7. 외부설정과 프로필2

#1.인강/9. 스프링부트/강의#

프로젝트 설정

프로젝트 설정 순서

1. `external-read-start`의 폴더 이름을 `external-read`로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
```

```
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //test lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
- `Lombok`
- 테스트 코드에서 lombok을 사용할 수 있도록 설정을 추가했다.

실행

`ExternalReadApplication.main()`을 실행해서 해당 메인 메서드가 실행되고, 정상 종료되면 성공이다.

외부 설정 사용 - Environment

다음과 같은 외부 설정들은 스프링이 제공하는 `Environment`를 통해서 일관된 방식으로 조회할 수 있다.

외부 설정

- 설정 데이터(`application.properties`)
- OS 환경변수
- 자바 시스템 속성
- 커맨드 라인 옵션 인수

다양한 외부 설정 읽기

스프링은 `Environment`는 물론이고 `Environment`를 활용해서 더 편리하게 외부 설정을 읽는 방법들을 제공한다.

스프링이 지원하는 다양한 외부 설정 조회 방법

- `Environment`
- `@Value` - 값 주입
- `@ConfigurationProperties` - 타입 안전한 설정 속성

이번 시간에는 조금 복잡한 예제를 가지고 외부 설정을 읽어서 활용하는 다양한 방법들을 학습해보자.
예제에서는 가상의 데이터소스를 하나 만들고, 여기에 필요한 속성들을 외부 설정값으로 채운 다음 스프링 빈으로 등록할 것이다. 이 예제는 외부 설정값을 어떤식으로 활용하는지 이해를 돕기 위해 만들었고, 실제 DB에 접근하지는 않는다.

MyDataSource

```
package hello.datasource;

import jakarta.annotation.PostConstruct;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;

import java.time.Duration;
import java.util.List;

@Slf4j
@Data
public class MyDataSource {

    private String url;
    private String username;
    private String password;
    private int maxConnection;

    private Duration timeout;

    private List<String> options;

    public MyDataSource(String url, String username, String password, int
maxConnection, Duration timeout, List<String> options) {
        this.url = url;
        this.username = username;
        this.password = password;
```

```
        this.maxConnection = maxConnection;
        this.timeout = timeout;
        this.options = options;
    }

    @PostConstruct
    public void init() {
        log.info("url={}", url);
        log.info("username={}", username);
        log.info("password={}", password);
        log.info("maxConnection={}", maxConnection);
        log.info("timeout={}", timeout);
        log.info("options={}", options);
    }
}
```

- `url`, `username`, `password`: 접속 url, 이름, 비밀번호
- `maxConnection`: 최대 연결 수
- `timeout`: 응답 지연시 타임아웃
- `options`: 연결시 사용하는 기타 옵션들

`@PostConstruct`에서 확인을 위해 설정된 값을 출력한다.

application.properties

```
my.datasource.url=local.db.com
my.datasource.username=local_user
my.datasource.password=local_pw
my.datasource.etc.max-connection=1
my.datasource.etc.timeout=3500ms
my.datasource.etc.options=CACHE,ADMIN
```

- 외부 속성은 설정 데이터(`appliation.properties`)를 사용한다.
- 여기서는 별도의 프로필은 사용하지 않았다. 환경에 따라서 다른 설정값이 필요하다면 각 환경에 맞는 프로필을 적용하면 된다.

> **참고 - properties 캐밥 표기법**
> `properties`는 자바의 낙타 표기법(`maxConnection`)이 아니라 소문자와 `-`(dash)를 사용하는 캐밥 표기법(`max-connection`)을 주로 사용한다. 참고로 이곳에 자바의 낙타 표기법을 사용한다고 해서 문제가 되는 것은 아니다. 스프링은 `properties`에 캐밥 표기법을 권장한다.

이제 외부 속성을 읽어서 앞서 만든 `MyDataSource`에 값을 설정하고 스프링 빈으로 등록해보자.

MyDataSourceEnvConfig

```
package hello.config;

import hello.datasource.MyDataSource;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;

import java.time.Duration;
import java.util.List;

@Slf4j
@Configuration
public class MyDataSourceEnvConfig {

    private final Environment env;

    public MyDataSourceEnvConfig(Environment env) {
        this.env = env;
    }

    @Bean
    public MyDataSource myDataSource() {
        String url = env.getProperty("my.datasource.url");
        String username = env.getProperty("my.datasource.username");
        String password = env.getProperty("my.datasource.password");
        int maxConnection = env.getProperty("my.datasource.etc.max-connection",
Integer.class);
        Duration timeout = env.getProperty("my.datasource.etc.timeout",
Duration.class);
        List<String> options = env.getProperty("my.datasource.etc.options",
List.class);

        return new MyDataSource(url, username, password, maxConnection, timeout,
options);
    }
}
```

- `MyDataSource`를 스프링 빈으로 등록하는 자바 설정이다.
- `Environment`를 사용하면 외부 설정의 종류와 관계없이 코드 안에서 일관성 있게 외부 설정을 조회할 수 있다.
- `Environment.getProperty(key, Type)`를 호출할 때 타입 정보를 주면 해당 타입으로 변환해준다. (스

프링 내부 변환기가 작동한다.)

