3. 스프링 부트와 내장 톰캣

#1.인강/9. 스프링부트/강의#

WAR 배포 방식의 단점

웹 애플리케이션을 개발하고 배포하려면 다음과 같은 과정을 거쳐야 한다.

- 톰캣 같은 웹 애플리케이션 서버(WAS)를 별도로 설치해야 한다.
- 애플리케이션 코드를 WAR로 빌드해야 한다.
- 빌드한 WAR 파일을 WAS에 배포해야 한다.

웹 애플리케이션을 구동하고 싶으면 웹 애플리케이션 서버를 별도로 설치해야 하는 구조이다.
과거에는 이렇게 웹 애플리케이션 서버와 웹 애플리케이션 빌드 파일(WAR)이 분리되어 있는것이 당연한 구조였다.
그런데 이런 방식은 다음과 같은 단점이 있다.

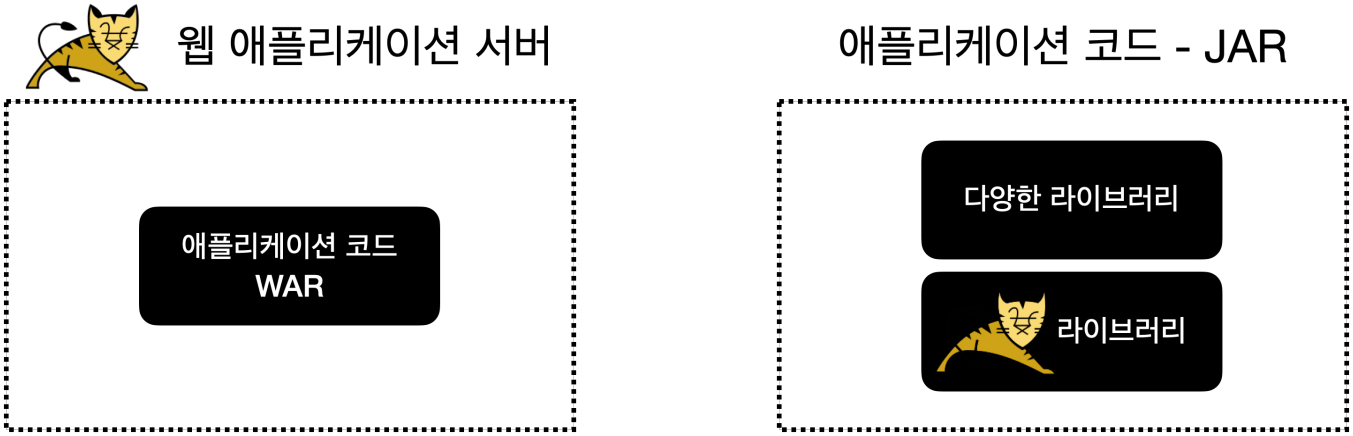
단점

- 톰캣 같은 WAS를 별도로 설치해야 한다.
- 개발 환경 설정이 복잡하다.
  - 단순한 자바라면 별도의 설정을 고민하지 않고, `main()` 메서드만 실행하면 된다.
  - 웹 애플리케이션은 WAS 실행하고 또 WAR와 연동하기 위한 복잡한 설정이 들어간다.
- 배포 과정이 복잡하다. WAR를 만들고 이것을 또 WAS에 전달해서 배포해야 한다.
- 톰캣의 버전을 변경하려면 톰캣을 다시 설치해야 한다.

고민

누군가는 오래전부터 이런 방식의 불편함을 고민해왔다. 단순히 자바의 `main()` 메서드만 실행하면 웹 서버까지 같이 실행되도록 하면 되지 않을까? 톰캣도 자바로 만들어져 있으니 톰캣을 마치 하나의 라이브러리 처럼 포함해서 사용해도 되지 않을까? 쉽게 이야기해서 톰캣 같은 웹서버를 라이브러리로 내장해버리는 것이다.
이런 문제를 해결하기 위해 톰캣을 라이브러리로 제공하는 내장 톰캣(embed tomcat) 기능을 제공한다.

외장 서버 VS 내장 서버



- 왼쪽 그림은 웹 애플리케이션 서버에 WAR 파일을 배포하는 방식, WAS를 실행해서 동작한다.
- 오른쪽 그림은 애플리케이션 JAR 안에 다양한 라이브러리들과 WAS 라이브러리가 포함되는 방식, `main()` 메서드를 실행해서 동작한다.

내장 톰캣1 - 설정

프로젝트 설정 순서

1. `embed-start`의 폴더 이름을 `embed`로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'java'
}
```

```
group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    //스프링 MVC 추가
    implementation 'org.springframework:spring-webmvc:6.0.4'

    //내장 톰캣 추가
    implementation 'org.apache.tomcat.embed:tomcat-embed-core:10.1.5'
}

tasks.named('test') {
    useJUnitPlatform()
}

//일반 Jar 생성
task buildJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'hello.embed.EmbedTomcatSpringMain'
    }
    with jar
}

//Fat Jar 생성
task buildFatJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'hello.embed.EmbedTomcatSpringMain'
    }
    duplicatesStrategy = DuplicatesStrategy.WARN
    from { configurations.runtimeClasspath.collect { it.isDirectory() ? it :
zipTree(it) } }
    with jar
}
```

- `tomcat-embed-core` : 톰캣 라이브러리이다. 톰캣을 라이브러리로 포함해서 톰캣 서버를 자바 코드로 실행할 수 있다.
    - 서블릿 관련 코드도 포함하고 있다.
- `buildJar` , `buildFatJar` 관련된 부분은 뒤에서 다시 설명한다.

앞서 개발했던 서블릿과 스프링 컨트롤러 코드는 그대로 유지해두었다.

- `hello.servlet.HelloServlet`
- `hello.spring.HelloController`
- `hello.spring.HelloConfig`

내장 톰캣2 - 서블릿

이제 본격적으로 내장 톰캣을 사용해보자. 내장 톰캣은 쉽게 이야기해서 톰캣을 라이브러리로 포함하고 자바 코드로 직접 실행하는 것이다.

EmbedTomcatServletMain

