✨ SpringBoot WEB 동작원리

- 작성자 : 윤요섭 사원

🏘 강의 소개

> 스프링 부트의 핵심 기능을 코드로 직접 구현하면서 스프링 부트의 동작 원리와 스프링 부트에 적용된 스프링 프레임워크의 활용법을 익히게 되는 강의

🗀 목차

1. Introducing Spring Boot

1. 스프링 부트 소개

스프링 부트(Spring boot)는 **스프링을 기반**으로 실무 환경에 사용가능한 수준의 **독립실행형 애플리케이션**을 복잡한 고민 없이 빠르게 작성할 수 있게 도와주는 여러가지 도구의 모음이다.

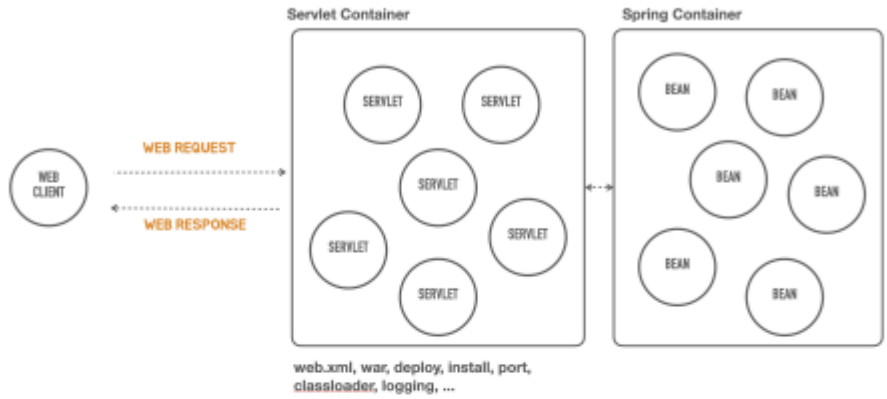
✔ 스프링 부트의 핵심

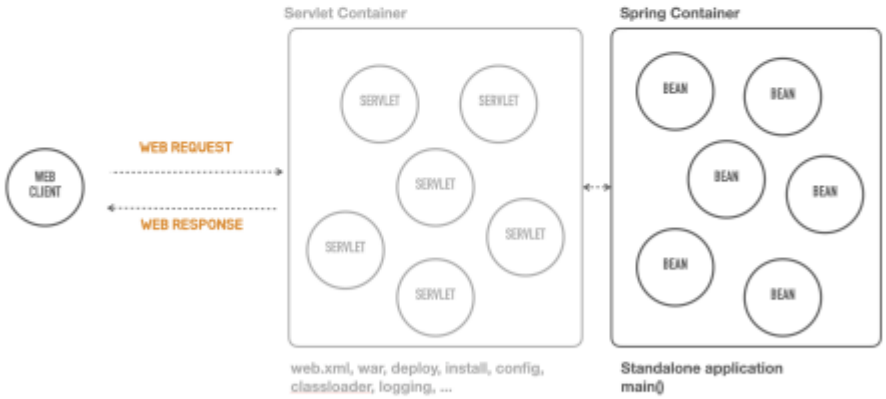
```
1. Containerless(Containerless web application architecture)

2. 강한 주장을 가진(Opinionated) 도구
```

1-1. Containerless

컨테이너리스 개발

"컨테이너 없는" 웹 애플리케이션 아키텍처란?





스프링 애플리케이션 개발에 요구되는 서블릿 컨테이너의 설치, WAR 폴더 구조, web.xml, WAR 빌드, 컨테이너로 배치, 포트 설정, 클래스로더, 로깅 등과 같은 필요하지만 애플리케이션 개발의 핵심이 아닌 단순 반복 작업을 제거해주는 개발 도구와 아키텍처 지원한다

설치된 컨테이너로 배포하지 않고 독립실행형(standalone) 자바 애플리케이션으로 동작

✔ Serverless

- 서버에 대한 설치, 관리 이런 부분을 개발자들이 신경쓰지 않고 서버 어플리케이션 개발해서 배포하고 운영하는 것이 가능하도록 만든 방법

Containerless는 Serverless의 개념과 유사하다. 그렇다면 여기서 말하는 Container란 무엇인가? 그리고 Containerless는 무엇을 뜻하는 것인가?

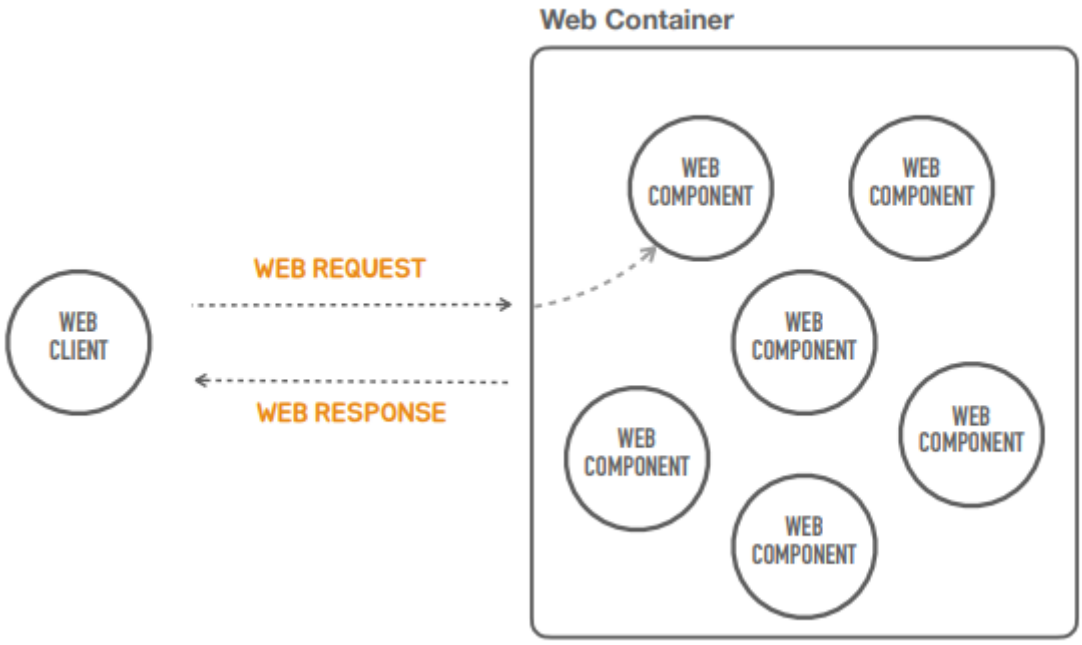
😥 Container란?

✔ Web Program 예시

```
Web Program은 서버에서 동작하면서 기능을 제공한다. 예시로, WEB COMPONENT (회원가입) 을 만들었다치자.

WEB COMPONENT 혼자서는 일을 하지 못한다. WEB COMPONENT는 WEB CLIENT가 필요하기 때문이다.
```

```
WEB CLIENT가 WEB REQUEST를 해줘야 WEB COMPONENT가 Dynamic Content를 만들어서, WEB RESPONSE 해줄 수 있다.
```

이 때, WEB COMPONENT는 항상 Web Container 안에 있어야 한다.

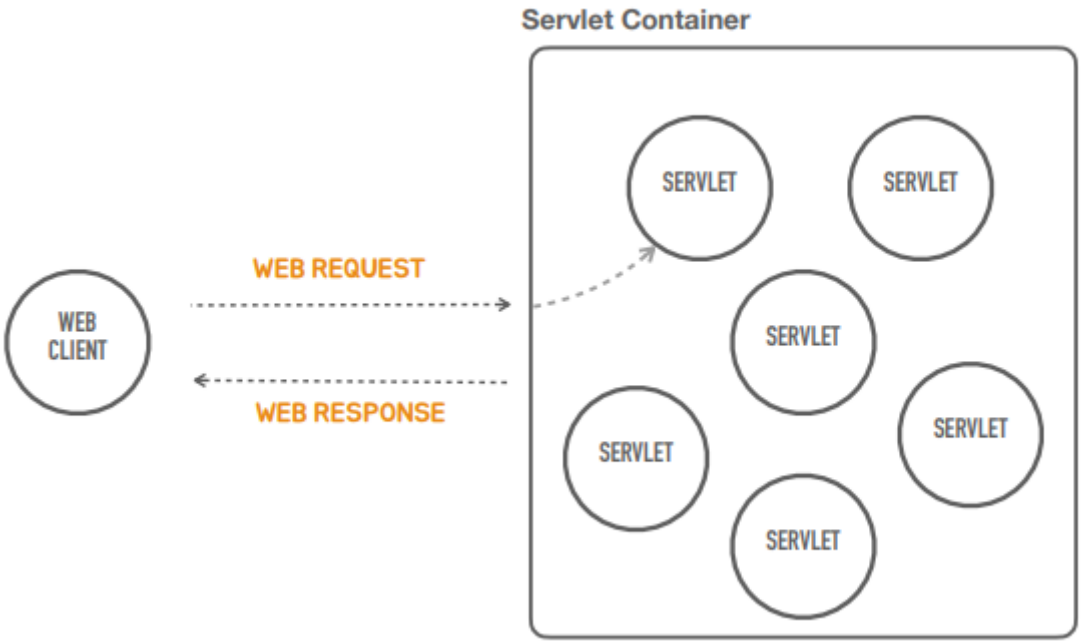


Web Container는 WEB COMPONENT의 life cycle을 관리하는 역할을 하며, Client로 부터 들어온 요청을 어느 WEB COMPONENT가 담당할지 정해줘야 한다.

