03-1. 섹션 ~ 3까지 정리

D.I.Y 클래스 - 나만의 스프링 만들기

> Spring Boot 클론 코딩

목차

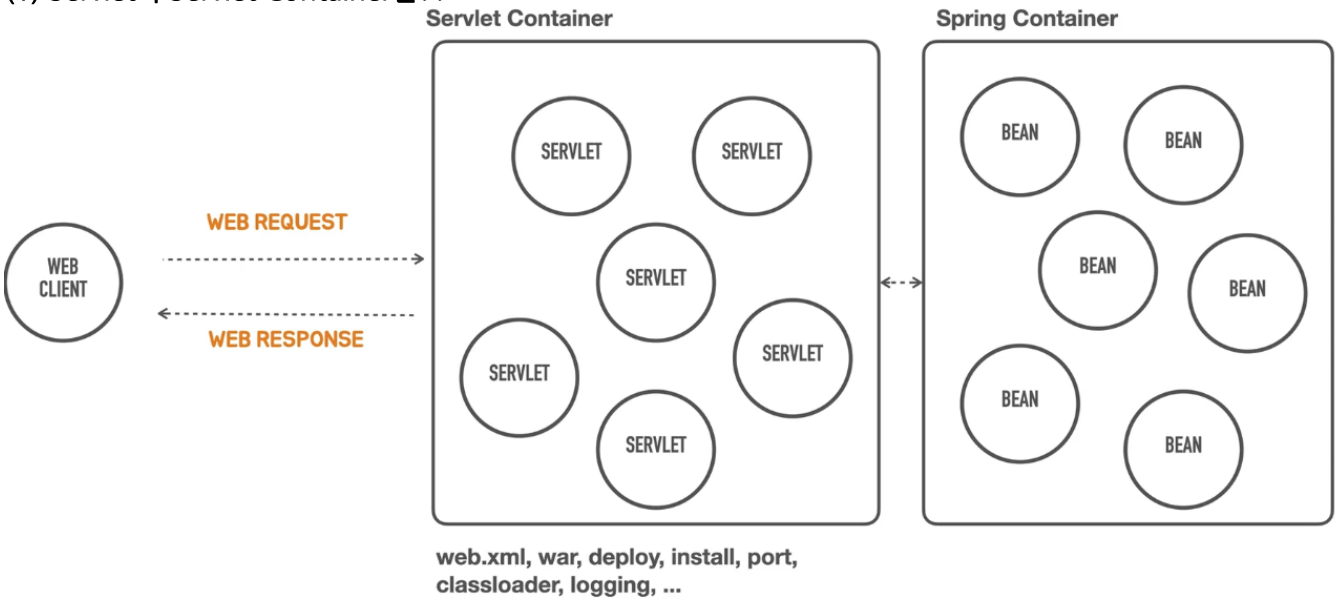
0. 강의 목표 리마인드

강의 목표

스프링 부트의 핵심 기능을 직접 구현하면서
"스프링 부트의 동작 원리"와 "스프링 부트가 스프링을 어떻게 이용하여 동작하는지" 이해하자

1. Servlet container? 아니 Servlet은 뭔데??

(1) Servlet과 Servlet Container란??



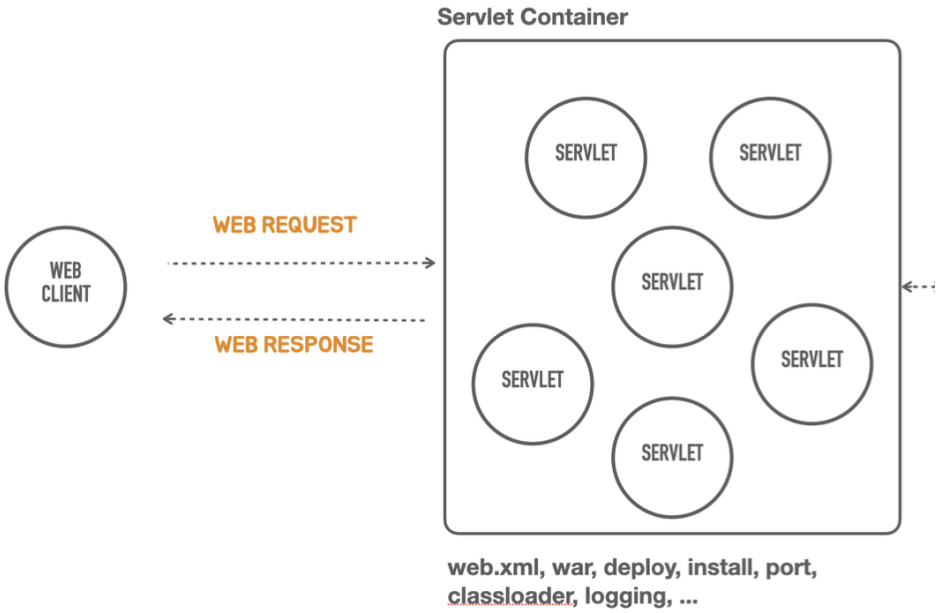
사실,,, Java에서

- Web container → Servlet container
- Web Component → Servlet

Servlet (Web Component)