```
package hello.embed;

import hello.servlet.HelloServlet;
import org.apache.catalina.Context;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;

public class EmbedTomcatServletMain {

    public static void main(String[] args) throws LifecycleException {
        System.out.println("EmbedTomcatServletMain.main");
        //톰캣 설정
        Tomcat tomcat = new Tomcat();
        Connector connector = new Connector();
        connector.setPort(8080);
        tomcat.setConnector(connector);

        //서블릿 등록
        Context context = tomcat.addContext("", "/");
        tomcat.addServlet("", "helloServlet", new HelloServlet());
        context.addServletMappingDecoded("/hello-servlet", "helloServlet");
        tomcat.start();
    }
}
```

- 톰캣 설정
  - 내장 톰캣을 생성하고, 톰캣이 제공하는 커넥터를 사용해서 `8080` 포트에 연결한다.
- 서블릿 등록
  - 톰캣에 사용할 `contextPath`와 `docBase`를 지정해야 한다. 이 부분은 크게 중요하지 않으므로 위 코드와 같이 적용하자.
  - `tomcat.addServlet()`을 통해서 서블릿을 등록한다.
  - `context.addServletMappingDecoded()`을 통해서 등록한 서블릿의 경로를 매핑한다.
- 톰캣 시작
  - `tomcat.start()` 코드를 사용해서 톰캣을 시작한다.

등록한 `HelloServlet` 서블릿이 정상 동작하는지 확인해보자.

실행

`EmbedTomcatServletMain.main()` 메서드를 실행하자.

- http://localhost:8080/hello-servlet

결과

```
hello servlet!
```

내장 톰캣을 사용한 덕분에 IDE에 별도의 복잡한 톰캣 설정 없이 `main()` 메서드만 실행하면 톰캣까지 매우 편리하게 실행되었다. 물론 톰캣 서버를 설치하지 않아도 된다!

> **참고**
>
> 내장 톰캣을 개발자가 직접 다룰일은 거의 없다. 스프링 부트에서 내장 톰캣 관련된 부분을 거의 대부분 자동화해서 제공하기 때문에 내장 톰캣을 깊이있게 학습하는 것은 권장하지 않는다.(백엔드 개발자는 이미 공부해야 할 것이 너무 많다.)
>
> 내장 톰캣이 어떤 방식으로 동작하는지 그 원리를 대략 이해하는 정도면 충분하다.

주의

실행시 특정 환경에서 다음과 같은 오류가 발생하는 경우가 있다.

```
Caused by: java.lang.IllegalArgumentException: The main resource set specified
[...\tomcat\tomcat.8080\webapps] is not valid
```

이런 경우에는 다음을 참고해서 코드를 추가해주면 된다.

```
...

//서블릿 등록
Context context = tomcat.addContext("", "/");
```

```
//== 코드 추가 시작==
File docBaseFile = new File(context.getDocBase());
if (!docBaseFile.isAbsolute()) {
    docBaseFile = new File(((org.apache.catalina.Host)
context.getParent()).getAppBaseFile(), docBaseFile.getPath());
}
docBaseFile.mkdirs();
//== 코드 추가 종료==
tomcat.addServlet("", "helloServlet", new HelloServlet());
context.addServletMappingDecoded("/hello-servlet", "helloServlet");
tomcat.start();
```

내장 톰캣3 - 스프링

이번에는 내장 톰캣에 스프링까지 연동해보자.

EmbedTomcatSpringMain

```
package hello.embed;

import hello.spring.HelloConfig;
import org.apache.catalina.Context;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;
import
org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class EmbedTomcatSpringMain {

    public static void main(String[] args) throws LifecycleException {
        System.out.println("EmbedTomcatSpringMain.main");
        //톰캣 설정
        Tomcat tomcat = new Tomcat();
        Connector connector = new Connector();
        connector.setPort(8080);
        tomcat.setConnector(connector);

        //스프링 컨테이너 생성
        AnnotationConfigWebApplicationContext appContext = new
```

```
AnnotationConfigWebApplicationContext();
        appContext.register(HelloConfig.class);

        //스프링 MVC 디스패처 서블릿 생성, 스프링 컨테이너 연결
        DispatcherServlet dispatcher = new DispatcherServlet(appContext);

        //디스패처 서블릿 등록
        Context context = tomcat.addContext("", "/");
        tomcat.addServlet("", "dispatcher", dispatcher);
        context.addServletMappingDecoded("/", "dispatcher");

        tomcat.start();
    }
}
```

- 스프링 컨테이너를 생성하고, 내장 톰캣에 디스패처 서블릿을 등록했다.
- 이미 앞서 설명한 내용과 같아서, 이해하는데 어려움은 없을 것이다.

실행

- `EmbedTomcatSpringMain.main()` 메서드를 실행하자.
- http://localhost:8080/hello-spring

결과

