
- 메이븐에서는 pom.xml 파일이 존재하는 곳이 프로젝트의 근본이라고 간주하는 것 가능

## Maven으로 파일 제작해보기

- spring initializer 에서 나같은 경우에는

위와 같이 파일을 생성해서 프로젝트를 만들었다.

The screenshot shows the Spring Initializr web application interface. It has a dark theme with a green logo and text. The interface is divided into sections for configuring a new project.

**Project**

- ☒ Maven Project
- ☐ Gradle Project

**Language**

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

**Spring Boot**

- ☐ 2.7.0 (SNAPSHOT)
- ☐ 2.6.3 (SNAPSHOT)
- ☒ 2.6.2
- ☐ 2.5.9 (SNAPSHOT)
- ☐ 2.5.8

**Project Metadata**

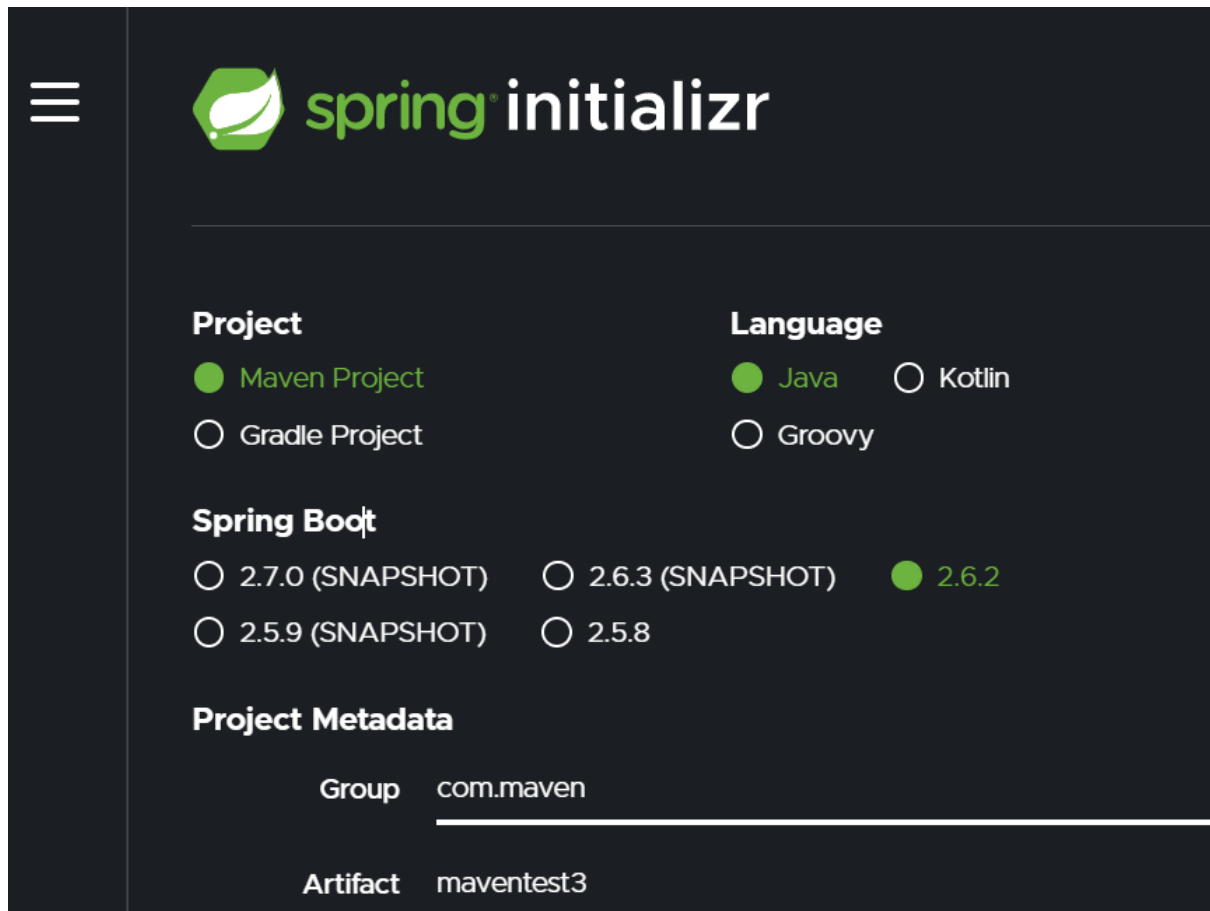
Group: com.maven

Artifact: maventest3

- pom.xml 구조 살펴보기

### 1. properties : 속성 나타내는 부분

1. 버전 나타내는 부분



The image shows the Spring Initializr web interface. It has a dark theme with a green logo and text. The interface is divided into sections for Project, Language, Spring Boot, and Project Metadata.

**Project**

- ☒ Maven Project
- ☐ Gradle Project

**Language**

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

**Spring Boot**

- ☐ 2.7.0 (SNAPSHOT)
- ☐ 2.6.3 (SNAPSHOT)
- ☒ 2.6.2
- ☐ 2.5.9 (SNAPSHOT)
- ☐ 2.5.8

**Project Metadata**

Group: com.maven

Artifact: maventest3

1. parent - 상속과 관련된 부분

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.6.2</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

1. initializer 설정에서 정했던 부분들

```

<name>maventest3</name> ✓
<description>maven test</description> ✓
<properties>
  <java.version>11</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

```

## 2. dependency : 사용할 외부 라이브러리 명시해주는 부분

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

## 3. build : 빌드를 어떠한 방식으로 진행할 것인지 명시해주는 부분

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

## Gradle

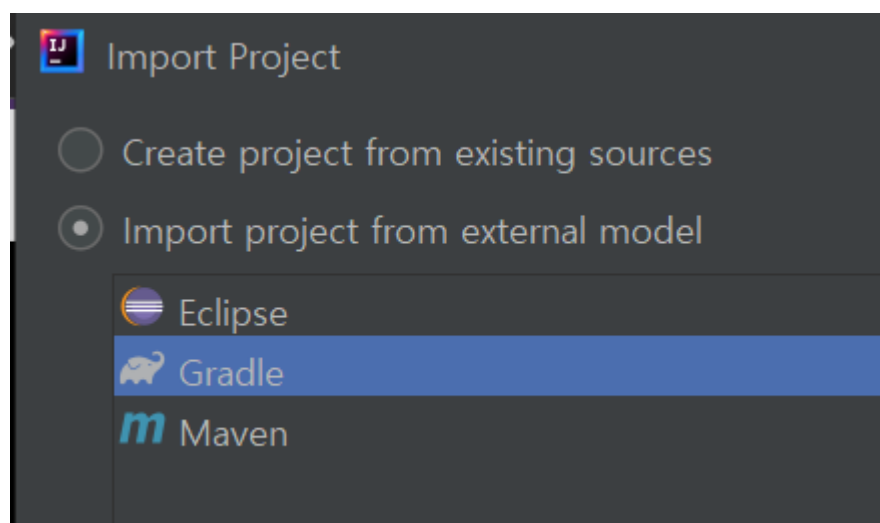
- JAVA를 위한 Build Automation 도구
- C, C++, JS, 등을 위해서도 사용 가능

### bundle.gradle

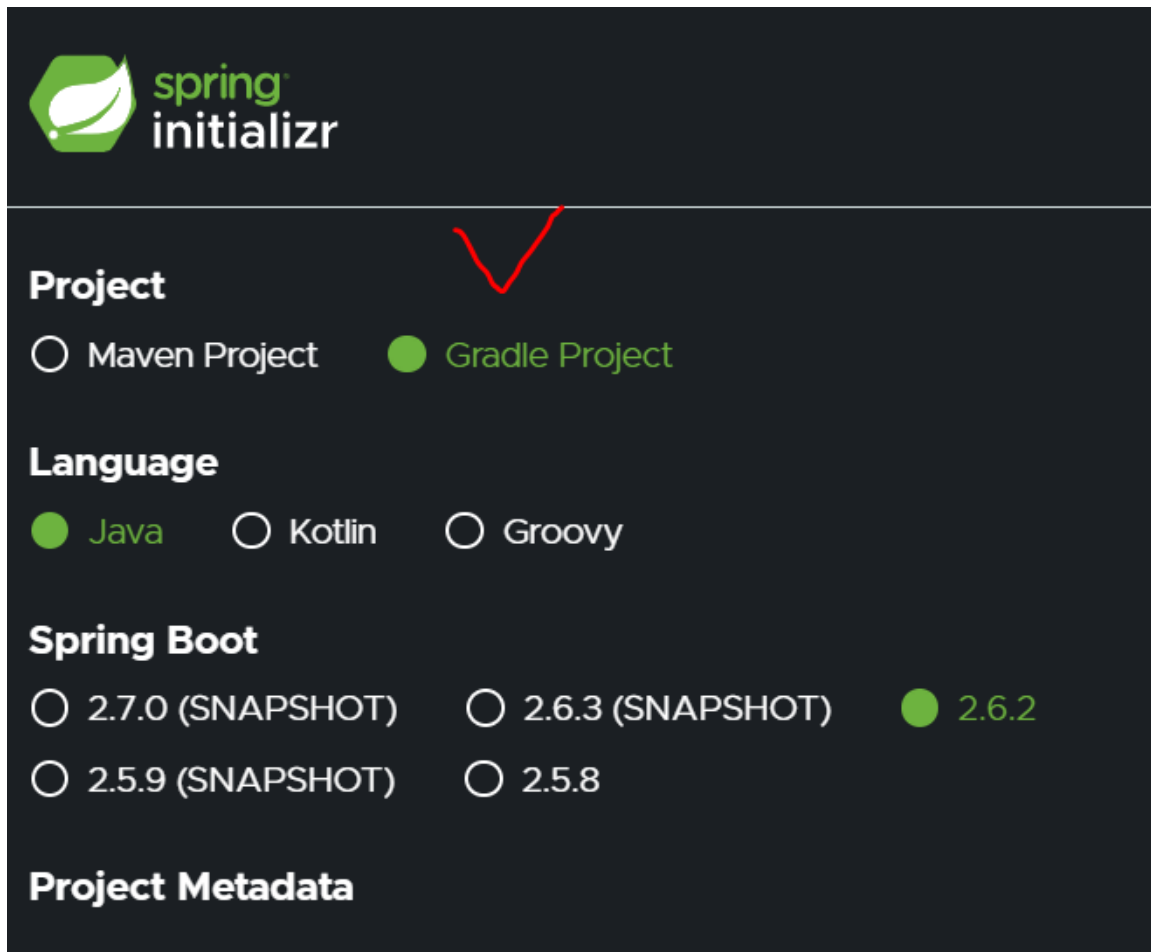
- groovy라는 언어로 프로젝트 정의
- sub-project 등을 포함시키는 용도의 settings.gradle 도 존재
- kotlin을 사용해 정의 가능 (build.gradle.kts)