Servlet

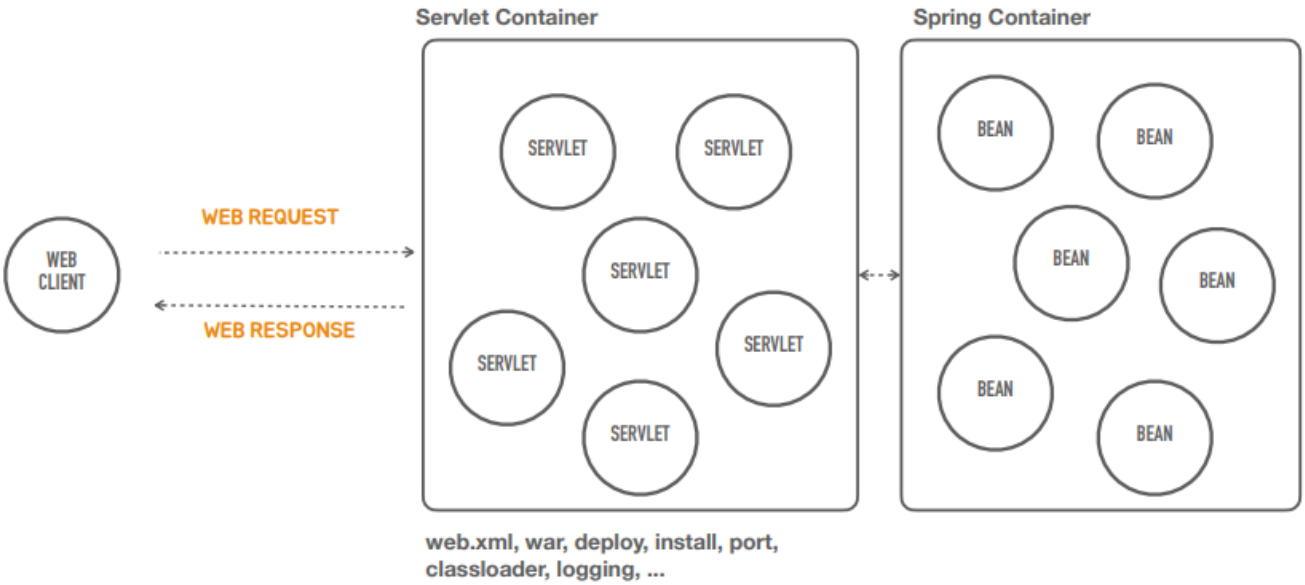
자바에서는 WEB COMPONENT를 SERVLET이라고 부른다.

```
- Servlet : Server + applet로 applet는 클라이언트 브라우저에서 jvm이 돌아가는 기술이다.
            현재는 Servlet은 대부분 사용하지 않고 Spring Framework에서 Controller 역할로 사용한다.

- JAVA에서의 웹 컴포넌트: SERVLET

- JAVA에서의 웹 컨테이터: Servlet Container
```



- Servlet Container의 예시 : Tomcat 등등

Spring Container



> **Spring은 Servlet Container를 대체하는 것이 아니라, Servlet Container 뒤에 존재하면서, 기능을 담당하는 Component(BEAN)들을 Servlet을 통해서 웹으로 들어온 요청을 받아서 Spring Container에게 넘겨주면, 어느 빈이 요청을 처리할 지 결정한다.**

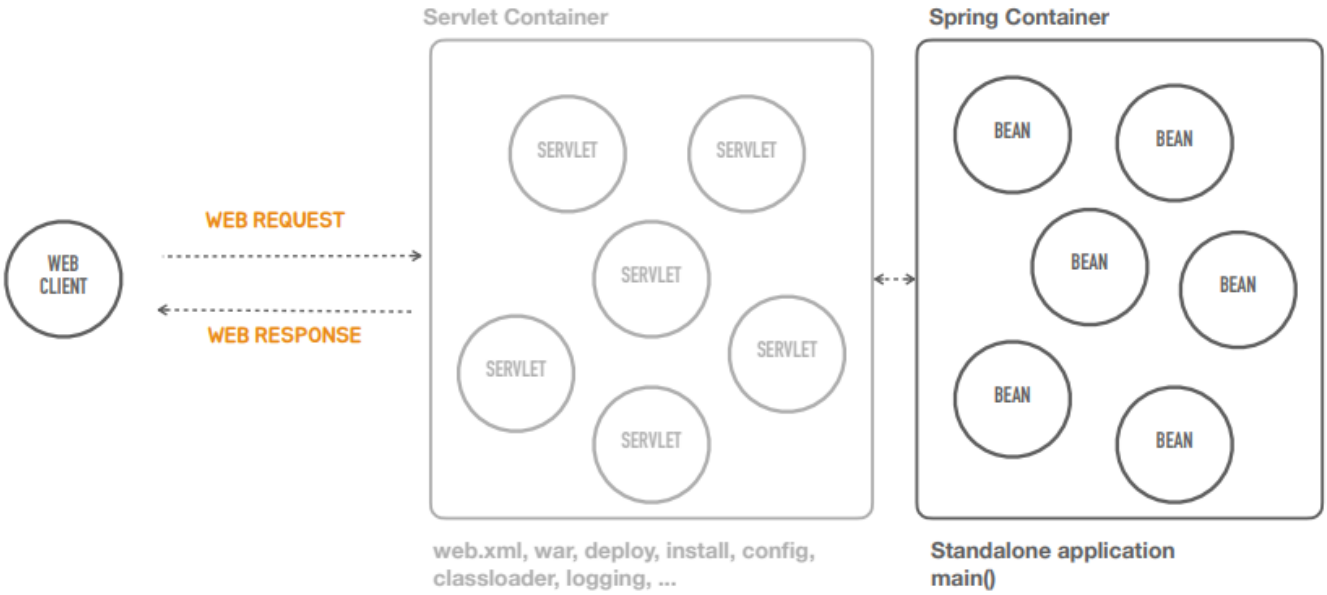
😨 그렇다면 Spring Container가 아예 Servlet Container를 대체하면 안 될까?

- 기본적으로 자바의 표준 웹 기술을 사용하기 위해서는 Servlet Container가 있어야 한다.

😥 Spring에서 Servlet Container 하나를 띄우기 위해서는...

- 스프링에서 Servlet Container 하나를 띄우기 위해서, web.xml을 설정해줘야 서블릿 컨테이너가 작동하고, 서블릿 컨테이너를 통해서 스프링 컨테이너가 호출 된다.
- 또한, Servlet Container 위에 돌아가는 애플리케이션은 확장자가 .war 형식의 파일로 빌드가 되어야 하고, 폴더 구조(구성)도 맞춰야 한다.
- Servlet Container는 독립적인 서버 프로그램이므로, war파일을 배포해주어야 한다.
- Servlet Container의 port, class loader, logging 등등 많은 정보를 설정해야 한다.

Containerless web application architecture



> **Containerless는 Servlet Container가 필요는 하지만(동작을 해야 되지만), 이것을 설치하고 관리하고 신경쓰기 위해서 개발자가 시간을 들이고, 지식을 학습하고 적용하는 수고를 없이 할 수 있었으면 좋겠다에서 출발한 개념이다.**

😨 Containerless가 적용된다면?