```
hello spring!
```

`main()` 메서드를 실행하면 다음과 같이 동작한다.

- 내장 톰캣을 생성해서 `8080` 포트로 연결하도록 설정한다.
- 스프링 컨테이너를 만들고 필요한 빈을 등록한다.
- 스프링 MVC 디스패처 서블릿을 만들고 앞서 만든 스프링 컨테이너에 연결한다.
- 디스패처 서블릿을 내장 톰캣에 등록한다.
- 내장 톰캣을 실행한다.

코드를 보면 알겠지만, 서블릿 컨테이너 초기화와 거의 같은 코드이다.
다만 시작점이 개발자가 `main()` 메서드를 직접 실행하는가, 서블릿 컨테이너가 제공하는 초기화 메서드를 통해서 실행하는가의 차이가 있을 뿐이다.

내장 톰캣4 - 빌드와 배포1

이번에는 애플리케이션에 내장 톰캣을 라이브러리로 포함했다. 이 코드를 어떻게 빌드하고 배포하는지 알아보자.
자바의 `main()` 메서드를 실행하기 위해서는 `jar` 형식으로 빌드해야 한다.

그리고 `jar` 안에는 `META-INF/MANIFEST.MF` 파일에 실행할 `main()` 메서드의 클래스를 지정해주어야 한다.

`META-INF/MANIFEST.MF`

```
Manifest-Version: 1.0
Main-Class: hello.embed.EmbedTomcatSpringMain
```

Gradle의 도움을 받으면 이 과정을 쉽게 진행할 수 있다. 다음 코드를 참고하자.

`build.gradle - buildJar` 참고

```
task buildJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'hello.embed.EmbedTomcatSpringMain'
    }
    with jar
}
```

해당 코드는 `embed-start`에서 포함해두었다.

다음과 같이 실행하자.

jar 빌드

`./gradlew clean buildJar`

[윈도우 OS]

`gradlew clean buildJar`

다음 위치에 `jar` 파일이 만들어졌을 것이다.

`build/libs/embed-0.0.1-SNAPSHOT.jar`

jar 파일 실행

`jar` 파일이 있는 폴더로 이동한 후에 다음 명령어로 `jar` 파일을 실행해보자.

`java -jar embed-0.0.1-SNAPSHOT.jar`

실행 결과

```
... % java -jar embed-0.0.1-SNAPSHOT.jar
Error: Unable to initialize main class hello.embed.EmbedTomcatSpringMain
Caused by: java.lang.NoClassDefFoundError: org/springframework/web/context/
WebApplicationContext
```

실행 결과를 보면 기대했던 내장 톰캣 서버가 실행되는 것이 아니라, 오류가 발생하는 것을 확인할 수 있다.

오류 메시지를 잘 읽어보면 스프링 관련 클래스를 찾을 수 없다는 오류이다.

무엇이 문제일까?

문제를 확인하기 위해 `jar` 파일의 압축을 풀어보자.

jar 압축 풀기

- 우리가 빌드한 jar 파일의 압축을 풀어서 내용물을 확인해보자.
- `build/libs` 폴더로 이동하자.
- 다음 명령어를 사용해서 압축을 풀자
  - `jar -xvf embed-0.0.1-SNAPSHOT.jar`

JAR를 푼 결과

- `META-INF`
  - `MANIFEST.MF`
- `hello`
  - `servlet`
    - `HelloServlet.class`
  - `embed`
    - `EmbedTomcatSpringMain.class`
    - `EmbedTomcatServletMain.class`
  - `spring`
    - `HelloConfig.class`
    - `HelloController.class`

JAR를 푼 결과를 보면 스프링 라이브러리나 내장 톰캣 라이브러리가 전혀 보이지 않는다. 따라서 해당 오류가 발생한 것이다.

과거에 WAR 파일을 풀어본 기억을 떠올려보자.

WAR를 푼 결과

- `WEB-INF`
  - `classes`
    - `hello/servlet/TestServlet.class`
  - `lib`

        - `jakarta.servlet-api-6.0.0.jar`
    - `index.html`

WAR는 분명 내부에 라이브러리 역할을 하는 `jar` 파일을 포함하고 있었다.

jar 파일은 jar파일을 포함할 수 없다.

- WAR와 다르게 JAR 파일은 내부에 라이브러리 역할을 하는 JAR 파일을 포함할 수 없다. 포함한다고 해도 인식이 안된다. 이것이 JAR 파일 스펙의 한계이다. 그렇다고 WAR를 사용할 수 도 없다. WAR는 웹 애플리케이션 서버(WAS) 위에서만 실행할 수 있다.
- 대안으로는 라이브러리 jar 파일을 모두 구해서 MANIFEST 파일에 해당 경로를 적어주면 인식이 되지만 매우 번거롭고, Jar 파일안에 Jar 파일을 포함할 수 없기 때문에 라이브러리 역할을 하는 jar 파일도 항상 함께 가지고 다녀야 한다. 이 방법은 권장하기 않기 때문에 따로 설명하지 않는다.

내장 톰캣5 - 빌드와 배포2

FatJar

대안으로는 `fat jar` 또는 `uber jar` 라고 불리는 방법이다.
Jar 안에는 Jar를 포함할 수 없다. 하지만 클래스는 얼마든지 포함할 수 있다.
라이브러리에 사용되는 `jar` 를 풀면 `class` 들이 나온다. 이 `class` 를 뽑아서 새로 만드는 `jar` 에 포함하는 것이다.
이렇게 하면 수 많은 라이브러리에서 나오는 `class` 때문에 뚱뚱한(fat) `jar` 가 탄생한다. 그래서 `Fat Jar` 라고 부르는 것이다.

`build.gradle - buildFatJar` 참고