## Gradle 로 파일 제작

- spring initializer 에서 gradle 로 파일 생성



- gradle 로 선택



The image shows the Spring Initializr web application interface. At the top is the Spring Initializr logo. Below it, a red checkmark is drawn above the 'Project' section. The 'Project' section has two radio buttons: 'Maven Project' (unselected) and 'Gradle Project' (selected). The 'Language' section has three radio buttons: 'Java' (selected), 'Kotlin' (unselected), and 'Groovy' (unselected). The 'Spring Boot' section has four radio buttons: '2.7.0 (SNAPSHOT)' (unselected), '2.6.3 (SNAPSHOT)' (unselected), '2.6.2' (selected), and '2.5.9 (SNAPSHOT)' (unselected). The '2.5.8' option is also present but unselected. The 'Project Metadata' section is visible at the bottom but empty.

**Project**

☐ Maven Project ☒ Gradle Project

**Language**

☒ Java ☐ Kotlin ☐ Groovy

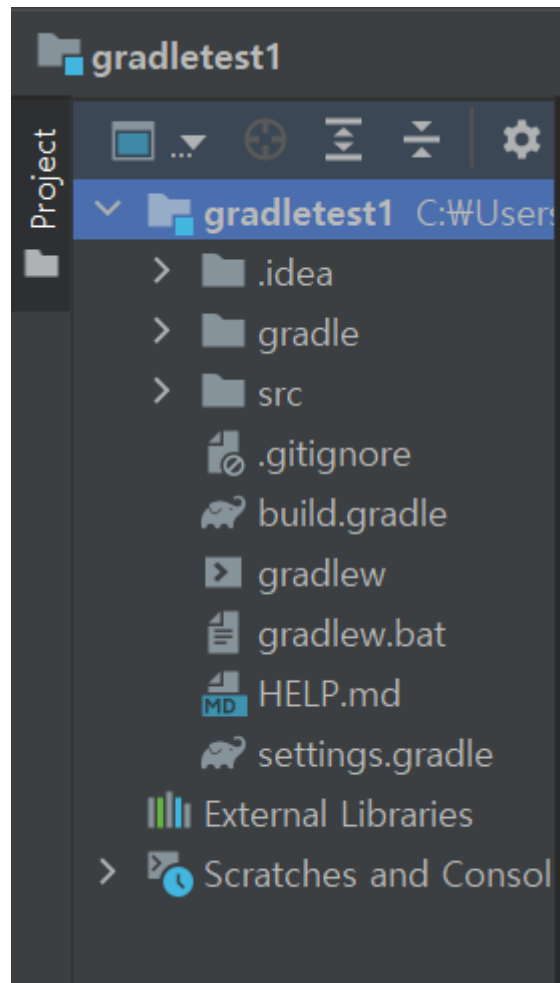
**Spring Boot**

☐ 2.7.0 (SNAPSHOT) ☐ 2.6.3 (SNAPSHOT) ☒ 2.6.2

☐ 2.5.9 (SNAPSHOT) ☐ 2.5.8

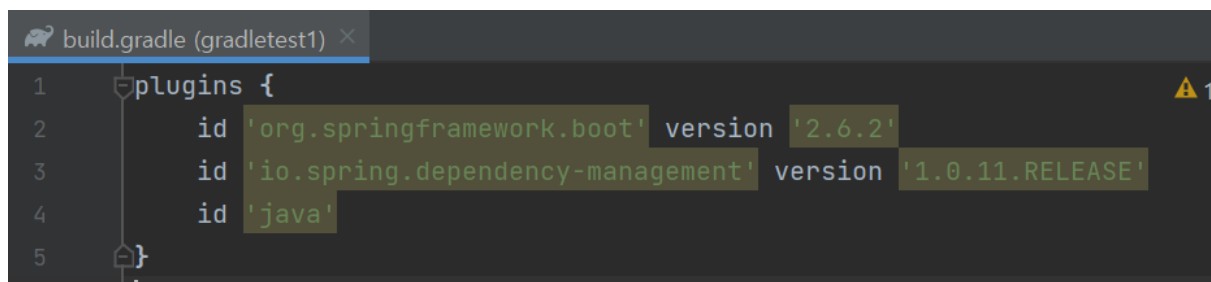
**Project Metadata**

- 아래와 같이 프로젝트가 잘 생성됨



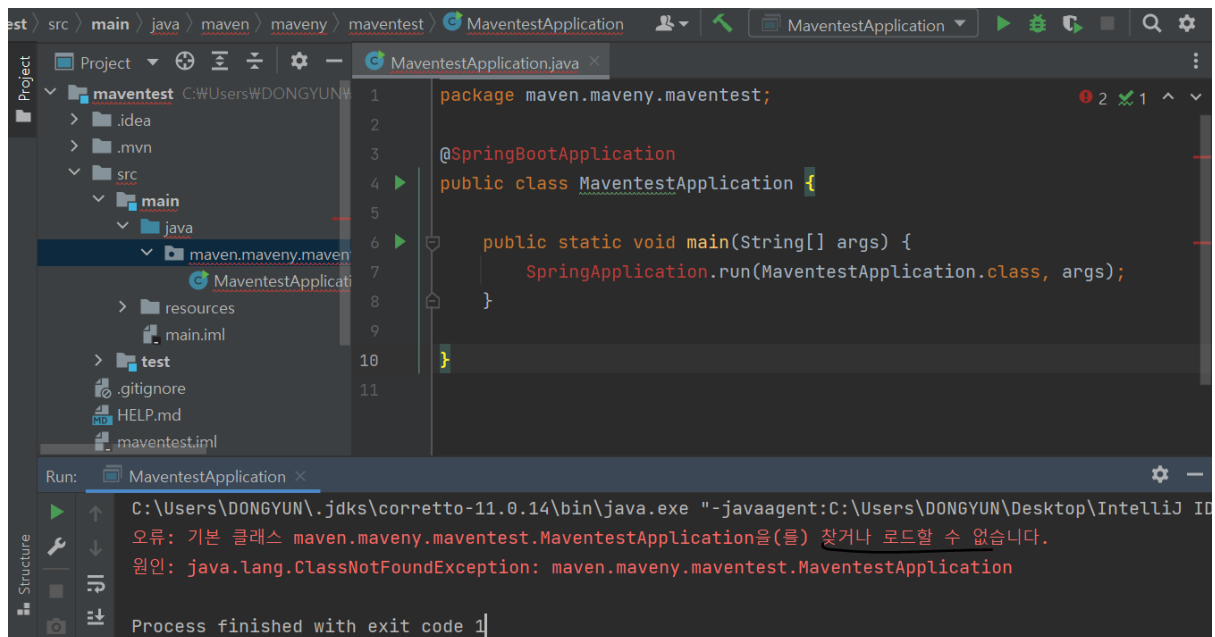
---

## 1. plugin : 할 수 있는 태스크들



- 어떤 식으로 버전 관리, 의존성 관리 할 건지
- 
- gradle 의 장점
    - => incremental build (점진적인 빌드)
  - 일정 부분까지 빌드가 되면 그 이후로는 그 부분까지 다시 빌드할 필요가 없음
  - BUT 초기 단계에선 maven 보다 조금 더 느린 경우가 있음
-

## (+) IntelliJ, Maven으로 파일 생성 시 java.lang.ClassNotFoundException에러



- maven으로 파일을 생성하니 위와 같은 에러가 뜨는 상황이 발생

나는 모든 해결방법을 해보았는데 절대 에러가 풀리지 않아서 그냥  
spring initializer에서 파일을 다시 만들 때, 버전을 다른 아이로 바꿔주  
면서 에러를 해결하고자 했음

