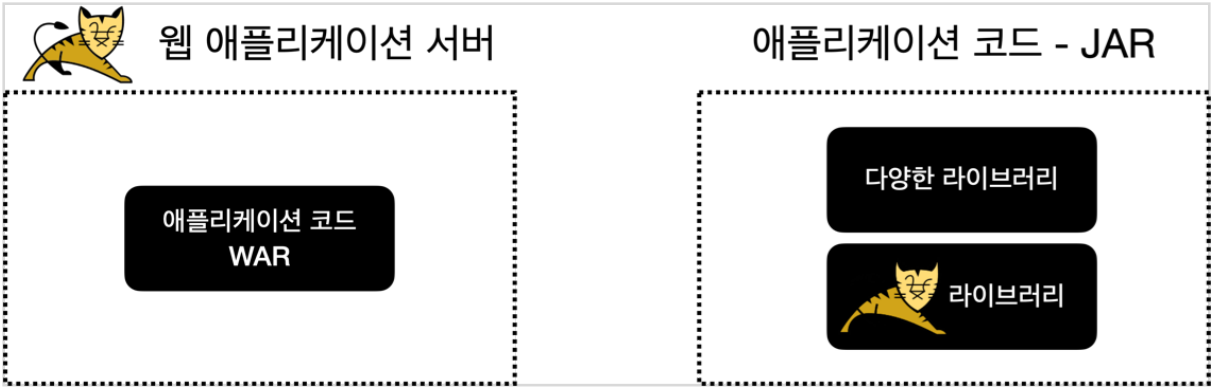
2. 웹 서버와 서블릿 컨테이너

#2.인강/9. 스프링부트/강의#

목차

웹 서버와 스프링 부트 소개

외장 서버 VS 내장 서버



전통적인 방식

과거에 자바로 웹 애플리케이션을 개발할 때는 먼저 서버에 톰캣 같은 WAS(웹 애플리케이션 서버)를 설치했다. 그리고 WAS에서 동작하도록 서블릿 스펙에 맞추어 코드를 작성하고 WAR 형식으로 빌드해서 war 파일을 만들었다. 이렇게 만들어진 war 파일을 WAS에 전달해서 배포하는 방식으로 전체 개발 주기가 동작했다.

이런 방식은 WAS 기반 위에서 개발하고 실행해야 한다. IDE 같은 개발 환경에서도 WAS와 연동해서 실행되도록 복잡한 추가 설정이 필요하다.

최근 방식

최근에는 스프링 부트가 내장 톰캣을 포함하고 있다. 애플리케이션 코드 안에 톰캣 같은 WAS가 라이브러리로 내장되어 있다는 뜻이다. 개발자는 코드를 작성하고 JAR로 빌드한 다음에 해당 JAR를 원하는 위치에서 실행하기만 하면 WAS도 함께 실행된다.
쉽게 이야기해서 개발자는 `main()` 메서드만 실행하면 되고, WAS 설치나 IDE 같은 개발 환경에서 WAS와 연동하는 복잡한 일은 수행하지 않아도 된다.

그런데 스프링 부트는 어떤 원리로 내장 톰캣을 사용해서 실행할 수 있는 것일까?
지금부터 과거로 돌아가서 톰캣도 직접 설치하고, WAR도 빌드하는 전통적인 방식으로 개발을 진행해보자. 그리고 어떤 불편한 문제들이 있어서 최근 방식으로 변화했는지 그 과정을 함께 알아보자.

이미 잘 아는 옛날 개발자들은 WAS를 설치하고 war를 만들어서 배포하는 것이 익숙하겠지만, 최근 개발자들은 이런 것을 경험할 일이 없다. 그래도 한번은 알아둘 가치가 있다. 과거에 어떻게 했는지 알아야 현재의 방식이 왜 이렇게 사용되고 있는지 더 깊이있는 이해가 가능하다. 서블릿 컨테이너도 설정하고, 스프링 컨테이너도 만들어서 등록하고, 디스패처 서블릿을 만들어서 스프링 MVC와 연결하는 작업을 스프링 부트 없이 직접 경험해보자. 그러면 스프링 부트가 웹 서버와 어떻게 연동되는지 자연스럽게 이해할 수 있을 것이다.
참고로 여기서는 `web.xml` 대신에 자바 코드로 서블릿을 초기화 한다. 옛날 개발자라도 대부분 `web.xml`을 사용했지 자바 코드로 서블릿 초기화를 해본 적은 없을 것이므로 꼭 한번 코드로 함께 따라해보자.

참고
저와 같은 옛날 개발자들은 앞 부분에서 톰캣을 설치하고 WAR를 만들고 실행하는 부분은 이미 잘 아실 것이라 생각한다. 이 분들을 편안하게 따라서 진행하면 된다. 아직 이런 것을 경험하지 못한 스프링 부트로 개발을 시작하는 신입, 주니어 개발자들을 배려한다 생각하자.

톰캣 설치

> **자바 버전 주의!**
> **자바 17 버전** 또는 그 이상을 설치하고 사용해주세요. 강의에서는 **스프링 3.0을 사용하는데 자바 17이 최소 요구 버전**입니다.

톰캣 다운로드

- https://tomcat.apache.org/download-10.cgi
- Download 메뉴에서 Apache Tomcat 10 버전의 톰캣 다운로드
  - Core에 있는 `zip`을 선택
- 다운로드 후 압축 풀기
- 다운로드 링크: https://dlcdn.apache.org/tomcat/tomcat-10/v10.0.27/bin/apache-tomcat-10.0.27.zip

톰캣 실행 설정

- **MAC, 리눅스 사용자**
- `톰캣폴더/bin` 폴더로 이동
- 권한 주기: `chmod 755 *`
- 실행: `./startup.sh`
- 종료: `./shutdown.sh`

> **참고**
> MAC, 리눅스 사용자는 권한을 주지 않으면 `permission denied` 라는 오류가 발생할 수 있다.

윈도우 사용자

- `톰캣폴더/bin` 폴더로 이동
- 실행: `startup.bat`
- 종료: `shutdown.bat`

실행 확인

톰캣을 실행한 상태로 다음 URL에 접근하면 톰캣 서버가 실행된 화면을 확인할 수 있다.

- http://localhost:8080

> **참고 - 실행 로그 확인**
> 톰캣의 실행 로그는 `톰캣폴더/logs/catalina.out` 파일로 확인할 수 있다.

실행이 잘 되지 않을 때 해결 방안

톰캣을 실행했지만 `http://localhost:8080` 에 접근이 되지 않으면 우선 실행 로그를 확인해야 한다.
실행 로그 `톰캣폴더/logs/catalina.out`

만약 다음과 같은 메시지가 보인다면 어떤 프로그램이 이미 8080 포트를 사용하고 있는 것이다.
`java.net.BindException: Address already in use`

해결방안1 - 해당 포트를 사용하는 프로그램을 종료한다.

8080 포트를 사용하는 프로그램을 찾아서 종료한다. 만약 종료가 어렵다면 컴퓨터를 재부팅 하는 것도 방법이다.

[MAC OS]

`sudo lsof -i :8080` 프로세스 ID(PID) 조회

`sudo kill -9 PID` : 프로세스 종료

[윈도우]

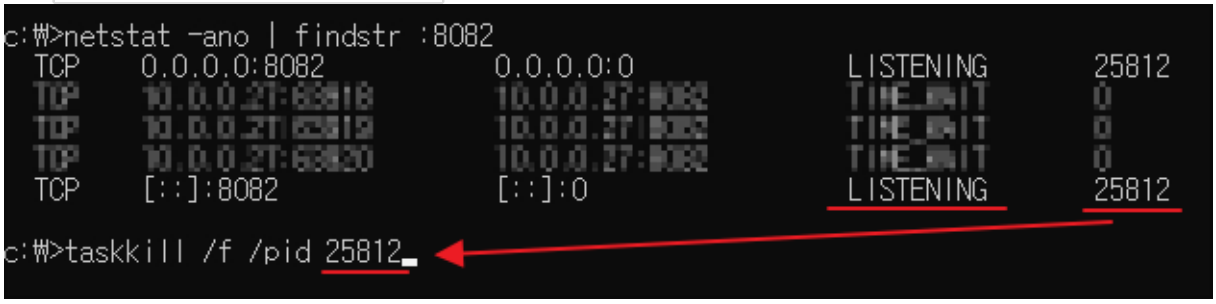
cmd를 열고 아래 명령어 순차적으로 실행

현재 포트를 사용중인 프로세스 찾기

`netstat -ano | findstr :포트번호`

프로세스 강제 종료하기

`taskkill /f /pid 프로세스번호`



해결방안2 - 톰캣 서버 포트를 변경한다

다음 톰캣 설정 파일을 수정한다. 여기에 보면 8080 이라는 부분이 있는데, 이 부분을 다른 포트로 변경한다.

그리고 톰캣 서버를 종료하고 다시 시작한 다음 다른 포트로 접근한다.

예) 9080 포트로 변경했으면 http://localhost:9080 으로 접근한다.

`톰캣폴더/conf/server.xml`

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

이제 이 톰캣 서버에 배포할 애플리케이션 코드를 작성해보자.

프로젝트 설정

사전 준비물

- Java 17 이상 설치
- IDE: IntelliJ 또는 Eclipse 설치

프로젝트 설정 순서

1. `server-start`의 폴더 이름을 `server`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'
    id 'war'
}

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

repositories {
    mavenCentral()
}

dependencies {
    //서블릿
    implementation 'jakarta.servlet:jakarta.servlet-api:6.0.0'
}

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

- `id 'war'` : 톰캣 같은 웹 애플리케이션 서버(WAS) 위에서 동작하는 WAR 파일을 만들어주는 플러그인이다.
- `jakarta.servlet-api` : 서블릿을 사용할 때 필요한 라이브러리이다.

먼저 간단한 HTML과 순수 서블릿으로 동작하는 웹 애플리케이션을 만들어보자.

간단한 HTML 등록

웹 서버가 정적 리소스를 잘 전달하는지 확인하기 위해 HTML을 하나 만들어보자.

- `/src/main` 하위에 `webapp` 이라는 폴더를 만들자
- 다음 HTML 파일을 생성하자.

`/src/main/webapp/index.html`

```
<html>
<body>index html</body>
</html>
```

서블릿 등록

전체 설정이 잘 동작하는지 확인하기 위해 간단한 서블릿을 하나 만들어보자.

웹 서버를 통해 이 서블릿이 실행되어야 한다.

TestServlet 등록

```
package hello.servlet;

import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;

/**
 * http://localhost:8080/test
 */
@WebServlet(urlPatterns = "/test")
public class TestServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
        System.out.println("TestServlet.service");
        resp.getWriter().println("test");
    }
```

```
    }
```

- `/test`로 요청이 오면 이 서블릿이 실행된다.
- `TestServlet.service`를 로그에 출력한다.
- `test`를 응답한다. 웹 브라우저로 요청하면 이 서블릿이 실행되고 화면에 `test`가 출력되어야 한다.