- Dynamic Web Page를 만들 때 사용되는 자바 기반의 웹 애플리케이션 프로그래밍 기술
- 서블릿은 자바 클래스로 웹 애플리케이션을 작성한 뒤 이후 웹 서버 안에 있는 웹 컨테이너에서 이것을 실행
- 웹 컨테이너에서는 서블릿 인스턴스를 생성 후 서버에서 실행되다가 웹 브라우저에서 서버에 요청(Request)을 하면 요청에 맞는 동작을 수행하고 웹 브라우저에 HTTP형식으로 응답(response)함
- 하나의 서비스를 담당하는 컴포넌트, ex. 회원가입 서비스

Servlet Container (Web Container)

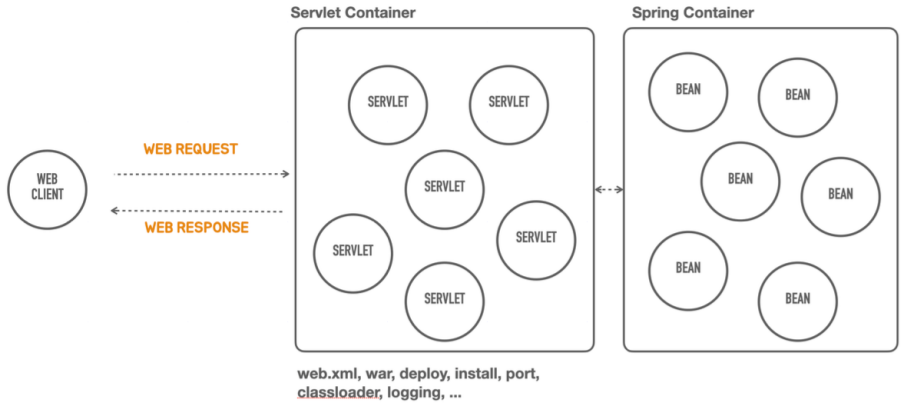


-

- 서버에 있는 web component(servlet)를 관리하는 역할
- web component를 메모리에 올리고 (ex. new로 인스턴스 생성) 서비스가 되는 동안 메모리에서 관리해주는 역할
- life cycle 관리
- web client가 요청한 작업을 룰을 따라 어느 web component에 할당할 것인지 결정(라우팅, 맵핑)

(+) Spring Container != Servlet Container

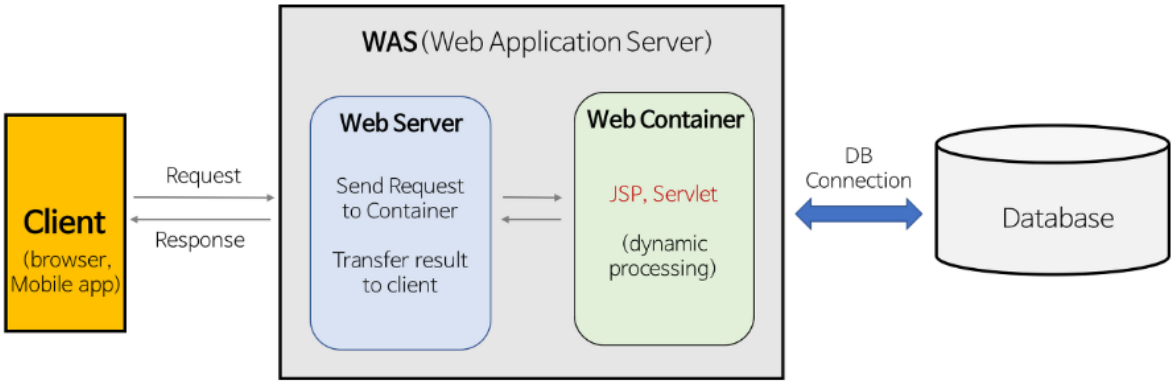
- Servlet에선 http 관련 업무(포트 설정, ...)만 하고 비즈니스 로직은 Spring을 통해 처리한다.
- Spring container는 Servlet Container 뒤쪽에 있으면서 특정 기능을 담당하는 Component(=Bean)들을 관리하고 매핑해줌



-

---

2. Containerless



WAS의 구조 및 동작

Web Server란?

- 클라이언트의 request(요청)을 받아 정적인 컨텐츠(html, css, js)를 response(응답)하는 서버.
- 예) Apache, Nginx, IIS, WebtoB 등

Web Container란? (=Servlet Container)