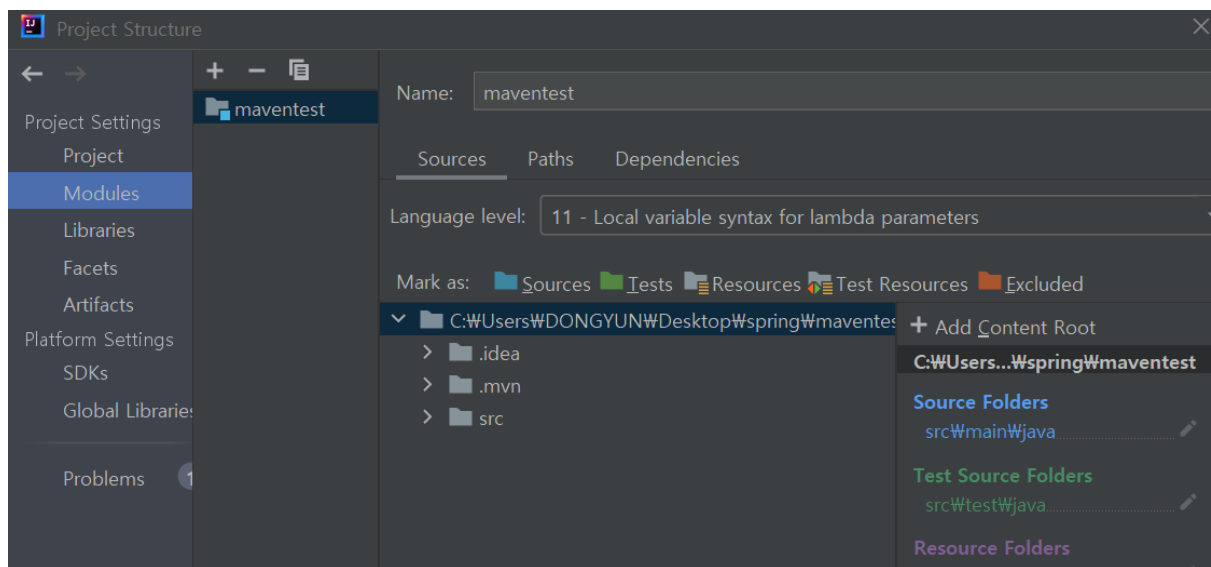
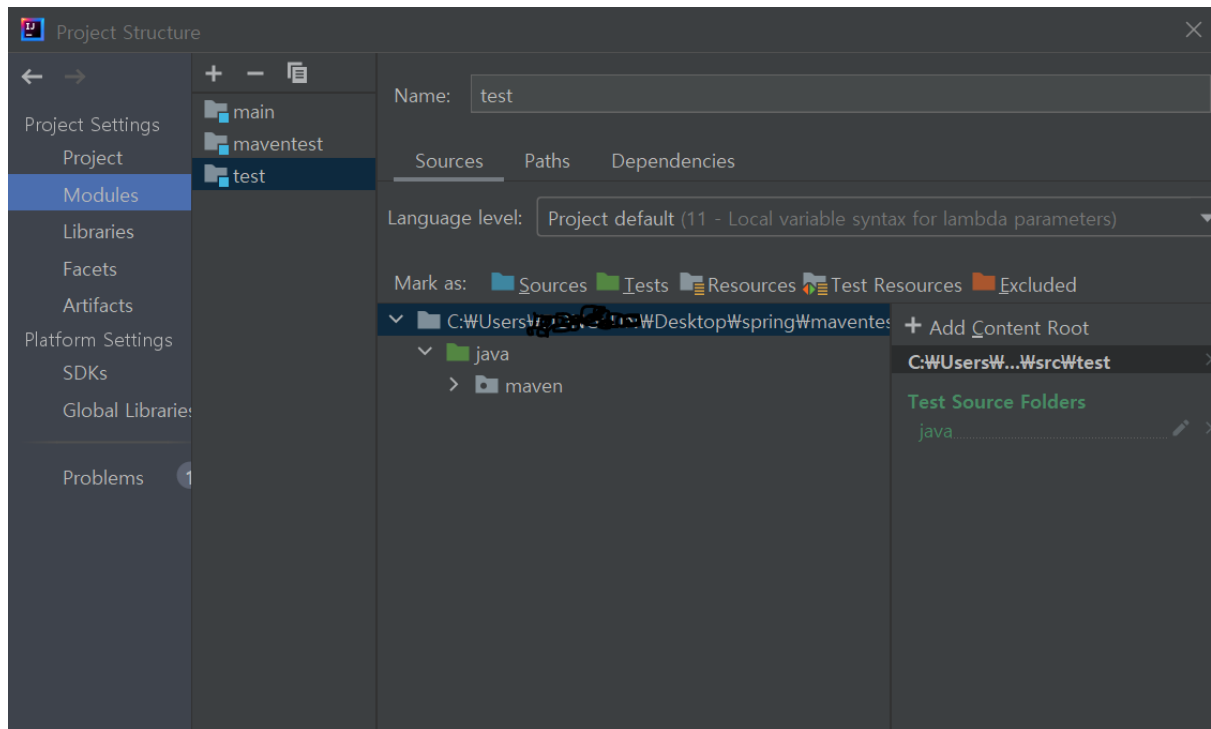
메이븐 에러 <- 블로그를 참조

- 관련 스택오버플로우
- 나랑 똑같은 에러나신 분 스택오버플로우
- 윗 블로그에서 써주신 것에 따르면 메이븐에서 Dependency로 설정된 라이브러리 (\*\*\*.jar) 들이 인  
식되지 않는 오류\*\*가 발생하곤 합니다. 라고 한다

