06 자동 구성 기반 애플리케이션

메타 애노테이션과 합성 애노테이션

Meta-annotation

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component    // Meta Annotation
public @interface Service {

}
```

애노테이션에 적용한 애노테이션을 메타 애노테이션이라고 한다. 스프링은 코드에서 사용된 애노테이션의 메타 애노테이션의 효력을 적용해준다.

@Service 애노테이션이 부여된 클래스는 @Service의 메타 애노테이션인 @Component가 직접 사용된 것처럼 컴포넌트 스캔의 대상이 된다.

Composed-annotation

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller    // Meta Annotation
@ResponseBody  // Meta Annotation
public @interface RestController {
   ...
}
```

합성(composed) 애노테이션은 하나 이상의 메타 애노테이션이 적용된 애노테이션을 말한다. 합성 애노테이션을 사용하면 모든 메타 애노테이션이 적용된 것과 동일한 효과를 같는다.

@RestController를 클래스에 적용하면 @Component와 @ResponseBody를 둘 다 사용한 것과 동일한 결과를 가져온다.

합성 애노테이션 적용

@Configuration과 @ComponentScan 애노테이션을 메타 애노테이션으로 가지는 새로운 애노테이션을 만들어서 HellobootApplication에 적용한다.

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

@Configuration도 @Component를 메타 애노테이션으로 가지는 애노테이션이다.

빈 오브젝트의 역할과 구분

애플리케이션 로직 빈

애플리케이션의 비즈니스 로직을 담고 있는 클래스로 만들어지는 빈. 컴포넌트 스캐너에 의해서 빈 구성 정보가 생성되고 빈 오브젝트로 등록된다.

애플리케이션 인프라스트럭처 빈

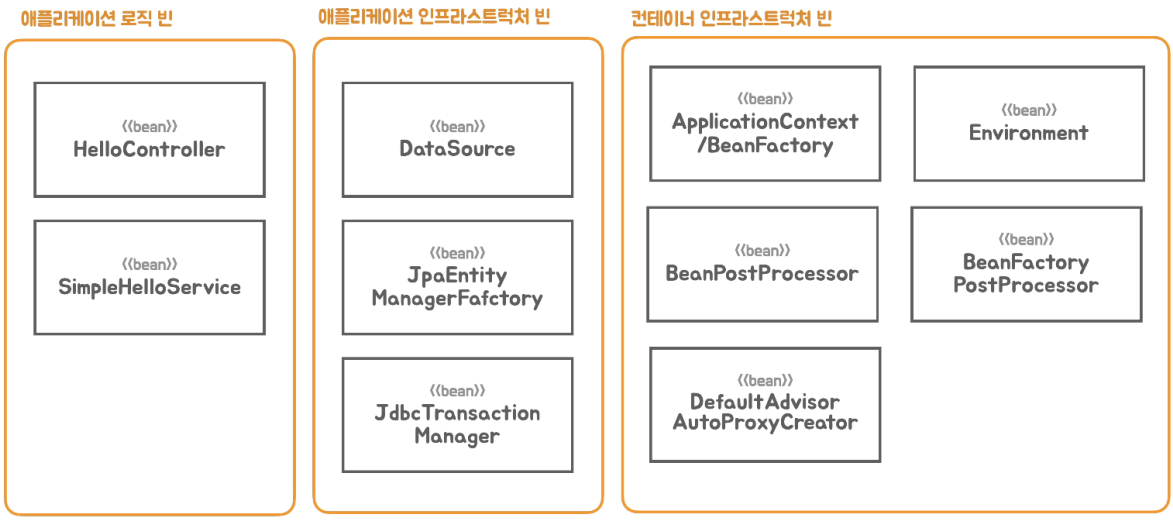
빈 구성 정보에 의해 컨테이너에 등록되는 빈이지만 애플리케이션의 로직이 아니라 애플리케이션이 동작하는데 꼭 필요한 기술 기반을 제공하는 빈이다.

전통적인 스프령 애플리케이션에서는 빈으로 등록되지 않지만 스프링 부트에서 구성 정보에 의해 빈으로 등록되어지는 ServletWebServerFactory나 DispatcherServlet 등도 애플리케이션 인프라 빈이라고 볼 수 있다.