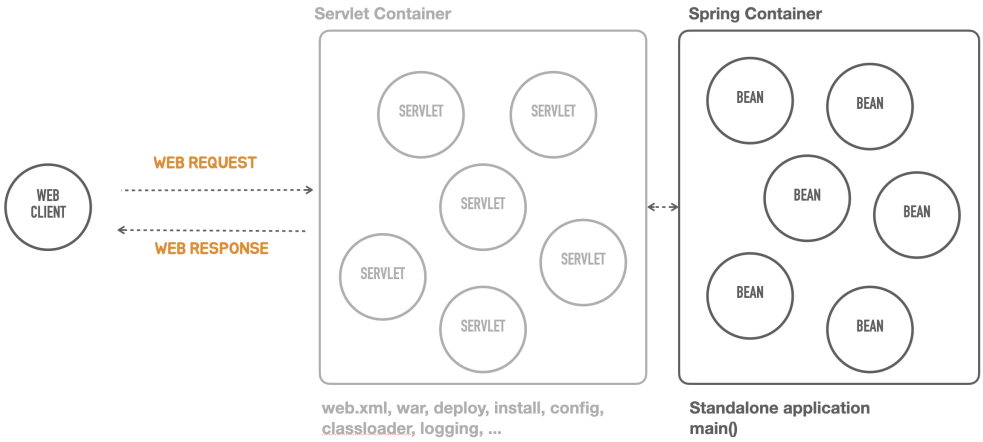
- 동적인 데이터들을 처리하여 정적인 페이지로 생성해주는 소프트웨어 모듈
- Servlet이라는 Component를 담을 수 있음
- Servlet, JSP를 실행할 수 있는 프로그램
    - Servlet : 특정 기능이나 관련 기능이 재사용 가능한 형태로 만들어진 프로그램, 특정 기능을 하는 서비스, Component
- 웹 서버가 입력 받은 정보를 통해 동적인 데이터를 처리하여 정적인 페이지 생성

- == 서블릿 컨테이너
- 동작 원리
    - 서버에서 JSP를 요청하면 컨테이너가 JSP파일을 서블릿으로 변환하여 컴파일을 수행하고, 서블릿 수행결과를 웹서버에게 전달

WAS 란?

- Web Application Server
- Web Server + Web Container를 결합한 서버
- 예) Tomcat, WebLogic, WebSphere, Jeus, JBoss 등

(2-1) Containerless



servlet container 설정의 번거로움

- spring container만 개발해서 띄우고 싶은데 servlet container도 개발해야해
    - XML파일 작성 불가피
    - 초반 작업임
- folder 구조도 맞춰야함
- servlet container는 독립적인 서버 프로그램이기 때문에 tomcat 같은걸 환경에 맞게 설치하고 실행시켜야함
    - war로 압축된 servlet container를 배포해야함
- servlet container의 설정도 많음
    - logging 등

이것들이 초반 설정 작업들이고 개발하는 동안 계속 신경써야되는 것들이 아님 -> 매번 번거로움

게다가 tomcat 말고 다른 웹서버를 쓰면 더 번거로워짐

그럼 그냥 servlet container를 없애고 spring container로 합치면 외않되?

- 기본적으로 java 표준 웹 기술 이용하기 위해선 Servlet Container가 존재해야 하므로, Spring Container가 이를 대체할 수 없음
    - 비즈니스 로직을 Spring 통해 처리하겠다는 것이지, Servlet 필요 없다는 이야기가 아님
- → 서블릿 컨테이너가 필요는 하지만, 이를 설치/관리하는 것에 들이는 수고를 줄일 수 없을까? → Containerless로 가자!!

servlet container를 신경쓸 필요 없는 web architecture를 만들어줘

따라서 servlet container가 없는 web architecture를 만들어줘! -> sevlet container를 신경쓰지 않게 해줘! == spring boot

- 독립실행형 애플리케이션 (standalone application)
    - servlet container를 초기에 띄우는 작업이 필요하지 않아? How?
    - main method를 실행하면 servlet container와 관련된 모든 작업들이 실행되는 것 (= containerless)

(2-2) Spring boot로 가기 위한 Servlet과 Servlet container를 만들어보자

servlet container를 직접 설치하지 않고 어떻게 동작하게 만들것이냐, 직접 신경쓰지 않도록 할 것이냐? 고민해보니

servlet container를 설치하는 대신 standalone으로 spring boot를 구성해서 servlet container를 알아서 띄워주게 하는 작업을 해야함

근데 servlet container 관련 작업들을 자동화하려면 일단 servlet container가 있어야하잖아?!

Servlet Container 생성

HellobootApplication.java

```
public class HellobootApplication {

    public static void main(String[] args) {

        //Tomcat servlet webserver        ->  webserver
        //ServletWebServerFactory serverFactory = new TomcatServletWebServerFactory(); ->
        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        //servlet container
        WebServer webServer = serverFactory.getWebServer();

        // Tomcat servlet container
        webServer.start();
    }

}
```

요청 결과

```
❯ http -v :8080
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 404
Connection: keep-alive
Content-Language: en
Content-Length: 682
Content-Type: text/html;charset=utf-8
Date: Tue, 11 Apr 2023 06:54:45 GMT
Keep-Alive: timeout=60

<!doctype html><html lang="en"><head><title>HTTP Status 404 – Not Found</title><style type="text/css">body
{font-family:Tahoma,Arial,sans-serif;} h1, h2, h3, b {color:white;background-color:#525D76;} h1 {font-
size:22px;} h2 {font-size:16px;} h3 {font-size:14px;} p {font-size:12px;} a {color:black;} .line
{height:1px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP Status 404 – Not Found</h1>
<hr class="line" /><p><b>Type</b> Status Report</p><p><b>Description</b> The origin server did not find a
current representation for the target resource or is not willing to disclose that one exists.</p><hr
class="line" /><h3>Apache Tomcat/9.0.73</h3></body></html>
```