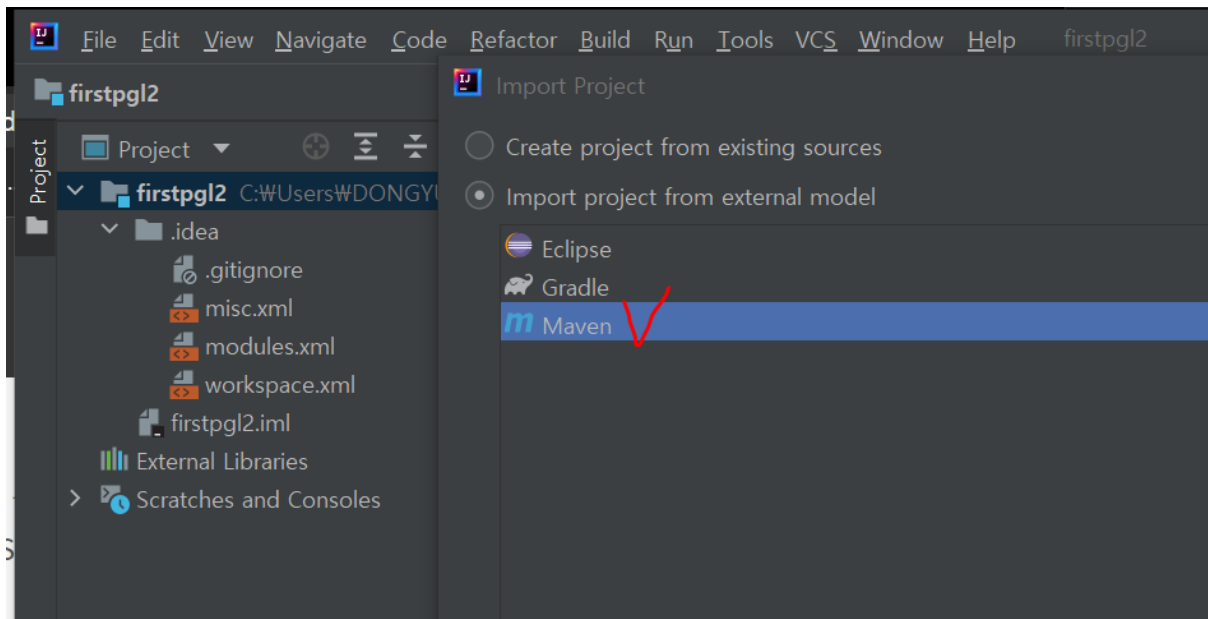
=> 해결방법

(1) 시도 1 : main, test 지우는 것 => 실패

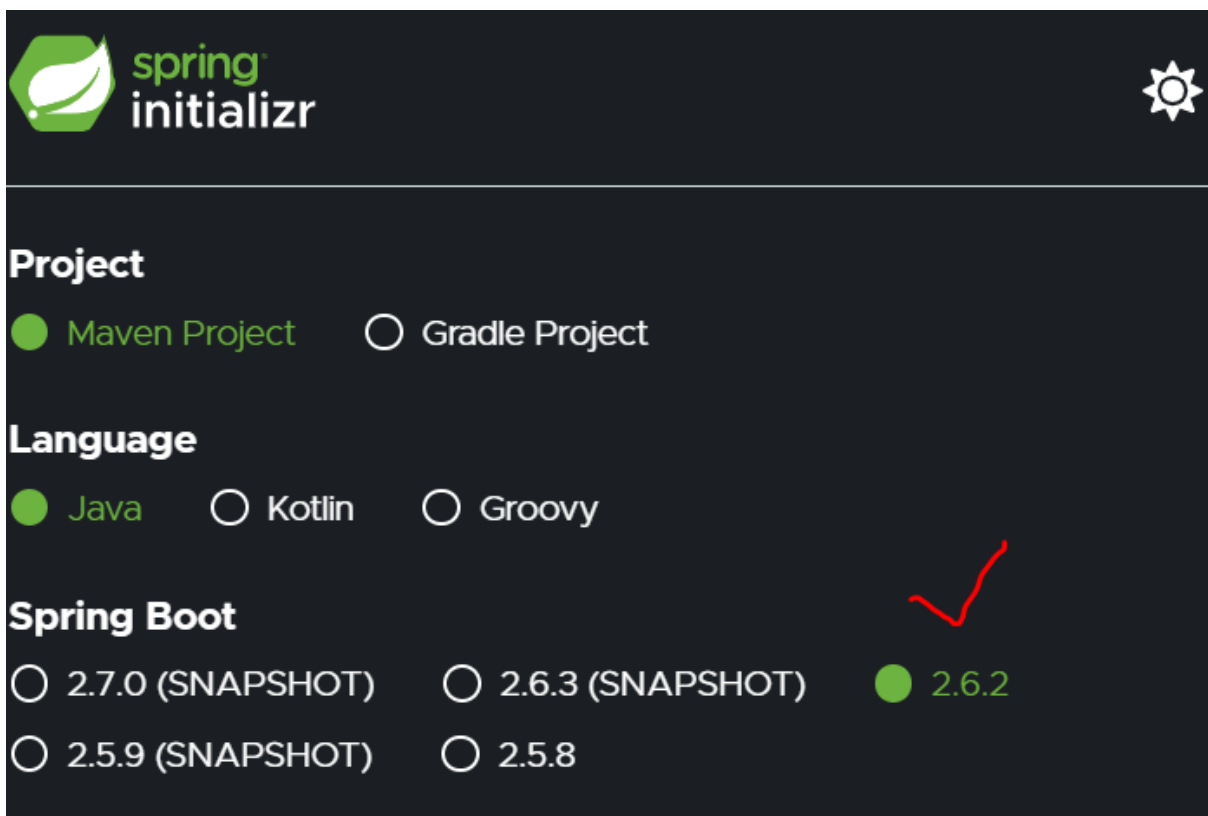




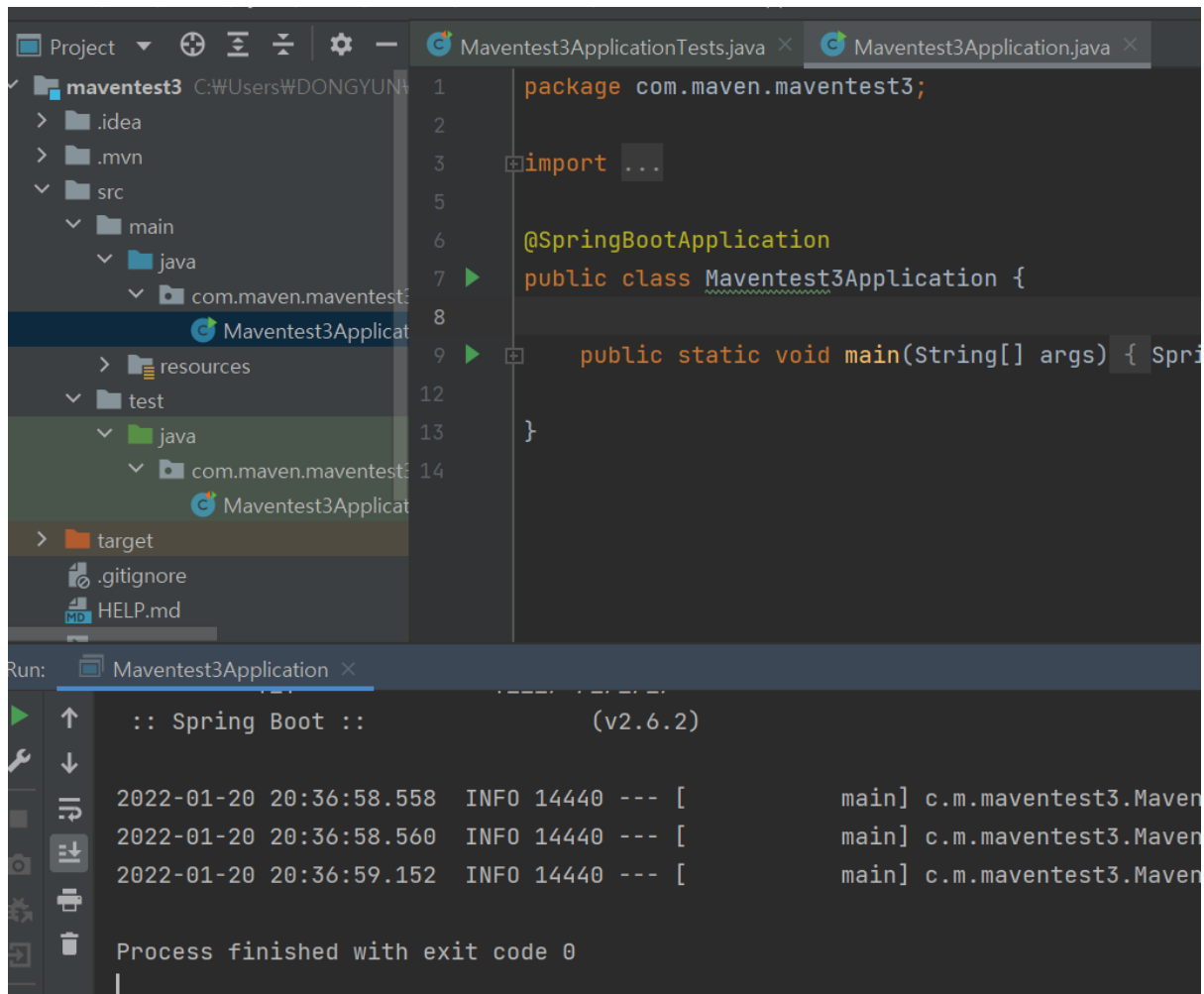
(2) 시도 2 : => 실패



(3) 시도 3 : => spring boot 버전을 잘 골라야 하는 듯



원래는 2.6.~ 아니고 다른 버전으로 선택하고 만들 땐 계속 오류가 났었는데, 2.6~로 선택해 주니 아무 에러 없이 잘 돌아간다.. 만약 나와 같은 에러가 나는 사람들이 있다면 spring initializer에서 다시 파일을 버전에 맞게 생성하는 것을 추천드림



## 일반적인 웹 서비스의 배포 방식

사용자브라우저 <-- 컴퓨터 통신 영역 --> 물리적컴퓨터(서버)

- 신호들이 서버로 전달될 때 신호들은 ip 주소를 보고 ip 주소에 해당하는 서버로 신호를 전달해준다.
- IP : 포트 , 번호로 구성되어 있는데 IP 주소로는 신호를 "어떤 물리적 서버" 에 전달해줄 지, 포트 번호로는 해당 서버에서 "어떤 웹서버"로 신호를 최종적으로 전달해줄 지 판단해줄 수 있다.
- 웹 서버 : HTTP 요청을 받아서 그 요청을 전달하거나, 요청에 맞게 파일을 전달하는 역할
- 따라서 브라우저로부터 온 신호들은 최종적으로 웹 서버들에게 전달이 됨 (ex) APACHE, Nginx
- 그런데 웹 서버에서 또 다른 웹 애플리케이션 서버 , 파일 시스템(어떤 정보를 달라 라고 요청)으로 전달이 될 수도 있다 ...

(+ 웹 서버와 웹 애플리케이션 시스템은 서로 다르다!)

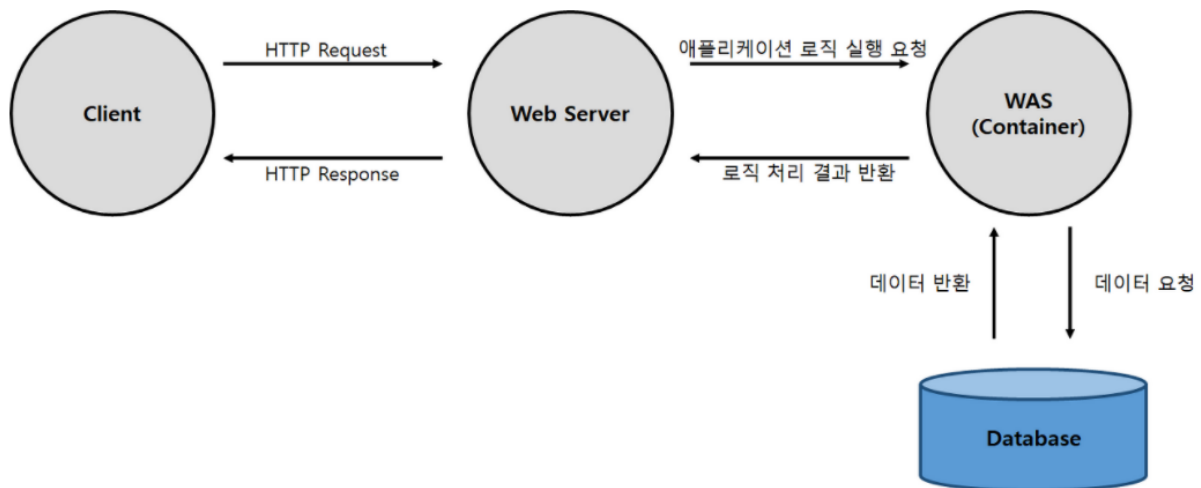
## 웹 서버 VS 웹 애플리케이션 서버

출처 :

블로그 출처1

출처 :

이미지,내용 출처2



### 웹 서버

(EX : Nginx, APACHE)

- 정적자원의 파일을 웹 클라이언트에게 제공할 때 사용
- 동적인 페이지 처리가 필요하다면 웹 애플리케이션 서버에 요청을 넘김

### 웹 애플리케이션 서버

(EX : 톰캣 - 호랑이얼굴고양이)

- 웹 서버로부터 요청을 받아서 처리, 결과는 웹 서버로 반환
  - 주로 동적인 페이지 생성을 위한 데이터베이스 연동의 작업을 진행
- 
- 웹 애플리케이션 서버는 웹 아카이브 파일들, 어플리케이션 파일들을 열기 위한 서버
  - 웹 어플리케이션 서버는 스프링 / 스프링 부트에 접근하여 자신에게 요청된 것을 수행하는 것
  - 이때 그냥 SPRING 을 사용했다면 WAR 파일의 형태로 있었을 것이고 / Spring 부트라면 그 자체로..