이 서블릿을 실행하려면 톰캣 같은 웹 애플리케이션 서버(WAS)에 이 코드를 배포해야 한다.

WAR 빌드와 배포

WAS에 우리가 만든 코드를 빌드하고 배포해보자.

프로젝트 빌드

- 프로젝트 폴더로 이동
- 프로젝트 빌드
  - `./gradlew build`
  - **[윈도우 OS]:** `gradlew build`
- WAR 파일 생성 확인
  - `build/libs/server-0.0.1-SNAPSHOT.war`

참고

`build.gradle`에 보면 `war` 플러그인이 사용된 것을 확인할 수 있다. 이 플러그인이 `war` 파일을 만들어준다.

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

WAR 압축 풀기

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

WAR를 푼 결과

- `WEB-INF`
  - `classes`
    - `hello/servlet/TestServlet.class`
  - `lib`
    - `jakarta.servlet-api-6.0.0.jar`
- `index.html`

WAR를 푼 결과를 보면 `WEB-INF`, `classes`, `lib` 같은 특별한 폴더들이 보인다. 이 부분을 알아보자.

JAR, WAR 간단 소개

JAR 소개

자바는 여러 클래스와 리소스를 묶어서 `JAR`(Java Archive)라고 하는 압축 파일을 만들 수 있다.
이 파일은 JVM 위에서 직접 실행되거나 또는 다른 곳에서 사용하는 라이브러리로 제공된다.
직접 실행하는 경우 `main()` 메서드가 필요하고, `MANIFEST.MF` 파일에 실행할 메인 메서드가 있는 클래스를 지정해두어야 한다.

실행 예) `java -jar abc.jar`
Jar는 쉽게 이야기해서 클래스와 관련 리소스를 압축한 단순한 파일이다. 필요한 경우 이 파일을 직접 실행할 수도 있고, 다른 곳에서 라이브러리로 사용할 수도 있다.

WAR 소개

WAR(Web Application Archive)라는 이름에서 알 수 있듯 WAR 파일은 웹 애플리케이션 서버(WAS)에 배포할 때 사용하는 파일이다.
JAR 파일이 JVM 위에서 실행된다면, WAR는 웹 애플리케이션 서버 위에서 실행된다.
웹 애플리케이션 서버 위에서 실행되고, HTML 같은 정적 리소스와 클래스 파일을 모두 함께 포함하기 때문에 JAR와 비교해서 구조가 더 복잡하다.
그리고 WAR 구조를 지켜야 한다.

WAR 구조

- `WEB-INF`
  - `classes` : 실행 클래스 모음
  - `lib` : 라이브러리 모음
  - `web.xml` : 웹 서버 배치 설정 파일(생략 가능)
- `index.html` : 정적 리소스

- `WEB-INF` 폴더 하위는 자바 클래스와 라이브러리, 그리고 설정 정보가 들어가는 곳이다.
- `WEB-INF` 를 제외한 나머지 영역은 HTML, CSS 같은 정적 리소스가 사용되는 영역이다.

WAR 배포

이렇게 생성된 WAR 파일을 톰캣 서버에 실제 배포해보자.

- 1. 톰캣 서버를 종료한다. `./shutdown.sh`
- 2. `톰캣폴더/webapps` 하위를 모두 삭제한다.
- 3. 빌드된 `server-0.0.1-SNAPSHOT.war` 를 복사한다.
- 4. `톰캣폴더/webapps` 하위에 붙여넣는다.
  - `톰캣폴더/webapps/server-0.0.1-SNAPSHOT.war`
- 5. 이름을 변경한다.
  - `톰캣폴더/webapps/ROOT.war`
- 6. 톰캣 서버를 실행한다. `./startup.sh`

주의!

`ROOT.war` 에서 `ROOT` 는 대문자를 사용해야 한다.

- **MAC, 리눅스 사용자**
- `톰캣폴더/bin` 폴더
  - 실행: `./startup.sh`
  - 종료: `./shutdown.sh`

윈도우 사용자

- `톰캣폴더/bin` 폴더
  - 실행: `startup.bat`
  - 종료: `shutdown.bat`

실행 결과 확인

- http://localhost:8080/index.html
- http://localhost:8080/test

실행해보면 `index.html` 정적 파일과 `/test` 로 만들어둔 `TestServlet` 모두 잘 동작하는 것을 확인할 수 있다.

> **참고**
> 진행이 잘 되지 않으면 `톰캣폴더/logs/catalina.out` 로그를 꼭 확인해보자.

실제 서버에서는 이렇게 사용하면 되지만, 개발 단계에서는 `war` 파일을 만들고, 이것을 서버에 복사해서 배포하는 과정이 너무 번잡하다.
인텔리J나 이클립스 같은 IDE는 이 부분을 편리하게 자동화해준다.

톰캣 설정 - 인텔리J 유료 버전

인텔리J 유로 버전과 무료 버전에 따라 IDE에서 톰캣을 설정하는 방법이 다르다. 여기서는 유료 버전을 설명한다. 무료 버전을 사용하면 다음 영상을 참고하자.

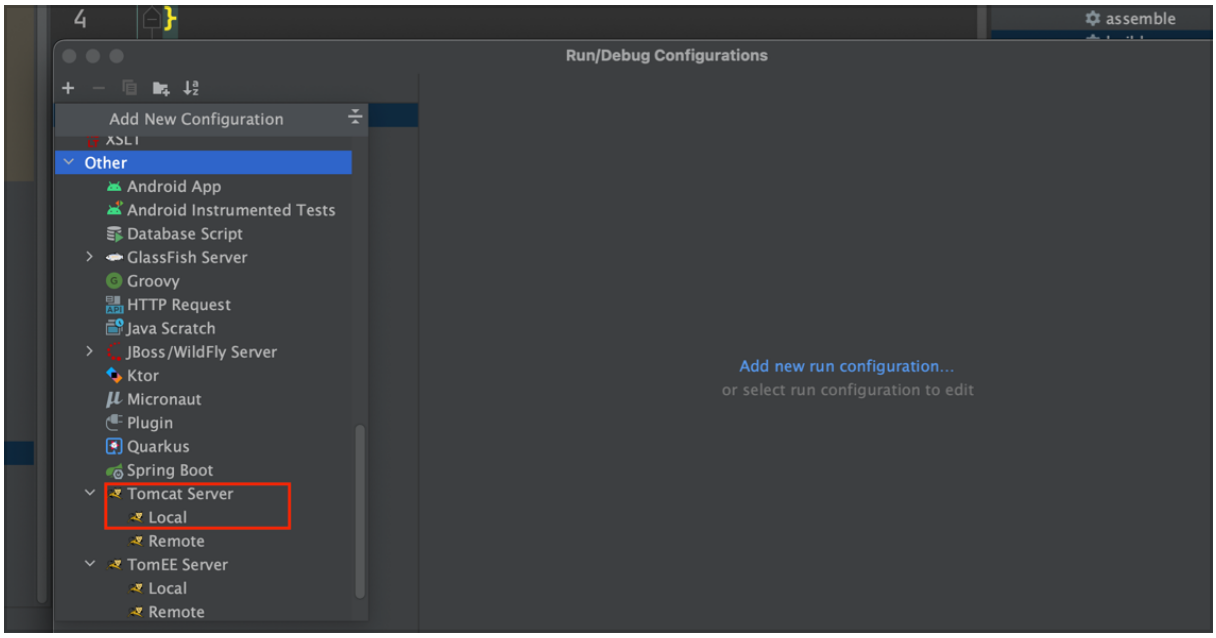
> **참고**
> 이클립스 IDE 설정은 다루지 않는다. "이클립스 gradle 톰캣"이라는 키워드로 검색하면 수 많은 예시를 확인할 수 있다.

톰캣 설정 - 인텔리J 유료 버전

인텔리J 유로 버전에는 톰캣 지원이 포함되어 있다.

- 다음으로 이동한다.
  - `메뉴 -> Run... -> Edit Configurations`

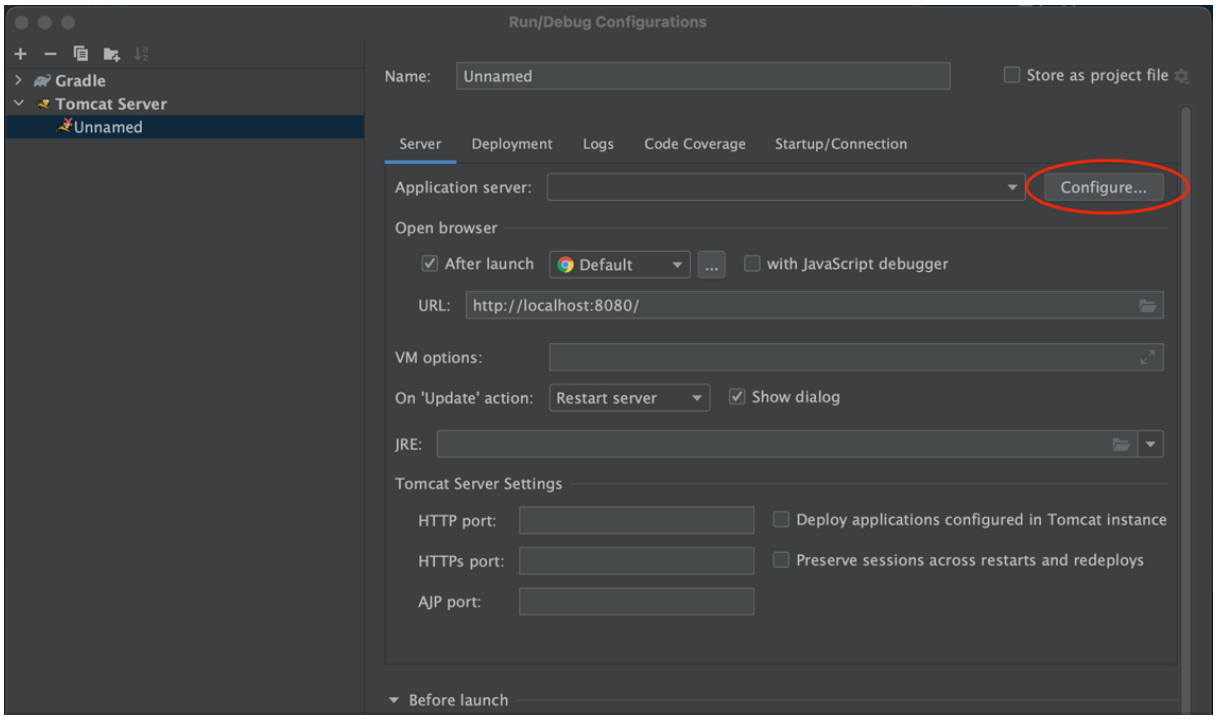
- 왼쪽 상단의 플러스 버튼을 클릭한다.