Servlet Container에 servlet 등록

서블릿 컨테이너에 웹 컴포넌트(서블릿)를 넣어보자

서블릿 컨테이너가 웹 클라이언트로부터 요청을 받으면 어떤 서블릿이 요청을 처리할지 결정하게 함 == mapping

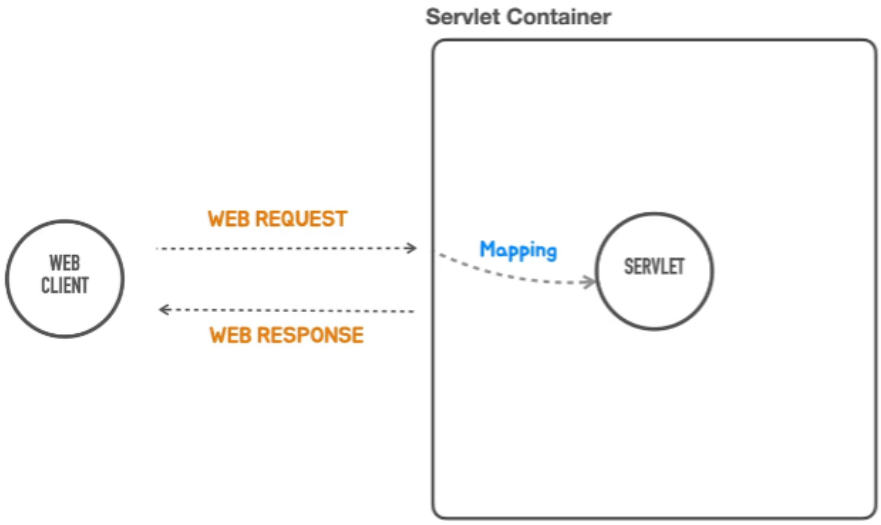
HellobootApplication.java

```
public class HellobootApplication {

    public static void main(String[] args) {

        //Tomcat servlet webserver
        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        //servlet container
        //servlet container      object
        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("hello", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
                    //    ,
                                        //HttpServletRequest     HttpServletResponse

                    String name = req.getParameter("name");

                    resp.setStatus(HttpStatus.OK.value());
                    resp.setHeader(HttpHeaders.CONTENT_TYPE, MediaType.TEXT_PLAIN_VALUE);
                    resp.getWriter().println("Hello " + name); //getWriter.print: object to string .
                }
            }).addMapping("/hello"); //mapping
        });

        // Tomcat servlet container
        webServer.start();
    }

}
```

요청 결과

```
❯ 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: 13
Content-Type: text/plain;charset=ISO-8859-1
Date: Tue, 11 Apr 2023 08:02:11 GMT
Keep-Alive: timeout=60

Hello Spring
```

지금까지,,,



- servlet container를 생성
- servlet container 안에 Hello spring을 뽑는 Servlet 생성 후 등록
- http 요청 테스트 완

근데 또 귀찮아,,,

서블릿은 요청마다 매핑이 필요

서블릿이 늘어나고 매핑받고 하다 보니까 **여러 서블릿에서 공통적으로 필요한 작업이 각 서블릿 안에 중복되는거임(불편)**

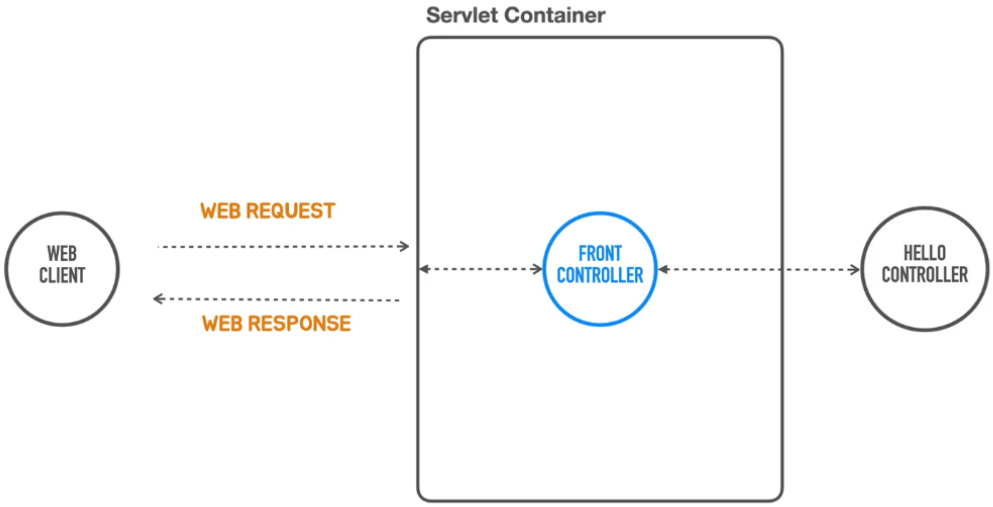
그렇다고 서블릿이 직접적으로 웹 request/ response object를 다뤄줘야하는 방식은 자연스럽지 못함

기본적인 서블릿을 가지고 개발하기 어려움이 있다고 판단

따라서 프론트 컨트롤러 등장

(2-3) 프론트 컨트롤러 도입

프론트 컨트롤러는



- 모든 서블릿에 공통적으로 등장하는 어떤 처리하는 코드를 처리하는 / 공통 로직 처리하는 프론트 컨트롤러 두고, 공통 로직 끝나면 각 부분 처리하는 개별적인 곳으로 위임
- 전처리 + 후처리 둘다 가능
    - FrontController 가 처리해주는 일반적인 공통 작업 : 인증/보안/다국어처리 등
- 서블릿을 프론트 컨트롤러로 만들기 위해, 모든 요청을 하나의 서블릿이 담당하도록 매핑해줌