## 총 정리

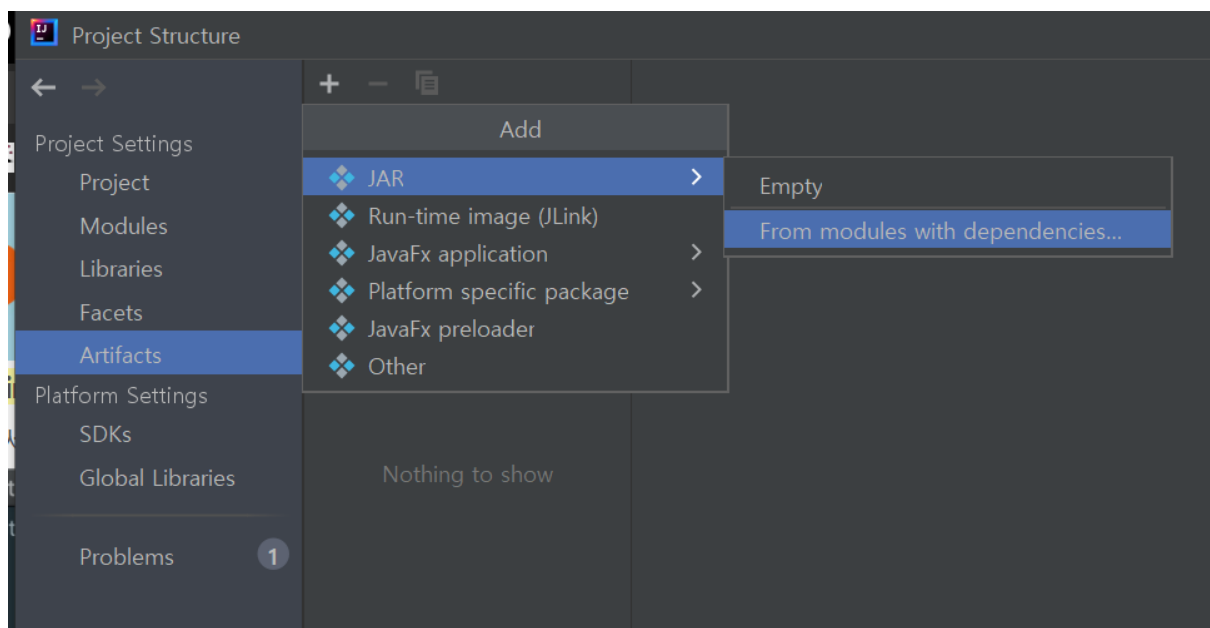
- 하드웨어 서버 - 웹 서버 - (스프링 부트와 같은 웹 어플리케이션 시스템으로 넘겨주기 / 혹은 파일로 넘겨주기)

## Java와 Jar

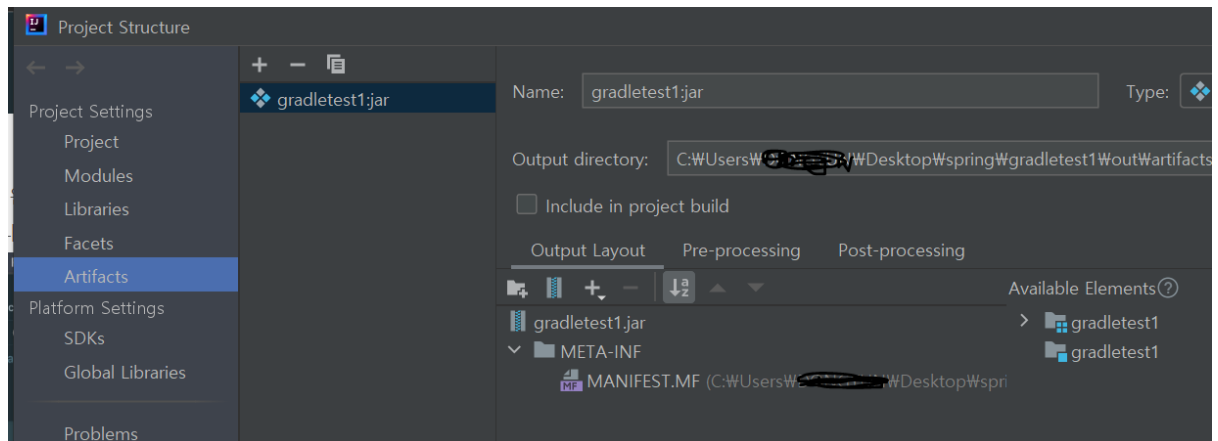
- 스프링 부트 프로젝트를 빌드하면 jar 파일이 나오게 된다
- jar : JAVA ARchive
- JAVA로 작성 후 컴파일 된 JAVA Bytecode와 실행을 위해 필요한 다양한 자원을 배포를 위해 모아놓은 파일의 형태  
=> 다른 형태의 압축파일과 같다고 볼 수 있음

## Jar 파일 생성해보기

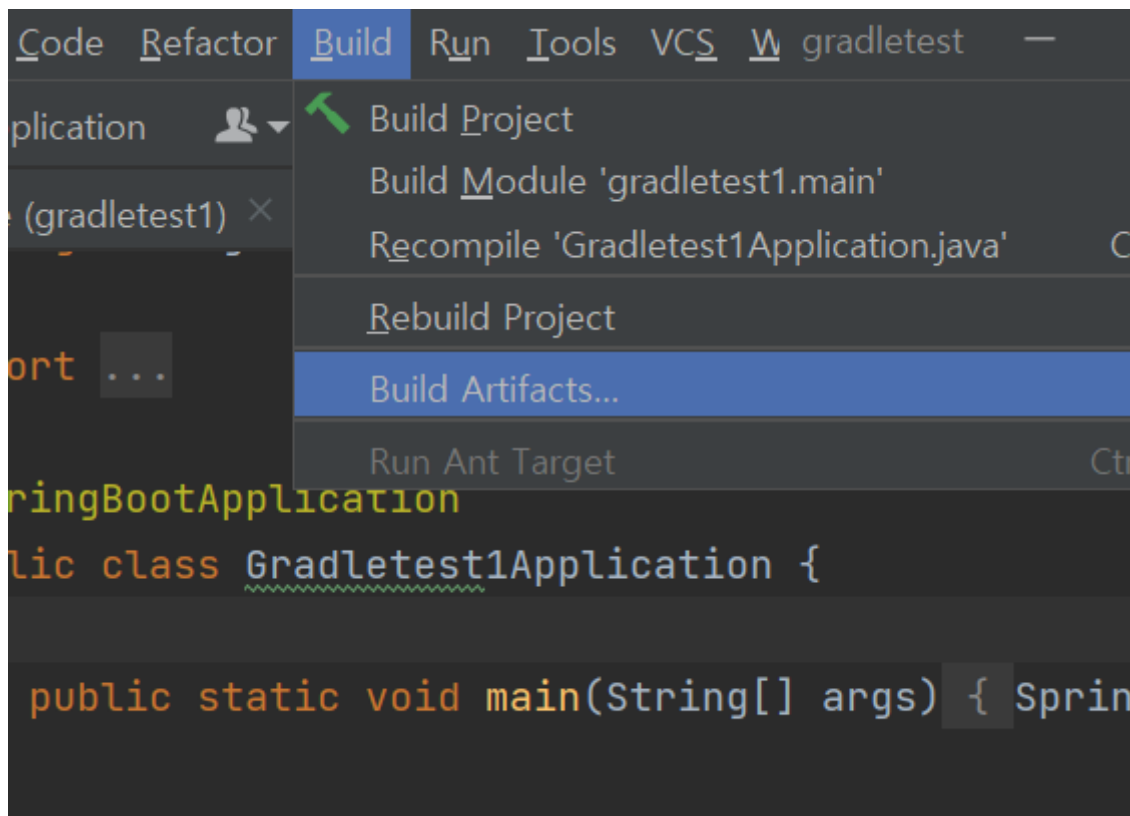
1. 플젝 세팅 들어가서 Artifact -> add -> jar 선택 ,



1. 사용할 모듈 선택하고 ok



1. 해당 모듈에 들어가서 빌드 Artifact

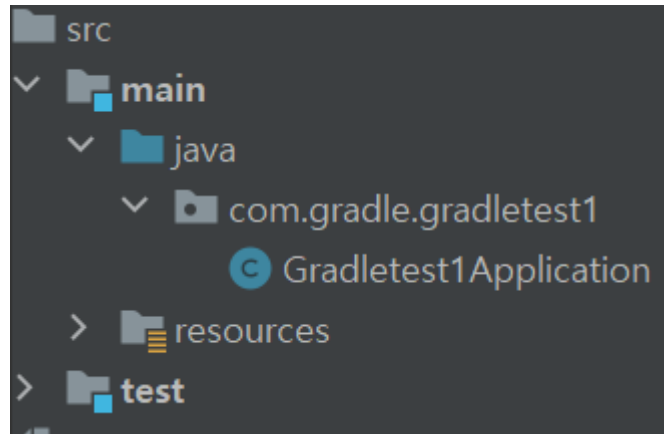


1. 아래와 같이 잘 빌드 되었다 뜨면 jar 파일 잘 생성된 것

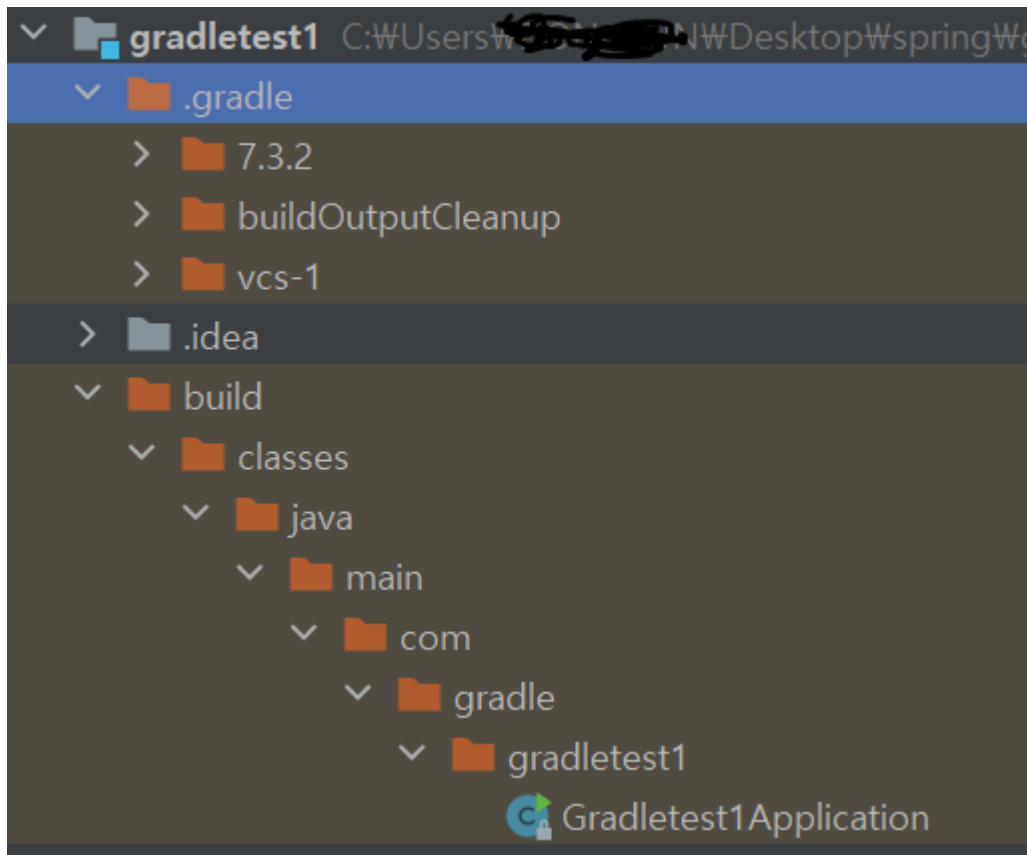
```
Build Output X
405 ms Copying archives...
Running 'after' tasks
javac 11.0.14 was used to compile java sources
Finished, saving caches...
Executing post-compile tasks...
Loading Ant configuration...
Running Ant tasks...
Synchronizing output directories...
2022-01-20 오후 10:40 - Build completed successfully in 4 sec, 405 ms
```

