7장 스프링 핵심 기술의 응용2

7.4 인터페이스 상속을 통한 안전한 기능확장

애플리케이션을 새로 시작하지 않고 특정 SQL의 내용만을 변경하고 싶다면 어떻게 해야할까.

7.4.1 DI와 기능의 확장

- DI를 의식하는 설계
  - 오브젝트들이 서로의 세부적인 구현에 얽매이지 않고 유연한 방식으로 의존관계를 맺으며 독립적으로 발전할 수 있게 해준다.

    유연하고 확장 가능한 좋은 오브젝트 설계와 DI 프로그래밍모델은 서로 상승 작용을 한다.
  - 최소한 두 개 이상의 의존관계를 가지고 서로 협력해서 일하는 오브젝트가 필요하고 적절한 책임에 따라 오브젝트를 분리해야줘야 한다.

    항상 의존 오브젝트는 자유롭게 확장될 수 있다는 점을 염두에 둬야 한다.

    DI 는 런타임시에 의존 오브젝트를 다이내믹하게 연결해줘서 유연한 확장을 하는게 목적이다.
  - DI란 결국 미래를 프로그래밍하는 것이다.
- DI와 인터페이스 프로그래밍
  - DI를 DI답게 만들려면 두 개의 오브젝트가 인터페이스를 통해 느슨하게 연결돼야 한다.
  - 다형성을 얻기 위해서
    - 프록시, 데코레이터, 어댑터, 테스트 대역 등의 다양한 목적을 위해 인터페이스를 통한 다형성이 활용된다.
  - 인터페이스 분리 원칙(ISP)을 통해 클라이언트와 의존 오브젝트 사이의 관계를 명확하게 해줄 수 있다.

    인터페이스는 하나의 오브젝트가 여러 개를 구현할 수 있으므로 하나의 오브젝트를 바라보는 창이 여러가지일 수 있다는 뜻이다.

    응집도가 높은 작은 단위로 설계해도 목적과 관심이 각기 다른 클라이언트가 있다면 인터페이스를 통해 이를 적절하게 분리해줄 필요가 있다.

7.4.2 인터페이스 상속

DI는 결국 잘 설계된 오브젝트 의존관계에 달려 있다. 인터페이스를 적절하게 분리하고 확장하는 방법을 통해 오브젝트 사이의 의존관계를 명확하게 해주고, 기존 의존관계에 영향을 주지 않으면서 유연한 확장성을 얻는 방법이 무엇인지 항상 고민해야한다.

7.5 DI를 이용해 다양한 구현 방법 적용하기

운영 중인 시스템에서 사용하는 정보를 실시간으로 변경하는 작업을 만들 때 가장 먼저 고려해야 할 사항은 동시성 문제다.

7.5.1 ConcurrentHashMap을 이용한 수정 가능 SQL 레지스트리

멀티스레드 환경에서 안전하게 HashMap을 조작하려면 동기화된 해시 데이터 조작에 최적화되도록 만들어진 ConcurrentHashMap을 사용하는 방법이 일반적으로 권장된다. ConcurrentHashMap은 데이터 조작 시 전체 데이터에 대해 락을 걸지 않고 조회는 락을 아예 사용하지 않는다.

- 수정 가능 SQL 레지스트리 테스트

  ▼ code

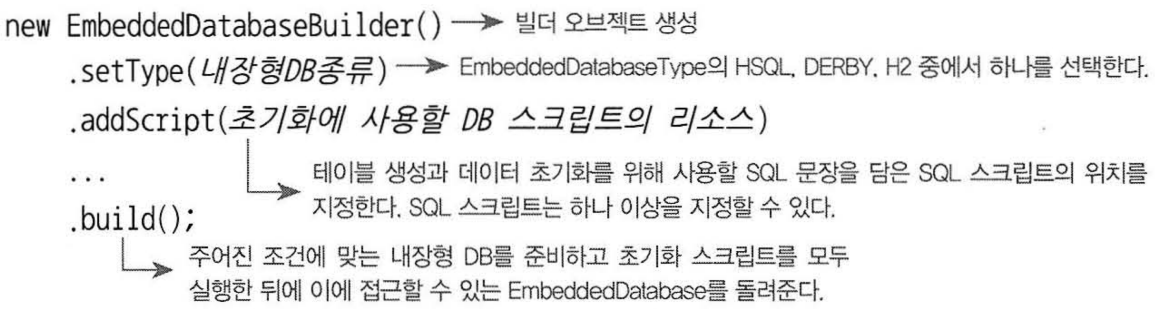
```
public class ConcurrentHashMapSqlRegistry implements UpdatableSqlRegistry {
    private Map<String, String> sqlMap = new ConcurrentHashMap<>();

    public String findSql(String key) throws SqlNotFoundExcpeion {
        String sql = sqlMap.get(key);
        if (sql == null) {
            throw new SqlNotFoundExcpeion();
        }
        return sql;
    }

    public void registerSql(String key, String sql) {
        sqlMap.put(key, sql);
    }

    public void updateSql(String key, String sql) throws SqlNotFoundExcpeion {
        if (sqlMap.get(key) == null) {
            throw new SqlNotFoundExcpeion();
        }
        sqlMap.put(key, sql);
    }

    public void updateSql(Map<String, String> sqlmap) throws SqlNotFoundExcpeion {
        for (Map.Entry<String, String> entry : sqlmap.entrySet()) {
            updateSql(entry.getKey(), entry.getValue());
        }
    }
}
```

