Spring Boot에서 민감정보 은닉하기

1. DB 접근정보 분리하기

```
기존 application.properties에 정의해놓은 datasource에 대한 data들을 또다른
properties파일로 분류해 작성해준다.
본 필자는 config.properties로 작성했다.
이렇게 분류한 config.properties 파일은 별도 local 경로에 저장하던지 하여 프로젝트를 공유
하거나 할 때에 같이 올라가지 않도록 한다.
물리적으로 정보를 은닉하는 방법이다.

필자는 이해를 돕기위해 classpath 경로에 만들어두었다.
이해했다면 따로 빼서 관리하도록 하자!
간단하니 사진은 생략!
```

2. DB접근정보 Configuration 객체 만들기

```
config.properties로 분류해 작성해두었다.
이는 DB접근에 대한 내용으로 DB접근 시 필요한 정보이다.(당연히)
DB접근이 필요한 소스에서 Bean객체화 하여 사용할 것이다.

첫번째로 @Configuration을 이용해 등록하도록한다.
필자는 GlobalpropertySource.java파일로 작성했다.
```

```java
package com.example.demo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.annotation.PropertySources;

// Bean객체를 생성하기위한 class임을 명시한다.
@Configuration
// 가져올 설정파일의 경로를 입력한다.
// 현재는 classpath에 올렸지만 따로 관리하여 정보은닉을 하도록한다.
@PropertySources({
    @PropertySource( value = "classpath:config.properties", ignoreResourceNotFound = true ),
})
public class GlobalPropertySource {
    // ${가져올 값} -> 설정파일에서 정의한 값을 불러온다.
    @Value("${spring.datasource.driver-class-name}")
    private String driverClassName;

    @Value("${spring.datasource.url}")
    private String url;

    @Value("${spring.datasource.username}")
    private String username;

    @Value("${spring.datasource.password}")
    private String password;

    public String getDriverClassName() {
        return driverClassName;
    }

    public String getUrl() {
        return url;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

}
```

PropertySource로 정의되어있는 설정파일의 경로를 입력한다.
@value annotation을 통해 PropertySource경로에 정의되어있는 값에 접근할 수 있다.
getter를 생성하여 DataSourceBuilder로 생성할 수 있도록 만들어놓는다.

3. DB접근정보 실질적으로 사용하기!

현재 작성한 소스에서 DB접근을 위해 DataSource에 접근하는 부분이 있다.
demo/db/DatabaseConfig.java경로이다.

```java
@Configuration
@MapperScan(basePackages="com.example.demo.db.mapper")
@MapperScan(basePackages="com.example.demo.security.Mapper")
@EnableTransactionManagement
public class DatabaseConfig {

    @Autowired
    GlobalPropertySource globalPropertySource;
    // 원래라면 root-context.xml에서 정의했을 부분이다.
    // application.properties에서 정의한 dataSource의 정보를 이용한다.
    // mybatis 기본이용법이니 참고자료는 구글링하시길


    //DataSource를 custom하기 위해 사용한다.
    @Bean
    @Primary
    // 클래스에서 정의한 메소드를 사용해 DataSource형태를 정의한다.
    public DataSource customDataSource(){
      return DataSourceBuilder
      .create()
      .url(globalPropertySource.getUrl())
      .driverClassName(globalPropertySource.getDriverClassName())
      .username(globalPropertySource.getUsername())
      .password(globalPropertySource.getPassword())
      .build();
    }


    // 기존 parameter 에서 datasource로 자동설정된 정보를 사용했지만
    // custom한 DataSource 정보를 사용하도록 바꾸어준다.
    @Bean
    public SqlSessionFactory sqlSessionFactory(DataSource customDataSource) throws Exception {
        final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
        sessionFactory.setDataSource(customDataSource);
        PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
        sessionFactory.setMapperLocations(resolver.getResources("classpath:mybatis/mapper/*.xml"));
        return sessionFactory.getObject();
    }

    @Bean
    public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) throws Exception {
      final SqlSessionTemplate sqlSessionTemplate = new SqlSessionTemplate(sqlSessionFactory);
      return sqlSessionTemplate;
    }


}
```

`@Autowired`를 사용해 `GlobalPropertySource`를 불러온다.
이를 사용해 `@Bean,@Primary`인 `customDataSource()` 생성자를 작성한다.
이는 `DataSource`자료형을 반환하며 `DataSourceBuilder`를 통해 생성한다.
앞서 정의해놓은 `getter`를 사용해 만들어준다.

이 정보를 `sqlSessionFactory`에서 사용해주도록하자

이렇게 설정해놓고 실행시켜보도록 한다.
민감정보 은닉하기 완료!

Made by 장경석