1장 발표자료

1. Servlet(서블릿)

클라이언트의 요청을 처리하고, 그 결과를 반환하는 Servlet 클래스의 구현 규칙을 지킨 자바 웹 프로그래밍 기술

간단히 말해서, 서블릿이란 자바를 사용하여 웹을 만들기 위해 필요한 기술입니다.
클라이언트가 어떠한 요청을 하면 그에 대한 결과를 다시 전송해주어야 하는데, 이러한 역할을 하는 자바 프로그램입니다.
예를 들어, 어떠한 사용자가 로그인을 하려고 할 때. 사용자는 아이디와 비밀번호를 입력하고, 로그인 버튼을 누릅니다.
그때 서버는 클라이언트의 아이디와 비밀번호를 확인하고, 다음 페이지를 띄워주어야 하는데, 이러한 역할을 수행하는
것이 바로 서블릿(Servlet)입니다. 그래서 서블릿은 자바로 구현 된 *CGI라고 흔히 말합니다.

Q) CGI(Common Gateway Interface)란?

CGI는 특별한 라이브러리나 도구를 의미하는 것이 아니고, 별도로 제작된 웹 서버와 프로그램간의 교환방식입니다. CGI방식은 어떠한 프로그래밍 언어로도 구현이 가능하며,
별도로 만들어 놓은 프로그램에 HTML의 Get or Post 방법으로 클라이언트의 데이터를 환경변수로 전달하고, 프로그램의 표준 출력 결과를 클라이언트에게 전송하는 것입니다.
즉, 자바 어플리케이션 코딩을 하듯 웹 브라우저용 출력 화면을 만드는 방법입니다.

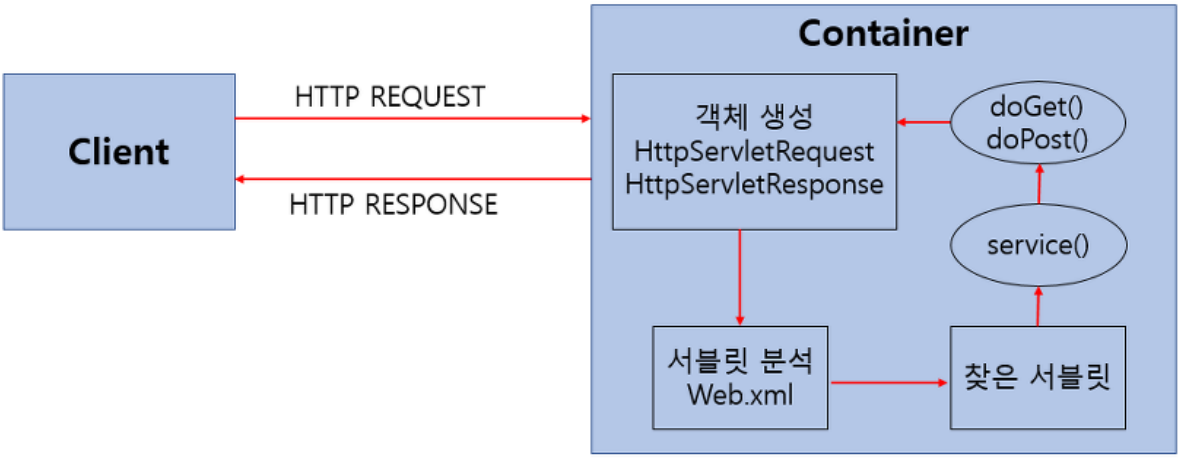
[Servlet 특징]

- 클라이언트의 요청에 대해 동적으로 작동하는 웹 어플리케이션 컴포넌트
- html을 사용하여 요청에 응답한다.
- Java Thread를 이용하여 동작한다. (서블릿 컨테이너는 멀테 쓰레드로 운용, 원래는 쓰레드의 생성을 관리해야 했지만 서블릿 컨테이너가 이를 대신 관리해 줌)
- MVC 패턴에서 Controller로 이용된다.

