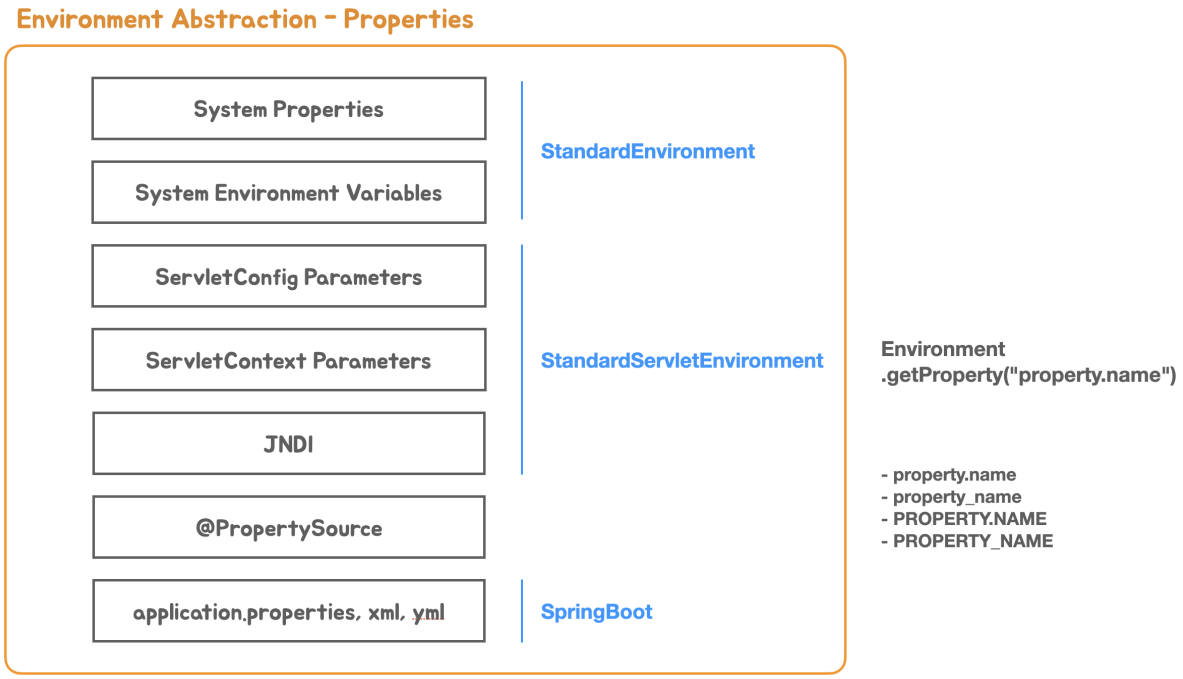
08 외부 설정을 활용하는 자동 구성

스프링의 Environment 추상화



스프링의 Environment 추상화는 애플리케이션의 두 가지 환경 정보 모델인 profile과 properties 를 제공한다.

자동 구성 정보의 일부 내용을 변경하거나 설정해야할 때 Environment를 통해서 프로퍼티 값을 가져와 활용할 수 있다. 커스톰 빈 등록을 하는 방법에 비해서 간단하게 자동 구성의 디폴트 설정을 변경하는 게 가능하다.

프로퍼티 정보는 시스템 프로퍼티, 환경 변수, 서블릿 파라미터, JNDI 등에서 우선순위에 따라서 가져온다. 애플리케이션 코드에서 @PropertySource로 프로퍼티 값을 가져올 대상을 지정할 수 있다.

스프링 부트는 기본적으로 application.properties, application.xml, application.yml 등의 파일에서 프로퍼티를 읽어오는 기능을 추가했다.

자동 구성에 Environment 프로퍼티 적용

스프링 부트의 모든 애플리케이션 초기화 작업이 끝나고 나면 실행되는 코드를 만들 때 ApplicationRunner 인터페이스를 구현한 오브젝트 또는 람다식을 빈으로 등록하면 된다.

```
@Bean
ApplicationRunner applicationRunner(Environment environment) {
    return args -> {
        String name = environment.getProperty("my.name");
        System.out.println("my.name: " + name);
    };
}
```

자동 구성 클래스의 메소드에도 Environment를 주입 받아서 빈 속성으로 지정할 프로퍼티 값을 가져올 수 있다.

```
@Bean("tomcatWebServerFactory")
@ConditionalOnMissingBean
```

```
public ServletWebServerFactory servletWebServerFactory(Environment env) {
    TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
    factory.setContextPath(env.getProperty("contextPath"));
    return factory;
}
```

@Value와 PropertySourcesPlaceholderConfigurer

@Value 애노테이션은 엘리먼트로 치환자(placeholder)를 지정하고 컨테이너 초기화시 프로퍼티 값으로 이를 대체할 수 있다.