- `env.getProperty("my.datasource.etc.max-connection", Integer.class)` : 문자 ➜ 숫자로 변환
- `env.getProperty("my.datasource.etc.timeout", Duration.class)` : 문자 ➜ `Duration` (기간) 변환
- `env.getProperty("my.datasource.etc.options", List.class)` : 문자 ➜ `List` 변환 (`A,B` ➜ `[A,B]`)

스프링은 다양한 타입들에 대해서 기본 변환 기능을 제공한다.

> **속성 변환기 - 스프링 공식 문서**
> https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config.typesafe-configuration-properties.conversion

ExternalReadApplication - 수정

```
package hello;

import hello.config.MyDataSourceEnvConfig;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;

@Import(MyDataSourceEnvConfig.class)
@SpringBootApplication(scanBasePackages = "hello.datasource")
public class ExternalReadApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExternalReadApplication.class, args);
    }

}
```

- 설정 정보를 빈으로 등록해서 사용하기 위해 `@Import(MyDataSourceEnvConfig.class)` 를 추가했다.
- `@SpringBootApplication(scanBasePackages = "hello.datasource")`
  - 예제에서는 `@Import` 로 설정 정보를 계속 변경할 예정이므로, 설정 정보를 바꾸면서 사용하기 위해 `hello.config` 의 위치를 피해서 컴포넌트 스캔 위치를 설정했다.
  - `scanBasePackages` 설정을 하지 않으면 현재 위치인 `hello` 패키지부터 그 하위가 모두 컴포넌트 스캔이 된다. 따라서 `@Configuration` 을 포함하고 있는 `MyDataSourceEnvConfig` 이 항상 컴포넌트 스캔의 대상이 된다.

실행 결과

```
url=local.db.com
username=local_user
password=local_pw
maxConnection=1
timeout=PT3.5S
options=[CACHE, ADMIN]
```

정리

`application.properties`에 필요한 외부 설정을 추가하고, `Environment`를 통해서 해당 값들을 읽어서, `MyDataSource`를 만들었다. 향후 외부 설정 방식이 달라져도, 예를 들어서 설정 데이터(`application.properties`)를 사용하다가 커맨드 라인 옵션 인수나 자바 시스템 속성으로 변경해도 애플리케이션 코드를 그대로 유지할 수 있다.

단점

이 방식의 단점은 `Environment`를 직접 주입받고, `env.getProperty(key)`를 통해서 값을 꺼내는 과정을 반복해야 한다는 점이다. 스프링은 `@Value`를 통해서 외부 설정값을 주입 받는 더욱 편리한 기능을 제공한다.

외부설정 사용 - @Value

`@Value`를 사용하면 외부 설정값을 편리하게 주입받을 수 있다.
참고로 `@Value`도 내부에서는 `Environment`를 사용한다.

MyDataSourceValueConfig

```
package hello.config;

import hello.datasource.MyDataSource;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;



import java.time.Duration;
import java.util.List;

@Slf4j
```

```java
@Configuration
public class MyDataSourceValueConfig {

    @Value("${my.datasource.url}")
    private String url;
    @Value("${my.datasource.username}")
    private String username;
    @Value("${my.datasource.password}")
    private String password;
    @Value("${my.datasource.etc.max-connection}")
    private int maxConnection;
    @Value("${my.datasource.etc.timeout}")
    private Duration timeout;
    @Value("${my.datasource.etc.options}")
    private List<String> options;

    @Bean
    public MyDataSource myDataSource1() {
        return new MyDataSource(url, username, password, maxConnection, timeout,
options);
    }

    @Bean
    public MyDataSource myDataSource2(
            @Value("${my.datasource.url}") String url,
            @Value("${my.datasource.username}") String username,
            @Value("${my.datasource.password}") String password,
            @Value("${my.datasource.etc.max-connection}") int maxConnection,
            @Value("${my.datasource.etc.timeout}") Duration timeout,
            @Value("${my.datasource.etc.options}") List<String> options) {

        return new MyDataSource(url, username, password, maxConnection, timeout,
options);
    }

}
```

- `@Value`에 `${}`를 사용해서 외부 설정의 키 값을 주면 원하는 값을 주입 받을 수 있다.
- `@Value`는 필드에 사용할 수도 있고, 파라미터에 사용할 수도 있다.
  - `myDataSource1()`은 필드에 주입 받은 설정값을 사용한다.
  - `myDataSource2()`는 파라미터를 통해서 설정 값을 주입 받는다.