```
task buildFatJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'hello.embed.EmbedTomcatSpringMain'
    }
    duplicatesStrategy = DuplicatesStrategy.WARN
    from { configurations.runtimeClasspath.collect { it.isDirectory() ? it :
zipTree(it) } }
    with jar
}
```

해당 코드는 `embed-start` 에서 포함해두었다.

jar 빌드

`./gradlew clean buildFatJar`

[윈도우OS]

`gradlew clean buildFatJar`

빌드시 `Encountered duplicate path` 경고가 나올 수 있는데 이 부분은 무시하자

다음 위치에 `jar` 파일이 만들어졌을 것이다.
`build/libs/embed-0.0.1-SNAPSHOT.jar`
용량을 확인해보면 `10M` 이상의 상당히 큰 사이즈가 나왔다.

jar 파일 실행

`jar` 파일이 있는 폴더로 이동한 후에 다음 명령어로 `jar` 파일을 실행해보자.
`java -jar embed-0.0.1-SNAPSHOT.jar`

실행 결과

```
... % java -jar embed-0.0.1-SNAPSHOT.jar
EmbedTomcatSpringMain.main
...
INFO: Starting Servlet engine: [Apache Tomcat/9.0.65]
...
INFO: Starting ProtocolHandler ["http-nio-8080"]
```

실행

- http://localhost:8080/hello-spring

결과

```
hello spring!
```

드디어 정상 동작하는 것을 확인할 수 있다.

Fat Jar의 정체를 확인하기 위해 `jar` 파일의 압축을 풀어보자.

jar 압축 풀기

- 우리가 빌드한 jar 파일의 압축을 풀어서 내용물을 확인해보자.
- `build/libs` 폴더로 이동하자.
- 다음 명령어를 사용해서 압축을 풀자
  - `jar -xvf embed-0.0.1-SNAPSHOT.jar`

Jar를 풀어보면 우리가 만든 클래스를 포함해서, 수 많은 라이브러리에서 제공되는 클래스들이 포함되어 있는 것을 확인할 수 있다.

Fat Jar 정리

Fat Jar의 장점

- Fat Jar 덕분에 하나의 jar 파일에 필요한 라이브러리들을 내장할 수 있게 되었다.
- 내장 톰캣 라이브러리를 jar 내부에 내장할 수 있게 되었다.
- 덕분에 하나의 jar 파일로 배포부터, 웹 서버 설치+실행까지 모든 것을 단순화 할 수 있다.

이전에 살펴보았던 WAR를 외부 서버에 배포하는 방식의 단점을 다시 확인해보자.

WAR 단점과 해결

- 톰캣 같은 WAS를 별도로 설치해야 한다.
  - 해결: WAS를 별도로 설치하지 않아도 된다. 톰캣 같은 WAS가 라이브러리로 jar 내부에 포함되어 있다.
- 개발 환경 설정이 복잡하다.
  - 단순한 자바라면 별도의 설정을 고민하지 않고, `main()` 메서드만 실행하면 된다.
  - 웹 애플리케이션은 WAS를 연동하기 위한 복잡한 설정이 들어간다.
  - 해결: IDE에 복잡한 WAS 설정이 필요하지 않다. 단순히 `main()` 메서드만 실행하면 된다.
- 배포 과정이 복잡하다. WAR를 만들고 이것을 또 WAS에 전달해서 배포해야 한다.
  - 해결: 배포 과정이 단순하다. JAR를 만들고 이것을 원하는 위치에서 실행만 하면 된다.
- 톰캣의 버전을 업데이트 하려면 톰캣을 다시 설치해야 한다.
  - 해결: gradle에서 내장 톰캣 라이브러리 버전만 변경하고 빌드 후 실행하면 된다.

Fat Jar의 단점

Fat Jar는 완벽해 보이지만 몇가지 단점을 여전히 포함하고 있다.

- 어떤 라이브러리가 포함되어 있는지 확인하기 어렵다.
  - 모두 `class`로 풀려있으니 어떤 라이브러리가 사용되고 있는지 추적하기 어렵다.
- 파일명 중복을 해결할 수 없다.
  - 클래스나 리소스 명이 같은 경우 하나를 포기해야 한다. 이것은 심각한 문제를 발생한다. 예를 들어서

서블릿 컨테이너 초기화에서 학습한 부분을 떠올려 보자.

- `META-INF/services/jakarta.servlet.ServletContainerInitializer` 이 파일이 여러 라이브러리(`jar`)에 있을 수 있다.
- `A` 라이브러리와 `B` 라이브러리 둘다 해당 파일을 사용해서 서블릿 컨테이너 초기화를 시도한다. 둘다 해당 파일을 `jar` 안에 포함한다.
- `Fat Jar`를 만들면 파일명이 같으므로 `A`, `B` 라이브러리가 둘다 가지고 있는 파일 중에 하나의 파일만 선택된다. 결과적으로 나머지 하나는 포함되지 않으므로 정상 동작하지 않는다.

편리한 부트 클래스 만들기

지금까지 진행한 내장 톰캣 실행, 스프링 컨테이너 생성, 디스패처 서블릿 등록의 모든 과정을 편리하게 처리해주는 나만의 부트 클래스를 만들어보자. 부트는 이름 그대로 시작을 편하게 처리해주는 것을 뜻한다.

MySpringApplication

```
package hello.boot;

import org.apache.catalina.Context;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Tomcat;
import
org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import java.util.List;

public class MySpringApplication {

    public static void run(Class configClass, String[] args) {
        System.out.println("MySpringBootApplication.run args=" + List.of(args));
        //톰캣 설정
        Tomcat tomcat = new Tomcat();
        Connector connector = new Connector();
        connector.setPort(8080);
        tomcat.setConnector(connector);