1. 실제로는 Servlet Container가 동작하지만, web.xml, war 등이 프로젝트에서 알아서 스프링부트를 통해서 제공된다.
2. 개발자는 이런 부분을 신경쓰지 않고 일단 개발을 시작하고 서버를 띄우고 동작을 시킬 수 있다.
3. 서블릿 컨테이너를 동작 시키기 위해 설치하고 실행하는 작업 조차도 제거하고 main 메서드 호출 하나로 전체가 다 동작가능하게 개발할 수 있다.

> 이렇게 main 메서드 호출 하나로 전체가 다 동작하는 방식으로 개발할 수 있는 방식을 독립실행형 어플리케이션(standalone application)이라고 한다.

😱 스프링 부트가 Containerless 특성을 어떻게 갖출 수 있었을까?

1-2. 강한 주장을 가진(Opinionated) 도구

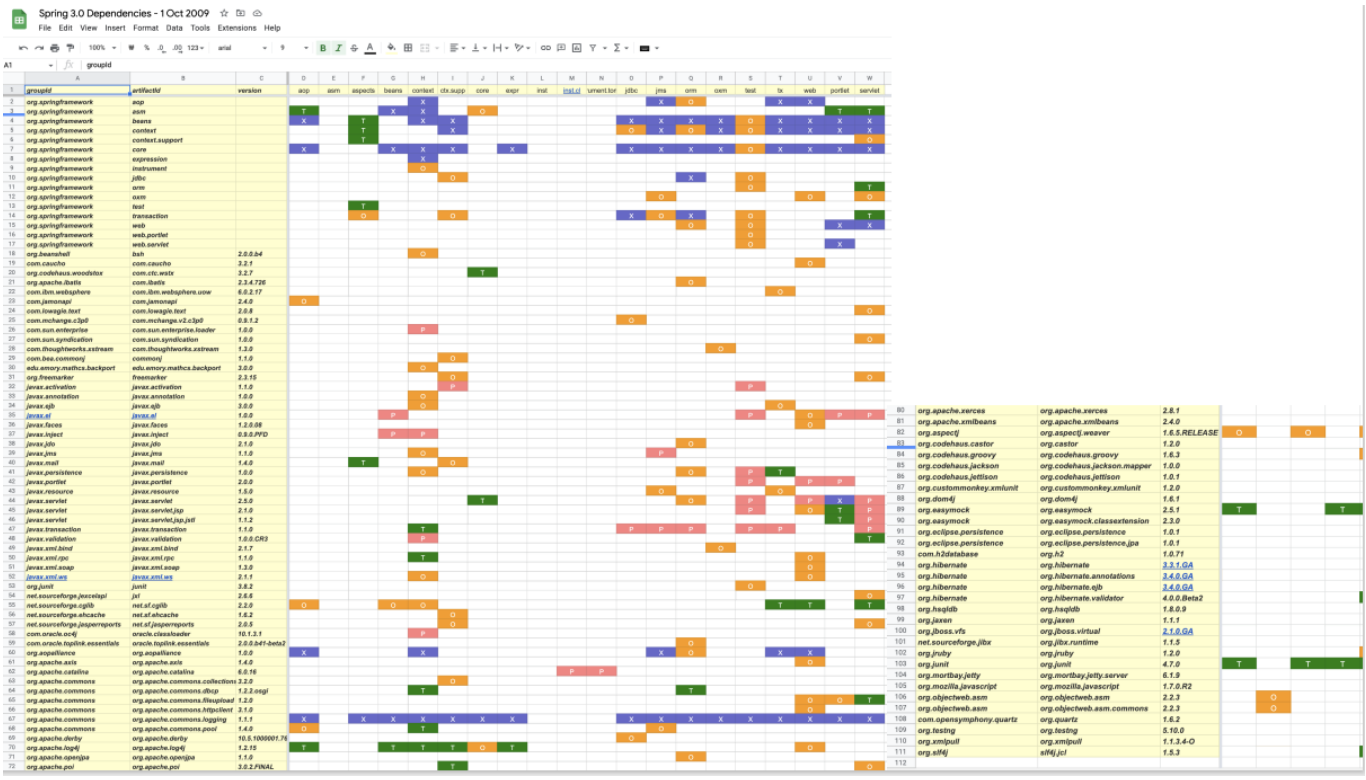
스프링 부트가 Containerless 특성을 갖출 수 있었던 것은 강한 주장을 가진(Opinonated) 도구이기 때문이다.

스프링 프레임워크의 설계 철학을 보면, 스프링 부트가 왜 강한 주장을 가진 도구인지 알 수 있다.

✔ 스프링 프레임워크의 설계 철학

1. 극단적인 유연함 추구
2. 다양한 관점을 수용
3. Not opinionated
4. 수많은 선택지를 다 포용

그러나, 이러한 스프링 프레임워크의 설계 철학으로 인해, 사용 기술과 라이브러리를 선택하는 고민을 스프링 프레임워크를 사용하는 개발자들이 직접 다 해야하는 불편함이 있다. (많은 시간 소요)



- Spriong core framework안의 각 모듈들이 사용하는 라이브러리에 대한 정보 예시

✔ 스프링 부트의 설계 철학

```
1. Opionionated - 자기 주장이 강한, 자기 의견을 고진하는, 독선적인
2. 일단 정해주는 대로 빠르게 개발하고 고민은 나중에
3. 스프링을 잘 활용하는 뛰어난 방법을 제공
```

스프링 부트의 경우, 빠르게 시작할 수 있고, 광범위한 스프링 관련 기술들을 포용하고 있다.

✔ 스프링부트가 결정해주는 것은?

1. 사용 기술과 의존 라이브러리 결정

- 업계에서 검증된 스프링 생태계 프로젝트, 표준 자바 기술, 오픈소스 기술의 종류와 의존관계, 사용버전을 정해줌
- 각 기술을 스프링에 적용하는 방식(DI 구성)과 디폴트 설정값 제공

```
🐵 : 스프링부트가 강한 주장을 가지고 사용 기술과 의존 라이브러리를 미리 결정해줌으로써 사용이 편리한 건 좋으나, 고급스럽게 확장해서 나아갈 때, customizing에 있어서 어려움이 있지 않을까?

🙈 : 스프링 부트는 유연한 확장을 통해 개발자가 다른 옵션을 선택하고 싶을 시, 간단하게 원하는 옵션을 반영할 수 있도록 지원한다.
```

2. 유연한 확장

- 스프링 부트에 내장된 디폴트 구성을 커스터마이징 하는 매우 자연스럽고 유연한 방법을 제공
- 스프링 부트가 스프링을 사용하는 방식을 이해한다면 언제라도 스프링 부트를 제거하고 원하는 방식으로 재구성 가능
- 스프링 부트처럼 기술과 구성을 간편하게 제공하는 나만의 모듈 작성

스프링 부트의 사용 기술과 의존 라이브러리 결정과 유연한 확장으로 사용자들이 사용하기 편리해졌지만, 이로 인해 스프링 부트를 이용한 개발의 오해와 한계가 생기기도 한다.

✔ 스프링 부트를 이용한 개발의 오해와 한계

- 어플리케이션 기능 코드만 잘 작성하면 된다.
- 스프링을 몰라도 개발을 잘 할 수 있다.

- 스프링 부트가 직접적으로 보여주지 않는 것은 몰라도 된다.
- 뭔가 기술적인 필요가 생기면 검색을 해서 해결한다.

따라서, 스프링 부트의 오해와 한계를 넘어서기 위해서는 스프링 부트가 스프링의 기술을 어떻게 활용하는지, 스프링부트가 선택한 기술, 자동으로 만들어 주는 구성, 디폴트 설정이 어떤 것인지 등을 알아야 한다.