기본값

만약 키를 찾지 못할 경우 코드에서 기본값을 사용하려면 다음과 같이 `:` 뒤에 기본값을 적어주면 된다.

- 예) `@Value("${my.datasource.etc.max-connection:1}")`: `key`가 없는 경우 `1`을 사용한다.

ExternalReadApplication - 수정

```
package hello;

import hello.config.*;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;

//@Import(MyDataSourceEnvConfig.class)
@Import(MyDataSourceValueConfig.class)
@SpringBootApplication(scanBasePackages = "hello.datasource")
public class ExternalReadApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExternalReadApplication.class, args);
    }

}
```

- `@Import(MyDataSourceEnvConfig.class)`를 주석처리 한다.
- `@Import(MyDataSourceValueConfig.class)`를 추가 한다.

실행 결과

```
url=local.db.com
username=local_user
password=local_pw
maxConnection=1
timeout=PT3.5S
options=[CACHE, ADMIN]
```

스프링 빈을 2개 등록해서 같은 실행 결과가 두 번 나온다.

정리

`application.properties`에 필요한 외부 설정을 추가하고, `@Value`를 통해서 해당 값들을 읽어서, `MyDataSource`를 만들었다.

단점

`@Value`를 사용하는 방식도 좋지만, `@Value`로 하나하나 외부 설정 정보의 키 값을 입력받고, 주입 받아와야 하는 부

분이 번거롭다. 그리고 설정 데이터를 보면 하나하나 분리되어 있는 것이 아니라 정보의 묶음으로 되어 있다. 여기서는 `my.datasource` 부분으로 묶여있다. 이런 부분을 객체로 변환해서 사용할 수 있다면 더 편리하고 더 좋을 것이다.

외부설정 사용 - @ConfigurationProperties 시작

Type-safe Configuration Properties

스프링은 외부 설정의 묶음 정보를 객체로 변환하는 기능을 제공한다. 이것을 **타입 안전한 설정 속성**이라 한다.
객체를 사용하면 타입을 사용할 수 있다. 따라서 실수로 잘못된 타입이 들어오는 문제도 방지할 수 있고, 객체를 통해서 활용할 수 있는 부분들이 많아진다. 쉽게 이야기해서 외부 설정을 자바 코드로 관리할 수 있는 것이다. 그리고 설정 정보 그 자체도 타입을 가지게 된다.

MyDataSourcePropertiesV1

```
package hello.datasource;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

import java.time.Duration;
import java.util.ArrayList;
import java.util.List;

@Data
@ConfigurationProperties("my.datasource")
public class MyDataSourcePropertiesV1 {

    private String url;
    private String username;
    private String password;
    private Etc etc = new Etc();

    @Data
    public static class Etc {
        private int maxConnection;
        private Duration timeout;
        private List<String> options = new ArrayList<>();
    }

}
```

- 외부 설정을 주입 받을 객체를 생성한다. 그리고 각 필드를 외부 설정의 키 값에 맞추어 준비한다.
- `@ConfigurationProperties`이 있으면 외부 설정을 주입 받는 객체라는 뜻이다. 여기에 외부 설정 KEY의 묶음 시작점인 `my.datasource`를 적어준다.
- 기본 주입 방식은 자바빈 프로퍼티 방식이다. `Getter`, `Setter`가 필요하다. (롬복의 `@Data`에 의해 자동 생성된다.)

설정 속성을 실제 어떻게 사용하는지 확인해보자.

MyDataSourceConfigV1

```
package hello.config;

import hello.datasource.MyDataSource;
import hello.datasource.MyDataSourcePropertiesV1;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;

@Slf4j
@EnableConfigurationProperties(MyDataSourcePropertiesV1.class)
public class MyDataSourceConfigV1 {

    private final MyDataSourcePropertiesV1 properties;

    public MyDataSourceConfigV1(MyDataSourcePropertiesV1 properties) {
        this.properties = properties;
    }

    @Bean
    public MyDataSource dataSource() {
        return new MyDataSource(
                properties.getUrl(),
                properties.getUsername(),
                properties.getPassword(),
                properties.getEtc().getMaxConnection(),
                properties.getEtc().getTimeout(),
                properties.getEtc().getOptions());
    }

}
```

- `@EnableConfigurationProperties(MyDataSourcePropertiesV1.class)`

    - 스프링에게 사용할 `@ConfigurationProperties`를 지정해주어야 한다. 이렇게 하면 해당 클래스는 스프링 빈으로 등록되고, 필요한 곳에서 주입 받아서 사용할 수 있다.
- `private final MyDataSourcePropertiesV1 properties` 설정 속성을 생성자를 통해 주입 받아서 사용한다.