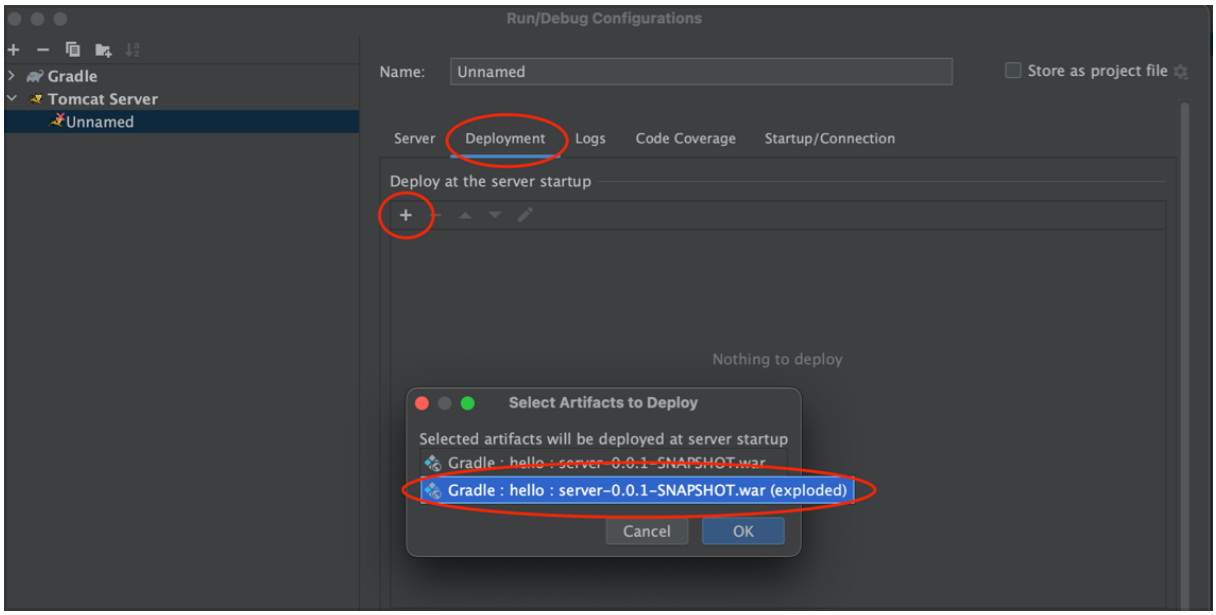
- Other 하위에 있는 Tomcat Server에 Local을 선택한다.

  주의!

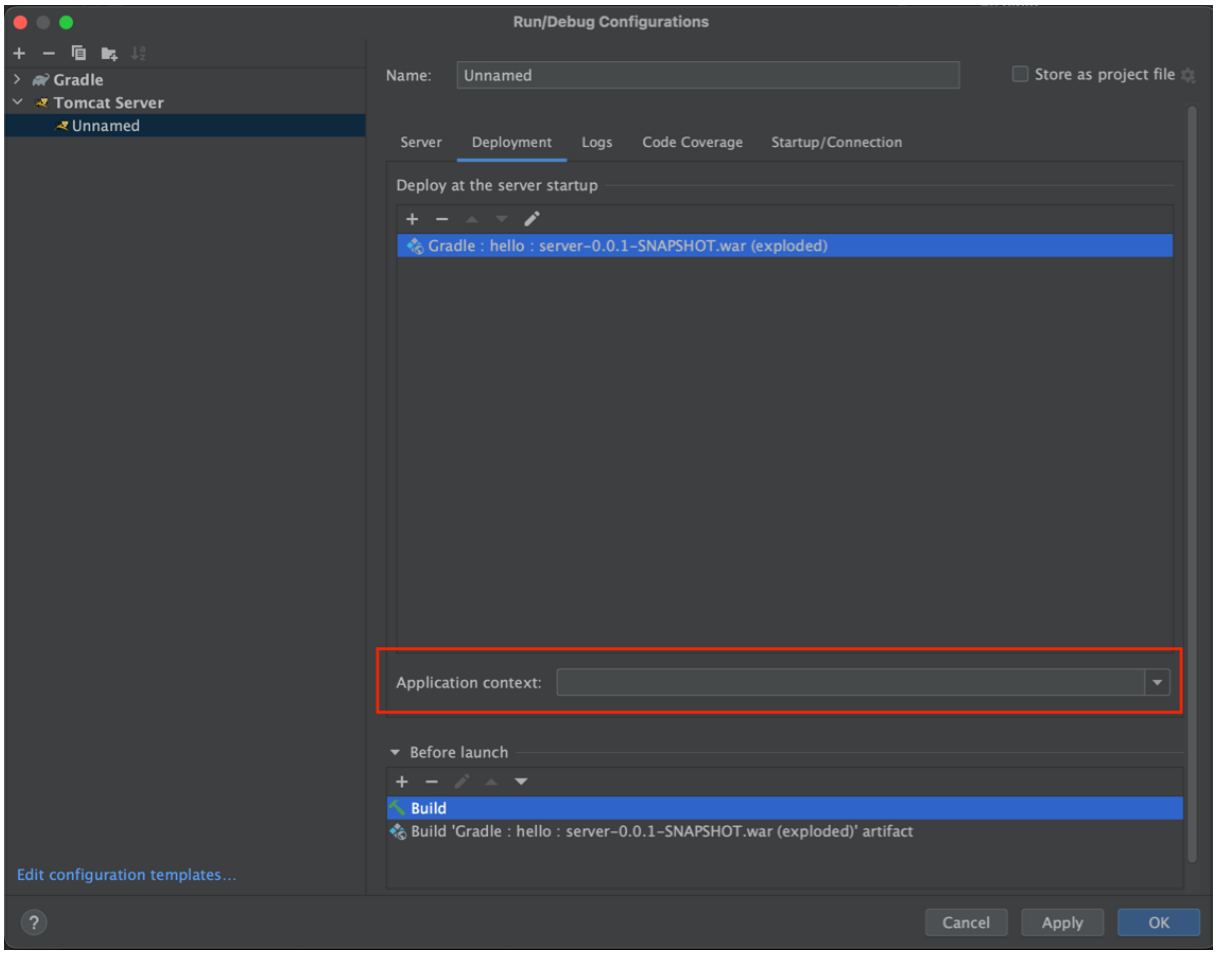
  `Tomcat Server`를 선택해야 한다. `TomEE Server`를 선택하면 안된다.



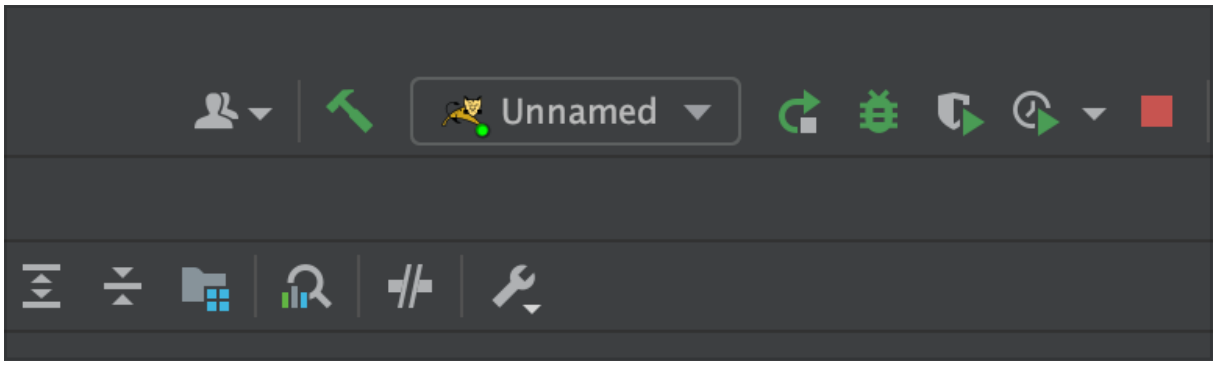
- Configure… 부분을 선택한다.
- Tomcat Home: 부분에 설치된 톰캣 폴더를 선택한다.



- `Deployment` 메뉴를 선택한다.
- `+` 버튼을 선택한다.
- 끝에 (exploded)로 끝나는 `war` 파일을 선택한다.



- Application context 박스 안에 있는 내용을 모두 지워준다.



설정한 톰캣을 선택하고 실행한다.

> **주의!**
> `java.net.BindException: Address already in use` 오류 메시지가 로그에 보이면 앞서 실행한 톰캣 서버가 이미 8080 포트를 점유하고 있을 가능성이 높다. `shutdown.sh`를 실행해서 앞서 실행한 톰캣 서버를 내리자 (잘 안되면 컴퓨터를 재부팅 하는 것도 방법이다.)

실행

- http://localhost:8080
- http://localhost:8080/test

톰캣 설정 - 인텔리J 무료 버전

무료 버전은 설정이 조금 복잡하니 차근차근 확인하면서 진행하자

gradle 설정

`build.gradle`에 다음 내용을 추가하자.

```
//war 풀기, 인텔리J 무료버전 필요
task explodedWar(type: Copy) {
    into "$buildDir/exploded"
    with war
}
```

다음 명령어를 실행한다.

`./gradlew explodedWar`

[윈도우 OS]: `gradlew explodedWar`

`build/exploded` 폴더가 새로 생성되고, 여기에 `WEB-INF`를 포함한 `war`가 풀려 있는 모습을 확인할 수

있다.

Tomcat runner 설정

인텔리J 무료 버전에서는 `Tomcat runner`라는 플러그인을 설치해야 한다.

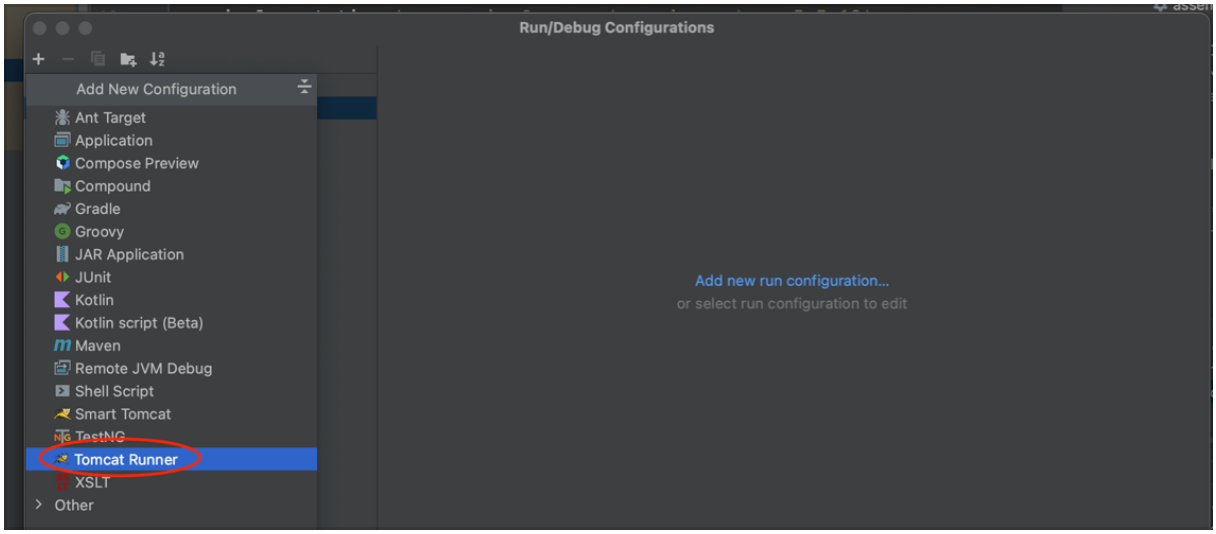
> **참고**
> 인텔리J 유로 버전에서는 `Tomcat runner` 플러그인을 찾을 수 없다.