"프레임워크를 효과적으로 재사용하기 위해서는 프레임워크의 최종 모습뿐만 아니라 현재의 모습을 띠게 되기까지 진화한 과정을 살펴 보는 것이 가장 효과적이다. 프레임워크의 진화 과정 속에는 프레임워크의 구성 원리 및 설계 원칙, 재사용 가능한 컨텍스트와 변경 가능성에 관련된 다양한 정보가 들어 있기 때문이다. "

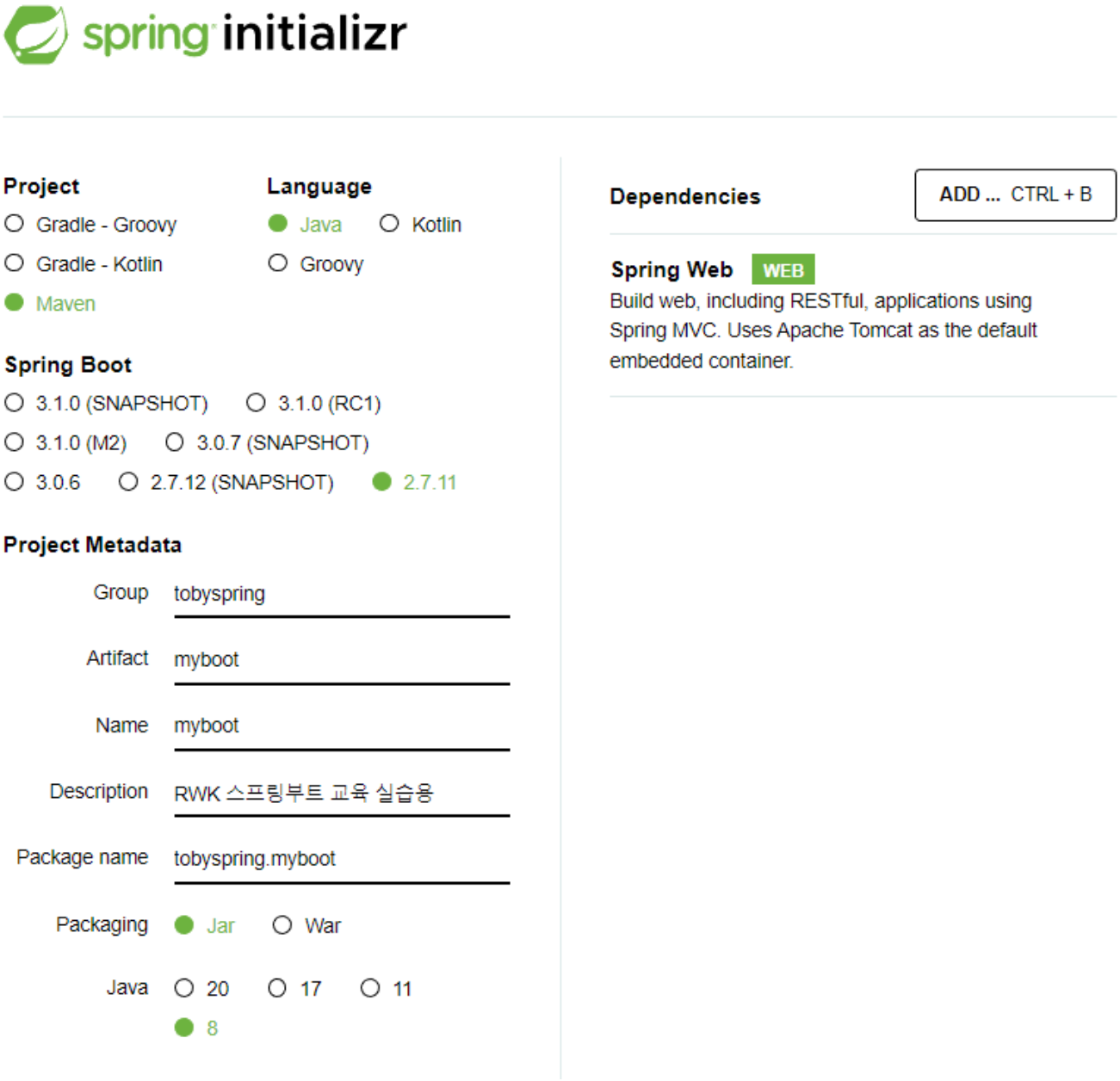
- 조영호 (프레임워크 3부)

2. Getting Started with Spring boot

2-1. Hello API 테스트

✔ 개발 환경 구성

Spring boot 실습을 위해, spring initializr에서 다음과 같이 구성



✔ Hello API 테스트

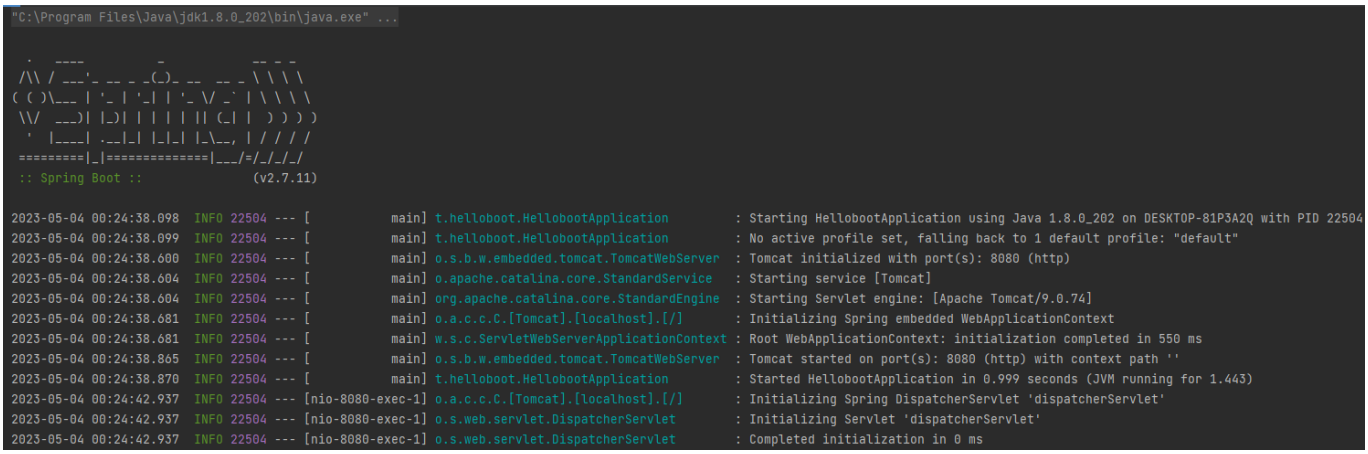
- HellobootApplication.java

```java
package tobyspring.helloboot;

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

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

- HelloController.java

```java
package tobyspring.helloboot;

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

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(String name){
        System.out.println("HelloController >> ");
        System.out.println("input name : "+name);
        return "Hello"+name;
    }
}
```

✔ HellobootApplication Start



✔ WEB에서 GET 방식으로 REQUEST 요청

```
http://localhost:8080/hello?name=Spring
```

✔ Spring console 출력 결과

```
HelloController >>
input name : Spring
```

✔ 화면 Body 출력 결과

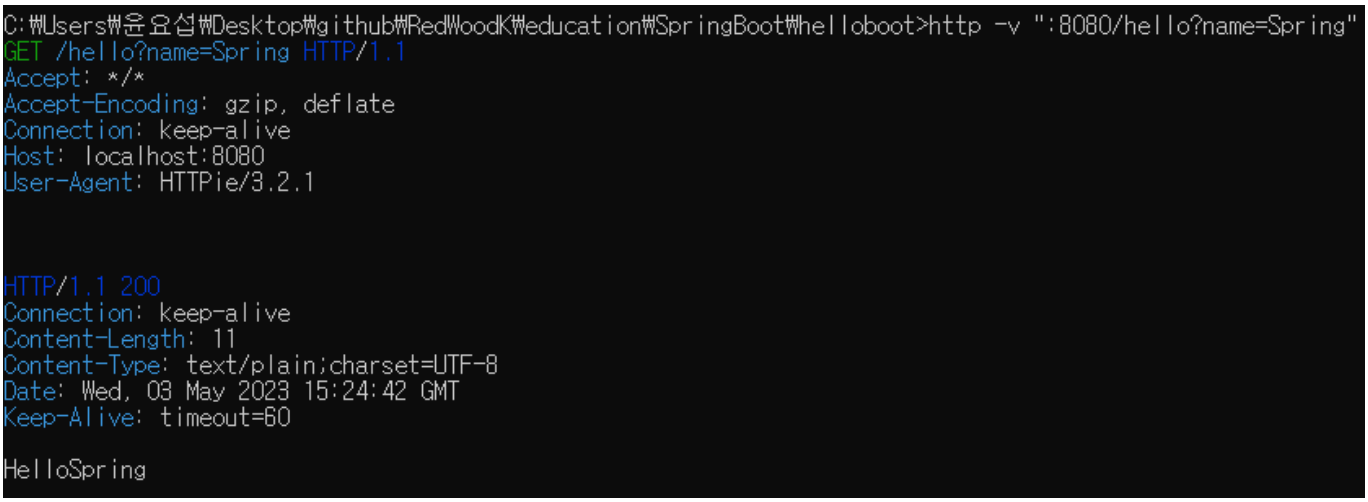
```
HelloSpring
```

2-2. HTTP 요청과 응답

- 사용한 테스트 도구 : httpie