ExternalReadApplication - 수정

```
//@Import(MyDataSourceValueConfig.class)
@Import(MyDataSourceConfigV1.class)
@SpringBootApplication(scanBasePackages = "hello.datasource")
public class ExternalReadApplication {...}
```

- `@Import(MyDataSourceValueConfig.class)`를 주석처리 한다.
- `@Import(MyDataSourceConfigV1.class)`를 추가 한다.

실행 결과

```
url=local.db.com
username=local_user
password=local_pw
maxConnection=1
timeout=PT3.5S
options=[CACHE, ADMIN]
```

타입 안전

`ConfigurationProperties`를 사용하면 타입 안전한 설정 속성을 사용할 수 있다.

`maxConnection=abc`로 입력하고 실행해보자.

실행 결과

```
Failed to bind properties under 'my.datasource.etc.max-connection' to int:

    Property: my.datasource.etc.max-connection
    Value: "abc"
    Origin: class path resource [application.properties] - 4:34
    Reason: failed to convert java.lang.String to int ...
```

실행 결과를 보면 숫자가 들어와야 하는데 문자가 들어와서 오류가 발생한 것을 확인할 수 있다. 타입이 다르면 오류가 발생하는 것이다. 실수로 숫자를 입력하는 곳에 문자를 입력하는 문제를 방지해준다. 그래서 타입 안전한 설정 속성이라고 한다. `ConfigurationProperties`로 만든 외부 데이터는 타입에 대해서 믿고 사용할 수 있다.

정리

- `application.properties`에 필요한 외부 설정을 추가하고, `@ConfigurationProperties`를 통해서

`MyDataSourcePropertiesV1`에 외부 설정의 값들을 설정했다. 그리고 해당 값들을 읽어서 `MyDataSource`를 만들었다.

표기법 변환

- `maxConnection`은 표기법이 서로 다르다. 스프링은 캐밥 표기법을 자바 낙타 표기법으로 중간에서 자동으로 변환해준다.
  - `application.properties`에서는 `max-connection`
  - 자바 코드에서는 `maxConnection`

@ConfigurationPropertiesScan

- `@ConfigurationProperties`를 하나하나 직접 등록할 때는 `@EnableConfigurationProperties`를 사용한다.
  - `@EnableConfigurationProperties(MyDataSourcePropertiesV1.class)`
- `@ConfigurationProperties`를 특정 범위로 자동 등록할 때는 `@ConfigurationPropertiesScan`을 사용하면 된다.

`@ConfigurationPropertiesScan` 예시

```
@SpringBootApplication
@ConfigurationPropertiesScan({ "com.example.app", "com.example.another" })
public class MyApplication {}
```

빈을 직접 등록하는 것과 컴포넌트 스캔을 사용하는 차이와 비슷하다.

문제

`MyDataSourcePropertiesV1`은 스프링 빈으로 등록된다. 그런데 `Setter`를 가지고 있기 때문에 누군가 실수로 값을 변경하는 문제가 발생할 수 있다. 여기에 있는 값들은 외부 설정값을 사용해서 초기에만 설정되고, 이후에는 변경하면 안된다. 이럴 때 `Setter`를 제거하고 대신에 생성자를 사용하면 중간에 데이터를 변경하는 실수를 근본적으로 방지할 수 있다.
이런 문제가 없을 것 같지만, 한번 발생하면 정말 잡기 어려운 버그가 만들어진다.
대부분의 개발자가 `MyDataSourcePropertiesV1`의 값은 변경하면 안된다고 인지하고 있지만, 어떤 개발자가 자신의 문제를 해결하기 위해 `setter`를 통해서 값을 변경하게 되면, 애플리케이션 전체에 심각한 버그를 유발할 수 있다. 좋은 프로그램은 제약이 있는 프로그램이다.

외부설정 사용 - @ConfigurationProperties 생성자

`@ConfigurationProperties`는 Getter, Setter를 사용하는 자바빈 프로퍼티 방식이 아니라 생성자를 통해서 객체를 만드는 기능도 지원한다. 다음 코드를 통해서 확인해보자.