Tomcat runner 설치

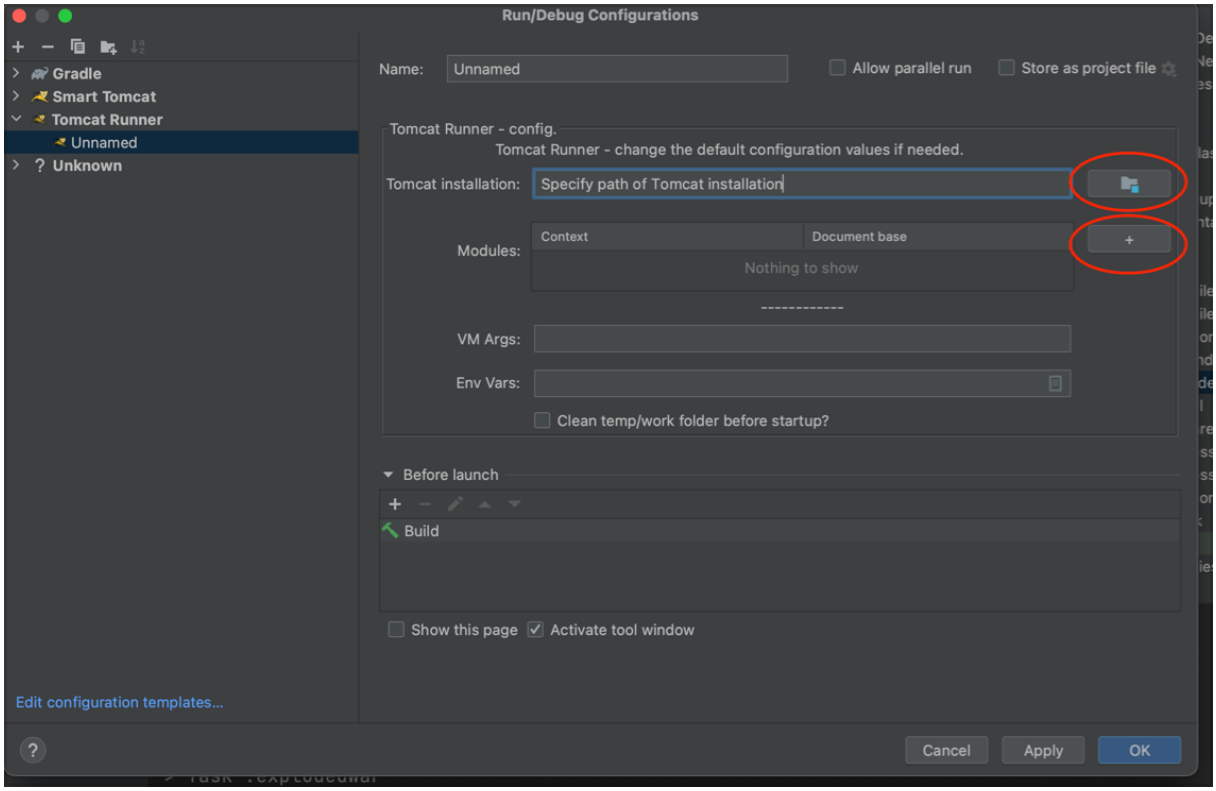
- 메뉴 ➡ IntelliJ IDEA ➡ Preferences(윈도우즈는 File ➡ Settings) ➡ plugin ➡ `tomcat runner` 검색 설치 실행 (재시작)
- `tomcat runner`에서 사용할 톰캣 서버를 별도로 하나 만들어둔다.
  - 여기서는 다운로드 받은 톰캣 서버의 압축을 새로 풀어서 `tomcat-runner`라는 이름의 폴더로 만들어두겠다.
  - **주의!** 이 플러그인은 기존 톰캣 서버의 설정을 변경하기 때문에, 이 톰캣 서버를 다른곳에서 함께 사용하면 문제가 발생할 수 있다. 꼭 별도의 톰캣 서버를 준비해두자.

- 다음으로 이동한다.
  - `메뉴 -> Run -> Edit Configurations`

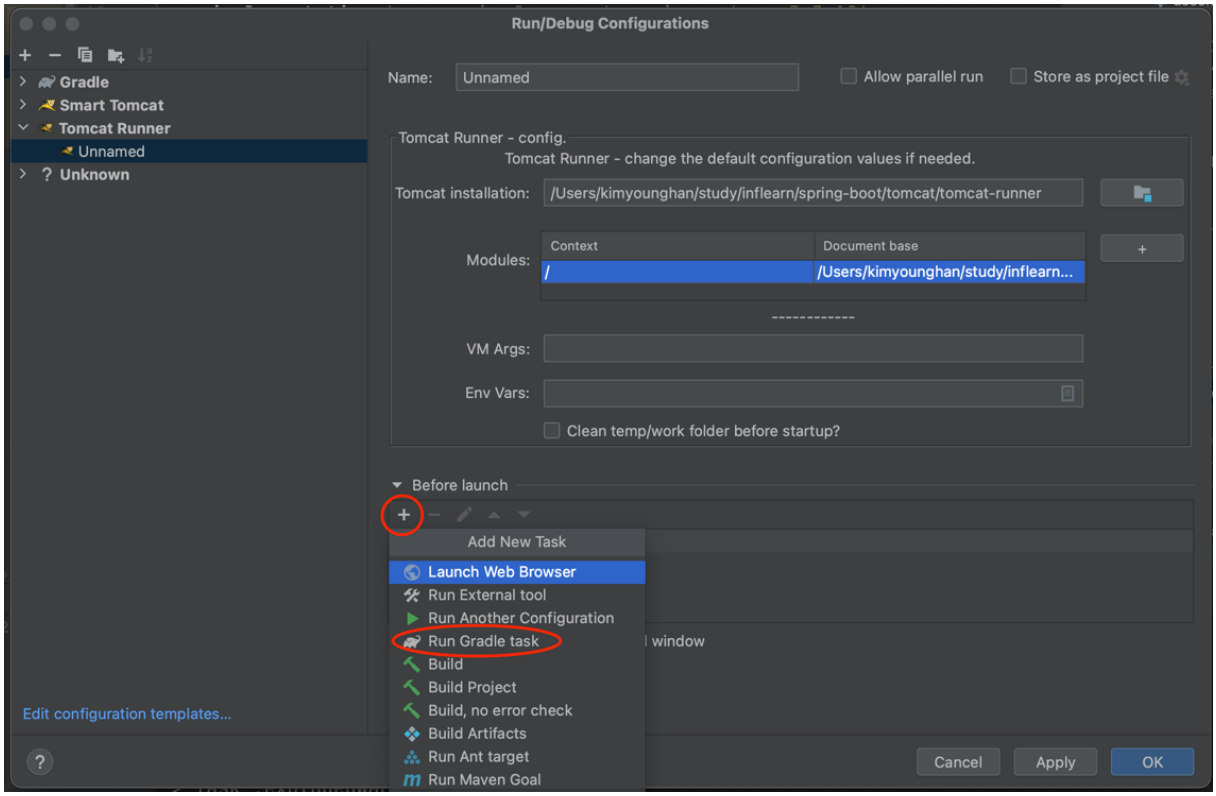
- 왼쪽 상단의 플러스 버튼을 클릭한다.



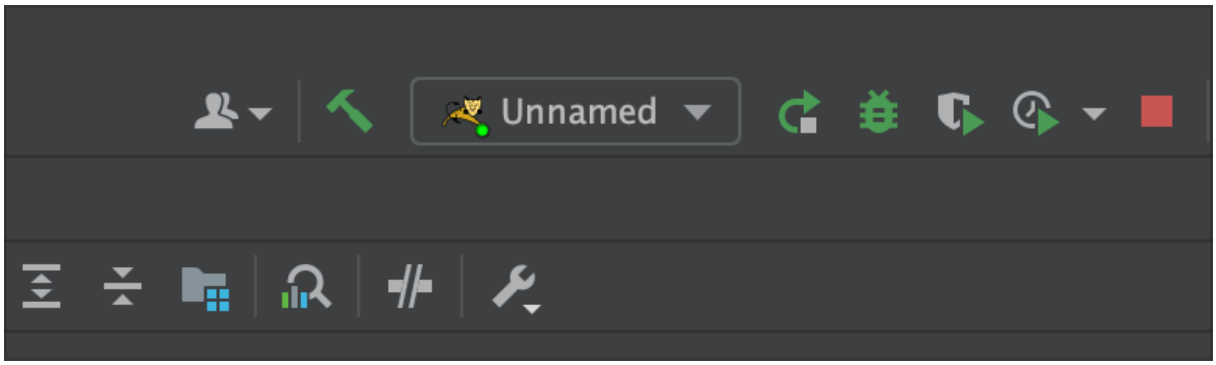
- Tomcat Runner를 선택한다.



- Tomcat installation에 설치된 톰캣 서버를 지정한다. 참고로 `tomcat runner` 전용으로 준비한 톰캣 서버를 사용해야 한다. 그렇지 않으면 향후 다른 곳에서 해당 톰캣 서버를 사용할 때 설정이 꼬이는 문제가 발생한다. 여기서는 `tomcat-runner` 라는 이름의 폴더에 새로운 톰캣 서버를 준비해두었다.
- Modules 부분에 `+` 버튼을 선택한다.
  - Context: 부분에는 `/` 를 입력한다.
  - Doc. base: 부분에는 `build/exploded` 폴더를 지정한다.
  - Modules 부분을 수정하고 싶으면 더블 클릭해서 삭제한 다음에 다시 입력해야 한다.



- Before launch 아래 있는 `+` 버튼을 선택한다.
- Run Gradle task를 선택한다.
  - Gradle project: 현재 프로젝트를 선택한다.
  - Tasks: `explodedWar` 를 입력한다.
  - 이렇게 하면 서버를 실행하기 전에 새로 빌드하면서 `gradle explodedWar` 를 실행하게 된다.