        //스프링 컨테이너 생성
```

```
        AnnotationConfigWebApplicationContext appContext = new
AnnotationConfigWebApplicationContext();
        appContext.register(configClass);

        //스프링 MVC 디스패처 서블릿 생성, 스프링 컨테이너 연결
        DispatcherServlet dispatcher = new DispatcherServlet(appContext);

        //디스패처 서블릿 등록
        Context context = tomcat.addContext("", "/");
        tomcat.addServlet("", "dispatcher", dispatcher);
        context.addServletMappingDecoded("/", "dispatcher");
        try {
            tomcat.start();
        } catch (LifecycleException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- 기존 코드를 모아서 편리하게 사용할 수 있는 클래스를 만들었다. `MySpringApplication.run()`을 실행하면 바로 작동한다.
- `configClass`: 스프링 설정을 파라미터로 전달받는다.
- `args`: `main(args)`를 전달 받아서 사용한다. 참고로 예제에서는 단순히 해당 값을 출력한다.
- `tomcat.start()`에서 발생하는 예외는 잡아서 런타임 예외로 변경했다.

@MySpringBootApplication

```
package hello.boot;

import org.springframework.context.annotation.ComponentScan;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@ComponentScan
public @interface MySpringBootApplication {
}
```

- 컴포넌트 스캔 기능이 추가된 단순한 애노테이션이다.
- 시작할 때 이 애노테이션을 붙여서 사용하면 된다.

HelloConfig - 수정

```java
package hello.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

//@Configuration //주석 처리
public class HelloConfig {

    @Bean
    public HelloController helloController() {
        return new HelloController();
    }
}
```

- `@Configuration`을 주석처리했다.
- 여기서는 편리하게 컴포넌트 스캔을 사용할 예정이어서 `@Configuration`를 주석처리했다.

MySpringBootAppMain

```java
package hello;

import hello.boot.MySpringApplication;
import hello.boot.MySpringBootApplication;

@MySpringBootApplication
public class MySpringBootMain {

    public static void main(String[] args) {
        System.out.println("MySpringBootMain.main");
        MySpringApplication.run(MySpringBootMain.class, args);
    }

}
```

- 패키지 위치가 중요하다. `hello`에 위치했다.
- 여기에 위치한 이유는 `@MySpringBootApplication`에 컴포넌트 스캔이 추가되어 있는데, 컴포넌트 스캔의 기본 동작은 해당 애노테이션이 붙은 클래스의 현재 패키지 부터 그 하위 패키지를 컴포넌트 스캔의 대상으로 사용하기 때문이다 애노테이션이 붙은 `hello.MySpringBootMain` 클래스의 패키지 위치는 `hello`이므로 그 하위의 `hello.spring.HelloController`를 컴포넌트 스캔한다.
- `MySpringApplication.run(설정 정보, args)` 이렇게 한줄로 실행하면 된다.
- 이 기능을 사용하는 개발자는 `@MySpringBootApplication` 애노테이션과 `MySpringApplication.run()` 메서드만 기억하면 된다.
- 이렇게 하면 내장 톰캣 실행, 스프링 컨테이너 생성, 디스패처 서블릿, 컴포넌트 스캔까지 모든 기능이 한번

에 편리하게 동작한다.

스프링 부트

지금까지 만든 것을 라이브러리로 만들어서 배포한다면? ➜ 그것이 바로 스프링 부트이다.

일반적인 스프링 부트 사용법

```
@SpringBootApplication
public class BootApplication {

    public static void main(String[] args) {
        SpringApplication.run(BootApplication.class, args);
    }

}
```

스프링 부트는 보통 예제와 같이 `SpringApplication.run()` 한줄로 시작한다.

이제 본격적으로 스프링 부트를 사용해보자.

스프링 부트와 웹 서버 - 프로젝트 생성

스프링 부트는 지금까지 고민한 문제를 깔끔하게 해결해준다.

- 내장 톰캣을 사용해서 빌드와 배포를 편리하게 한다.
- 빌드시 하나의 Jar를 사용하면서, 동시에 Fat Jar 문제도 해결한다.
- 지금까지 진행한 내장 톰캣 서버를 실행하기 위한 복잡한 과정을 모두 자동으로 처리한다.

스프링 부트로 프로젝트를 만들어보고 스프링 부트가 어떤 식으로 편리한 기능을 제공하는지 하나씩 알아보자.

프로젝트 생성

프로젝트 설정 순서

1. `boot-start`의 폴더 이름을 `boot`로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

또는 스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트를 생성해도 된다.

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 3.0.x
- Project Metadata
  - Group: hello
  - Artifact: boot
  - Name: boot
  - Package name: hello.boot
  - Packaging: Jar
  - Java: 17

- Dependencies: **Spring Web**

build.gradle

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
```

```
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`BootApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

컨트롤러 하나를 간단하게 등록해서 잘 동작하는지 확인해보자.