```
httpie는 python에서 개발된 유틸리티로 http 개발이나 디버깅 용도로 사용된다. 사용성이 쉬우면서 json이 내장되어 있다. 가독성이 뛰어나며 기타 장점들이 있음(윈도우에서는 pip으로 설치)
```

- 출처 및 설치 방법(Windows): luji 블로그



✨ HTTP 통신 관련하여 요청, 응답 요소

```
Request
    - Request Line : Method, Path, HTTP Version
    - Headers
    - Message Body
Response
    - Status Line: HTTP Version, Status Code, Status Text
    - Headers
    - Message Body
```

✨ HTTP 통신 관련하여 요청, 응답 요소 상세 내용

```
Request:
    - Method :
        - GET :
            - 웹 브라우저 주소 표시줄에 사용자가 입력한
              내용이 표시됨
            - 요청 데이터를 header에 전송
        - POST :
            - 웹 브라우저 주소 표시줄에 내용 표시 안 됨
            - 요청 데이터를 body에 전송
Response:
    - Stauts Code:
        - 200 : 서버에서 잘 동작을 해서 결과를 잘 만들어서 클라이언트에게 전달할 때
        - 404 : 요청한 정보에서 요청한 파일을 찾지 못했을 때
```

```
        - 500 : 요청한 정보를 파일을 잘 찾았는데 서버에서 에러가 났을 때
    - Message Body:
        -  Message Body 들어가는 데이터 타입을 HTTP Header Content-type 필드에 적어준다.
        -  Content-type : Content-Type 개체 헤더는 리소스의 media type을 나타내기 위해 사용된다.
            - 1. media-type : 리소스 혹은 데이터의 MIME type.
            - 2. charset : 문자 인코딩 표준
```

- HTTP 프로토콜 더 자세히 알아보기

3. Standalone application

✔ 스프링 부트에서 @RestController와 @GetMapping("/hello")으로 HttpRequest 파라미터를 받아서, HelloSpring까지 출력하는 테스트를 했는데, 해당 원리에 대해 더 자세히 알기 위해, Servlet으로 실습을 해보도록 한다.

3-1. 독립 실행형 서블릿 애플리케이션 실습

3-1-1. 빈 서블릿 컨테이너 띄우기

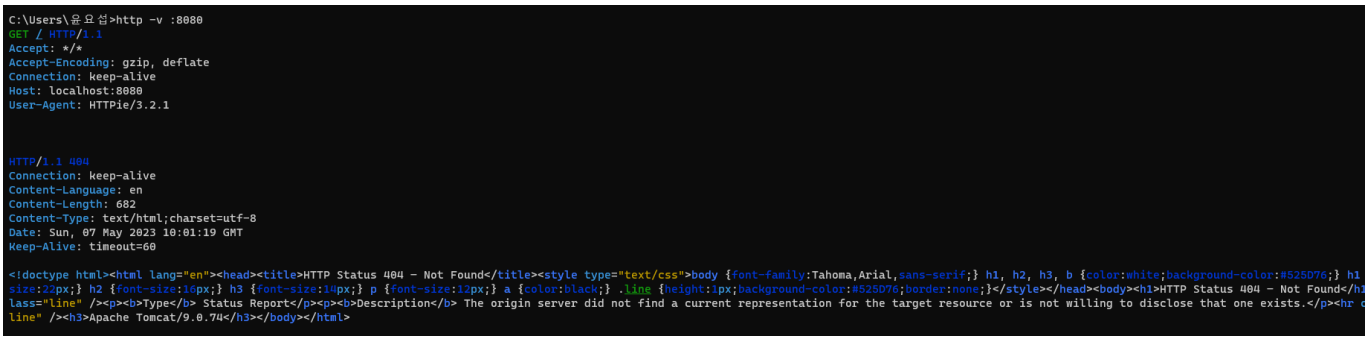
```java
package tobyspring.helloboot;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServer;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;

public class HelloBootApplication {
    public static void main(String[] args) {
        // Servlet Container 띄우기
        ServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();
        WebServer webServer = serverFactory.getWebServer();
        webServer.start();
    }
}
```

- 실행시 console 출력 결과

```
...
19:00:02.118 [main] INFO org.springframework.boot.web.embedded.tomcat.TomcatWebServer - Tomcat initialized with port(s): 8080 (http)
5월 07, 2023 7:00:02 오후 org.apache.coyote.AbstractProtocol init
정보: Initializing ProtocolHandler ["http-nio-8080"]
5월 07, 2023 7:00:02 오후 org.apache.catalina.core.StandardService startInternal
정보: Starting service [Tomcat]
```

```
5월 07, 2023 7:00:02 오후 org.apache.catalina.core.StandardEngine startInternal
정보: Starting Servlet engine: [Apache Tomcat/9.0.74]
5월 07, 2023 7:00:02 오후 org.apache.coyote.AbstractProtocol start
정보: Starting ProtocolHandler ["http-nio-8080"]
19:00:02.303 [main] INFO
org.springframework.boot.web.embedded.tomcat.TomcatWebServer - Tomcat started on
port(s): 8080 (http) with context path ''
```

- http 테스트 결과



- 자원을 찾지 못했을 뿐, 톰켓 서버까지는 잘 뜬 것을 확인 할 수 있다.

3-1-2. Servlet 등록 후, Servlet Mapping한 뒤, 테스트하기

- 서블릿 컨테이너가 웹 클라이언트로부터 요청을 받으면 어떤 서블릿한테 기능을 수행하게 할지 결정하는 것을 Mapping을 한다라고 한다.

> **/hello 요청 들어왔을 때, Hello Servlet으로 response하는 코드 작성**

```
package tobyspring.helloboot;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServer;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;

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


public class HelloBootApplication {
    public static void main(String[] args) {
        // Servlet Container 띄우기
        ServletWebServerFactory serverFactory = new
TomcatServletWebServerFactory();
        // ServletWebServerFactory :: getWebServer() : 서블릿 컨테이너에 서블릿 등록
하는데 필요한 작업 수행
```

```
        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("hello", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
                    String name = req.getParameter("name");

                    // 응답 코드 (Status Code)
                    resp.setStatus(HttpStatus.OK.value()); // 200
                    // Headers
                    resp.setHeader(HttpHeaders.CONTENT_TYPE,
MediaType.TEXT_PLAIN_VALUE); // "Content-Type", "text/plain"
                    // Message Body
                    resp.getWriter().println("Hello Servlet"+name);
                }
            }).addMapping("/hello");
        });
        webServer.start();
    }
}
```

http 테스트 결과

```
C:\Users\윤요섭>http -v :8080/hello?name="Spring"
GET /hello?name=Spring HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 200
Connection: keep-alive
Content-Length: 21
Content-Type: text/plain;charset=ISO-8859-1
Date: Sun, 07 May 2023 10:39:22 GMT
Keep-Alive: timeout=60