설정한 톰캣을 선택하고 실행한다.

> **주의!**
> `java.net.BindException: Address already in use` 오류 메시지가 로그에 보이면 앞서 실행한 톰캣 서버가 이미 8080 포트를 점유하고 있을 가능성이 높다. `./shutdown.sh` 를 실행해서 앞서 실행한 톰캣 서버를 내리자 (잘 안되면 컴퓨터를 재부팅 하는 것도 방법이다.)

실행

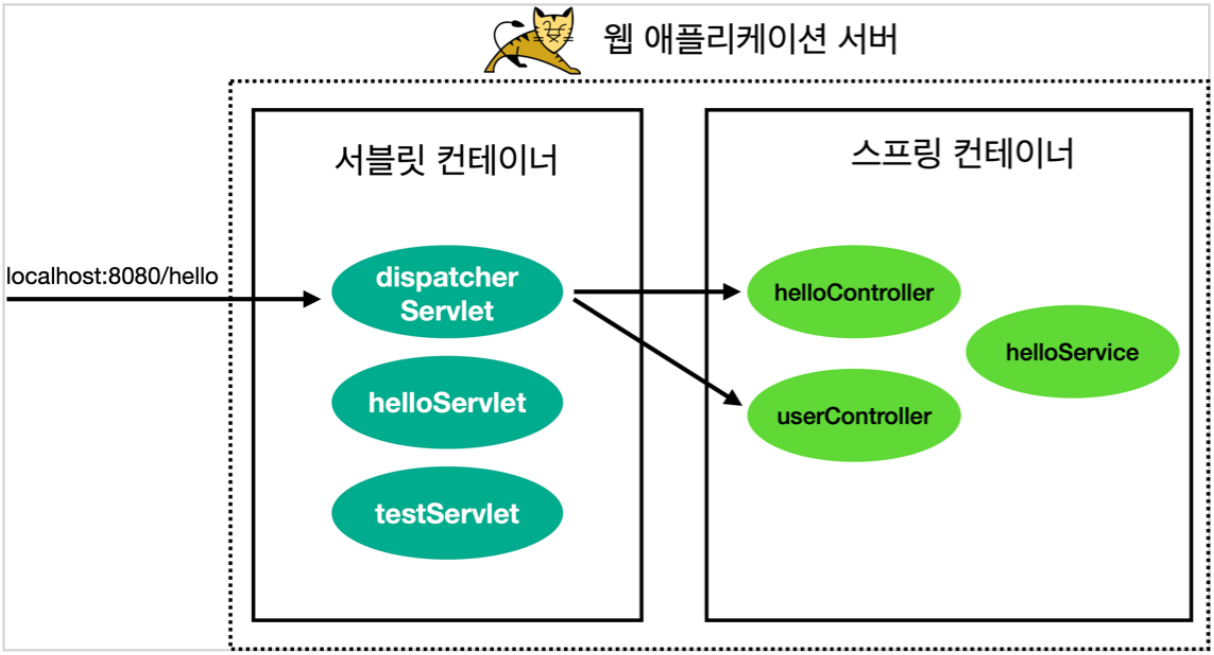
- http://localhost:8080

- http://localhost:8080/test

서블릿 컨테이너 초기화1

- WAS를 실행하는 시점에 필요한 초기화 작업들이 있다. 서비스에 필요한 필터와 서블릿을 등록하고, 여기에 스프링을 사용한다면 스프링 컨테이너를 만들고, 서블릿과 스프링을 연결하는 디스페처 서블릿도 등록해야 한다.
- WAS가 제공하는 초기화 기능을 사용하면, WAS 실행 시점에 이러한 초기화 과정을 진행할 수 있다.
- 과거에는 `web.xml`을 사용해서 초기화했지만, 지금은 서블릿 스펙에서 자바 코드를 사용한 초기화도 지원한다.

서블릿 컨테이너와 스프링 컨테이너



지금부터 서블릿 컨테이너의 초기화 기능을 알아보고 이어서 이 초기화 기능을 활용해 스프링 만들고 연결해보자.

서블릿 컨테이너 초기화 개발

서블릿은 `ServletContainerInitializer`라는 초기화 인터페이스를 제공한다. 이름 그대로 서블릿 컨테이너를 초기화 하는 기능을 제공한다.
서블릿 컨테이너는 실행 시점에 초기화 메서드인 `onStartup()`을 호출해준다. 여기서 애플리케이션에 필요한 기능들을 초기화 하거나 등록할 수 있다.

ServletContainerInitializer

```
public interface ServletContainerInitializer {
    public void onStartup(Set<Class<?>> c, ServletContext ctx) throws
ServletException;
}
```

- `Set<Class<?>> c` : 조금 더 유연한 초기화를 기능을 제공한다. `@HandlesTypes` 애노테이션과 함께 사용한다. 이후에 코드로 설명한다.
- `ServletContext ctx` : 서블릿 컨테이너 자체의 기능을 제공한다. 이 객체를 통해 필터나 서블릿을 등록할 수 있다.

방금 본 서블릿 컨테이너 초기화 인터페이스를 간단히 구현해서 실제 동작하는지 확인해보자.

`hello/container/MyContainerInitV1`

```
package hello.container;

import jakarta.servlet.ServletContainerInitializer;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import java.util.Set;

public class MyContainerInitV1 implements ServletContainerInitializer {

    @Override
    public void onStartup(Set<Class<?>> c, ServletContext ctx) throws
ServletException {
        System.out.println("MyContainerInitV1.onStartup");
        System.out.println("MyContainerInitV1 c = " + c);
        System.out.println("MyContainerInitV1 ctx = " + ctx);
    }
}
```

이것이 끝이 아니다. 추가로 WAS에게 실행할 초기화 클래스를 알려줘야 한다.
다음 경로에 파일을 생성하자

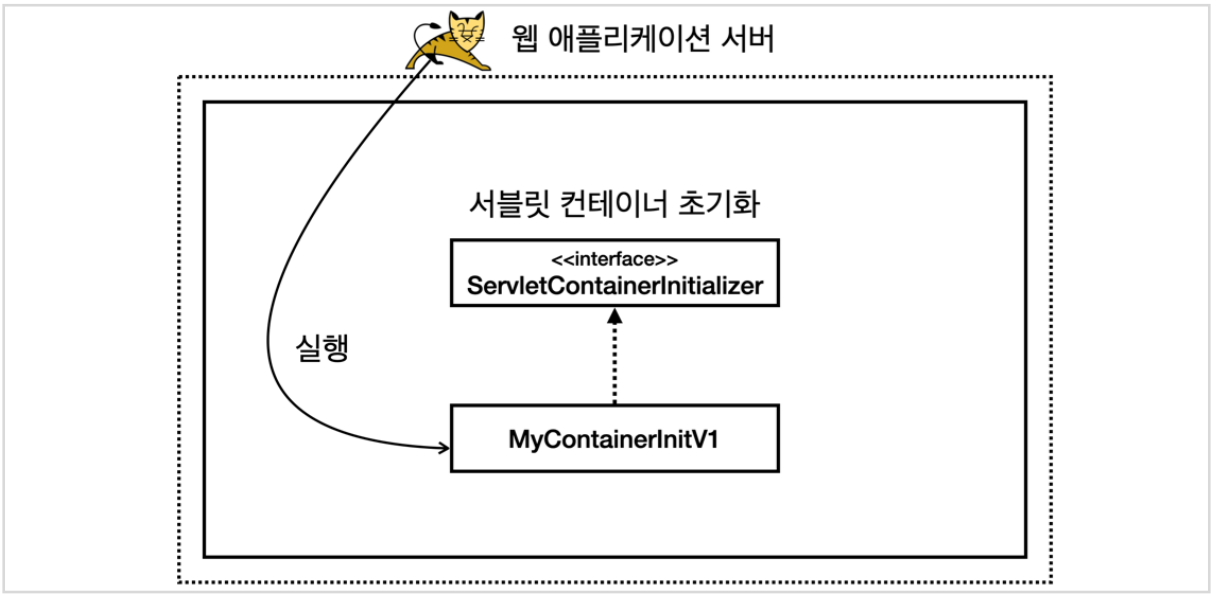
`resources/META-INF/services/jakarta.servlet.ServletContainerInitializer`

```
hello.container.MyContainerInitV1
```

이 파일에 방금 만든 `MyContainerInitV1` 클래스를 패키지 경로를 포함해서 지정해주었다.
이렇게 하면 WAS를 실행할 때 해당 클래스를 초기화 클래스로 인식하고 로딩 시점에 실행한다.

> **주의!**
> 경로와 파일 이름을 주의해서 작성해야한다.
> `META-INF` 는 대문자이다.
> `services` 는 마지막에 `s` 가 들어간다.
> 파일 이름은 `jakarta.servlet.ServletContainerInitializer` 이다.

그림



WAS를 실행해보자.

실행 결과 로그

```
MyContainerInitV1.onStartup
MyContainerInitV1 c = null
MyContainerInitV1 ctx =
org.apache.catalina.core.ApplicationContextFacade@65112751
```

WAS를 실행할 때 해당 초기화 클래스가 실행된 것을 확인할 수 있다.

서블릿 컨테이너 초기화2

서블릿 컨테이너 초기화를 조금 더 자세히 알아보자.
여기서는 `HelloServlet`이라는 서블릿을 서블릿 컨테이너 초기화 시점에 프로그래밍 방식으로 직접 등록해줄 것이다.

서블릿을 등록하는 2가지 방법

- `@WebServlet` 애노테이션
- 프로그래밍 방식

HelloServlet

```
package hello.servlet;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

public class HelloServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("HelloServlet.service");
        resp.getWriter().println("hello servlet!");
    }
}
```

- 이 서블릿을 등록하고 실행하면 다음과 같은 결과가 나온다. 다음 내용을 통해서 서블릿을 등록해보자.
  - 로그: `HelloServlet.service`
  - HTTP 응답: `hello servlet!`

애플리케이션 초기화

서블릿 컨테이너는 조금 더 유연한 초기화 기능을 지원한다.
여기서는 이것을 **애플리케이션 초기화**라 하겠다.
이 부분을 이해하려면 실제 동작하는 코드를 봐야 한다.

AppInit

```
package hello.container;

import jakarta.servlet.ServletContext;

public interface AppInit {
    void onStartup(ServletContext servletContext);
}
```

- 애플리케이션 초기화를 진행하려면 먼저 인터페이스를 만들어야 한다. 내용과 형식은 상관없고, 인터페이스는 꼭 필요하다. 예제 진행을 위해서 여기서는 `AppInit` 인터페이스를 만들자.

앞서 개발한 애플리케이션 초기화(`AppInit`) 인터페이스를 구현해서 실제 동작하는 코드를 만들어보자.