MyDataSourcePropertiesV2

```
package hello.datasource;

import lombok.Getter;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;

import java.time.Duration;
import java.util.List;

@Getter
@ConfigurationProperties("my.datasource")
public class MyDataSourcePropertiesV2 {

    private String url;
    private String username;
    private String password;
    private Etc etc;

    public MyDataSourcePropertiesV2(String url, String username, String
password, @DefaultValue Etc etc) {
        this.url = url;
        this.username = username;
        this.password = password;
        this.etc = etc;
    }

    @Getter
    public static class Etc {
        private int maxConnection;
        private Duration timeout;
        private List<String> options;

        public Etc(int maxConnection, Duration timeout, @DefaultValue("DEFAULT")
List<String> options) {
            this.maxConnection = maxConnection;
```

```
            this.timeout = timeout;
            this.options = options;
        }
    }

}
```

- 생성자를 만들어 두면 생성자를 통해서 설정 정보를 주입한다.
- `@Getter` 롬복이 자동으로 `getter`를 만들어준다.
- `@DefaultValue`: 해당 값을 찾을 수 없는 경우 기본값을 사용한다.
  - `@DefaultValue Etc etc`
    - `etc`를 찾을 수 없을 경우 `Etc` 객체를 생성하고 내부에 들어가는 값은 비워둔다. (`null`, `0`)
  - `@DefaultValue("DEFAULT") List<String> options`
    - `options`를 찾을 수 없을 경우 `DEFAULT`라는 이름의 값을 사용한다.

> **참고** `@ConstructorBinding`
> 스프링 부트 3.0 이전에는 생성자 바인딩 시에 `@ConstructorBinding` 애노테이션을 필수로 사용해야 했다.
> 스프링 부트 3.0 부터는 생성자가 하나일 때는 생략할 수 있다. 생성자가 둘 이상인 경우에는 사용할 생성자에 `@ConstructorBinding` 애노테이션을 적용하면 된다.

`MyDataSourcePropertiesV2`를 사용해보자.

MyDataSourceConfigV2

```
package hello.config;

import hello.datasource.MyDataSource;
import hello.datasource.MyDataSourcePropertiesV2;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;

@Slf4j
@EnableConfigurationProperties(MyDataSourcePropertiesV2.class)
public class MyDataSourceConfigV2 {

    private final MyDataSourcePropertiesV2 properties;

    public MyDataSourceConfigV2(MyDataSourcePropertiesV2 properties) {
        this.properties = properties;
```

```
    }

    @Bean
    public MyDataSource dataSource() {
        return new MyDataSource(
                properties.getUrl(),
                properties.getUsername(),
                properties.getPassword(),
                properties.getEtc().getMaxConnection(),
                properties.getEtc().getTimeout(),
                properties.getEtc().getOptions());
    }

}
```

- `MyDataSourcePropertiesV2`를 적용하고 빈을 등록한다. 기존 코드와 크게 다르지 않다.

ExternalReadApplication - 수정

```
//@Import(MyDataSourceConfigV1.class)
@Import(MyDataSourceConfigV2.class)
@SpringBootApplication(scanBasePackages = "hello.datasource")
public class ExternalReadApplication {...}
```

- `@Import(MyDataSourceConfigV1.class)`를 주석처리 한다.
- `@Import(MyDataSourceConfigV2.class)`를 추가 한다.

실행 결과

```
url=local.db.com
username=local_user
password=local_pw
maxConnection=1
timeout=PT3.5S
options=[CACHE, ADMIN]
```

정리

`application.properties`에 필요한 외부 설정을 추가하고, `@ConfigurationProperties`의 생성자 주입을 통해서 값을 읽어들였다. `Setter`가 없으므로 개발자가 중간에 실수로 값을 변경하는 문제가 발생하지 않는다.

문제

타입과 객체를 통해서 숫자에 문자가 들어오는 것 같은 기본적인 타입 문제들은 해결이 되었다. 그런데 타입은 맞는데 숫자의 범위가 기대하는 것과 다르면 어떻게 될까? 예를 들어서 `max-conneciton`의 값을 `0`으로 설정하면 커넥션

이 하나도 만들어지지 않는 심각한 문제가 발생한다고 가정해보자.
`max-conneciton`은 최소 `1` 이상으로 설정하지 않으면 애플리케이션 로딩 시점에 예외를 발생시켜서 빠르게 문제를 인지할 수 있도록 하고 싶다.

외부설정 사용 - @ConfigurationProperties 검증

`@ConfigurationProperties`를 통해서 숫자가 들어가야 하는 부분에 문자가 입력되는 문제와 같은 타입이 맞지 않는 데이터를 입력하는 문제는 예방할 수 있다. 그런데 문제는 숫자의 범위라던가, 문자의 길이 같은 부분은 검증이 어렵다.
예를 들어서 최대 커넥션 숫자는 최소 `1` 최대 `999`라는 범위를 가져야 한다면 어떻게 검증할 수 있을까? 이메일을 외부 설정에 입력했는데, 만약 이메일 형식에 맞지 않는다면 어떻게 검증할 수 있을까?
개발자가 직접 하나하나 검증 코드를 작성해도 되지만, 자바에는 자바 빈 검증기(java bean validation)이라는 훌륭한 표준 검증기가 제공된다.