- HTTP 프로토콜 서비스를 지원하는 javax.servlet.http.HttpServlet 클래스를 상속받는다.
- UDP보다 처리 속도가 느리다.
- HTML 변경 시 Servlet을 재컴파일해야 하는 단점이 있다.

일반적으로 웹서버는 정적인 페이지만을 제공합니다. 그렇기 때문에 동적인 페이지를 제공하기 위해서 웹서버는
다른 곳에 도움을 요청하여 동적인 페이지를 작성해야 합니다. 동적인 페이지로는 임의의 이미지만을 보여주는 페이지와 같이
사용자가 요청한 시점에 페이지를 생성해서 전달해 주는 것을 의미합니다. 여기서 웹서버가 동적인 페이지를 제공할 수 있도록
도와주는 어플리케이션이 서블릿이며, 동적인 페이지를 생성하는 어플리케이션이 CGI입니다.

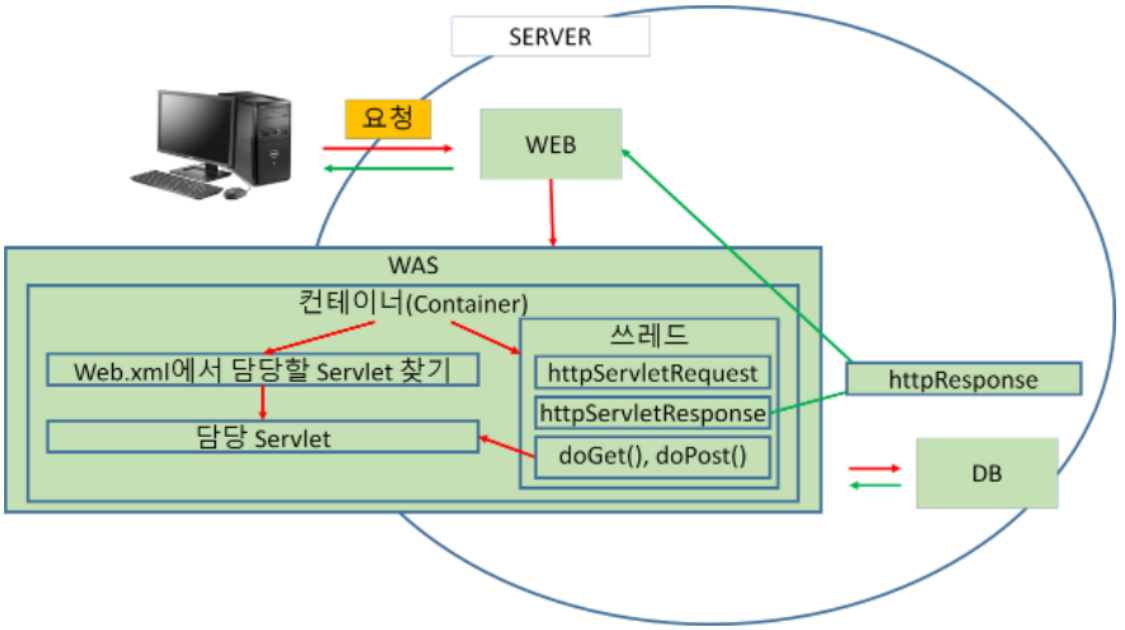
[Servlet 동작 방식]