AppInitV1Servlet

```
package hello.container;

import hello.servlet.HelloServlet;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletRegistration;

/**
 * http://localhost:8080/hello-servlet
 */
public class AppInitV1Servlet implements AppInit {

    @Override
    public void onStartup(ServletContext servletContext) {
        System.out.println("AppInitV1Servlet.onStartup");

        //순수 서블릿 코드 등록
        ServletRegistration.Dynamic helloServlet =
                servletContext.addServlet("helloServlet", new HelloServlet());
        helloServlet.addMapping("/hello-servlet");
    }
}
```

- 여기서는 프로그래밍 방식으로 `HelloServlet` 서블릿을 서블릿 컨테이너에 직접 등록한다.
- HTTP로 `/hello-servlet`를 호출하면 `HelloServlet` 서블릿이 실행된다.

서블릿을 등록하는 2가지 방법

- `@WebServlet` 애노테이션
- 프로그래밍 방식

참고 - 프로그래밍 방식을 사용하는 이유

`@WebServlet`을 사용하면 애노테이션 하나로 서블릿을 편리하게 등록할 수 있다. 하지만 애노테이션 방식을 사용하면 유연하게 변경하는 것이 어렵다. 마치 하드코딩 된 것 처럼 동작한다. 아래 참고 예시를 보면 `/test` 경로를 변경하고 싶으면 코드를 직접 변경해야 바꿀 수 있다.
반면에 프로그래밍 방식은 코딩을 더 많이 해야하고 불편하지만 무한한 유연성을 제공한다.
예를 들어서

- `/hello-servlet` 경로를 상황에 따라서 바꾸어 외부 설정을 읽어서 등록할 수 있다.
- 서블릿 자체도 특정 조건에 따라서 `if` 문으로 분기해서 등록하거나 뺄 수 있다.
- 서블릿을 내가 직접 생성하기 때문에 생성자에 필요한 정보를 넘길 수 있다.

예제에서는 단순화를 위해 이런 부분들을 사용하지는 않았지만 프로그래밍 방식이 왜 필요하지? 라고 궁금하신 분들을 위해서 적어보았다.

참고 - 예시

```
@WebServlet(urlPatterns = "/test")
public class TestServlet extends HttpServlet {}
```

서블릿 컨테이너 초기화(`ServletContainerInitializer`)는 앞서 알아보았다. 그런데 애플리케이션 초기화(`AppInit`)는 어떻게 실행되는 것일까? 다음 코드를 만들어 보자.

MyContainerInitV2

```
package hello.container;

import jakarta.servlet.ServletContainerInitializer;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.HandlesTypes;
import java.util.Set;
```

```
@HandlesTypes(AppInit.class)
public class MyContainerInitV2 implements ServletContainerInitializer {

    @Override
    public void onStartup(Set<Class<?>> c, ServletContext ctx) throws
ServletException {
        System.out.println("MyContainerInitV2.onStartup");
        System.out.println("MyContainerInitV2 c = " + c);
        System.out.println("MyContainerInitV2 container = " + ctx);

        for (Class<?> appInitClass : c) {
            try {
                //new AppInitV1Servlet()과 같은 코드
                AppInit appInit = (AppInit)
appInitClass.getDeclaredConstructor().newInstance();
                appInit.onStartup(ctx);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

애플리케이션 초기화 과정

- 1. `@HandlesTypes` 애노테이션에 애플리케이션 초기화 인터페이스를 지정한다.
  - 여기서는 앞서 만든 `AppInit.class` 인터페이스를 지정했다.
- 2. 서블릿 컨테이너 초기화(`ServletContainerInitializer`)는 파라미터로 넘어오는 `Set<Class<?>> c` 에 애플리케이션 초기화 인터페이스의 구현체들을 모두 찾아서 클래스 정보로 전달한다.
  - 여기서는 `@HandlesTypes(AppInit.class)` 를 지정했으므로 `AppInit.class` 의 구현체인 `AppInitV1Servlet.class` 정보가 전달된다.
  - 참고로 객체 인스턴스가 아니라 클래스 정보를 전달하기 때문에 실행하려면 객체를 생성해서 사용해야 한다.
- 3. `appInitClass.getDeclaredConstructor().newInstance()`
  - 리플렉션을 사용해서 객체를 생성한다. 참고로 이 코드는 `new AppInitV1Servlet()` 과 같다 생각하면 된다.
- 4. `appInit.onStartup(ctx)`
  - 애플리케이션 초기화 코드를 직접 실행하면서 서블릿 컨테이너 정보가 담긴 `ctx` 도 함께 전달한다.

MyContainerInitV2 등록

`MyContainerInitV2`를 실행하려면 서블릿 컨테이너에게 알려주어야 한다. 설정을 추가하자.

`resources/META-INF/services/jakarta.servlet.ServletContainerInitializer`

```
hello.container.MyContainerInitV1
hello.container.MyContainerInitV2
```

- `hello.container.MyContainerInitV2` 추가

WAS를 실행해보자.

실행 로그

```
MyContainerInitV1.onStartup
MyContainerInitV1 c = null
MyContainerInitV1 ctx =
org.apache.catalina.core.ApplicationContextFacade@38dd0980
MyContainerInitV2.onStartup
MyContainerInitV2 c = [class hello.container.AppInitV1Servlet]
MyContainerInitV2 container =
org.apache.catalina.core.ApplicationContextFacade@38dd0980
AppInitV1Servlet.onStartup
```

서블릿을 실행해보자.

실행

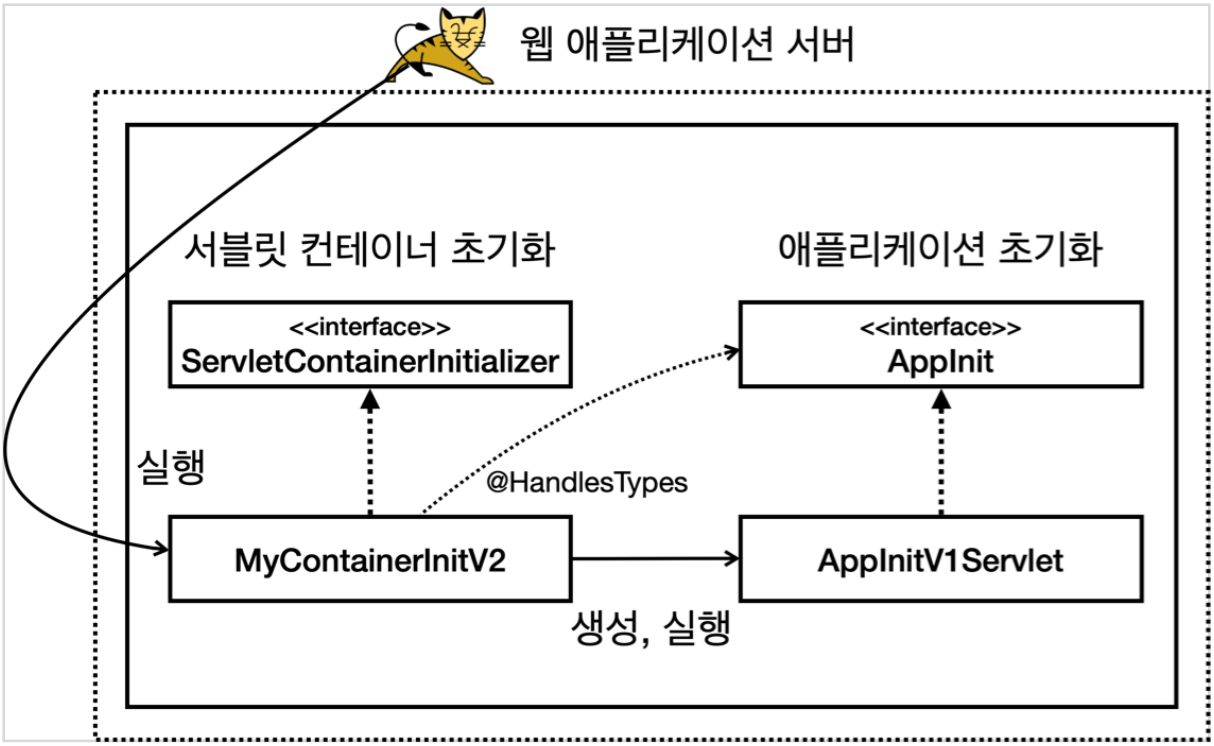
- http://localhost:8080/hello-servlet

결과

```
hello servlet!
```

정리

그림



초기화는 다음 순서로 진행된다.

- 1. 서블릿 컨테이너 초기화 실행
  - `resources/META-INF/services/jakarta.servlet.ServletContainerInitializer`
- 2. 애플리케이션 초기화 실행
  - `@HandlesTypes(AppInit.class)`

참고

서블릿 컨테이너 초기화만 있어도 될 것 같은데, 왜 이렇게 복잡하게 애플리케이션 초기화라는 개념을 만들었을까?

편리함

- 서블릿 컨테이너를 초기화 하려면 `ServletContainerInitializer` 인터페이스를 구현한 코드를 만들어야 한다. 여기에 추가로 `META-INF/services/jakarta.servlet.ServletContainerInitializer` 파일에 해당 코드를 직접 지정해주어야 한다.
- 애플리케이션 초기화는 특정 인터페이스만 구현하면 된다.

의존성

- 애플리케이션 초기화는 서블릿 컨테이너에 상관없이 원하는 모양으로 인터페이스를 만들 수 있다. 이를 통해 애플리케이션 초기화 코드가 서블릿 컨테이너에 대한 의존을 줄일 수 있다. 특히 `ServletContext ctx`가 필요없는 애플리케이션 초기화 코드라면 의존을 완전히 제거할 수도 있다.

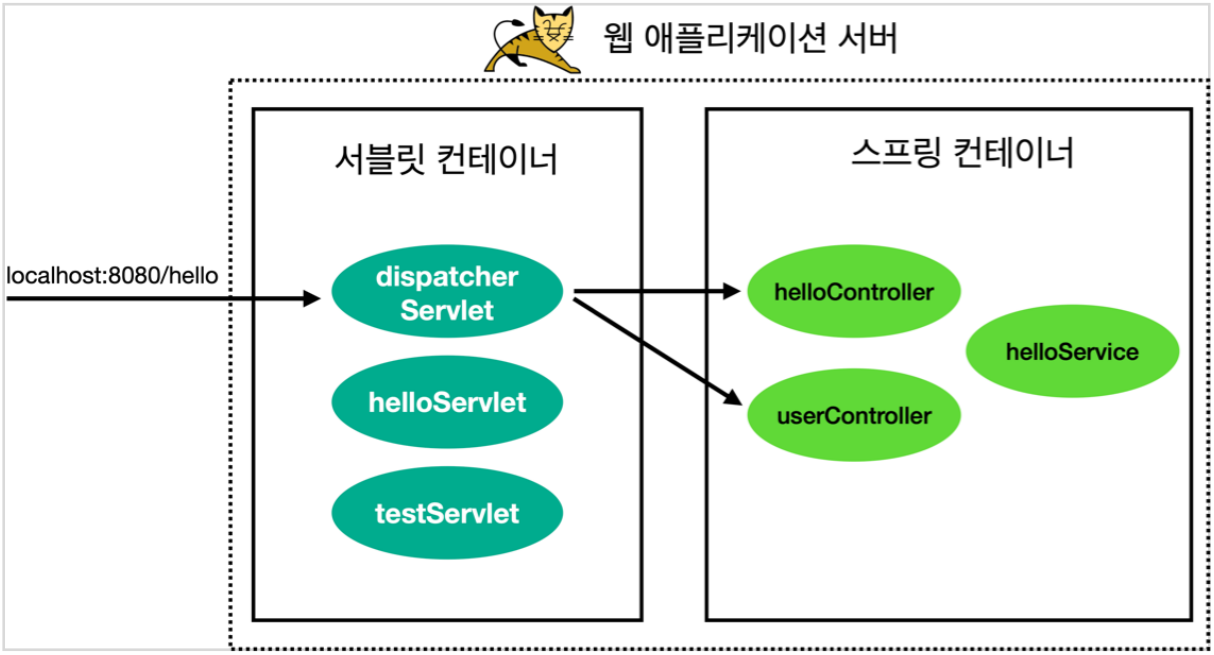
스프링 컨테이너 등록

이번에는 WAS와 스프링을 통합해보자.
앞서 배운 서블릿 컨테이너 초기화와 애플리케이션 초기화를 활용하면 된다.

다음과 같은 과정이 필요할 것이다.

- 스프링 컨테이너 만들기
- 스프링MVC 컨트롤러를 스프링 컨테이너에 빈으로 등록하기
- 스프링MVC를 사용하는데 필요한 디스패처 서블릿을 서블릿 컨테이너 등록하기

서블릿 컨테이너와 스프링 컨테이너



현재 라이브러리에는 스프링 관련 라이브러리가 전혀 없다. 스프링 관련 라이브러리를 추가하자.

`build.gradle` - `spring-webmvc` 추가