## 1. jar 파일 확인

- 프로젝트 구조



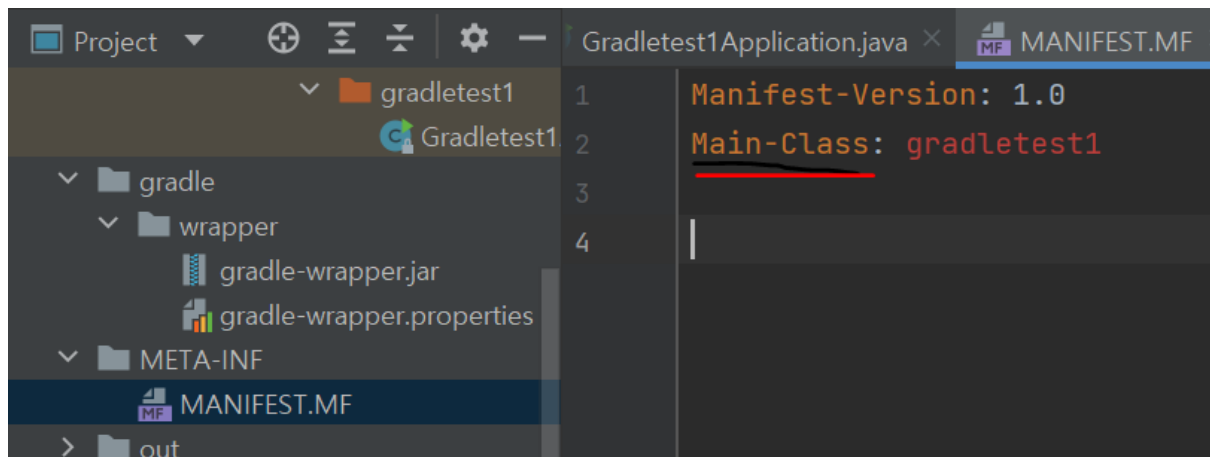
- JAR 파일 구조



=> jar 은 빌드된 결과물

=> jre 가 있는 환경에서 실행할 수 있도록 해주는 것

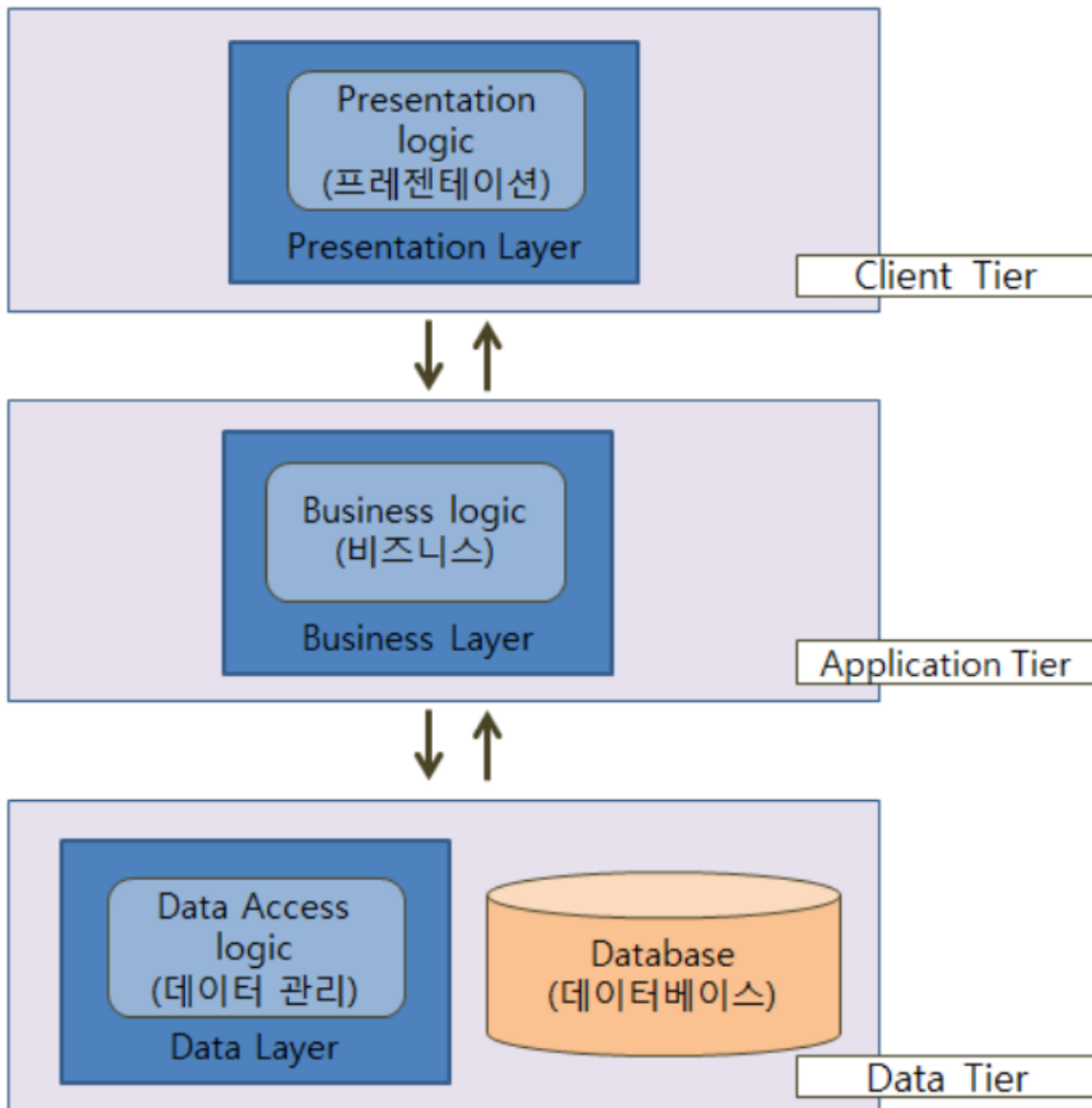
- MANIFEST.MF 자세히 살펴보기



- 위의 MAIN 클래스가 JAR 의 원천이라고 할 수 있음
- 만약 우리가 배포를 하게 되면 이 jar 파일을 서버로 가져다가 / 혹은 이미지로 가져다가 작업을 한다



## 웹 어플리케이션 구조



### 이미지 출처 블로그

1. presentation 레이어 : 사용자와 직접적으로 맞닿는 부분
2. logic 레이어 : 요청을 처리하는 결정 짓는 부분
3. data 레이어 : 데이터의 표현을 저장, 불러오는 부분 (ㄹㅇ 실제 데이터는 아니라는 말)