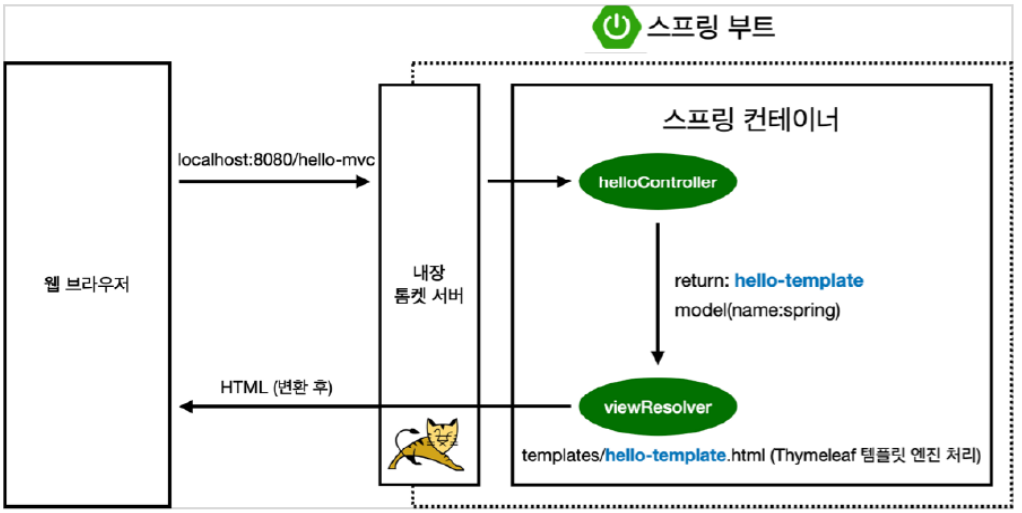
1. 사용자(클라이언트)가 URL을 입력하면 HTTP Request가 Servlet Container로 전송합니다.
2. 요청을 전송받은 Servlet Container는 HttpServletRequest, HttpServletResponse 객체를 생성합니다.
3. web.xml을 기반으로 사용자가 요청한 URL이 어느 서블릿에 대한 요청인지 찾습니다.
4. 해당 서블릿에서 service메소드를 호출한 후 클리아언트의 GET, POST여부에 따라 doGet() 또는 doPost()를 호출합니다.

5. doGet() or doPost() 메소드는 동적 페이지를 생성한 후 HttpServletResponse 객체에 응답을 보냅니다.
6. 응답이 끝나면 HttpServletRequest, HttpServletResponse 두 객체를 소멸시킵니다.

2. Web WAS?



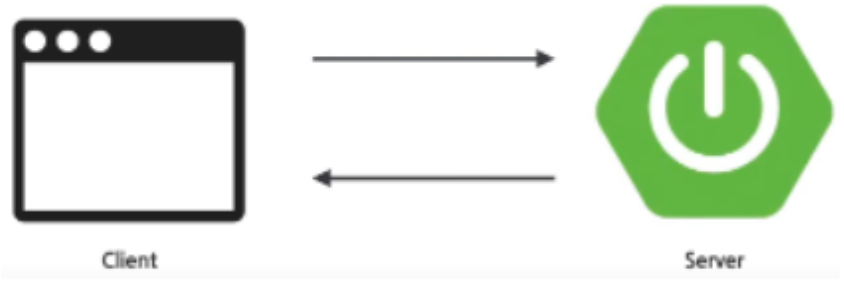
MVC, 템플릿 엔진 이미지



WAS(Web Application Server)

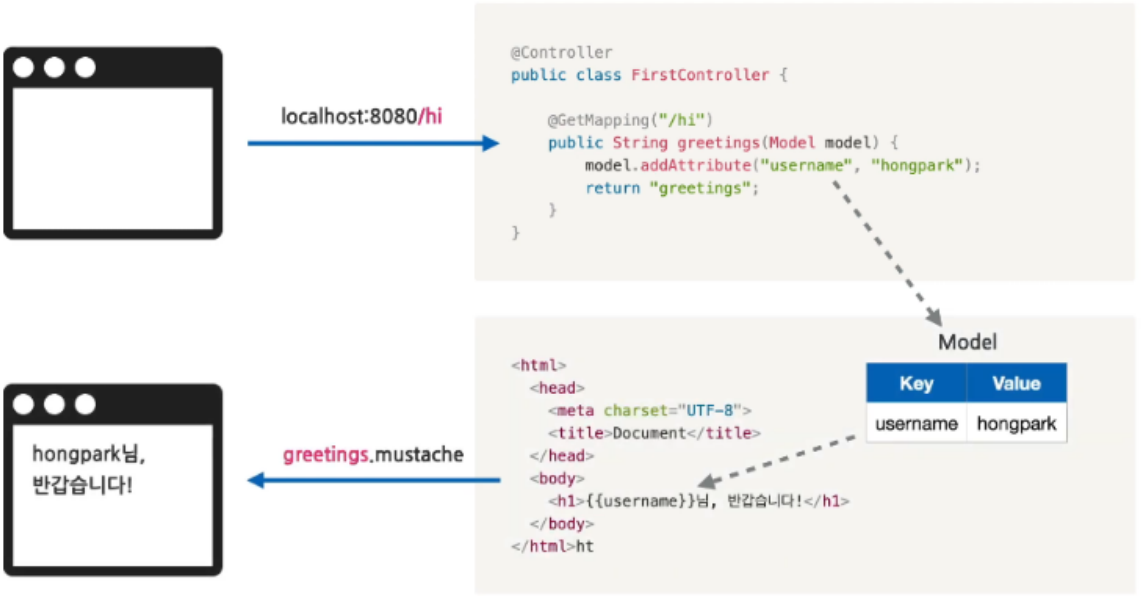
- 동적 요청이 들어옴
  - 새 쓰레드에 아래의 것들을 만듦
    - httpServletRequest 객체 : 정보 뭉텅이(전달받은 데이터) -> 나중에 새로운 동적 페이지를 만들 때 사용