HelloController

```
package hello.boot.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello-spring")
    public String hello() {
        System.out.println("HelloController.hello");
        return "hello spring!";
    }
}
```

실행

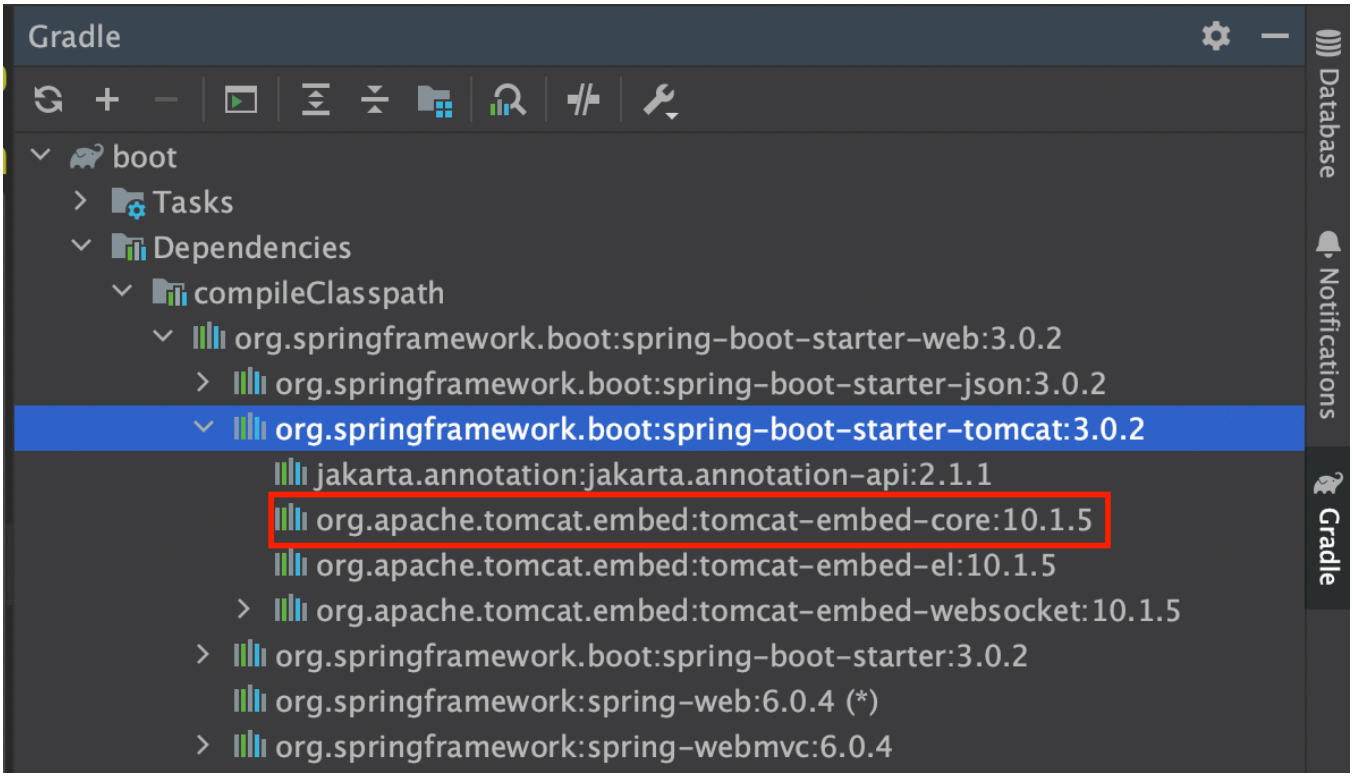
- `BootApplication.main()`
- http://localhost:8080/hello-spring

결과

```
http://localhost:8080/hello-spring
```

내장 톰캣 의존관계 확인

`spring-boot-starter-web`를 사용하면 내부에서 내장 톰캣을 사용한다.



- 라이브러리 의존관계를 따라가보면 내장 톰캣(`tomcat-embed-core`)이 포함된 것을 확인할 수 있다.

라이브러리 버전

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- 스프링 부트를 사용하면 라이브러리 뒤에 버전 정보가 없는 것을 확인할 수 있다.
- 스프링 부트는 현재 부트 버전에 가장 적절한 외부 라이브러리 버전을 자동으로 선택해준다. (이 부분에 대한 자세한 내용은 뒤에서 다룬다.)

스프링 부트와 웹 서버 - 실행 과정

스프링 부트의 실행 과정

```
@SpringBootApplication
public class BootApplication {

    public static void main(String[] args) {
        SpringApplication.run(BootApplication.class, args);
    }
```

```
}
```

- 스프링 부트를 실행할 때는 자바 `main()` 메서드에서 `SpringApplication.run()`을 호출해주면 된다.
- 여기에 메인 설정 정보를 넘겨주는데, 보통 `@SpringBootApplication` 애노테이션이 있는 현재 클래스를 지정해주면 된다.
- 참고로 현재 클래스에는 `@SpringBootApplication` 애노테이션이 있는데, 이 애노테이션 안에는 컴포넌트 스캔을 포함한 여러 기능이 설정되어 있다. 기본 설정은 현재 패키지와 그 하위 패키지 모두를 컴포넌트 스캔한다.

이 단순해 보이는 코드 한줄 안에서는 수 많은 일들이 발생하지만 핵심은 2가지다.

- 1. 스프링 컨테이너를 생성한다.
- 2. WAS(내장 톰캣)를 생성한다.

스프링 부트 내부에서 스프링 컨테이너를 생성하는 코드

`org.springframework.boot.web.servlet.context.ServletWebServerApplicationContextFactory`

```
class ServletWebServerApplicationContextFactory implements
ApplicationContextFactory {
  ...
  private ConfigurableApplicationContext createContext() {
    if (!AotDetector.useGeneratedArtifacts()) {
      return new AnnotationConfigServletWebServerApplicationContext();
    }
  return new ServletWebServerApplicationContext();
}
```

- `new AnnotationConfigServletWebServerApplicationContext()` 이 부분이 바로 스프링 부트가 생성하는 스프링 컨테이너이다.
- 이름 그대로 애노테이션 기반 설정이 가능하고, 서블릿 웹 서버를 지원하는 스프링 컨테이너이다.