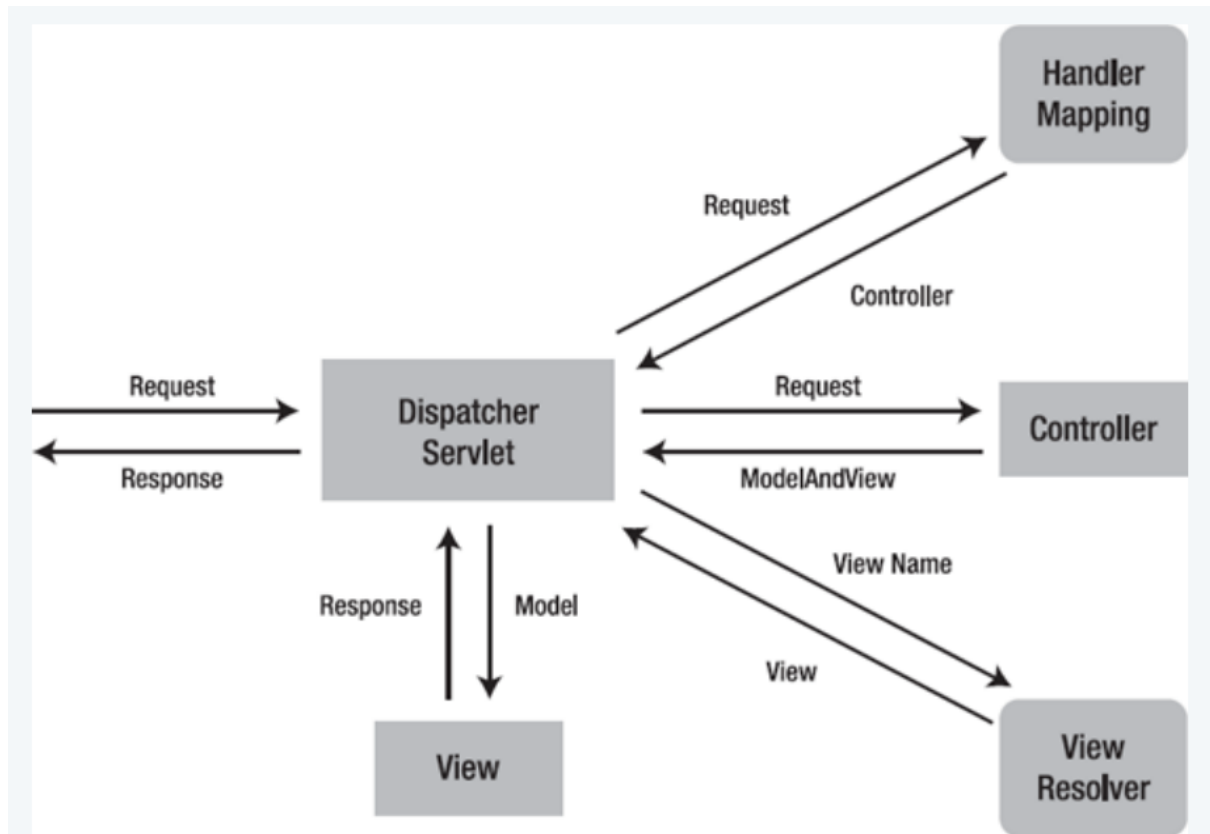
## Spring Boot 프로젝트 구조

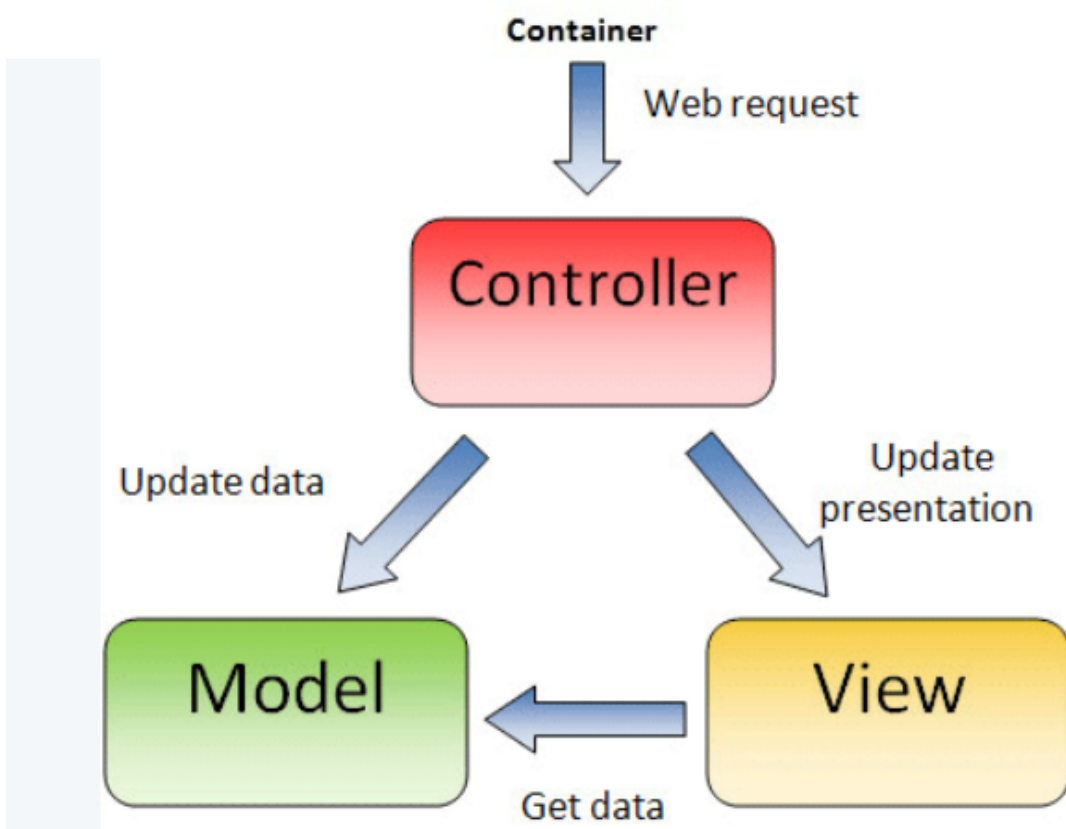
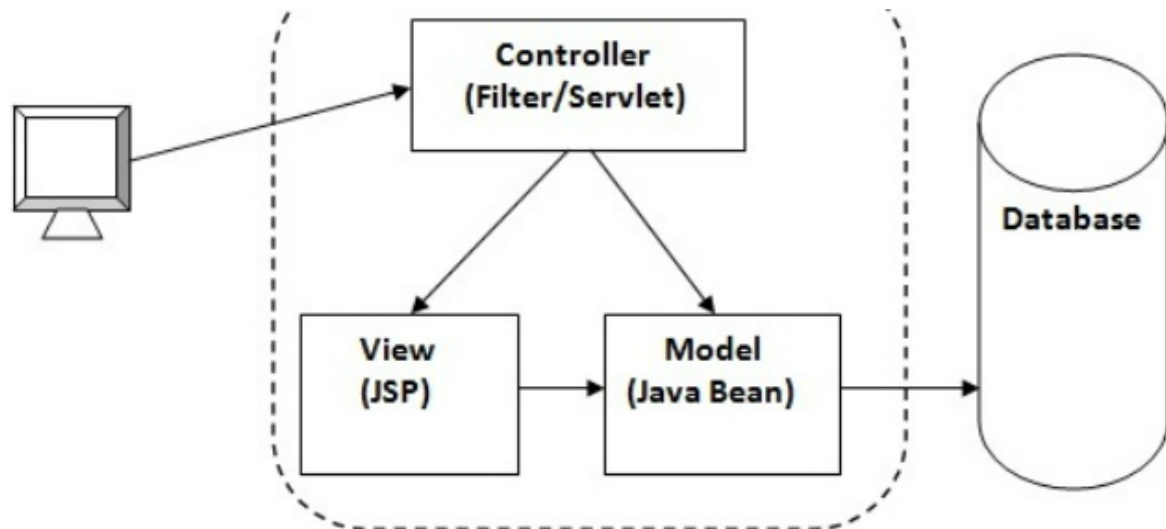
1. controller - 디스패처 서버로부터 직접적으로 요청 받음 & 해당 요청을 검증
2. service - 컨트롤러가 검증한 데이터 받아서 사용자의 입력에 따른 데이터 조작을 수행

3. 레포지토리 - 데이터를 받아서 데이터를 저장하고, db로부터 데이터 불러와서 돌려주는 역할 수행 (db는 스프링 부트 외부에 존재!)

(+)

아래 기술한 스프링 구조 내용, 이미지 출처 블로그 주소





controller가 모든 요청 받는다

1. 주소(URL)에 해당하는 모델 찾아서 기능 수행
2. 해당하는 데이터를 보여줄 View를 찾는다.
3. (Controller 파일 들어가면 다 명시되어있음)

#### Model

- DB 작업

- 일반적으로 POJO로 구성된다.
- Java Beans

## View

- 사용자가 보는 페이지
- Model data의 렌더링(컨텐츠를 가져와서 구성)을 담당하며, HTML output을 생성한다.
- JSP

## Controller

- View와 Model 사이의 인터페이스(컨트롤) 역할을 담당한다
- 사용자 요청을 수신하여 그에 따라 적절한 결과를 Model에 담아 View에 전달한다.
- 즉, Model Object와 이 Model을 화면에 출력할 View Name을 반환한다.
- Controller → Service → Dao → DB
- Servlet

## DispatcherServlet

Spring Framework가 제공하는 Servlet 클래스

사용자의 요청을 받는다.

Dispatcher가 받은 요청은 HandlerMapping으로 넘어간다.

## HandlerMapping

사용자의 요청을 처리할 Controller를 찾는다. (Controller URL Mapping)

요청 url에 해당하는 Controller 정보를 저장하는 table을 가진다.

즉, 클래스에 @RequestMapping("/url") annotation을 명시하면 해당 URL에 대한 요청이 들어왔을 때 table에 저장된 정보에 따라 해당 클래스 또는 메서드에 Mapping한다.

## ViewResolver

Controller가 반환한 View Name(the logical names)에 prefix, suffix를 적용하여 View Object(the physical view files)를 반환한다.

예를 들어

view name: home,

prefix: /WEB-INF/views/,

suffix: .jsp는 "/WEB-INF/views/home.jsp"라는 위치의 View(JSP)에 Controller에게 받은 Model을 전달한다.

이 후에 해당 View에서 이 Model data를 이용하여 적절한 페이지를 만들어 사용자에게 보여준다.

## 장점

Navigation control is centralized - 컨트롤러가 페이지들을 결정하는 모든 로직을 가지고 있다.

유지보수가 쉽다

확장성이 좋다

테스트 하기 용이하다

## 단점

컨트롤러 코드를 직접 써야 한다. 컨트롤러가 수정되면 다시 컴파일해서 다시 deploy 해야 한다.