HellobootApplication.java

```
public class HellobootApplication {

    public static void main(String[] args) {
        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        //servlet container
        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("frontController", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

                    //  front controller servlet     ( servlet container )
                    if(req.getRequestURI().equals("/hello") && req.getMethod().equals(HttpMethod.GET.name())){
                        String name = req.getParameter("name");

                        resp.setStatus(HttpStatus.OK.value());
                        resp.setHeader(HttpHeaders.CONTENT_TYPE, MediaType.TEXT_PLAIN_VALUE);
                        resp.getWriter().println("Hello " + name); //getWriter.print: object to string .
                    } else if (req.getRequestURI().equals("/user")) {
                        //user
                    } else{
                        resp.setStatus(HttpStatus.NOT_FOUND.value());
                    }
                }
            }).addMapping("/*"); // /        -> front controller
        });

        // Tomcat servlet container
        webServer.start();
    }

}
```

요청 결과

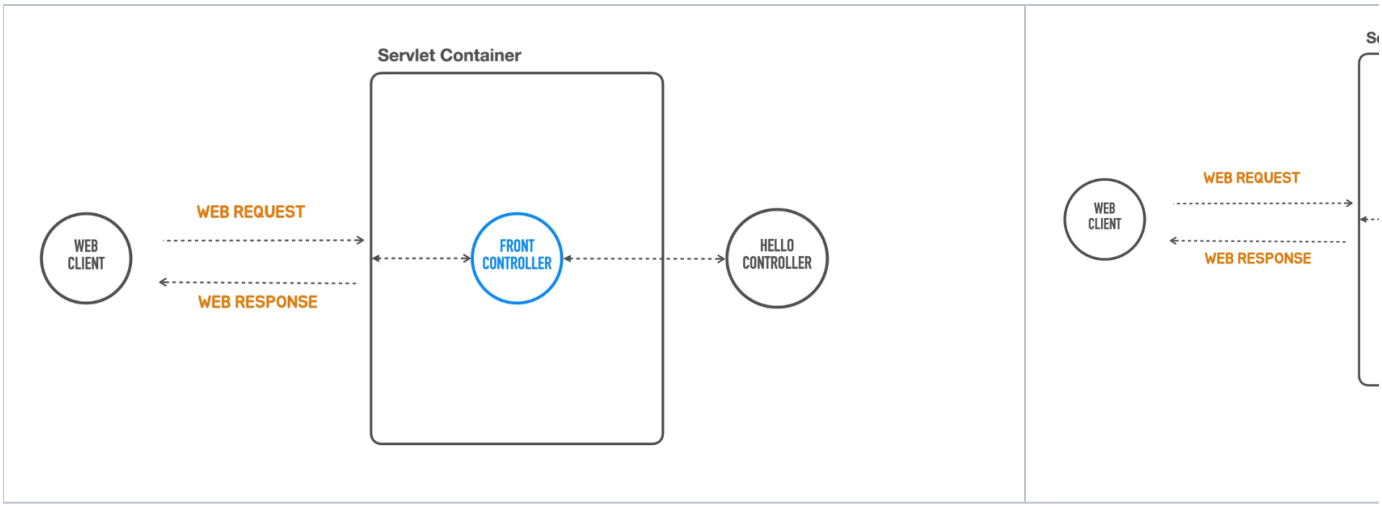
```
❯ http -v ":8080/hello?name=Spring"
GET /hello?name=Spring HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 200
Connection: keep-alive
Content-Length: 13
Content-Type: text/plain;charset=ISO-8859-1
Date: Tue, 11 Apr 2023 08:21:03 GMT
Keep-Alive: timeout=60

Hello Spring
```

```
❯ http -v POST ":8080/hello?name=Spring"
POST /hello?name=Spring HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 0
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 404
Connection: keep-alive
Content-Length: 0
Date: Tue, 11 Apr 2023 08:21:16 GMT
Keep-Alive: timeout=60
```

---

3. 독립 실행형 스프링 애플리케이션

현재 구조 : 독립 실행형 서블릿 애플리케이션	다음 목표 구조 : 독립 실행형 스프링 애



- 독립 실행형 애플리케이션
    - main() 호출 시 서블릿 컨테이너 설치/배포 등의 일련의 작업이 수행되는 애플리케이션
- Servlet에선 http 관련 업무(포트 설정, ...)만 하고 비즈니스 로직은 Spring을 통해 처리하겠다.