```
dependencies {
    //서블릿
    implementation 'jakarta.servlet:jakarta.servlet-api:6.0.0'

    //스프링 MVC 추가
    implementation 'org.springframework:spring-webmvc:6.0.4'
}
```

- `spring-webmvc` 라이브러리를 추가하면 스프링 MVC 뿐만 아니라 `spring-core` 를 포함한 스프링 핵심 라이브러리들도 함께 포함된다.

HelloController

```
package hello.spring;

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!";
    }
}
```

- 간단한 스프링 컨트롤러다.
- 실행하면 HTTP 응답으로 `hello spring!` 이라는 메시지를 반환한다.

HelloConfig

```
package hello.spring;

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

@Configuration
public class HelloConfig {

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

- 컨트롤러를 스프링 빈으로 직접 등록한다. 참고로 여기서는 컴포넌트 스캔을 사용하지 않고 빈을 직접 등록했다.

이제 애플리케이션 초기화를 사용해서 서블릿 컨테이너에 스프링 컨테이너를 생성하고 등록하자.

AppInitV2Spring

```
package hello.container;

import hello.spring.HelloConfig;
import
org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletRegistration;

/**
 * http://localhost:8080/spring/hello-spring
 */
public class AppInitV2Spring implements AppInit {

    @Override
    public void onStartup(ServletContext servletContext) {
        System.out.println("AppInitV2Spring.onStartup");

        //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 서블릿을 서블릿 컨테이너에 등록 (이름 주의! dispatcherV2)
        ServletRegistration.Dynamic servlet =
                servletContext.addServlet("dispatcherV2", dispatcher);

        // /spring/* 요청이 디스패처 서블릿을 통하도록 설정
```

```
            servlet.addMapping("/spring/*");
        }
    }
```

- `AppInitV2Spring`는 `AppInit`을 구현했다. `AppInit`을 구현하면 애플리케이션 초기화 코드가 자동으로 실행된다. 앞서 `MyContainerInitV2`에 관련 작업을 이미 해두었다.

스프링 컨테이너 생성

- `AnnotationConfigWebApplicationContext`가 바로 스프링 컨테이너이다.
  - `AnnotationConfigWebApplicationContext` 부모를 따라가 보면 `ApplicationContext` 인터페이스를 확인할 수 있다.
  - 이 구현체는 이름 그대로 애노테이션 기반 설정과 웹 기능을 지원하는 스프링 컨테이너로 이해하면 된다.
- `appContext.register(HelloConfig.class)`
  - 컨테이너에 스프링 설정을 추가한다.

스프링 MVC 디스패처 서블릿 생성, 스프링 컨테이너 연결

- `new DispatcherServlet(appContext)`
- 코드를 보면 스프링 MVC가 제공하는 디스패처 서블릿을 생성하고, 생성자에 앞서 만든 스프링 컨테이너를 전달하는 것을 확인할 수 있다. 이렇게 하면 디스패처 서블릿에 스프링 컨테이너가 연결된다.
- 이 디스패처 서블릿에 HTTP 요청이 오면 디스패처 서블릿은 해당 스프링 컨테이너에 들어있는 컨트롤러 빈들을 호출한다.

디스패처 서블릿을 서블릿 컨테이너에 등록

- `servletContext.addServlet("dispatcherV2", dispatcher)`
  - 디스패처 서블릿을 서블릿 컨테이너에 등록한다.
- `/spring/*` 요청이 디스패처 서블릿을 통하도록 설정
  - `/spring/*` 이렇게 경로를 지정하면 `/spring`과 그 하위 요청은 모두 해당 서블릿을 통하게 된다.
    - `/spring/hello-spring`
    - `/spring/hello/go`

주의! 서블릿을 등록할 때 이름은 원하는 이름을 등록하면 되지만 같은 이름으로 중복 등록하면 오류가 발생한다. 여기서는 `dispatcherV2` 이름을 사용했는데, 이후에 하나 더 등록할 예정이기 때문에 이름에 유의하자

실행

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

결과

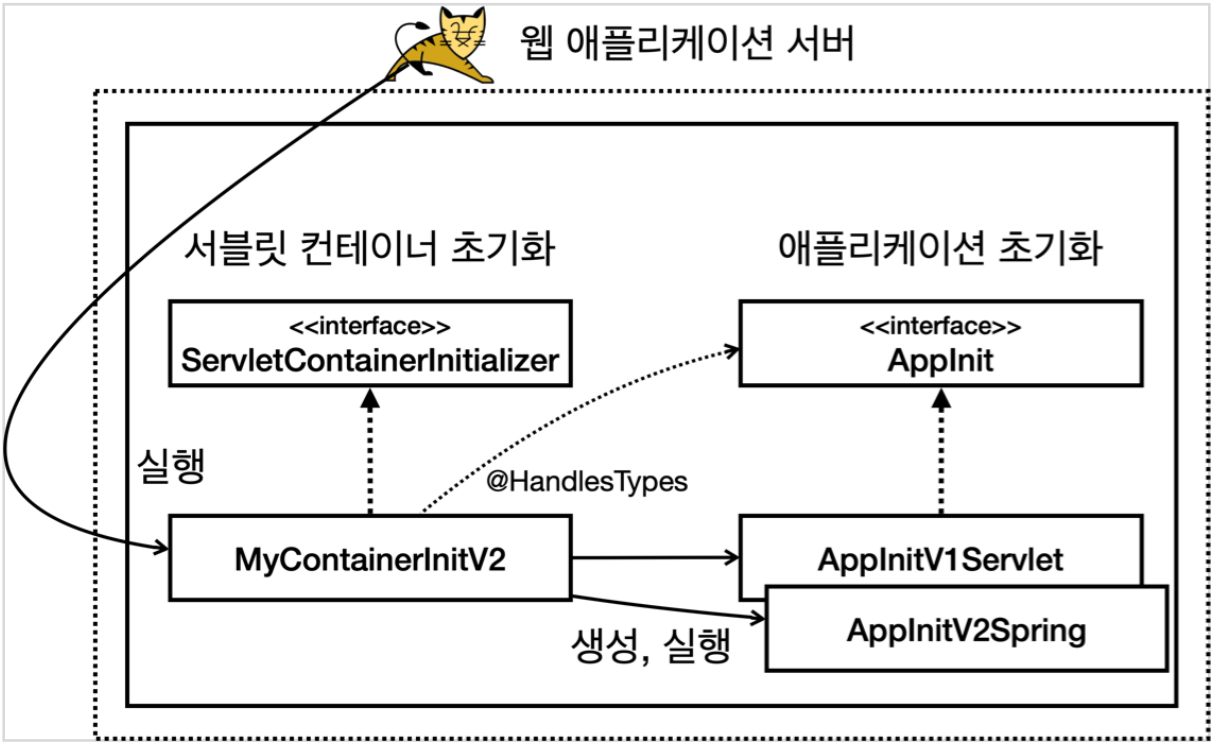
```
hello spring!
```

실행 과정 정리

`/spring/hello-spring`
실행을 `/spring/*` 패턴으로 호출했기 때문에 다음과 같이 동작한다.

- `dispatcherV2` 디스패처 서블릿이 실행된다. (`/spring`)
- `dispatcherV2` 디스패처 서블릿은 스프링 컨트롤러를 찾아서 실행한다. (`/hello-spring`)
  - 이때 서블릿을 찾아서 호출하는데 사용된 `/spring` 을 제외한 `/hello-spring` 가 매핑된 컨트롤러 `HelloController`)의 메서드를 찾아서 실행한다. (쉽게 이야기해서 뒤에 `*` 부분으로 스프링 컨트롤러를 찾는다.)

그림



스프링 MVC 서블릿 컨테이너 초기화 지원

지금까지의 과정을 생각해보면 서블릿 컨테이너를 초기화 하기 위해 다음과 같은 복잡한 과정을 진행했다.

- `ServletContainerInitializer` 인터페이스를 구현해서 서블릿 컨테이너 초기화 코드를 만들었다.
- 여기에 애플리케이션 초기화를 만들기 위해 `@HandlesTypes` 애노테이션을 적용했다.
- `/META-INF/services/jakarta.servlet.ServletContainerInitializer` 파일에 서블릿 컨테이너

초기화 클래스 경로를 등록했다.

서블릿 컨테이너 초기화 과정은 상당히 번거롭고 반복되는 작업이다.
스프링 MVC는 이러한 서블릿 컨테이너 초기화 작업을 이미 만들어두었다. 덕분에 개발자는 서블릿 컨테이너 초기화 과정은 생략하고, 애플리케이션 초기화 코드만 작성하면 된다.
스프링이 지원하는 애플리케이션 초기화를 사용하려면 다음 인터페이스를 구현하면 된다.