Hello ServletSpring
```

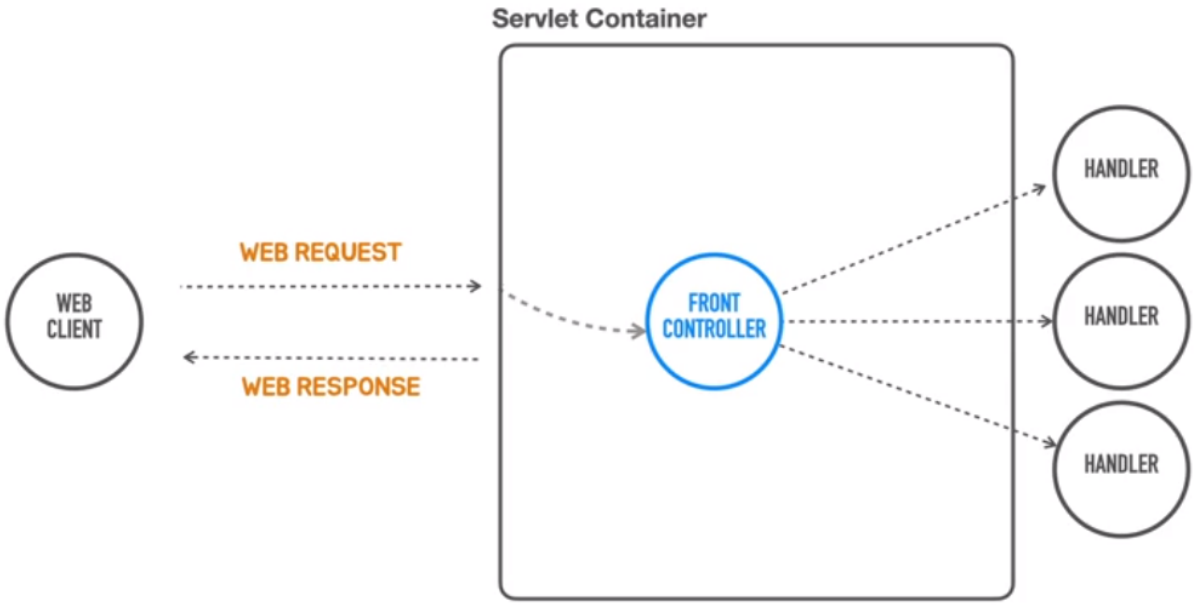
- Servlet Mapping한 뒤, HttpServletRequest의 파라미터를 받아서, HttpServletResponse에서 세팅한 대로 잘 출력 된 것을 확인할 수 있다.

3-1-3. Front Controller

1. 서블릿이 여러 개일 때, 서블릿 컨테이너로 부터 직접 요청 받아서 모든 작업을 수행하고 리턴하다보면, 대부분의 서블릿에서 필요로 하는 공통적인 작업이 중복되어 생긴다.

2. 서블릿은 웹 요청과 응답을 직접적으로 다뤄줘야 하는 방식이기 때문에 기본적인 서블릿의 기능만 가지고 개발하는데 한계가 있다.

이를 개선하기 위해 **FRONT CONTROLLER**의 개념이 등장한다.



😲 Front Controller가 등장하면서 처리 방식이 어떻게 달라졌는지?

✔ before

기존의 Servlet은 각 URL에 맞게 매핑을 해서 각각의 Servlet이 다른 URL을 맡아서 처리하는 방식

✔ after

모든 서블릿의 공통적인 작업을 중앙화된 Front Controller 오브젝트에서 처리하고 요청의 종류에 따라서 로직을 처리하는 다른 오브젝트한테 요청을 다시 위임하여 전달하는 방식

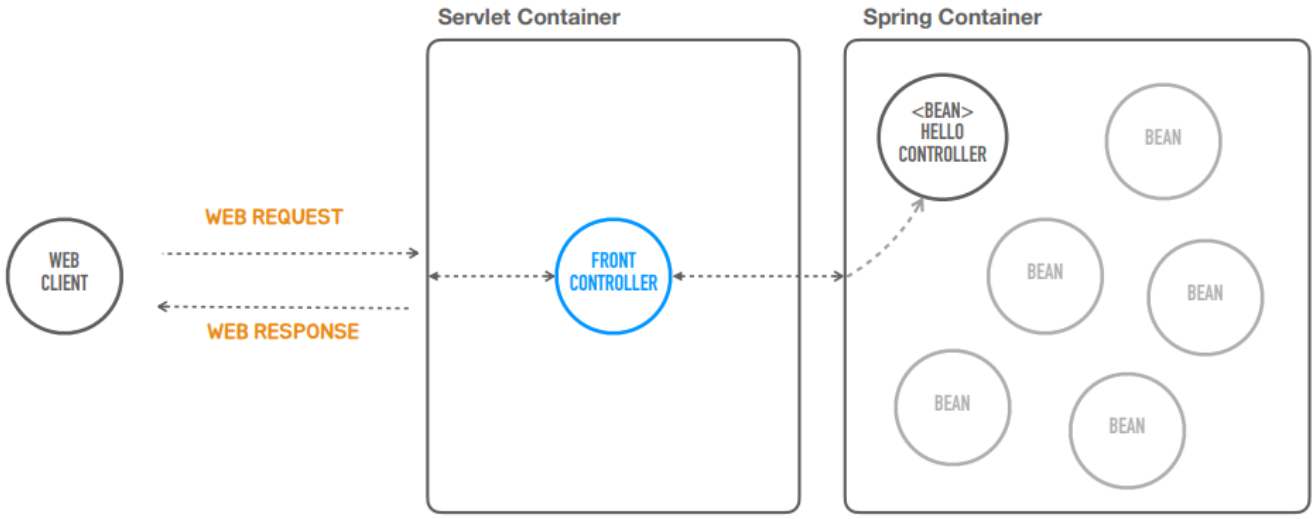
- Front Controller 활용 예제

```
...
public class HelloBootApplication {
    public static void main(String[] args) {
        ServletWebServerFactory serverFactory = new
TomcatServletWebServerFactory();
        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("frontcontroller", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
                    // URI: /hello, GET METHOD 일 때만, 정상적인 응답 코드로 반환해준
다.
                    if(req.getRequestURI().equals("/hello") &&
req.getMethod().equals(HttpMethod.GET.name())){
                        String name = req.getParameter("name");
                        resp.setStatus(HttpStatus.OK.value()); // 200
```

```
                        resp.setHeader(HttpHeaders.CONTENT_TYPE,
MediaType.TEXT_PLAIN_VALUE); // "Content-Type", "text/plain"
                        resp.getWriter().println("Hello"+name);
                    }else{
                        resp.setStatus(HttpStatus.NOT_FOUND.value()); // 404
                    }
                }
            }).addMapping("/*");
        });
        webServer.start();
    }
}
```

3-2. 독립 실행형 스프링 애플리케이션

Servlet Container의 Front Controller에서 Spring Container에 있는 BEAN을 등록하고 가져와서 사용할 수 있다.



3-2-1. ApplicationContext::getBean()

스프링에서 스프링 컨테이너를 대표하는 인터페이스인 ApplicationContext를 구현한 GenericApplicationContext를 이용하여 빈을 등록하고, ApplicationContext의 getBean() 메소드를 통해 컨테이너가 관리하는 빈 오브젝트를 가져올 수 있다.

```
GenericApplicationContext applicationContext = new GenericApplicationContext();
applicationContext.registerBean(HelloController.class);
applicationContext.refresh();
```

- 컨테이너에 필요한 정보를 등록하고 refresh()를 이용해서 초기화 작업을 진행한다.

```
HelloController helloController =
applicationContext.getBean(HelloController.class);
String ret = helloController.hello(name);
```

- 빈의 타입(클래스, 인터페이스) 정보를 이용해서 해당 타입의 빈을 요청한다.

> 스프링 애플리케이션에서는 싱글톤 패턴과 유사하게 애플리케이션이 동작하는 동안 딱 **하나의 오브젝트만**을 만들고 사용되게 만들어 준다. 이런 면에서 스프링 컨테이너는 **싱글톤 레지스트리**라고 한다.

3-2-2. Dependency Injection, Assembler

✔ HelloBootApplication 클래스에서 HelloController의 Bean을 받아 hello 함수를 호출하면, HelloService 인터페이스를 구현한 SimpleHelloService 클래스의 sayHello 메소드 호출해보는 코드를 작성해보자.

- HelloController.java

```
package tobyspring.helloboot;

import java.util.Objects;


public class HelloController {

    private final HelloService helloService;

    public HelloController(HelloService helloService){
        this.helloService = helloService;
    }

    public String hello(String name){
        // HelloService helloService = new SimpleHelloService();
        // objects.requireNonNull(name) : name이 null인지 check하여 null일 경우 예외 처리, null이 아닐 경우 그대로 return
        return helloService.sayHello(Objects.requireNonNull(name));
    }
}
```

- HelloService.java

```
package tobyspring.helloboot;

public interface HelloService {
    public String sayHello(String name);
}
```

- SimpleHelloService.java