스프링 컨테이너는

- Spring container 안에는 두가지가 필요한데
    - POJOs : 비즈니스 로직 담고 있는 비즈니스 오브젝트, Bean으로 등록할 Java Object
    - Configuration Meta Data : 비즈니스 오브젝트들을 어떤 식으로 구성할 지에 대한 정보
- 두 개를 조합해서 빈 구성 -> 사용 가능한 시스템을 만들어 냄(서버 애플리케이션)

(3-1)실습

1. spring container 생성
2. hello controller를 spring container에 집어 넣고
3. hello controller를 직접 생성해서 사용하는 대신에 spring container한테 요청해서 가져와서 사용

HellobootApplication.java

```
public class HellobootApplication {

    public static void main(String[] args) {
        //    (   )     == application context == spring container
        // container
        GenericApplicationContext genericApplicationContext = new GenericApplicationContext();

        //
        genericApplicationContext.registerBean(HelloController.class);

        //       -> application context bean object
        genericApplicationContext.refresh();



        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        //servlet container
        WebServer webServer = serverFactory.getWebServer(servletContext -> {

            //servlet container
            servletContext.addServlet("frontController", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

                    if(req.getRequestURI().equals("/hello") && req.getMethod().equals(HttpMethod.GET.name())){
// mapping
                        String name = req.getParameter("name");

                        //
                        //              .
                        // servlet container hello controller     controller
                        HelloController helloController = genericApplicationContext.getBean(HelloController.
class);

                        //  (  )
                        String ret = helloController.hello(name);

                        //resp.setStatus(HttpStatus.OK.value()); // 200
                        resp.setContentType(MediaType.TEXT_PLAIN_VALUE);
                        resp.getWriter().println(ret); //getWriter.print: object to string .
                    }
                    else{
                        resp.setStatus(HttpStatus.NOT_FOUND.value());
                    }
                }
            }).addMapping("/*");
        });

        // Tomcat servlet container
        webServer.start();
    }

}
```

(3-2) 의존 오브젝트 추가

지금까지 한 작업들이 spring container가 할수있는 여러 중요한 일들을 적용할 수 있는 기본 구조를 짜놨다는 것에 의의가 있음

controller는 기본적으로 web controller이기 때문에 웹을 통해 들어온 어떤 요청사항을 검증하고

이를 비즈니스 로직을 수행하는 다른 오브젝트에게 요청을 보내서 결과를 돌려받고 웹 클라이언트에게 어떤 형식으로 돌려줄 것인가 정도 결정하면 됨

→ 비즈니스 로직을 수행하는 다른 오브젝트 만들고 작업을 위임함으로써 책임이 많이 줄어든다.

여기서 잠깐! 매핑과 바인딩이란??

Mapping

- 요청에 들어있는 정보 활용해서 어떤 로직 수행하는 코드 호출할 것인지 결정

Binding

- 직접적으로 웹 요청/응답 다루는 오브젝트 사용하지 않고, 평범한 자바 타입으로 (웹 요청) 정보를 변환해서 사용
- DTO나 Java bean 형태로 데이터 집어넣어서 처리하는 오브젝트에 파라미터로 넘겨주는 거 등
- (아래 예제에서) 파라미터로 넘어온 name이란 값을 string type으로 Hello 메소드 호출할 때 인자값으로 넘겨주는 작업

이전 HelloController.java

HelloController.java

```
@RestController
@RequestMapping("")
public class HelloController {
    @GetMapping("/hello")
    public String hello(String name){
        return "Hello " + name;
    }
}
```

바꾼 HelloController.java

HelloController.java

```
/**
 * controller   :     +
 * ex) name parameter  null  check
 */
public class HelloController {
    public String hello(String name){
        SimpleHelloService helloService = new SimpleHelloService();
        //if(name == null) throw NullPointerException;

        // controller
        //   service
        return helloService.sayHello(Objects.requireNonNull(name));  // Objects.requireNonNull(obj) : obj null
, null
    }
}

// spring container
```

SimpleHelloService.java

```
//
public class SimpleHelloService {
    String sayHello(String name){
        return "Hello " + name;
    }
}
```

4. Dependency Injection

(4-1)Spring DI container

이전 코드에서는

HelloController가 SimpleHelloService를 의존하고 있음.

그리고 직접 service class 구현체를 new 생성자를 통해 만들고 controller 내부에서 사용하고 있었음.

문제점

- helloController는 SimpleHelloService가 변경되면 영향을 받음
    - 즉, 의존관계에 있으면 변경사항이 있을때마다 코드를 수정해야함
    - ex) hello controller는 service 클래스가 바뀌면 클래스 선언부를 바꿔줘야함
- 분리된 service 객체가 필요한 오브젝트에서 계속해서 service를 생성하면 같은 작업을 하는 service 오브젝트가 쓸데없이 많아짐
    - 싱글톤 스타일에 맞지 않음

DI에는 제 3의 존재가 필요함 -> 어셈블러

hello controller가 사용하는 오브젝트를 직접 new 키워드를 사용해서 만드는 대신에 **외부에서 그 오브젝트를 만들어서 hello Controller가 사용할 수 있도록 주입**을 해주는 것! -> **assembler**가 해줌!

Assembler가 직접 의존관계가 없는 클래스들의 오브젝트를 가져다가 서로 관계를 연결시켜주고 사용할 수 있도록 만들어주는 그런 역할을 함

이 assembler가 Spring Container다!

spring container는 기본적으로 안에 어떤 오브젝트를 만들때 딱 한번만 만듦

spring container가 가지고 있는 오브젝트를 필요로 하는 여러 오브젝트(서블릿)들이 있을텐데, 그들이 요청할때마다 새로운 오브젝트를 만들어서 주는게 아니라 처음에 만들어둔 동일, 유일한 특정 오브젝트를 리턴해줌 (== 싱글톤 패턴)

spring container == singletone registry = 싱글톤 패턴을 사용하지 않고도 마치 싱글톤 패턴을 쓰는 것처럼 오브젝트를 재사용함

정리

Spring container가 하는 역할

1. 우리가 메타정보를 주면 그걸 가지고 클래스의 싱글톤 오브젝트를 만들고
2. 이 오브젝트가 사용할 다른 의존 오브젝트가 있다면 그것을 주입해줌

주입을 해준다는 것은???