스프링 부트 내부에서 내장 톰캣을 생성하는 코드

`org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory`

```
@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
  ...
    Tomcat tomcat = new Tomcat();
  ...
    Connector connector = new Connector(this.protocol);
```

```
    ...
        return getTomcatWebServer(tomcat);
    }
```

- `Tomcat tomcat = new Tomcat()`으로 내장 톰캣을 생성한다.

그리고 어디선가 내장 톰캣에 디스패처 서블릿을 등록하고, 스프링 컨테이너와 연결해서 동작할 수 있게 한다.
어디서 많이 본 것 같지 않은가?
스프링 부트도 우리가 앞서 내장 톰캣에서 진행했던 것과 동일한 방식으로 스프링 컨테이너를 만들고, 내장 톰캣을 생성하고 그 둘을 연결하는 과정을 진행한다.

> **참고**
> 스프링 부트는 너무 큰 라이브러리이기 때문에 스프링 부트를 이해하기 위해 모든 코드를 하나하나 파보는 것은 추천하지 않는다.
> 스프링 부트가 어떤 식으로 동작하는지 개념을 이해하고, 꼭 필요한 부분의 코드를 확인하자.

지금까지 스프링 부트가 어떻게 톰캣 서버를 내장해서 실행하는지 스프링 부트의 비밀 하나를 풀어보았다. 다음에는 스프링 부트의 빌드와 배포 그리고 스프링 부트가 제공하는 `jar`의 비밀을 알아보자.

스프링 부트와 웹 서버 - 빌드와 배포

내장 톰캣이 포함된 스프링 부트를 직접 빌드해보자.

jar 빌드
`./gradlew clean build`

[윈도우OS]
`gradlew clean build`

다음 위치에 `jar` 파일이 만들어진다.

- `build/libs/boot-0.0.1-SNAPSHOT.jar`

jar 파일 실행
`jar`파일이 있는 폴더로 이동한 후에 다음 명령어로 `jar` 파일을 실행해보자.

```
java -jar boot-0.0.1-SNAPSHOT.jar
```

실행 결과

```
... % java -jar boot-0.0.1-SNAPSHOT.jar
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080
(http) with context path ''
hello.boot.BootApplication               : Started BootApplication in 0.961
seconds
```

스프링 부트 애플리케이션이 실행되고, 내장 톰캣이 8080 포트로 실행된 것을 확인할 수 있다.

컨트롤러가 잘 호출되는지 확인해보자.

실행

- http://localhost:8080/hello-spring

결과

```
http://localhost:8080/hello-spring
```

스프링 부트 jar 분석

`boot-0.0.1-SNAPSHOT.jar` 파일 크기를 보면 대략 `18M` 정도 된다. 참고로 버전에 따라서 용량은 변할 수 있다.
아마도 앞서 배운 Fat Jar와 비슷한 방식으로 만들어져 있지 않을까? 생각될 것이다.
비밀을 확인하기 위해 `jar` 파일의 압축을 풀어보자.

jar 압축 풀기

- `build/libs` 폴더로 이동하자.
- 다음 명령어를 사용해서 압축을 풀자
  - `jar -xvf boot-0.0.1-SNAPSHOT.jar`

JAR를 푼 결과

- `boot-0.0.1-SNAPSHOT.jar`
  - `META-INF`
    - `MANIFEST.MF`
  - `org/springframework/boot/loader`
    - `JarLauncher.class`: 스프링 부트 `main()` 실행 클래스
  - `BOOT-INF`
    - `classes`: 우리가 개발한 class 파일과 리소스 파일
      - `hello/boot/BootApplication.class`

        - `hello/boot/controller/HelloController.class`
        - ...
    - `lib`: 외부 라이브러리
        - `spring-webmvc-6.0.4.jar`
        - `tomcat-embed-core-10.1.5.jar`
        - ...
    - `classpath.idx`: 외부 라이브러리 경로
    - `layers.idx`: 스프링 부트 구조 경로

JAR를 푼 결과를 보면 Fat Jar가 아니라 처음보는 새로운 구조로 만들어져 있다. 심지어 jar 내부에 jar를 담아서 인식하는 것이 불가능한데, jar가 포함되어 있고, 인식까지 되었다. 지금부터 스프링 부트가 제공하는 jar에 대해서 알아보자.

참고
빌드 결과를 보면 `boot-0.0.1-SNAPSHOT-plain.jar` 파일도 보이는데, 이것은 우리가 개발한 코드만 순수한 jar로 빌드한 것이다. 무시하면 된다.

스프링 부트 실행 가능 Jar

Fat Jar는 하나의 Jar 파일에 라이브러리의 클래스와 리소스를 모두 포함했다. 그래서 실행에 필요한 모든 내용을 하나의 JAR로 만들어서 배포하는 것이 가능했다. 하지만 Fat Jar는 다음과 같은 문제를 가지고 있다.

Fat Jar의 단점

- 어떤 라이브러리가 포함되어 있는지 확인하기 어렵다.
    - 모두 `class`로 풀려있으니 어떤 라이브러리가 사용되고 있는지 추적하기 어렵다.
- 파일명 중복을 해결할 수 없다.
    - 클래스나 리소스 명이 같은 경우 하나를 포기해야 한다. 이것은 심각한 문제를 발생한다. 예를 들어서 서블릿 컨테이너 초기화에서 학습한 부분을 떠올려 보자.
    - `META-INF/services/jakarta.servlet.ServletContainerInitializer` 이 파일이 여러 라이브러리(`jar`)에 있을 수 있다.
    - `A` 라이브러리와 `B` 라이브러리 둘다 해당 파일을 사용해서 서블릿 컨테이너 초기화를 시도한다. 둘다 해당 파일을 `jar` 안에 포함한다.
    - `Fat Jar`를 만들면 파일명이 같으므로 `A`, `B` 둘중 하나의 파일만 선택된다. 결과적으로 나머지는 정상 동작하지 않는다.