`@ConfigurationProperties`은 자바 객체이기 때문에 스프링이 자바 빈 검증기를 사용할 수 있도록 지원한다.

자바 빈 검증기를 사용하려면 `spring-boot-starter-validation`이 필요하다. `build.gradle`에 다음 코드를 추가하자.

`build.gradle`

```
implementation 'org.springframework.boot:spring-boot-starter-validation' //추가
```

검증기를 추가해서 `ConfigurationProperties`을 만들어보자.

MyDataSourcePropertiesV3

```
package hello.datasource;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotEmpty;
import lombok.Getter;
import org.hibernate.validator.constraints.time.DurationMax;
import org.hibernate.validator.constraints.time.DurationMin;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;
```

```java
import java.time.Duration;
import java.util.List;

@Getter
@ConfigurationProperties("my.datasource")
@Validated
public class MyDataSourcePropertiesV3 {

    @NotEmpty
    private String url;
    @NotEmpty
    private String username;
    @NotEmpty
    private String password;
    private Etc etc;

    public MyDataSourcePropertiesV3(String url, String username, String
password, Etc etc) {
        this.url = url;
        this.username = username;
        this.password = password;
        this.etc = etc;
    }

    @Getter
    public static class Etc {
        @Min(1)
        @Max(999)
        private int maxConnection;
        @DurationMin(seconds = 1)
        @DurationMax(seconds = 60)
        private Duration timeout;
        private List<String> options;

        public Etc(int maxConnection, Duration timeout, List<String> options) {
            this.maxConnection = maxConnection;
            this.timeout = timeout;
            this.options = options;
        }
    }

}
```

- `@NotEmpty` `url`, `username`, `password`는 항상 값이 있어야 한다. 필수 값이 된다.
- `@Min(1) @Max(999) maxConnection`: 최소 `1`, 최대 `999`의 값을 허용한다.
- `@DurationMin(seconds = 1) @DurationMax(seconds = 60)`: 최소 1, 최대 60초를 허용한다.

`jakarta.validation.constraints.Max`
패키지 이름에 `jakarta.validation`으로 시작하는 것은 자바 표준 검증기에서 지원하는 기능이다.

`org.hibernate.validator.constraints.time.DurationMax`
패키지 이름에 `org.hibernate.validator`로 시작하는 것은 자바 표준 검증기에서 아직 표준화 된 기능은 아니고, 하이버네이트 검증기라는 표준 검증기의 구현체에서 직접 제공하는 기능이다. 대부분 하이버네이트 검증기를 사용하므로 이 부분이 크게 문제가 되지는 않는다.

MyDataSourceConfigV3

```
package hello.config;

import hello.datasource.MyDataSource;
import hello.datasource.MyDataSourcePropertiesV3;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;

@Slf4j
@EnableConfigurationProperties(MyDataSourcePropertiesV3.class)
public class MyDataSourceConfigV3 {

    private final MyDataSourcePropertiesV3 properties;

    public MyDataSourceConfigV3(MyDataSourcePropertiesV3 properties) {
        this.properties = properties;
    }

    @Bean
    public MyDataSource dataSource() {
        return new MyDataSource(
                properties.getUrl(),
                properties.getUsername(),
                properties.getPassword(),
                properties.getEtc().getMaxConnection(),
                properties.getEtc().getTimeout(),
                properties.getEtc().getOptions());
```

```
    }

}
```

- `MyDataSourceConfigV3`은 기존 코드와 크게 다르지 않다.

ExternalReadApplication - 수정

```
//@Import(MyDataSourceConfigV2.class)
@Import(MyDataSourceConfigV3.class)
@SpringBootApplication(scanBasePackages = "hello.datasource")
public class ExternalReadApplication {...}
```

- `@Import(MyDataSourceConfigV2.class)`를 주석처리 한다.
- `@Import(MyDataSourceConfigV3.class)`를 추가 한다.

값이 검증 범위를 넘어서게 설정해보자

그러면 애플리케이션 로딩 시점에 다음과 같은 오류 메시지를 확인 할 수 있다.