@Value의 치환자를 프로퍼티 값으로 교체하려면 PropertySourcesPlaceholderConfigurer 타입의 빈을 등록해줘야 한다. PropertySourcesPlaceholderConfigurer는 빈 팩토리의 후처리기로 동작해서 초기 구성 정보에서 치환자를 찾아서 교체하는 기능을 담당한다.

PropertySourcesPlaceholderConfigurer도 자동 구성 빈으로 등록되게 한다.

```
@MyAutoConfiguration
public class PropertyPlaceholderConfig {
    @Bean PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

프로퍼티 클래스의 분리

자동 구성에 적용할 프로퍼티의 갯수가 많아지고 프로퍼티를 처리할 로직이 추가되야 한다면, 프로퍼티를 다루는 기능을 별도의 클래스로 분리하는 것이 좋다.

기본적인 프로퍼티 클래스는 프로퍼티 값을 가지고 있는 단순한 클래스로 작성할 수 있다.

```
public class ServerProperties {
    private String contextPath;

    private int port;

    public String getContextPath() {
        return contextPath;
    }

    public void setContextPath(String contextPath) {
        this.contextPath = contextPath;
    }

    public int getPort() {
        return port;
    }

    public void setPort(int port) {
        this.port = port;
    }
}
```

이 클래스를 빈으로 등록하는 자동 구성 클래스를 추가한다. Environment에서 프로퍼티 값을 가져와 오브젝트에 주입하는 것은 스프링 부트의 Binder 클래스를 이용하면 편리하다.

```
@MyAutoConfiguration
public class ServerPropertiesConfig {
    @Bean
    public ServerProperties serverProperties(Environment environment) {
        return Binder.get(environment).bind("", ServerProperties.class).get();
    }
}
```

프로퍼티 클래스로 만든 빈은 자동 구성 빈을 만들 때 주입 받아서 사용한다.

```
@Bean("tomcatWebServerFactory")
@ConditionalOnMissingBean
public ServletWebServerFactory servletWebServerFactory(ServerProperties properties) {
    TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();

    factory.setContextPath(properties.getContextPath());
    factory.setPort(properties.getPort());

    return factory;
}
```

프로퍼티 빈의 후처리기 도입

프로퍼티 클래스를 빈 등록을 위한 자동 구성을 따로 만드는 필요한 곳에서 @Import 해서 사용할 수 있다.

@MyConfigurationProperties 라는 마커 애노테이션을 만들고, 이를 BeanPostProcessor를 만들어서 빈 오브젝트 생성 후에 후처리 작업을 진행시킬 수 있다.

```
public interface BeanPostProcessor {
  default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
    return bean;
  }

  default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    return bean;
  }
}
```

마커 애노테이션을 찾아서 프로퍼티를 주입하는 기능을 이 인터페이스를 구현해서 만들고 자동 구성으로 등록되게 한다.

```
@MyAutoConfiguration
public class PropertyPostProcessorConfig {
    @Bean BeanPostProcessor propertyPostProcessor(Environment env) {
        return new BeanPostProcessor() {
            @Override
            public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
                MyConfigurationProperties annotation = findAnnotation(bean.getClass(), MyConfigurationProperties.class);
                if (annotation == null) return bean;

                Map<String, Object> attrs = getAnnotationAttributes(annotation);
                String prefix = (String) attrs.get("prefix");

                return Binder.get(env).bindOrCreate(prefix, bean.getClass());
            }
        };
    }
}
```

마커 애노테이션에 prefix 엘리먼트를 지정하게 하고, 이를 이용해서 프로퍼티 이름 앞에 접두어를 붙이도록 할 수 있다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Component
public @interface MyConfigurationProperties {
    String prefix();
}
```

애노테이션과 ImportSelector를 조합해서 애노테이션의 엘리먼트 값으로 지정한 클래스를 빈으로 등록하는 방법도 가능하다.

```
public class MyConfigurationPropertiesImportSelector implements DeferredImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        MultiValueMap<String, Object> attr = importingClassMetadata.getAllAnnotationAttributes(EnableMyConfigurationProperties.class.g
        Class propertyClass = (Class) attr.getFirst("value");
        return new String[] { propertyClass.getName() };
    }
}
```

이때는 기본 엘리먼트 타입을 Class 타입으로 만들어 사용한다.