7.5.2 내장형 데이터베이스를 이용한 SQL 레지스트리 만들기

저장되는 데이터의 양이 많아지고 잦은 조회와 변경이 일어나는 환경이라면 인덱스를 이용한 최적화된 검색을 지원하고 동시에 많은 요청을 처리하면서 안정적인 변경 작업이 가능한 데이터 베이스를 이용한다.

- 스프링의 내장형 DB 지원 기능

  스프링은 내장형 DB를 손쉽게 이용할 수 있도록 내장형 DB 지원기능(서비스추상화)을 제공하고 있다. 내장형 DB를 초기화 하는 작업을 지원하는 편리한 내장형 DB 빌더를 제공한다.



- 내장형 DB를 이용한 SqlRegistry 만들기

```
public class EmbeddedDbSqlRegistry implements UpdatableSqlRegistry {
    SimpleJdbcTemplate jdbc;
    void setDataSource(DataSource dataSource){
        jdbc = new SimpleJdbcTemplate(dataSource);
    }
```

- UpdatableSqlRegistry 테스트 코드의 재사용

DAO는 DB까지 연동하는 테스트를 하는 편이 훨씬 간편하다.

▼ code

```
public abstract class AbstractUpdatableSqlRegistryTest {
    // UpdatableSqlRegistry인터페이스를 구현한 모든 클래스에 대한 테스트를 만들때 사용할 수 있다.
    UpdatableSqlRegistry sqlRegistry;

    @BeforeAll
    void setUp(){
        sqlRegistry = createUpdatableRegistry();
        sqlRegistry.registrySql("KEY1", "SQL1");
        sqlRegistry.registrySql("KEY2", "SQL2");
        sqlRegistry.registrySql("KEY3", "SQL3");
    }

    // 테스트 픽스처를 생성하는 부분만 추상 메서드로 만들어두고 서브 클래스에서 이를 구현하도록 한다.
    abstract protected UpdatableSqlRegistry createUpdatableRegistry();

    @Test
    void find(){
        checkFindResult("SQL1", "SQL2", "SQL3");
    }

    private void checkFindResult(String sql1, String sql2, String sql3) {
        assertThat(sqlRegistry.findSql("KEY1")).is(sql1);
        assertThat(sqlRegistry.findSql("KEY2")).is(sql2);
        assertThat(sqlRegistry.findSql("KEY3")).is(sql3);
    }
}

public class ConcurrentHashMapSqlRegistryTest extends AbstractUpdatableSqlRegistryTest {
    @Override
    protected UpdatableSqlRegistry createUpdatableRegistry() {
        return new ConcurrentHashMapSqlRegistry();
    }
}
```

- XML 설정을 통한 내장형 DB의 생성과 적용

7.5.3 트랜잭션 적용

HashMap 같은 컬렉션은 트랜잭션 개념을 적용하기가 매우 어렵다.

- 다중 SQL 수정에 대한 트랜잭션 테스트
- 코드를 이용한 트랜잭션 적용

  EmbeddedDbSqlRegistry가 DataSource를 DI 받아서 트랜잭션 매니저와 템플릿을 만들게 한다. 번거롭게 빈으로 등록하는 대신 EmbeddedDbSqlRegistry 내부에서 TransactionTemplate을 이용해 트랜잭션 기능을 사용한다.