WebApplicationInitializer

```
package org.springframework.web;

public interface WebApplicationInitializer {
  void onStartup(ServletContext servletContext) throws ServletException;
}
```

스프링이 지원하는 애플리케이션 초기화 코드를 사용해보자.

AppInitV3SpringMvc

```
package hello.container;

import hello.spring.HelloConfig;
import org.springframework.web.WebApplicationInitializer;
import
org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRegistration;

/**
 * http://localhost:8080/hello-spring
 *
 * 스프링 MVC 제공 WebApplicationInitializer 활용
 * spring-web
 * META-INF/services/jakarta.servlet.ServletContainerInitializer
 * org.springframework.web.SpringServletContainerInitializer
```

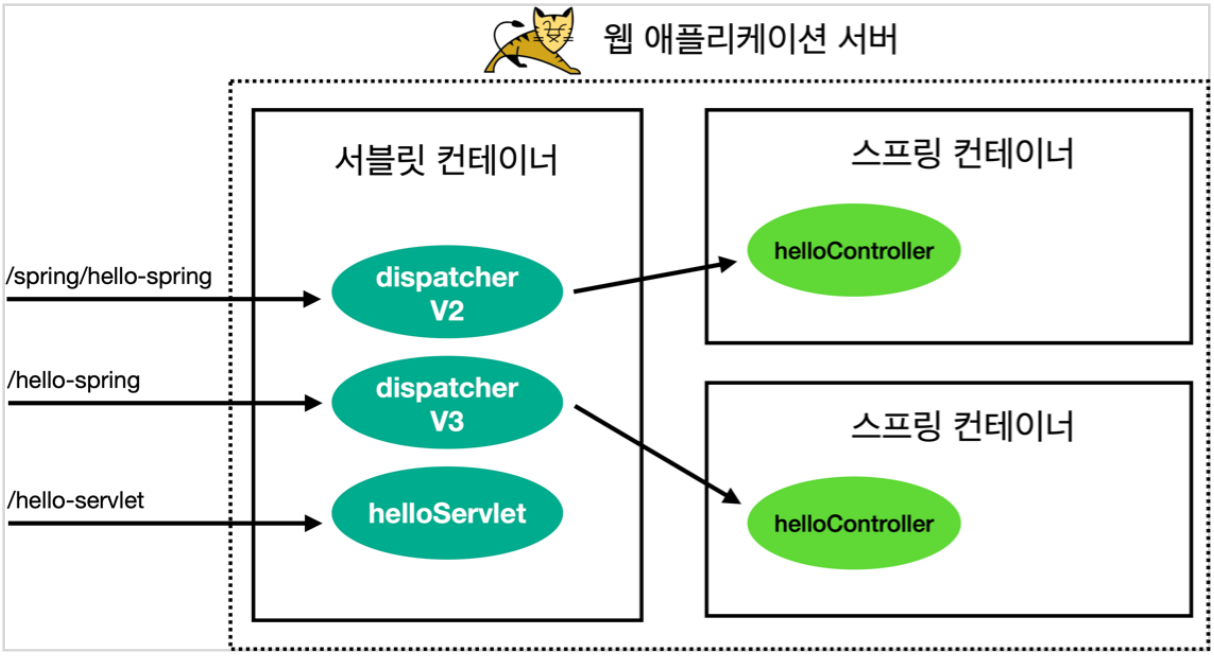
```
 */
public class AppInitV3SpringMvc implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws
ServletException {
        System.out.println("AppInitV3SpringMvc.onStartup");

        //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 서블릿을 서블릿 컨테이너에 등록 (이름 주의! dispatcherV3)
        ServletRegistration.Dynamic servlet =
                servletContext.addServlet("dispatcherV3", dispatcher);

        //모든 요청이 디스패처 서블릿을 통하도록 설정
        servlet.addMapping("/");
    }
}
```

- `WebApplicationInitializer` 인터페이스를 구현한 부분을 제외하고는 이전의 `AppInitV2Spring`과 거의 같은 코드이다.
  - `WebApplicationInitializer`는 스프링이 이미 만들어둔 애플리케이션 초기화 인터페이스이다.
- 여기서도 디스패처 서블릿을 새로 만들어서 등록하는데, 이전 코드에서는 `dispatcherV2`라고 했고, 여기서는 `dispatcherV3`라고 해주었다. 참고로 이름이 같은 서블릿을 등록하면 오류가 발생한다.
- `servlet.addMapping("/")` 코드를 통해 모든 요청이 해당 서블릿을 타도록 했다.
  - 따라서 다음과 같이 요청하면 해당 디스패처 서블릿을 통해 `/hello-spring`이 매핑된 컨트롤러 메서드가 호출된다.

실행

http://localhost:8080/hello-spring

정리 그림



- 현재 등록된 서블릿 다음과 같다.
  - `/` = `dispatcherV3`
  - `/spring/*` = `dispatcherV2`
  - `/hello-servlet` = `helloServlet`
  - `/test` = `TestServlet`
  - 이런 경우 우선순위는 더 구체적인 것이 먼저 실행된다.

참고

- 여기서는 이해를 돕기 위해 디스패처 서블릿도 2개 만들고, 스프링 컨테이너도 2개 만들었다.
- 일반적으로는 스프링 컨테이너를 하나 만들고, 디스패처 서블릿도 하나만 만든다. 그리고 디스패처 서블릿의 경로 매핑도 `/` 로 해서 하나의 디스패처 서블릿을 통해서 모든 것을 처리하도록 한다.

스프링 MVC가 제공하는 서블릿 컨테이너 초기화 분석

스프링은 어떻게 `WebApplicationInitializer` 인터페이스 하나로 애플리케이션 초기화가 가능하게 할까? 스프링도 결국 서블릿 컨테이너에서 요구하는 부분을 모두 구현해야 한다.

`spring-web` 라이브러리를 열어보면 서블릿 컨테이너 초기화를 위한 등록 파일을 확인할 수 있다. 그리고 이곳에 서블릿 컨테이너 초기화 클래스가 등록되어 있다.

`/META-INF/services/jakarta.servlet.ServletContainerInitializer`

```
org.springframework.web.SpringServletContainerInitializer
```

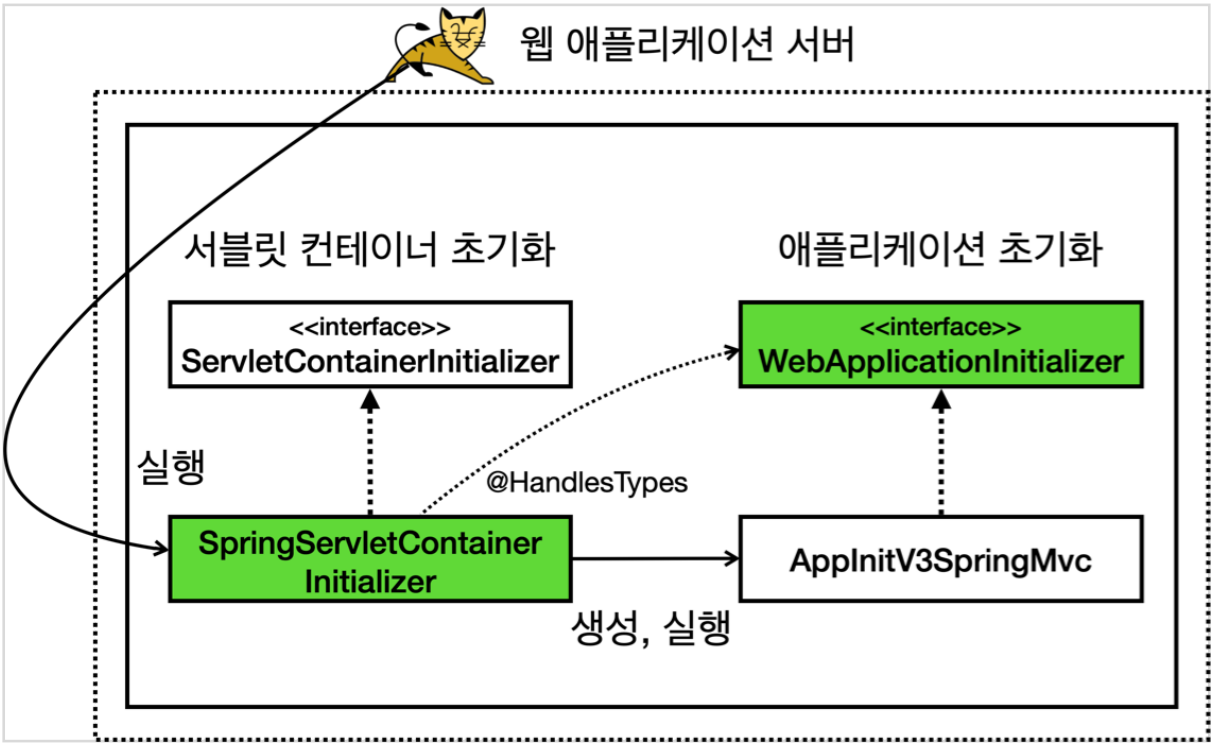
`org.springframework.web.SpringServletContainerInitializer` 코드를 확인해보자.

SpringServletContainerInitializer

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements
ServletContainerInitializer {}
```

- 코드를 보면 우리가 앞서 만든 서블릿 컨테이너 초기화 코드와 비슷한 것을 확인할 수 있다.
- `@HandlesTypes`의 대상이 `WebApplicationInitializer`이다. 그리고 이 인터페이스의 구현체를 생성하고 실행하는 것을 확인할 수 있다. 우리는 앞서 이 인터페이스를 구현했다.

그림



- 초록색 영역은 이미 스프링이 만들어서 제공하는 영역이다.

정리

스프링MVC도 우리가 지금까지 한 것 처럼 서블릿 컨테이너 초기화 파일에 초기화 클래스를 등록해두었다.
그리고 `WebApplicationInitializer` 인터페이스를 애플리케이션 초기화 인터페이스로 지정해두고, 이것을 생성해서 실행한다.
따라서 스프링 MVC를 사용한다면 `WebApplicationInitializer` 인터페이스만 구현하면 `AppInitV3SpringMvc`에서 본 것 처럼 편리하게 애플리케이션 초기화를 사용할 수 있다.

정리

지금까지 서블릿 컨테이너 초기화를 사용해서 필요한 서블릿도 등록하고, 스프링 컨테이너도 생성해서 등록하고 또 스프링 MVC가 동작하도록 디스패처 서블릿도 중간에 연결해보았다. 그리고 스프링이 제공하는 좀 더 편리한 초기화 방법도 알아보았다.

지금까지 알아본 내용은 모두 서블릿 컨테이너 위에서 동작하는 방법이다. 따라서 항상 톰캣 같은 서블릿 컨테이너에 배포를 해야만 동작하는 방식이다.

과거에는 서블릿 컨테이너 위에서 모든 것이 동작했지만, 스프링 부트와 내장 톰캣을 사용하면서 이런 부분이 바뀌기 시작했다.