`maxConnection=0`으로 설정한 예

```
    Property: my.datasource.etc.maxConnection
    Value: "0"
    Origin: class path resource [application.properties] - 4:34
    Reason: 1 이상이어야 합니다
```

정상 실행 결과

```
url=local.db.com
username=local_user
password=local_pw
maxConnection=1
timeout=PT3.5S
options=[CACHE, ADMIN]
```

정리

`ConfigurationProperties` 덕분에 타입 안전하고, 또 매우 편리하게 외부 설정을 사용할 수 있다. 그리고 검증기 덕분에 쉽고 편리하게 설정 정보를 검증할 수 있다.

가장 좋은 예외는 컴파일 예외, 그리고 애플리케이션 로딩 시점에 발생하는 예외이다. 가장 나쁜 예외는 고객 서비스 중에 발생하는 런타임 예외이다.

ConfigurationProperties 장점

- 외부 설정을 객체로 편리하게 변환해서 사용할 수 있다.

- 외부 설정의 계층을 객체로 편리하게 표현할 수 있다.
- 외부 설정을 타입 안전하게 사용할 수 있다.
- 검증기를 적용할 수 있다.

YAML

스프링은 설정 데이터를 사용할 때 `application.properties` 뿐만 아니라 `application.yml`이라는 형식도 지원한다.

YAML

YAML(YAML Ain't Markup Language)은 사람이 읽기 좋은 데이터 구조를 목표로 한다. 확장자는 `yaml`, `yml`이다. 주로 `yml`을 사용한다.

`application.properties` 예시

```
environments.dev.url=https://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=https://another.example.com
environments.prod.name=My Cool App
```

`application.yml` 예시

```
environments:
  dev:
    url: "https://dev.example.com"
    name: "Developer Setup"
  prod:
    url: "https://another.example.com"
    name: "My Cool App"
```

- YAML의 가장 큰 특징은 사람이 읽기 좋게 계층 구조를 이룬다는 점이다.
- YAML은 `space`(공백)로 계층 구조를 만든다. `space`는 1칸을 사용해도 되는데, 보통 2칸을 사용한다. 일관성 있게 사용하지 않으면 읽기 어렵거나 구조가 깨질 수 있다.
- 구분 기호로 `:`를 사용한다. 만약 값이 있다면 이렇게 `key: value` `:` 이후에 공백을 하나 넣고 값을 넣어주면 된다.

스프링은 YAML의 계층 구조를 `properties`처럼 평평하게 만들어서 읽어들인다. 쉽게 이야기해서 위의 `application.yml` 예시는 `application.properties` 예시처럼 만들어진다.

적용

프로젝트에 적용해보자.

`application.properties`를 사용하지 않도록 파일 이름을 변경하자.

`application.properties` ➜ `application_backup.properties`

```
my.datasource.url=local.db.com
my.datasource.username=local_user
my.datasource.password=local_pw
my.datasource.etc.max-connection=1
my.datasource.etc.timeout=3500ms
my.datasource.etc.options=CACHE,ADMIN
```

`src/main/resources/application.yml`를 생성하자.

```
my:
  datasource:
    url: local.db.com
    username: local_user
    password: local_pw
    etc:
      max-connection: 1
      timeout: 60s
      options: LOCAL, CACHE
```

실행해보면 `application.yml`에 입력한 설정 데이터가 조회되는 것을 확인할 수 있다.

> **주의**
>
> `application.properties`, `application.yml`을 같이 사용하면 `application.properties`가 우선권을 가진다.
>
> 이것을 둘이 함께 사용하는 것은 일관성이 없으므로 권장하지 않는다.
>
> 참고로 실무에서는 설정 정보가 많아서 보기 편한 `yml`을 선호한다.

YML과 프로필

YML에도 프로필을 적용할 수 있다.

`application.yml`

```yaml
my:
  datasource:
    url: local.db.com
    username: local_user
    password: local_pw
    etc:
      maxConnection: 2
      timeout: 60s
      options: LOCAL, CACHE
---
spring:
  config:
    activate:
      on-profile: dev
my:
  datasource:
    url: dev.db.com
    username: dev_user
    password: dev_pw
    etc:
      maxConnection: 10
      timeout: 60s
      options: DEV, CACHE
---
spring:
  config:
    activate:
      on-profile: prod
my:
  datasource:
    url: prod.db.com
    username: prod_user
    password: prod_pw
    etc:
      maxConnection: 50
      timeout: 10s
      options: PROD, CACHE
```

- `yml`은 `---` dash(`-`) 3개를 사용해서 논리 파일을 구분한다.
- `spring.config.active.on-profile`을 사용해서 프로필을 적용할 수 있다.
- 나머지는 `application.properties`와 동일하다.

@Profile

프로필과 외부 설정을 사용해서 각 환경마다 설정값을 다르게 적용하는 것은 이해했다.