        - httpServletResponse 객체 : 빈 객체 -> 나중에 DB와 httpServletRequest의 데이터들로 만든 결과 페이지를 담아 WEB으로 다시 돌려보냄(즉, 결과물)
        - doGet(), doPost() : Web.xml에서 해당 요청을 처리하는 survlet에 doGet(), doPost()함수를 실행
    - httpServletResponse를 WEB에 다시 보내면서 쓰레드는 사라짐
    - JVM, 톰캣
- 웹서비스는 클라이언트와 서버의 요청과 응답으로 동작
- 클라이언트 : 서비스를 사용하는 프로그램, 컴퓨터 → 브라우저
- 서버 : 서비스를 제공하는 프로그램, 컴퓨터 → 스프링부트



서버로서의 Spring Boot

서버의 유기적 역할 분담

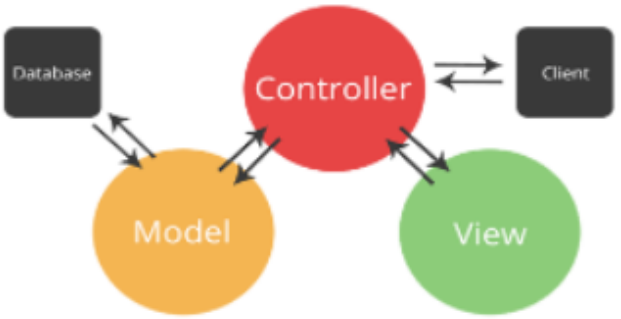


Controller의 model 객체가 값을 username에 연결시켜서 보내줌 → greeting.mustache에 변수로 삽입되어 출력

- Controller : 클라이언트로부터 요청을 받음, 유저가 보는 View와 정보가 있는 DB를 연결하며 어떻게 데이터를 처리할지 결정
- View : 최종 페이지 생성(mustache : template의 종류 중 하나)
- Model : 최종 페이지에 쓰일 데이터를 View에게 전달

1. **Controller**인 FirstController.java의 @GetMapping으로 인해 '/hi' 요청 시, niceToMeetYou() 메소드 실행

2. 메소드가 반환하는 return 값을 통해 **View** 페이지인 greeting.mustache 를 찾아서 보여줌
3. 뷰 페이지에서 {{username}} 변수를 사용하기 위해서는 Controller의 **Model**을 통해 model.addAttribute() 변수 등록



3. Bean에 대한 추가적인 정보

Bean 등록

1. **[@Bean 어노테이션과 @Configuration 어노테이션]**
2. **[@Component 어노테이션]**

@Configuration에 적용되는 프록시 패턴

@Configuration 어노테이션 안에는 @Component 어노테이션이 붙어있어서 @Configuration이 붙어있는 클래스 역시 스프링의 빈으로 등록이 된다. 그럼에도 불구하고 스프링이 @Configuration을 따로 만든 이유는 CGLib으로 프록시 패턴을 적용해 수동으로 등록하는 스프링 빈이 반드시 싱글톤으로 생성됨을 보장하기 위해서이다.