컨테이너 인프라스트럭처 빈

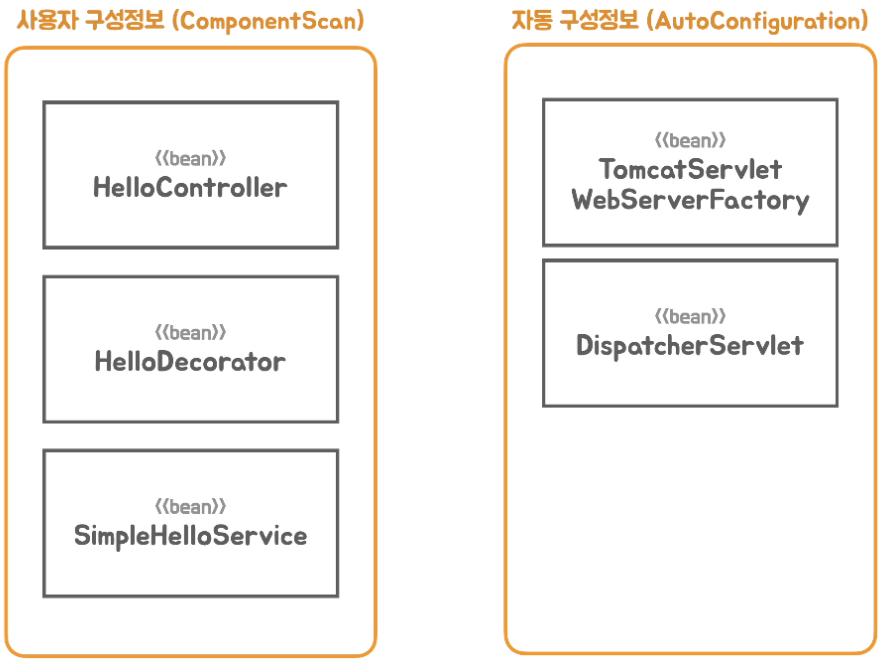
스프링 컨테이너의 기능을 확장해서 빈의 등록과 생성, 관계설정, 초기화 등의 작업에 참여하는 빈을 컨테이너 인프라스트럭처 빈, 줄여서 컨테이너 인프라 빈이라고 한다. 개발자가 작성한 구성 정보에 의해서 생성되는 게 아니라 컨테이너가 직접 만들고 사용하는 빈이기 때문에 애플리케이션 빈과 구분한다.

필요한 경우 일부 컨테이너 인프라 빈은 주입 받아서 활용할 수 있다.



스프링 부트의 빈 구성 정보는 컴포넌트 스캔에 의해서 등록되는 빈과 자동 구성에 의해서 등록되는 빈으로 구분된다. 일반적으로 애플리케이션 인프라 빈은 자동 구성에 의해서 등록되지만 개발자가 작성한 코드 구성 정보에 의해서도 등록할 수도 있다.

자동 구성 메카니즘을 확장하면 애플리케이션 로직을 담은 라이브러리를 만들어 자동 구성에 의해서 등록되도록 만드는 것도 가능하다.



인프라 빈 구성 정보의 분리

@Import 를 이용하면 스캔 대상이 아닌 클래스를 빈으로 등록하도록 추가할 수 있다.

보통 @Configuration 애노테이션이 붙은 클래스를 가져온다. @Component가 붙은 클래스도 빈으로 등록시킬 수 있다.

애플리케이션 인프라스트럭처 빈 구성 정보 클래스는 스프링 부트의 자동 구성 메카니즘에 의해서 등록이 되도록 분리하는 작업이 우선 필요하다. 분리된 클래스는 @Import로 포함시킨다.

자동으로 등록되는 대상인 자동 구성 클래스는 애노테이션을 하나 만들어서 그 안에서 @Import 시킨다. @Import도 메타 애노테이션으로 사용할 수 있다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import({DispatcherServletConfig.class, TomcatWebServerConfig.class})
public @interface EnableMyAutoConfiguration {
}
```

이 애노테이션을 애플리케이션 메인 애노테이션에도 메타 애노테이션으로 추가해준다.

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

동적인 자동 구성 정보 등록

```
public interface ImportSelector {

  String[] selectImports(AnnotationMetadata importingClassMetadata);
  ...
}
```

ImportSelector의 구현 클래스를 @Import하면 selectImports가 리턴하는 클래스 이름으로 @Configuration 클래스를 찾아서 구성 정보로 사용한다. 코드에 의해서 @import 대상을 외부에서 가져오고 선택할 수 있는 동적인 방법을 제공한다.

가져올 클래스 정보는 문자열 배열로 리턴한다.