```
package tobyspring.helloboot;

public class SimpleHelloService implements HelloService{
    @Override
```

```
    public String sayHello(String name){
        return "Simple Hello" + name;
    }
}
```

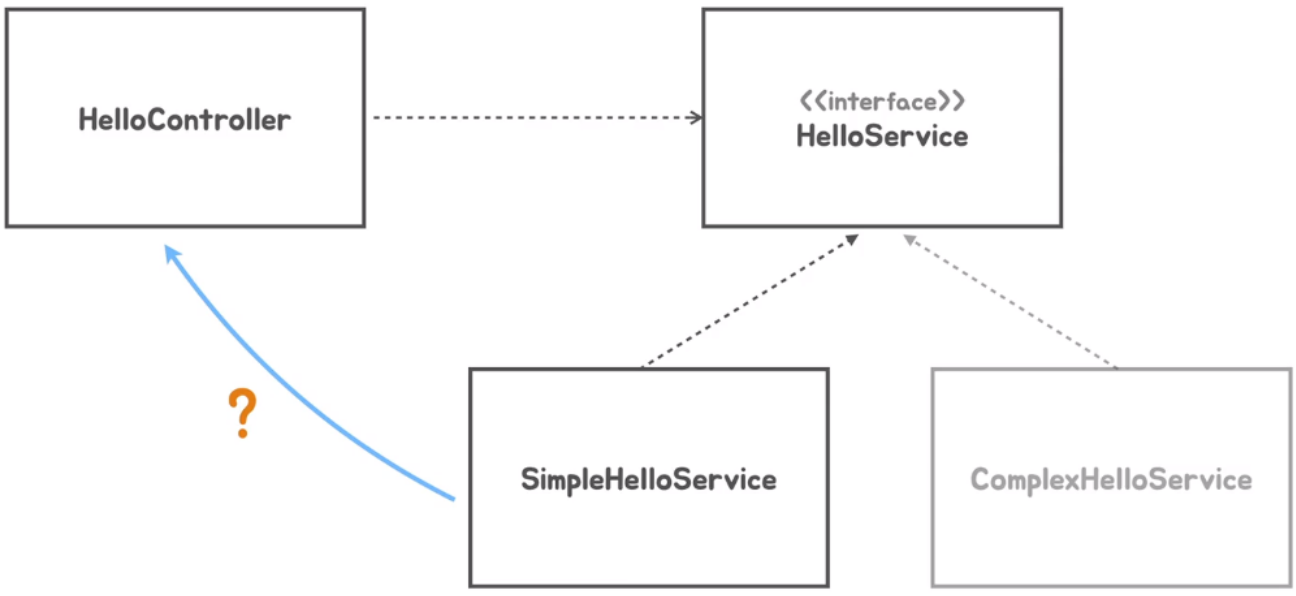
- ComplexHelloService

```
package tobyspring.helloboot;

public class ComplexHelloService implements HelloService{
    @Override
    public String sayHello(String name) {
        return "Complex Hello" + name;
    }
}
```

이런 상황에서 HelloController는 소스코드 상에서는 HelloService를 구현한 클래스와 직접 의존하고 있지 않지만, 문제는 어느 클래스의 오브젝트를 사용할 것인지 결정되어 있지 않다.

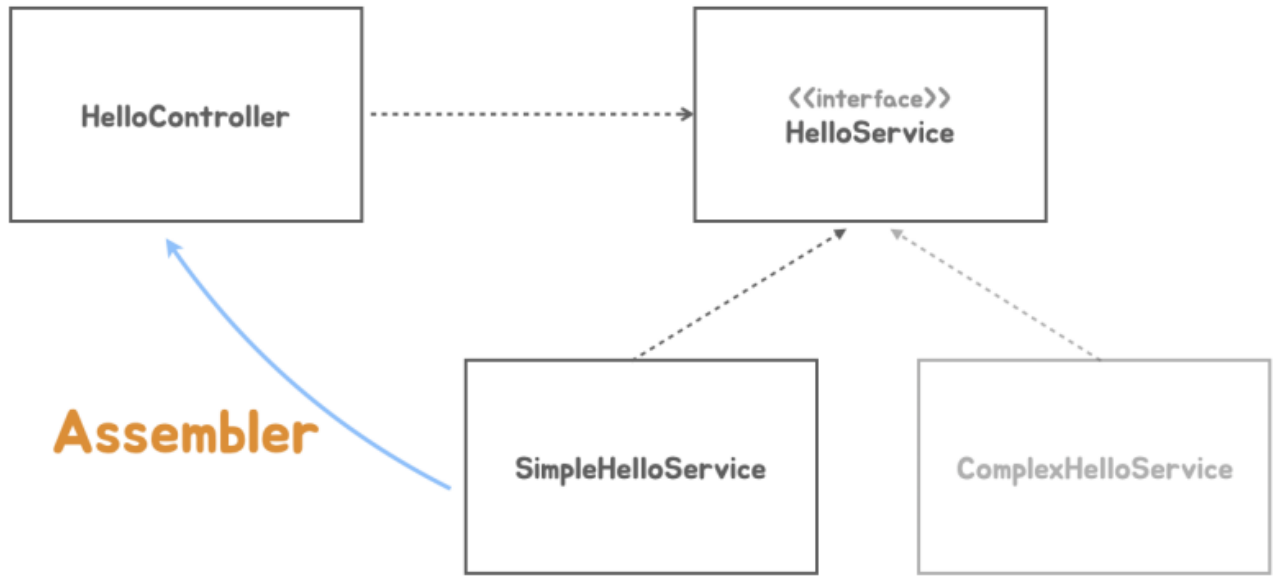
- console 출력 결과 에러

```
...
nested exception is
org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying
bean of type 'tobyspring.helloboot.HelloService' available: expected at least 1
bean which qualifies as autowire candidate. Dependency annotations: {}
...
```



따라서, HelloController와 SimpleHelloService의 연관관계를 어떻게든 이어주어야 한다.

> **이 작업을 해주는 과정을 Dependency Injection이라고 부른다.**



DI에는 두 개의 오브젝트가 동적으로 의존관계를 가지는 것을 도와주는 제 3의 존재가 필요한데, 이를 **어셈블러**라고 한다.

스프링 컨테이너는 DI를 가능하게 해주는 어셈블러로 동작한다.

✔ **SimpleHelloService.class를 Bean으로 등록하고 실행해보기**

- HelloBootApplication.java

```
applicationContext.registerBean(SimpleHelloService.class);
```

```
C:\Users\윤요섭>http -v GET ":8080/hello?name=Spring"
GET /hello?name=Spring HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 200
Connection: keep-alive
Content-Length: 20
Content-Type: text/plain;charset=ISO-8859-1
Date: Sun, 07 May 2023 12:41:10 GMT
Keep-Alive: timeout=60

Simple HelloSpring
```

첫 시작 부분인 HelloBootApplication.java에 SimpleHelloService.class를 Bean에 등록하면 SimpleHelloService 클래스의 sayHello메서드가 호출 되는 것을 확인 할 수 있다.

✔ 의존 오브젝트 DI 적용

😲 DI 어셈블러인 컨테이너가 정상적으로 주입하여 원하는 결과를 출력하였는데, 이 과정에서 어떻게 컨테이너가 주입을 할 수 있었을까?

- 다시 한 번, HelloController.java 들여다보기

```
public class HelloController {

    private final HelloService helloService;

    public HelloController(HelloService helloService){
        this.helloService = helloService;
    }
    ...
```

- 생성자 주입

```
의존 오브젝트를 생성자를 통해서 DI 어셈블러인 컨테이너가 주입을 할 수 있게 생성자 파라미터 정의한다. 주입 받은 오브젝트는 내부 멤버 필드로 저장해두고 이용할 수 있게 한다.
```

@Autowired와 같은 애노테이션으로 지정하여 명시적으로 의존 오브젝트를 주입하는 정보를 컨테이너에게 제공할 수 있으나, 이는 지양해야하는 방법이다.

참고자료: 다양한 의존성 주입 방법과 생성자 주입을 사용해야 하는 이유(망나니개발자님의 블로그)

3-2-3. DispatcherServlet으로 전환

앞에서는 Front Controller라는 Servlet에 Spring Container의 Bean을 등록하여 사용했지만, 스프링에는 사실 앞에서 만들었던 프론트 컨트롤러와 같은 역할을 담당하는 DispatcherServlet이 있다.

> DispatcherServlet은 서블릿으로 등록되어서 동작하면서, 스프링 컨테이너를 이용해서 요청을 전달할 핸들러인 컨트롤러 오브젝트를 가져와 사용한다.

- DispatcherServlet으로 전환한 HelloBootApplication.java