예를 들어 다음과 같이 스프링 빈으로 등록하고자 하는 클래스가 있다고 하자.

```
public class MangKyuResource {

}
```

CGlib Proxy

CGlib Proxy는 Enhancer를 바탕으로 Proxy를 구현하는 방식입니다
이 방식은 JDK Dynamic Proxy와는 다르게 Reflection을 사용하지 않고,
Extends(상속) 방식을 이용해서 Proxy화 할 메서드를 오버라이딩 하는 방식입니다
위의 클래스를 @Component를 이용해 자동으로 빈 등록을 한다면 스프링이 해당 클래스의 객체의 생성을 제어하게 되고(제어의 역전, IoC) 1개의 객체만 생성되도록 컨트롤할 수 있다.

하지만 위의 클래스를 @Bean을 사용해 직접 빈으로 등록해준다고 하자. 그러면 우리는 다음과 같이 해당 빈 등록 메소드를 여러 번 호출할 수 있게 된다.

```
@Configuration
public class MyBeanConfiguration {

    @Bean
    public MangKyuResource mangKyuResource() {
        return new MangKyuResource();
    }

    @Bean
    public MyFirstBean myFirstBean() {
        return new MyFirstBean(mangKyuResource());
    }

    @Bean
    public MySecondBean mySecondBean() {
        return new MySecondBean(mangKyuResource());
    }
}
```

실수로 위와 같이 빈을 생성하는 메소드를 여러 번 호출하였다면 불필요하게 여러 개의 빈이 생성이 된다. 스프링은 이러한 문제를 방지하고자 @Configuration이 있는 클래스를 객체로 생성할 때 CGLib 라이브러리를 사용해 프록시 패턴을 적용한다. 그래서 @Bean이 있는 메소드를 여러 번 호출하여도 항상 동일한 객체를 반환하여 싱글톤을 보장한다. 이를 이해하기 쉬운 코드로 나타내면 다음과 같다.

```
@Configuration
public class MyBeanConfigurationProxy extends MyBeanConfiguration {

    private Object source;

    @Override
    public MangKyuResource mangKyuResource() {
        if (mangKyuResource == null) {
            source = super.mangKyuResource();
        }

        return source;
    }

    @Override
    public MyFirstBean myFirstBean() {
        return super.myFirstBean();
    }
```

```
    @Override
    public MySecondBean mySecondBean() {
        return super.mySecondBean();
    }
}
```

CGLib은 상속을 사용해서 프록시를 구현하므로 다음과 같이 프록시가 구현된다고 이해할 수 있다.
물론 실제로는 이렇게 생성되지 않고 내부 클래스를 사용하는 등의 차이가 있으므로 이해를 돕기 위한 코드로만 생각하면 된다.

4. 스프링(Spring)의 싱글톤(Singleton)

[Spring에서 싱글톤을 사용하는 이유]
애플리케이션 컨텍스트에 의해 등록된 빈은 기본적으로 싱글톤으로 관리된다. 즉, 스프링에 여러 번 빈을 요청하더라도 매번 동일한 객체를 돌려준다는 것이다.
애플리케이션 컨텍스트가 싱글톤으로 빈을 관리하는 이유는 대규모 트래픽을 처리할 수 있도록 하기 위함이다.
스프링은 최초에 설계될 때부터 대규모의 엔터프라이즈 환경에서 요청을 처리할 수 있도록 고안되었다. 그리고 그에 따라 계층적으로 처리 구조(Controller, Service, Repository 등) 가 나뉘어지게 되었다.
그런데 매번 클라이언트에서 요청이 올 때마다 각 로직을 처리하는 빈을 새로 만들어서 사용한다고 생각해보자. 요청 1번에 5개의 객체가 만들어진다고 하고, 1초에 500번 요청이 온다고 하면 초당 2500개의 새로운 객체가 생성된다. 아무리 GC의 성능이 좋아졌다 하더라도 부하가 걸리면 감당이 힘들 것이다.
그래서 이러한 문제를 해결하고자 빈을 싱글톤 스코프로 관리하여 1개의 요청이 왔을 때 여러 쓰레드가 빈을 공유해 처리하도록 하였다.