## final 연결 흐름

하드웨어 서버 - nginx (webserver) - Spring 부트 (톰캣이라는 웹 애플리케이션 서버 내장)

- "스프링 부트 내부: 디스패처서버가 request 받고

1. 컨트롤러로 넘기고 서비스로 전달되어서 레포지토리에서 db에서 적절한 정보 꺼내오거나 데이터 저장하고 이를 다시 서비스 -> 컨트롤러에게 response 로 돌려주기
2. 뷰 resolver에게 전달해서 적절한 view를 response 로 돌려주기

## 1. Java Interface

(+) 이 부분에서 설명 어려워하는 분들 많을 것 같아서 인터페이스에 대한 개념 설명 필요할 듯

- interface

인터페이스 설명 참고 블로그

: 극단적으로 동일한 목적 하에 동일한 기능을 수행하게끔 강제하는 것

(ex) 소방차, 경찰차, 자가차 등 여러가지 자동차가 존재, 이때 자동차에 공통적인 기능이 존재할 것, 이 기능이 모든 종류의 자동차에 존재하도록 강제하고 싶을 때 interface에 자동차가 가져야할 공통적인 성질 정의, 추상메소드 제공을 통해서 동일기능 수행하게 하는 것

- 추상클래스, 인터페이스 잘 사용하는 것 중요
- 인터페이스를 잘 사용하면 서로 다른 구현체가 같은 목적을 위해 동작하도록 만들기 가능
- 사용하고자 하는 개체의 실제 자료형과 무관하게 동작하게 만들기 가능
- 함수의 인자와 반환 값은 인터페이스를 활용할 수 있도록 하자
- inputStream의 구현체가 많이 존재하는데, 이들은 다 inputStream의 기능을 가진다 => 즉 inputStream을 필요로 하는 기능엔 구분없이 사용 가능

## 2. IoC Container란

ioc 컨테이너 대한 설명 참고 출처 블로그

- ioc : inversion of control : 제어의 반전

기존에는 개발자가 프레임워크의 코드를 가져다가 썼지만, ioc에서는 프레임워크의 코드가 개발자의 코드를 가져다가 쓰는 것

=> 기존의 모든 제어를 클라이언트 코드가 가지도록 구현하던 것에서 기존 구조적 설계와 비교해, 프레임 워크가 제어를 나눠 가져가게 됨

=> 이로써 의존 관계의 방향이 달라지게 되는 것을 제어 반전이라고 함

- ioc container
  - => ioc를 구현하는 프레임워크로 객체를 관리하고 객체 생성 책임, 의존성 관리하는 컨테이너
  - => ioc를 담당하는 컨테이너
  - => 프레임워크를 기반으로 만든 애플리케이션이 작동을 할 때 개발자가 작성한 코드와 설정정보를 합쳐서 만들어져야 할 객체들을 직접 만들어주는 역할을 한다
  - => 이 객체가 이런 역할 할거라고 정의해두면 생성자를 통해서 객체를 만들어서 직접적으로 배치해줌
- 모든 의존성을 컨테이너 통해 받아오게 됨 -> 실수로 서로 다르게 구현 되는 경우 없음
- 즉 ioc 컨테이너로 관리를 하게 되면 필요한 객체를 외부에서 생성해서 ioc 컨테이너로 객체를 주입시키는 것

전통적인 프로그래밍에서 흐름은 프로그래머가 작성한 프로그램이 외부 라이브러리의 코드를 호출해 이용합니다.(프로그래머 코드 -> 외부 호출)

하지만 제어 반전이 적용된 구조에서는 외부 라이브러리의 코드가 프로그래머가 작성한 코드를 호출 (외부 코드 -> 프로그래머 코드 호출)

## 3. Spring Bean과 DI

- 스프링 ioc 컨테이너가 **개발자가 작성한 코드** 와 **설정정보** 를 합쳐서 만든 객체를 Bean이라고 부름
- Bean은 스프링 ioc 컨테이너가 관리하는 객체라고 할 수 있음
- 개발자가 비즈니스 로직 코드 작성

- 로직 코드 작성하면서 다른 객체를 사용해야 하는 시점에서 ioc 컨테이너에서 이미 만들어져있던 bean 객체를 전달해줌  
=> 개발자는 new로 선언돼있던 객체들 관리, 어떤 시점에서 만들어져있던 객체들을 신경써줄 필요가 없게 된다.
- Spring에서 구현을 요구하는 부분들을 인터페이스로 정의
- 이후 사용자가 정의한 구현체 Bean을 실제 서비스에서 사용하게 되는 것

ioc 컨테이너는 인터페이스 기반으로 작동을 하게 되는 것  
interface로 구현된 구현체들을 데리고 dependency injection을 진행하게 된다

## Dependency Injection (의존성 주입) 이란

ioc, di 아래 글 참고 블로그글

- **di** : 의존성 주입  
=> 하나의 객체 a가 다른 객체 b를 의존해야 할 때  
=> a의 코드 내부에서 b를 만들지 x  
=> 외부의 ioc 컨테이너에서 b를 만들고 생성자, setter 메소드 등의 방식을 활용해  
=> a의 내부로 주입시키는 것
- 위와 같은 di 진행방식을 통해서 모듈 간의 결합을 낮춰서 수정이 발생했을 때 코드 변경 지점을 최소화 할 수 있음

## Inversion of control 개념 추가

(출처는 위와 동일!)

=> 프레임워크가 개발자의 코드를 호출하면서 프로그램의 흐름을 주도하는 것

### 일반적인 흐름

객체 a 생성 -> 내부에서 의존성 객체 b 생성 -> 의존성 객체 b의 메소드를 호출

(ex) new를 통해 내부에서 객체 생성해서 사용하는 방식

```
class OwnerController {
    private OwnerRepository repository = new OwnerRepository();
}
```

=> 일반적인 흐름의 위 코드 설명 :

1. OwnerController 라는 a 가 OwnerRepository 라는 b 를 필요로 함
2. a의 내부에 new 를 활용해서 b라는 객체를 생성

**ioc 적용 흐름 :**

- 스프링에서는 모든 의존성 객체(Bean)을 스프링 컨테이너에서 직접 관리
- 이 Bean을 필요로 하는 곳에 주입해줌

객체 a,b 생성 -> 의존성 객체 b를 필요로 하는 곳에 b 주입 -> 이후 의존성 객체 b의 메소드를 호출

```
class OwnerController {  
    private final OwnerRepository repository;  
  
    public OwnerController(OwnerRepository repository) {  
        this.repository = repository;  
    }  
}
```

=> ioc 흐름의 뒷 코드 설명 :

1. OwnerController 이라는 a가 b를 필요로 할 때 생성자를 이용해서 외부에서 b를 호출해서 a 에 주입하는 중

## 4. 코드로 ioc 유무의 차이 이해하기

### 1. 일반적 interface 활용 코드(ioc x)

**Ainterface.java** (인터페이스)

```
package hello.hellospring;  
  
public interface AInterface {  
    void sayHello();  
}
```

**Acomponent.java** (클래스)

```
package hello.hellospring;
```



```
public class Acomponent {
    private AInterface A;
    public Acomponent(AInterface A){
        this.A = A;
        //this는 현재 생성자 그 자체 가리키는 것
    }
    public void sayHello(){
        this.A.sayHello();
    }
}
```

## static main 이 있는 곳

```
package hello.hellospring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloSpringApplication {

    public static void main(String[] args) {
        Acomponent acomp = new Acomponent(
            new AInterface(){
                @Override
                public void sayHello() {
                    System.out.println("this is temporary interface");
                }
            }
        );
        acomp.sayHello();
        SpringApplication.run(HelloSpringApplication.class, args);
    }
}
```

- > 결과 :

```
this is temporary interface
```

=> 현재는 new 를 활용해서 필요한 인터페이스를 내부에 생성함

## ioc 활용 시

- Acomponent 클래스 위에 `@Component` 붙이게 되면 스프링 ioc 에서 관리하는 객체로 변하게 됨

- 아래와 같이 @Component 붙이고 위의 코드 실행한다면

```
package hello.hellospring;

import org.springframework.stereotype.Component;

@Component
public class Acomponent {
    private AInterface A;
    public Acomponent(AInterface A){
        this.A = A;
        //this는 현재 생성자 그 자체 가리키는 것
    }
}
```

Parameter 0 of constructor in hello.hellospring.Acomponent required a bean of type 'hello.hellospring.AInterface' that could not be found.

라는 에러가 등장함 이유는 아래와 같은 것이다.

1. 이는 현재 Acomponent 가 @Component라는 annotation 달리게 되면서 ioc의 관리를 받게 되었다
2. Acomponent는 Ainterface를 필요로 하는 객체다.
3. ioc의 관리에 들어가게 되면 a,b의 bean 객체가 미리 생성해놓고 (ioc가 애네를 관리) -  
> a 가 b를 필요로 할 때 ioc 컨테이너는 bean객체에서 b를 데려다가 a에 주입시켜줌 (dependency injection) -> 이렇게 주입된 b를 a는 잘 사용함  
의 단계로 활용이 되어야 한다.

즉 ioc는 bean 형태의 객체로 관리를 해야하는데 현재 Ainterface의 bean 객체가 존재하지 않는다는 것 (Ainterface 는 그 자체만 있고 애를 implements한 다른 구현체들은 존재하지 않는 상태)

따라서 우리는 ioc를 고려해서 Ainterface를 implements 할 아이(bean)를 만들어주어야 함  
- bean 형태로 만들어주려면 @Component 붙여주면 된다.

따라서 Ainterface를 bean 객체로 하나 생성해주자

```
package hello.hellospring;
import org.springframework.stereotype.Component;

@Component
public class AinterfaceImplements implements AInterface{
    @Override
    public void sayHello() {
        System.out.println("방가방가 I am Bean!");
    }
}
```

스프링 부트를 다루는 것은 Bean을 얼마나 적절히 잘 사용하는지에 관한 것

Spring에서 구현을 요구하는 부분들을 interface를 통해서 정의, 이후 사용자가 정의한 구현체 Bean을 실제 서비스에서 사용!

## 5. Spring과 Spring Boot의 차이

Bean <= 개발자의 코드 + 설정

### 1) Spring Framework

- 스프링 프레임워크는 설정을 xml 파일로 만들어 주었다.  
(ex) 보안 설정 시 어떤 파일을 보안처리해야하는지 xml 로..  
=> xml 작성, 다루는 일이 까다로워서 입문하는 것 어려움
- JAVA Web Application (WAR 파일), 스스로 실행 불가  
=> 실행을 위해 톰캣과 같은 웹 애플리케이션 서버 필요
- 실행 시

Web Application Server -> 스프링(WAR파일들)

### 2) Spring Boot

- 설정이 Spring Boot Starter에 정의되어 있음
- 톰캣과 같은 웹어플리케이션 서버 프로그램이 내장 , JAR의 형태로 실행 가능  
=> JAR은 자바 명령어로 바로바로 실행 가능

- 실행 시

## | 스프링부트(JAR파일)

- Build a server that runs