  ▼ code

```
public class EmbeddedDbSqlRegistry implements UpdatableSqlRegistry {
    SimpleJdbcTemplate jdbc;
    TransactionTemplate transactionTemplate;

    void setDataSource(DataSource dataSource) {
        jdbc = new SimpleJdbcTemplate(dataSource);
        transactionTemplate = new TransactionTemplate(
                new DataSourceTransactionManager(dataSource));
    }
    @Override
    public void updateSql(final Map<String, String> sqlmap) throws SqlNotFoundExcpeion {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            // 트랜잭션 템플릿이 만드는 트랜잭션 경계 안에서 동작할 코드를 콜백 형태로 만들고 TransactionTemplate 의 execute() 메소드에 전달한다.
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                for (Map.Entry<String, String> entry : sqlmap.entrySet()) {
                    updateSql(entry.getKey(), entry.getValue());
                }
```

```
            }
        });
    }
```

내장형 DB의 트랜잭션 격리수준 지원

- HSQL 의 경우 1.8이하에서는 READ_UNCOMMITTED(레벨0)
- 낮은 격리 수준의 위험성을 피하려면 READ_COMMITTED 격리수준을 지원하는 HSQL1.9이상을 사용하거나 H2, Derby를 사용해야 한다.

7.6 스프링 3.1의 DI

객체지향언어인 자바의 특징과 장점을 극대화하는 프로그래밍 스타일과 이를 지원하는 도구로서의 스프링 정체성은 변하지 않았다 → 구 버젼 호환성을 유지하고 있다.

- 자바 언어의 변화와 스프링
  - 애노테이션의 메타정보 활용

    본래 리플렉션 API는 자바 코드나 컴포넌트를 작성하는데 사용되는 툴을 개발할 때 이용하도록 만들어졌는데, 자바 코드의 메타 정보를 데이터로 활용하는 스타일의 프로그래밍 방식에 더 많이 활용되고 있다.

    애노테이션은 기존의 자바 프로그래밍 방식(그 자체로 실행가능하고 상속하거나 참조하거나 호출하는 방식등)으로 활용할 수 없고, 애플리케이션이 동작할 때 메모리에 로딩되기도 하지만 자바 코드가 실행되는데 직접 참여하지 못한다.

    복잡한 리플렉션 API를 이용해 애플리케이션의 메타정보를 조회하고, 애노테이션 내에 설정된 값을 가져와 참고하는 방법이 전부다.

    애노테이션은 애플리케이션을 핵심로직을 담은 자바 코드와 이를 지원하는 IoC방식의 프레임워크, 그리고 프레임워크가 참조하는 메타정보라는 세 가지로 구성하는 방식에 잘 어울리기 때문에 애노테이션 활용이 늘어났다.

    XML은 작성하기 편하고 빌드 과정이 필요없으며, AOP를 위해 빈 생성과 관계 설정을 재구성하는 경우를 고려하면 더 유리했기 때문에 오브젝트 관계 설정용 DI 메타 정보로 적극 활용됐다.

    애노테이션은 코드의 동작에 직접 영향을 주진 않지만, 메타 정보로서 활용되는 데는 XML보다 유리하다. 애노테이션은 타입, 필드,메소드, 파라미터, 생성자, 로컬 변수의 한군데 이상 적용이 가능하고 다양한 부가 정보를 얻어낼 수 있다.

    리팩토링할때 XML은 번거롭고 안전하지 못하다. 하지만 어느 환경에서나 편집이 가능하고 빌드를 다시할 필요가 없다. 반면에 애노테이션은 변경할때 마다 새로 컴파일해야 한다.

    자바 개발의 흐름은 점차 XML같은 텍스트 형태의 메타정보 활용을 자바 코드에 내장된 애노테이션으로 대체하는 쪽으로가고 있다.

    스프링3.1에서는 핵심 로직을 담은 자바 코드와 DI 프레임워크, 그리고 DI를 위한 메타데이터로서의 자바 코드로 재구성되고 있다.

  - 정책과 관례를 이용한 프로그래밍

    애노테이션으로 간결하고 빠른 개발이 가능하기 때문에 이런 스타일의 프로그래밍 방식이 인기를 끌고 있다. 하지만 정책을 기억못하거나 잘못 알고 있을 경우 의도한 대로 동작하지 않는 코드가 만들어질 수 있다. 트랜잭션 속성의 문제 같은 경우는 디버깅도 매우 어렵다.

7.6.1 자바 코드를 이용한 빈 설정

XML에 담긴 DI 정보는 스프링 테스트 컨텍스트를 이용해서 테스트를 작성할 때 사용된다.

- 테스트 컨텍스트의 변경

  XML을 더 이상 사용하지 않게 하는 것이 최종 목적이다.

  먼저 DI 정보로 사용될 자바 클래스를 만든다 → @Configuration 을 달아주면 된다.

자바 클래스로 만들어진 DI설정 정보에서 XML의 설정 정보를 가져오게 만들 수 있다. → @ImportResource 를 이용하면 된다. → 단계적으로 XML의 내용을 옮기다가 XML에 더이상 아무런 DI 정보가 남지 않으면 그때 XML파일과 함께 @ImportResource를 제거하면 자바 코드로의 전환 작업이 마무리 된다.