실행 가능 Jar

스프링 부트는 이런 문제를 해결하기 위해 jar 내부에 jar를 포함할 수 있는 특별한 구조의 jar를 만들고 동시에 만든 jar를 내부 jar를 포함해서 실행할 수 있게 했다. 이것을 실행 가능 Jar(Executable Jar)라 한다. 이 실행 가능 Jar를 사용하면 다음 문제들을 깔끔하게 해결할 수 있다.

- 문제: 어떤 라이브러리가 포함되어 있는지 확인하기 어렵다.
  - 해결: jar 내부에 jar를 포함하기 때문에 어떤 라이브러리가 포함되어 있는지 쉽게 확인할 수 있다.
- 문제: 파일명 중복을 해결할 수 없다.
  - 해결: jar 내부에 jar를 포함하기 때문에 `a.jar`, `b.jar` 내부에 같은 경로의 파일이 있어도 둘다 인식할 수 있다.

참고로 실행 가능 Jar는 자바 표준은 아니고, 스프링 부트에서 새롭게 정의한 것이다.

지금부터 실행 가능 Jar를 자세히 알아보자.

실행 가능 Jar 내부 구조

- `boot-0.0.1-SNAPSHOT.jar`
  - `META-INF`
    - `MANIFEST.MF`
  - `org/springframework/boot/loader`
    - `JarLauncher.class`: 스프링 부트 `main()` 실행 클래스
  - `BOOT-INF`
    - `classes`: 우리가 개발한 class 파일과 리소스 파일
      - `hello/boot/BootApplication.class`
      - `hello/boot/controller/HelloController.class`
      - ...
    - `lib`: 외부 라이브러리
      - `spring-webmvc-6.0.4.jar`
      - `tomcat-embed-core-10.1.5.jar`
      - ...
    - `classpath.idx`: 외부 라이브러리 모음
    - `layers.idx`: 스프링 부트 구조 정보

Jar 실행 정보

`java -jar xxx.jar`를 실행하게 되면 우선 `META-INF/MANIFEST.MF` 파일을 찾는다. 그리고 여기에 있는 `Main-Class`를 읽어서 `main()` 메서드를 실행하게 된다. 스프링 부트가 만든 `MANIFEST.MF`를 확인해보자.

`META-INF/MANIFEST.MF`

```
Manifest-Version: 1.0
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: hello.boot.BootApplication
Spring-Boot-Version: 3.0.2
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
Build-Jdk-Spec: 17
```

- `Main-Class`
  - 우리가 기대한 `main()`이 있는 `hello.boot.BootApplication`이 아니라 `JarLauncher`라는 전혀 다른 클래스를 실행하고 있다.
  - `JarLauncher`는 스프링 부트가 빌드시에 넣어준다. `org/springframework/boot/loader/JarLauncher`에 실제로 포함되어 있다.
  - 스프링 부트는 jar 내부에 jar를 읽어들이는 기능이 필요하다. 또 특별한 구조에 맞게 클래스 정보도 읽어들여야 한다. 바로 `JarLauncher`가 이런 일을 처리해준다. 이런 작업을 먼저 처리한 다음 `Start-Class:`에 지정된 `main()`을 호출한다.
- `Start-Class`: 우리가 기대한 `main()`이 있는 `hello.boot.BootApplication`가 적혀있다.
- 기타: 스프링 부트가 내부에서 사용하는 정보들이다.
  - `Spring-Boot-Version`: 스프링 부트 버전
  - `Spring-Boot-Classes`: 개발한 클래스 경로
  - `Spring-Boot-Lib`: 라이브러리 경로
  - `Spring-Boot-Classpath-Index`: 외부 라이브러리 모음
  - `Spring-Boot-Layers-Index`: 스프링 부트 구조 정보
- 참고: `Main-Class`를 제외한 나머지는 자바 표준이 아니다. 스프링 부트가 임의로 사용하는 정보이다.

스프링 부트 로더

`org/springframework/boot/loader` 하위에 있는 클래스들이다.

`JarLauncher`를 포함한 스프링 부트가 제공하는 실행 가능 Jar를 실제로 구동시키는 클래스들이 포함되어 있다. 스프링 부트는 빌드시에 이 클래스들을 포함해서 만들어준다.

BOOT-INF

- `classes` : 우리가 개발한 class 파일과 리소스 파일
- `lib` : 외부 라이브러리
- `classpath.idx` : 외부 라이브러리 모음
- `layers.idx` : 스프링 부트 구조 정보

- WAR구조는 `WEB-INF` 라는 내부 폴더에 사용자 클래스와 라이브러리를 포함하고 있는데, 실행 가능 Jar도 그 구조를 본따서 만들었다. 이름도 유사하게 `BOOT-INF` 이다.
- `JarLauncher` 를 통해서 여기에 있는 `classes` 와 `lib` 에 있는 jar 파일들을 읽어들인다.

실행 과정 정리

1. `java -jar xxx.jar`
2. `MANIFEST.MF` 인식
3. `JarLauncher.main()` 실행
   - `BOOT-INF/classes/` 인식
   - `BOOT-INF/lib/` 인식
4. `BootApplication.main()` 실행

> **참고**
> 실행 가능 Jar가 아니라, IDE에서 직접 실행할 때는 `BootApplication.main()` 을 바로 실행한다.
> IDE가 필요한 라이브러리를 모두 인식할 수 있게 도와주기 때문에 `JarLauncher` 가 필요하지 않다.

정리