그런데 설정값이 다른 정도가 아니라 각 환경마다 서로 다른 빈을 등록해야 한다면 어떻게 해야할까?

예를 들어서 결제 기능을 붙여야 하는데, 로컬 개발 환경에서는 실제 결제가 발생하면 문제가 되니 가짜 결제 기능이 있는 스프링 빈을 등록하고, 운영 환경에서는 실제 결제 기능을 제공하는 스프링 빈을 등록한다고 가정해보자.

PayClient

```
package hello.pay;

public interface PayClient {
    void pay(int money);
}
```

- DI를 적극 활용하기 위해 인터페이스를 사용한다.

LocalPayClient

```
package hello.pay;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class LocalPayClient implements PayClient {
    @Override
    public void pay(int money) {
        log.info("로컬 결제 money={}", money);
    }
}
```

- 로컬 개발 환경에서는 실제 결제를 하지 않는다.

ProdPayClient

```
package hello.pay;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class ProdPayClient implements PayClient {
    @Override
    public void pay(int money) {
```

```
        log.info("운영 결제 money={}", money);
    }
}
```

- 운영 환경에서는 실제 결제를 시도한다.

OrderService

```
package hello.pay;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class OrderService {

    private final PayClient payClient;

    public void order(int money) {
        payClient.pay(money);
    }
}
```

`PayClient`를 사용하는 부분이다. 상황에 따라서 `LocalPayClient` 또는 `ProdPayClient`를 주입받는다.

PayConfig

```
package hello.pay;

import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Slf4j
@Configuration
public class PayConfig {

    @Bean
    @Profile("default")
    public LocalPayClient localPayClient() {
        log.info("LocalPayClient 빈 등록");
        return new LocalPayClient();
    }
```

```
    @Bean
    @Profile("prod")
    public ProdPayClient prodPayClient() {
        log.info("ProdPayClient 빈 등록");
        return new ProdPayClient();
    }
}
```

- `@Profile` 애노테이션을 사용하면 해당 프로필이 활성화된 경우에만 빈을 등록한다.
  - `default` 프로필(기본값)이 활성화 되어 있으면 `LocalPayClient`를 빈으로 등록한다.
  - `prod` 프로필이 활성화 되어 있으면 `ProdPayClient`를 빈으로 등록한다.

RunOrder

```
package hello.pay;

import lombok.RequiredArgsConstructor;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
@RequiredArgsConstructor
public class OrderRunner implements ApplicationRunner {

    private final OrderService orderService;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        orderService.order(1000);
    }

}
```

`ApplicationRunner` 인터페이스를 사용하면 스프링은 빈 초기화가 모두 끝나고 애플리케이션 로딩이 완료되는 시점에 `run(args)` 메서드를 호출해준다.

ExternalReadApplication 변경

```
package hello;

import hello.config.*;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;
```

```
@Import(MyDataSourceConfigV3.class)
@SpringBootApplication(scanBasePackages = {"hello.datasource", "hello.pay"})
public class ExternalReadApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExternalReadApplication.class, args);
    }

}
```

- 실행하기 전에 컴포넌트 스캔 부분에 `hello.pay` 패키지를 추가하자.

프로필 없이 실행

```
No active profile set, falling back to 1 default profile: "default"
LocalPayClient 빈 등록
...
로컬 결제 money=1000
```

- 프로필 없이 실행하면 `default` 프로필이 사용된다.
- `default` 프로필이 사용되면 `LocalPayClient`가 빈으로 등록되는 것을 확인할 수 있다.

prod 프로필 실행

`--spring.profiles.active=prod` 프로필 활성화 적용

```
The following 1 profile is active: "prod"
ProdPayClient 빈 등록
...
운영 결제 money=1000
```

- `prod` 프로필을 적용했다.
- `prod` 프로필이 사용되면 `ProdPayClient`가 빈으로 등록되는 것을 확인할 수 있다.

@Profile의 정체

```
package org.springframework.context.annotation;
...
@Conditional(ProfileCondition.class)
public @interface Profile {
    String[] value();
}
```

`@Profile`은 특정 조건에 따라서 해당 빈을 등록할지 말지 선택한다. 어디서 많이 본 것 같지 않은가? 바로

`@Conditional`이다.
코드를 보면 `@Conditional(ProfileCondition.class)`를 확인할 수 있다.
스프링은 `@Conditional` 기능을 활용해서 개발자가 더 편리하게 사용할 수 있는 `@Profile` 기능을 제공하는 것이다.

정리
`@Profile`을 사용하면 각 환경 별로 외부 설정 값을 분리하는 것을 넘어서, 등록되는 스프링 빈도 분리할 수 있다.

정리