```
public class HelloBootApplication {
    public static void main(String[] args){
        // DispatcherServlet이 사용하는 스프링 컨테이너를
        // GenericWebApplicationContext를 이용해서 작성
        GenericWebApplicationContext applicationContext = new
GenericWebApplicationContext(){
            @Override
            protected void onRefresh(){
                super.onRefresh();
```

```
                ServletWebServerFactory serverFactory = new
TomcatServletWebServerFactory();
                WebServer webServer = serverFactory.getWebServer(servletContext ->
{
                    servletContext.addServlet("dispatcherServlet", new
DispatcherServlet(this))
                            .addMapping("/*");
                });
                webServer.start();
            }
        };
        applicationContext.registerBean(HelloController.class);
        applicationContext.registerBean(SimpleHelloService.class);
        applicationContext.refresh();
    }
}
```

3-3. 스프링 애플리케이션 Web 살펴보기

위에서 Servlet Container부터 Dispatcher Servlet까지 복잡하게 코드를 구성하였지만, 이는 아래 코드와 같이 간략하게 작성할 수 있다.

- HelloBootApplication.java

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

- HelloController.java

```
@RestController
public class HelloController {

    private final HelloService helloService;

    public HelloController(HelloService helloService){
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    @ResponseBody
    public String hello(String name){
        return helloService.sayHello(Objects.requireNonNull(name));
    }
}
```

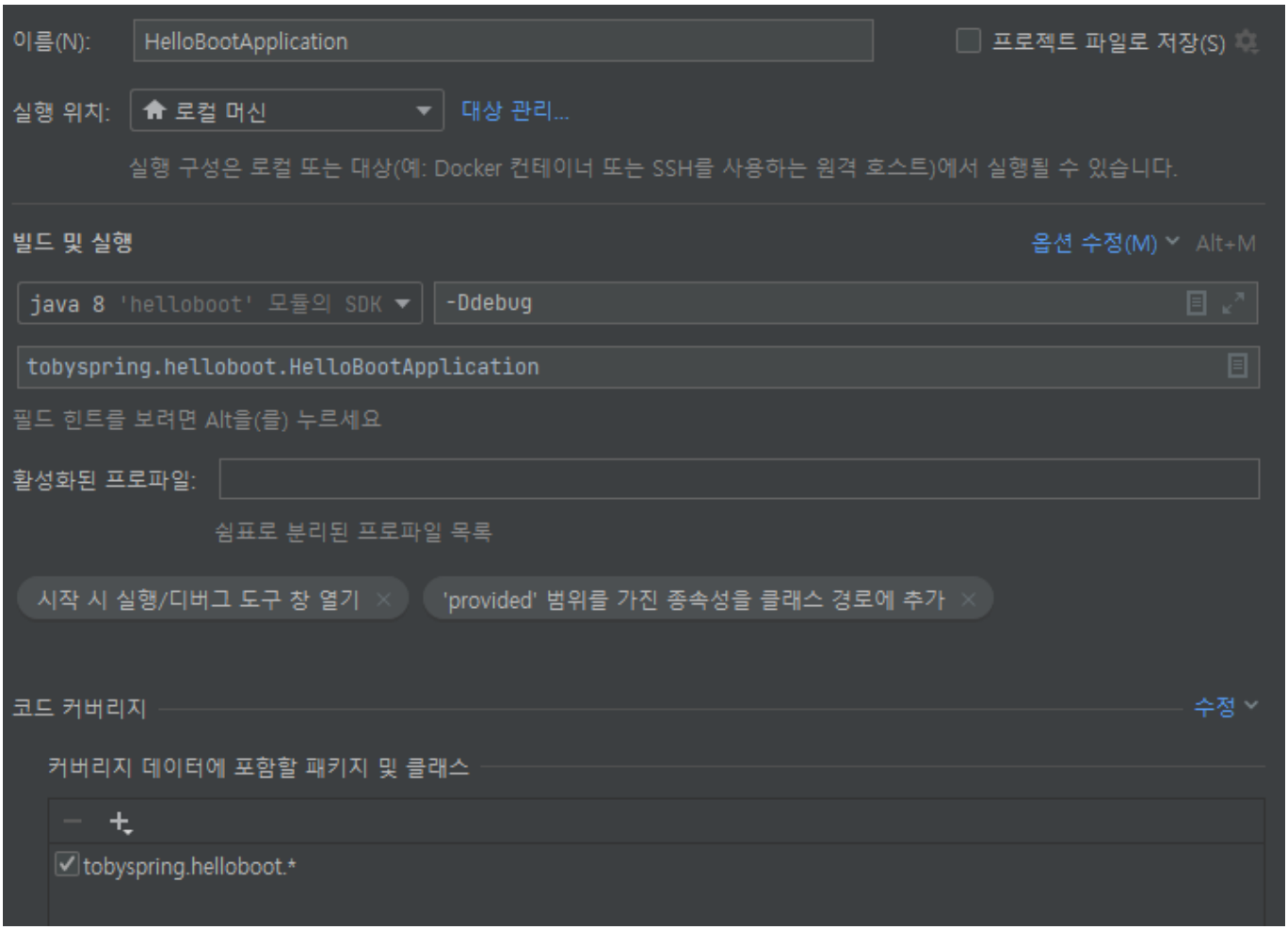
스프링의 강력한 도구의 지원과 Containerless의 특성으로 단순한 코드로 작성할 수 있게 된 것이다.

> **01.SpringBoot WEB 동작원리 편**에서는 Spring Boot의 두 가지 핵심인 Containerless와 Opinionated 특성에 대해 알기 위해, WEB의 동작 원리와 Spring Boot의 기본적인 특징들을 알아보았다.

3-3-1. 스프링 애플리케이션 Web 구성 살펴보기

그러나, 스프링 애플리케이션 Web 구성을 제대로 이해하기 위해서는 WEB의 동작방식 뿐만 아니라 스프링 WEB의 BEAN 어떻게 구성되어 있는지 확인하고 알아야 할 필요가 있다.

인텔리제이에서 VM 옵션에서 -Ddebug를 하면 CONDITIONS EVALUATION REPORT를 볼 수 있다.



✔ **CONDITIONS EVALUATION REPORT 출력 결과 보기**

```
============================
CONDITIONS EVALUATION REPORT
============================

Positive matches:
-----------------

   AopAutoConfiguration matched:
      - @ConditionalOnProperty (spring.aop.auto=true) matched
(OnPropertyCondition)

   AopAutoConfiguration.ClassProxyingConfiguration matched:
```

```
      - @ConditionalOnMissingClass did not find unwanted class
'org.aspectj.weaver.Advice' (OnClassCondition)
      - @ConditionalOnProperty (spring.aop.proxy-target-class=true) matched
(OnPropertyCondition)

   ApplicationAvailabilityAutoConfiguration#applicationAvailability matched:
      - @ConditionalOnMissingBean (types:
org.springframework.boot.availability.ApplicationAvailability; SearchStrategy:
all) did not find any beans (OnBeanCondition)

   DispatcherServletAutoConfiguration matched:
      - @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet' (OnClassCondition)
      - found 'session' scope (OnWebApplicationCondition)
    ...

Negative matches:
-----------------

   ActiveMQAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class
'javax.jms.ConnectionFactory' (OnClassCondition)

   AopAutoConfiguration.AspectJAutoProxyingConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class
'org.aspectj.weaver.Advice' (OnClassCondition)

   ArtemisAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class
'javax.jms.ConnectionFactory' (OnClassCondition)

   BatchAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class
'org.springframework.batch.core.launch.JobLauncher' (OnClassCondition)
...
```

이를 통해 매칭된 구성과 매칭되지 않은 구성에 대한 정보를 볼 수 있다.

3-3-2. 02.SpringBoot BEAN 구성원리 편에서는...

Spring Boot의 동작 방식을 이해하기 위해, WEB의 동작 원리와 Spring Boot의 기초적인 개념을 이번 편에서 알아보았지만,

스프링 부트에서 사용하는 기술과 관련된 자동 구성과 프로퍼티 등을 분석하고 어떻게 활용할 수 있는지 파악하기 위해서는 어노테이션의 활용, 조건부 자동 구성, 외부 설정을 이용한 자동 구성 등에 대한 내용도 알아야 할 필요가 있다.

따라서, **02.SpringBoot BEAN 구성원리 편**에서는 Spring Boot의 어노테이션의 활용과 자동 구성에 대해 알아보도록 한다.