```
public class MyAutoConfigImportSelector implements DeferredImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        return new String[] {
                "tobyspring.config.autoconfig.DispatcherServletConfig",
                "tobyspring.config.autoconfig.TomcatWebServerConfig"
        };
    }
}
```

자동 구성 정보 파일 분리

```
@Override
public String[] selectImports(AnnotationMetadata importingClassMetadata) {
    List<String> autoConfigs = new ArrayList<>();

    ImportCandidates.load(MyAutoConfiguration.class, classLoader).forEach(autoConfigs::add);

    return autoConfigs.toArray(new String[0]);
}
```

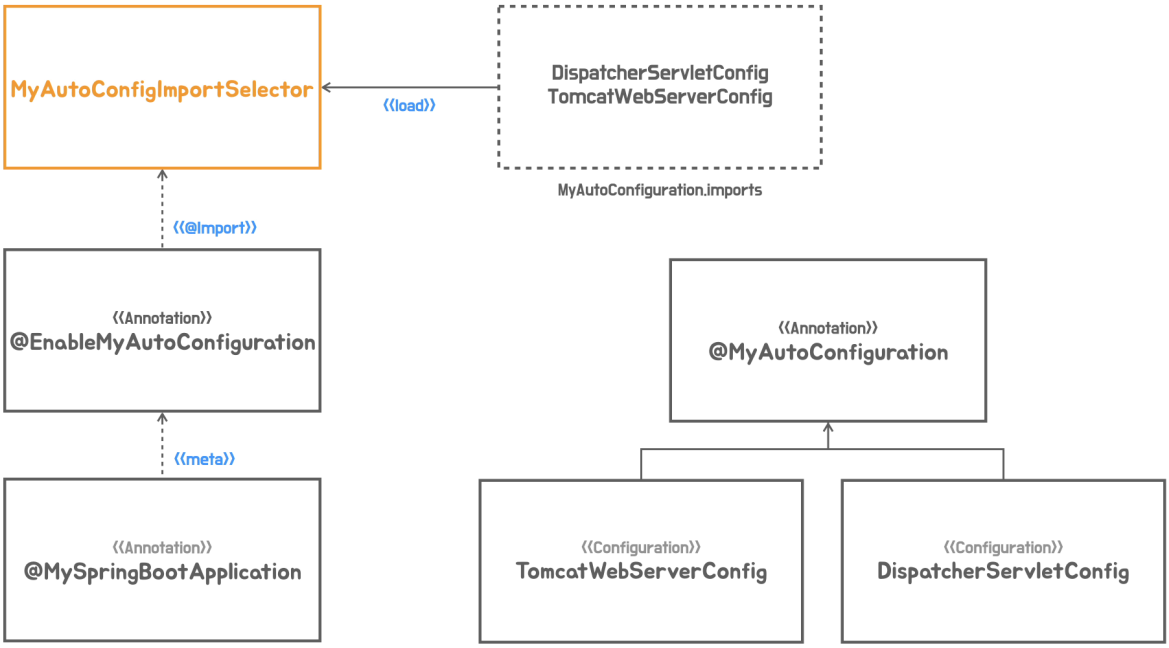
@MyAutoConfiguration 애노테이션을 만들고 이 클래스 이름 뒤에 .imports가 붙은 파일을 META-INF/spring 폴더 아래 만들어 selectImports()에서 가져와 컨테이너에 등록시킬 @Configuration 클래스 목록을 저장해둔다.

자동 구성 애노테이션 적용

@Configuration을 메타 애노테이션으로 가지는 @MyAutoConfiguration 애노테이션을 정의하고 인프라 빈 클래스에 @Configuration을 대체해서 부여한다.

```
@MyAutoConfiguration
public class DispatcherServletConfig {
    @Bean
    public DispatcherServlet dispatcherServlet() {
        return new DispatcherServlet();
    }
}
```

지금까지 적용한 자동 구성 정보를 다루는 구조는 다음과 같다.



@Configuration 클래스의 동작 방식

MyAutoConfiguration 애노테이션은 @Configuration을 메타 애노테이션으로 가지면서 proxyBeanMethods 엘리먼트를 false로 지정한다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration(proxyBeanMethods = false)
public @interface MyAutoConfiguration {
}
```

proxyBeanMethods는 디폴트로 true 값을 가진다. 이 경우 @Configuration이 붙은 클래스는 CGLib을 이용해서 프록시 클래스로 확장을 해서 @Bean이 붙은 메소드의 동작 방식을 변경한다. @Bean 메소드를 직접 호출해서 다른 빈의 의존 관계를 설정할 때 여러번 호출되더라도 싱글톤 빈처럼 참조할 수 있도록 매번 같은 오브젝트를 리턴하게 한다.

만약 @Bean 메소드 직접 호출로 빈 의존관계 주입을 하지 않는다면 굳이 복잡한 프록시 생성을 할 필요가 없다. 이 경우 proxyBeanMethods를 false로 지정해서도 된다. @Bean 메소드는 평범한 팩토리 메소드처럼 동작한다.

proxyBeanMethods는 스프링 5.2 버전부터 지원되기 시작했고 지금은 스프링과 스프링 부트의 상당히 많은 @Configuration 클래스 설정에 적용되고 있다.

@Bean은 @Configuration이 빈 클래스에서도 사용될 수 있다. 이를 @Bean 라이트 모드(lite mode)라고 부른다. 빈으로 등록되는 단순 팩토리 메소드로 사용된다.