다른 오브젝트의 레퍼런스를 넘겨주는 것이다

주입하는 방식 3가지

1. Hello Controller를 만들때 생성자 파라미터로 simple hello service의 오브젝트를 넣어줌 (물론 이때 파라미터 타입은 클래스가 구현하고 있는 hello service 인터페이스 타입으로 되어있음)
2. factory method로 bean 만들어서 넘기기
3. hello controller 클래스에 프로퍼티를 정의해서 setter method를 통해 simpleHelloService를 주입하는 방식

(4-2) 의존 오브젝트 DI 적용

기존 방식: Hello Controller가 Simple Hello Service라는 클래스의 오브젝트를 직접 생성해서 사용

→ 이젠 스프링 빈으로 등록하고 container 가 어셈블러로써 Hello Controller가 simple hello service를 사용할 수 있도록 주입(DI) 하는 작업 수행

실습

spring container에 두개의 bean을 띄우고 둘 사이의 의존관계 만들기 (container가 제공하는 DI로)

HellobootApplication

```
public class HellobootApplication {

    public static void main(String[] args) {
                // spring container
        GenericApplicationContext genericApplicationContext = new GenericApplicationContext();

        //
        genericApplicationContext.registerBean(HelloController.class);
        genericApplicationContext.registerBean(SimpleHelloService.class); // controller       ?

        //       -> application context bean object
        genericApplicationContext.refresh();

        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("frontController", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
                    if(req.getRequestURI().equals("/hello") && req.getMethod().equals(HttpMethod.GET.name())){
// mapping
                        String name = req.getParameter("name");

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

                        resp.setContentType(MediaType.TEXT_PLAIN_VALUE);
                        resp.getWriter().println(ret);
                    }
                    else{
                        resp.setStatus(HttpStatus.NOT_FOUND.value());
                    }
                }
            }).addMapping("/*");
        });

        // Tomcat servlet container
        webServer.start();
    }

}
```

이전 HelloController.java

HelloController.java

```
public class HelloController {
    public String hello(String name){
        SimpleHelloService helloService = new SimpleHelloService(); // <----------------
        return helloService.sayHello(Objects.requireNonNull(name));
    }
}
```

바꾼 HelloController.java

HelloController.java

```
public class HelloController {
    // controller     -> (Spring container) hello controller
    //
    private final HelloService helloService;


    // spring container Hello controller
    //     helloService
    // -> container     Hello service
    // ->
    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    public String hello(String name){
        return helloService.sayHello(Objects.requireNonNull(name));
    }
}
```

HelloService.java interface

```
package tobyspring.helloboot;

/**
 * intellij refactor -> extract interface  class interface
 */
public interface HelloService {
    String sayHello(String name);
}
```

SimpleHelloService.java

```
package tobyspring.helloboot;


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

(4-3) 조금 더 유연하게 가자면

hello controller는 헬로 서비스 인터페이스를 구현한 어떠한 클래스에 의존을 하는데

SimpleHelloService에서 ComplexHelloService로 바꾸었다고 해서 소스코드는 고치고 싶지 않다! → **유연한 구조를 원한 요구사항**

그래서 java에서는 인터페이스를 사용해서 인터페이스에 의존하도록 함, 그리고 인터페이스를 구현한 클래스를 만듦

hello controller는 클래스 대신 인터페이스로 서비스를 의존하면 인터페이스를 구현한 서비스 클래스를 아무리 많이 만들어도 특정 클래스에 의존하고 있지 않기 때문에 hello controller 고치지 않아도 됨

하지만 런타임에 필요하다면 SimpleHelloService의 오브젝트 대신 ComplexHelloService의 오브젝트를 사용하겠다 등 결정이 있을텐데, 그건 어떻게 하지?

그때는 어느 클래스의 오브젝트를 사용할 것인가 결정해야되야함

(4-4) @Primary OR @Qualifier

@Primary

"이 빈을 기본으로 주입해줘라." 라는 의미를 부여해주는 어노테이션.

빈을 등록할 때 붙여서 해당 빈이 같은 타입(같은 인터페이스의 구현체)의 빈 중 가장 높은 우선순위를 가진다고 명시함

SimpleHelloService.java

```
@Component
@Primary // <---
public class SimpleHelloService implements HelloService {
    @Override
    public String sayHello(String name){
        return "Hello " + name;
    }
}
```

@Qualifier

스프링 컨테이너에서 같은 타입에 대한 여러 빈을 찾았을 때, 그 중에 어떤 빈을 써야하는지에 대한 기준을 부여해주는 어노테이션이다.

SimpleHelloService.java

```
@Component
@Qualifier("MainHelloService")
public class SimpleHelloService implements HelloService {
    @Override
    public String sayHello(String name){
        return "Hello " + name;
    }
}
```