[Spring에서 관리하는 싱글톤의 단점]
Java로 기본적인 싱글톤 패턴을 구현하고자 하면 다음과 같은 단점들이 발생한다.

- private 생성자를 갖고 있어 상속이 불가능하다.
- 테스트하기 힘들다.
- 서버 환경에서는 싱글톤이 1개만 생성됨을 보장하지 못한다.
- 전역 상태를 만들 수 있기 때문에 객체지향적이지 못하다.

그래서 스프링은 직접 싱글톤 형태의 오브젝트를 만들고 관리하는 기능을 제공하는데, 그것이 바로 싱글톤 레지스트리(Singleton Registry) 이다.
스프링 컨테이너는 싱글톤을 생성하고, 관리하고 공급하는 컨테이너이기도 하다.
싱글톤 레지스트리의 장점은 다음과 같다.

- static 메소드나 private 생성자 등을 사용하지 않아 객체지향적 개발을 할 수 있다.
- 테스트를 하기 편리하다.

기본적으로 싱글톤이 멀티쓰레드 환경에서 서비스 형태의 객체로 사용되기 위해서는 내부에 상태정보를 갖지 않는 무상태(Stateless) 방식으로 만들어져야 한다. 만약 여러 쓰레드들이 동시에 상태를 접근하여 수정한다면 상당히 위험하기 때문이다.
직접 싱글톤을 구현한다면 상당히 많은 단점들이 있겠지만, Spring 프레임워크에서 직접 싱글톤으로 객체를 관리해주므로, 우리는 더욱 객체지향적인 개발을 할 수 있게 된 것이다.

5. Application Context의 Bean 등록 과정

[애플리케이션 컨텍스트(Application Context)란?]

애플리케이션 컨텍스트는 빈들의 생성과 의존성 주입 등의 역할을 하는 일종의 DI 컨테이너

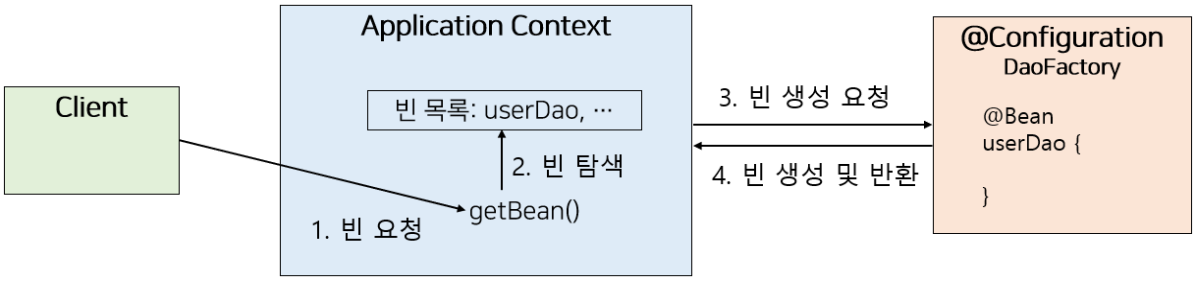
Spring에서는 빈의 생성과 관계설정 같은 제어를 담당하는 IoC(Inversion of Control) 컨테이너인 빈 팩토리(Bean Factory)가 존재
하지만 실제로는 빈의 생성과 관계설정 외에 추가적인 기능이 필요한데, 이러한 이유로 Spring에서는 빈 팩토리를 상속받아 확장한 애플리케이션 컨텍스트(Application Context)를 주로 사용
애플리케이션 컨텍스트는 별도의 설정 정보를 참고하고 IoC를 적용하여 빈의 생성, 관계설정 등의 제어 작업을 총괄
애플리케이션 컨텍스트에는 직접 오브젝트를 생성하고 관계를 맺어주는 코드가 없고, 그런 생성 정보와 연관 관계 정보에 대한 설정을 읽어 처리
예를 들어 @Configuration과 같은 어노테이션이 대표적인 IoC의 설정 정보

[빈(Bean) 요청 시 처리 과정]