```
@RestController
public class HelloController {
    private final HelloService helloService;
    public HelloController(@Qualifier("mainHelloService") HelloService helloService) {
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    @ResponseBody
    public String hello(String name){
        if(name == null || name.trim().length() == 0) throw new IllegalArgumentException();

        return helloService.sayHello(name);
    }

}
```

빈 등록시에 `@Qualifier("")` 어노테이션으로 선택 기준 이름을 등록해주고,

빈을 주입받는 곳에서 `@Qualifier("")`으로 원하는 선택 기준을 넣어주면 해당 이름으로 `@Qualifier`가 등록된 빈을 찾아서 주입해줌.

빈 검색 및 주입 우선순위

1. 타입을 기준으로 검색
2. 같은 타입에 대해 등록된 빈이 여러 개면 @Qualifier가 있는지 확인
3. @Qualifier가 없으면 @Primary로 지정된 빈이 있는지 확인
4. @Primary도 없으면 매개변수의 이름 중 가장 비슷한 이름의 빈을 주입

(4-5) @Component 스캔

이전까지 스프링 컨테이너에 빈을 등록하는 방식

1. class 정보를 registerBean() method에 넘겨줌
2. 팩토리 메소드를 만들어서 직접 인스턴스를 생성함

HellobootApplication

```
public class HellobootApplication {

    public static void main(String[] args) {
                // spring container
        GenericApplicationContext genericApplicationContext = new GenericApplicationContext();

        //       *********************************
        genericApplicationContext.registerBean(HelloController.class);
        genericApplicationContext.registerBean(SimpleHelloService.class);


        genericApplicationContext.refresh();

        TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();

        WebServer webServer = serverFactory.getWebServer(servletContext -> {
            servletContext.addServlet("frontController", new HttpServlet() {
                @Override
                protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
                    if(req.getRequestURI().equals("/hello") && req.getMethod().equals(HttpMethod.GET.name())){
                        String name = req.getParameter("name");

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

                        resp.setContentType(MediaType.TEXT_PLAIN_VALUE);
                        resp.getWriter().println(ret);
                    }
                    else{
                        resp.setStatus(HttpStatus.NOT_FOUND.value());
                    }
                }
            }).addMapping("/*");
        });

        webServer.start();
    }

}
```

더 간결하게 빈을 등록할 수 있는 방법이 없을까?

@Component, @ComponentScan를 사용하면 된다!

HellobootApplication.java

```
@Configuration

// @Component      ! ->
//
@ComponentScan
public class HellobootApplication {
    public static void main(String[] args) {
        AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext(){
            @Override
            protected void onRefresh() {
                super.onRefresh();

                TomcatServletWebServerFactory serverFactory = new TomcatServletWebServerFactory();
                WebServer webServer = serverFactory.getWebServer(servletContext -> {
                    servletContext.addServlet("dispatcherServlet",
                            new DispatcherServlet(this)
                    ).addMapping("/*");
                });
                webServer.start();
            }
        };

        applicationContext.register(HellobootApplication.class);
        applicationContext.refresh();


    }
}
```

HelloController.java

```
//@RequestMapping

//   !     !
// -> spring container
//@Component
//@MyComponent
//@Controller
@RestController // -> @Controller + @ResponseBody
                //  @RequestMapping   dispatcherServlet @RestController         ->
public class HelloController {
    private final HelloService helloService;
    public HelloController(HelloService helloService) { // <- HelloController bean  HelloController
HelloService bean !
        this.helloService = helloService;
    }

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

}
```

- Component scan 의 장점
  - 매번 구성정보를 추가할 필요없이 내가 작성하는 클래스가 빈으로 등록됨

- Component scan 의 단점
  - 빈으로 등록되는 클래스 많아지게 되면 내가 이 애플리케이션을 실행했을때 정확하게 어떤 것들이 등록되는지 찾아보기 번거롭다

그럼에도 **편리하기 때문에** 우리가 작성하는 애플리케이션 로직을 담은 코드를 빈으로 등록하는 방식에서는 거의 표준으로 쓰임

번거롭다는 단점은 패키지 구성 관리와 모듈 잘 나눠서 관리를 잘하면 커버칠 수 있음

@Component의 재밌는 특징

@Component를 **메타 애너테이션**으로 가지고 있는 애너테이션을 붙여도 @Component이 붙은 것과 동일한 효과를 낸다

메타 애너테이션이란?

보통 애너테이션은 클래스나 메소드 앞에 붙음. 하지만 애너테이션도 자바 코드로 만들어진 것이니까 애너테이션 코드 위에 애너테이션을 붙힐 수 있다.

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){
        if(name == null || name.trim().length() == 0) throw new IllegalArgumentException();

        return helloService.sayHello(name);
    }

}
```

Controller.java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component   // <-      component scan !
public @interface Controller {

        /**
         * The value may indicate a suggestion for a logical component name,
         * to be turned into a Spring bean in case of an autodetected component.
         * @return the suggested component name, if any (or empty String otherwise)
         */
        @AliasFor(annotation = Component.class)
        String value() default "";

}
```

spring에서 제공하는 메타 애너테이션 예시

- @Controller
- @Service
- @RestController → @Controller → @Component