클라이언트에서 해당 빈을 요청하면 애플리케이션 컨텍스트는 다음과 같은 과정을 거쳐 빈을 반환

1. ApplicationContext는 @Configuration이 붙은 클래스들을 설정 정보로 등록해 두고, @Bean이 붙은 메소드의 이름으로 빈 목록을 생성한다.

2. 클라이언트가 해당 빈을 요청한다.
3. ApplicationContext는 자신의 빈 목록에서 요청한 이름이 있는지 찾는다.
4. ApplicationContext는 설정 클래스로부터 빈 생성을 요청하고, 생성된 빈을 돌려준다.



애플리케이션 컨택스트는 @Configuration이 붙은 클래스들을 설정 정보로 등록해 두고, @Bean이 붙은 메소드의 이름으로 빈 목록을 생성한다. 그리고 클라이언트가 해당 빈을 요청한다면 애플리케이션 컨텍스트는 자신의 빈 목록에서 요청한 이름이 있는지 찾고, 있다면 해당 빈 생성 메소드(@Bean)을 호출하여 객체를 생성하고 돌려준다. (구체적으로는 Spring 내부에서 Reflection API를 이용해 빈 정의에 나오는 클래스 이름을 이용하거나 또는 빈 팩토리를 통해 빈을 생성한다.)

[애플리케이션 컨텍스트(Application Context)의 장점]

이렇듯 애플리케이션 컨텍스트를 사용하면 다음과 같은 장점들을 얻을 수 있다.

- 클라이언트는 @Configuration이 붙은 구체적인 팩토리 클래스를 알 필요가 없다.
    - 애플리케이션이 발전하면 팩토리 클래스가 계속해서 증가할 것이다. 애플리케이션 컨텍스트가 없다면 클라이언트는 원하는 객체를 가져오려면 어떤 팩토리 클래스에 접근해야 하는지 알아야 하는 번거로움이 생긴다. 반면에 애플리케이션 컨텍스트를 사용하면 팩토리가 아무리 많아져도 이에 직접 접근할 필요가 없어진다. 즉, 일관된 방식으로 원하는 빈을 가져올 수 있다.

- 애플리케이션 컨텍스트는 종합 IoC 서비스를 제공해준다.
    - 애플리케이션 컨텍스트는 객체의 생성과 관계 설정이 다가 아니다. 객체가 만들어지는 방식과 시점 및 전략 등을 다르게 가져갈 수 있고, 그 외에도 후처리나 정보의 조합 인터셉트 등과 같은 다양한 기능이 존재한다.

- 애플리케이션 컨텍스트를 통해 다양한 빈 검색 방법을 제공할 수 있다.
    - 애플리케이션 컨텍스트에서 빈 목록을 관리하여, 빈의 이름이나 타입 또는 어노테이션 설정 등으로 빈을 찾을 수 있다. 이러한 빈을 직접 찾는 방식은 의존성 검색(dependency lookup)으로 불린다.

애플리케이션 종류에 따라 각기 다른 종류의 ApplicationContext가 내부에서 만들어진다.

- 웹 애플리케이션이 아닌 경우
    - 애플리케이션 컨텍스트: AnnotationConfigApplicationContext
    - 웹서버: X
- 서블릿 기반의 웹 애플리케이션인 경우
    - 애플리케이션 컨텍스트: AnnotationConfigServletWebServerApplicationContext
    - 웹서버: Tomcat
- 리액티브 웹 애플리케이션인 경우
    - 애플리케이션 컨텍스트: AnnotationConfigReactiveWebServerApplicationContext
    - 웹서버: Reactor Netty

[DI 컨테이너와 애플리케이션 컨텍스트(Application Context)]

애플리케이션 컨텍스트는 이름 그대로 애플리케이션을 실행하기 위한 환경이다. 그럼에도 불구하고 애플리케이션 컨텍스트가 DI 컨테이너라고도 불리며 그러한 역할을 할 수 있는 이유는
ApplicationContext 상위에 빈들을 생성하는 BeanFactory 인터페이스를 부모로 상속받고 있기 때문이다.