- <context:annotaion-config/>제거

  <context:annotaion-config/>은 @PostConstruct를 붙인 메소드가 빈이 초기화된 후에 자동으로 실행되도록 사용했다. @Configuration이 붙은 설정 클래스를 사용하는 컨테이너가 사용되면 더 이상 위 태그를 넣을 필요가 없다. 컨테이너가 직접 @PostConstruct 애노테이션을 처리하는 빈 후처리기를 등록해준다.

- <bean>의 전환

  `@Bean`은 @Configuration이 붙은 DI 설정용 클래스에서 주로 사용되는 것으로, 메소드를 이용해서 빈 오브젝트의 생성과 의존관계 주입을 직접 자바 코드로 작성할 수 있게 해준다.

```
@Bean
    public DataSource dataSource() {
// 빈 내부에서 new 키워드로 빈 인스턴스를 만드는 경우엔 구현 클래스 타입으로 변수를 만든다.
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();

        // 클래스 타입의 드라이버 클래스
        dataSource.setDriverClass(Driver.class);
        dataSource.setUrl("jdbc:mysql://localhost:/springbook?characterEncoding=UTF=8");
        dataSource.setUsername("spring");
        dataSource.setPassword("book");

        return dataSource;
    }
```

```
@Bean
    public PlatformTransactionManager transactionManager(){
        DataSourceTransactionManager tm = new DataSourceTransactionManager();
        tm.setDataSource(dataSource());
        return tm;
    }
```

  XML에서 정의한 빈을 자바 코드에서 참조하려면? @Autowired를 붙은 필드를 선언해서 XML에 정의된 빈을 컨테이너가 주입해주게 해야 한다.

  @Resource는 필드 이름을 기준으로 빈을 찾는다.

  ▼ code

```
@Configuration
public class TestApplicationContext {

    @Bean
    public DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();

        dataSource.setDriverClass(Driver.class);
        dataSource.setUrl("jdbc:mysql://localhost:/springbook?characterEncoding=UTF=8");
        dataSource.setUsername("spring");
        dataSource.setPassword("book");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager tm = new DataSourceTransactionManager();
        tm.setDataSource(dataSource());
        return tm;
    }

    @Bean
    public UserDao userDao() {
```

```
        UserDaoJdbc dao = new UserDaoJdbc();
        dao.setDataSource(dataSource());
        dao.setSqlService(sqlService());
        return dao;
    }

    @Bean
    public SqlService sqlService() {
        OxmSqlService sqlService = new OxmSqlService();
        sqlService.setUnmarshaller(unmarshaller());
        sqlService.setSqlRegistry(sqlRegistry());
        return null;
    }

    @Resource
    Database embeddedDatabase;

    @Bean
    public SqlRegistry sqlRegistry() {
        EmbeddedDbSqlRegistry sqlRegistry = new EmbeddedDbSqlRegistry();
        sqlRegistry.setDataSource(this.embeddedDatabase);
        return sqlRegistry;
    }

    @Bean
    public Unmarshaller unmarshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("songi.lab.spring.toby.chap7.jaxb");
        return (Unmarshaller) marshaller;
    }

    @Bean
    public UserService userService() {
        UserServiceImpl service = new UserServiceImpl();
        service.setUserDao(userDao());
        return service;
    }
}
```

- 전용 태그 전환

  ▼ code

```
@Configuration
@EnableTransactionManagement
public class TestApplicationContext {
    /**
     *  DB 연결과 트랜잭션
     */
    @Bean
    public DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();

        dataSource.setDriverClass(Driver.class);
        dataSource.setUrl("jdbc:mysql://localhost:/springbook?characterEncoding=UTF=8");
        dataSource.setUsername("spring");
        dataSource.setPassword("book");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager tm = new DataSourceTransactionManager();
        tm.setDataSource(dataSource());
        return tm;
    }
    /**
     * 애플리케이션 로직 & 테스트
     */

    @Autowired SqlService sqlService;

    @Bean
```

```
    public UserDao userDao() {
        UserDaoJdbc dao = new UserDaoJdbc();
        dao.setDataSource(dataSource());
        dao.setSqlService(this.sqlService);
//        dao.setSqlService(sqlService());
        return dao;
    }

    @Bean
    public UserService userService() {
        UserServiceImpl service = new UserServiceImpl();
        service.setUserDao(userDao());
        return service;
    }

    /**
     * SQL 서비스
     */

    @Bean
    public SqlService sqlService() {
        OxmSqlService sqlService = new OxmSqlService();
        sqlService.setUnmarshaller(unmarshaller());
        sqlService.setSqlRegistry(sqlRegistry());
        return sqlService;
    }

    @Bean
    public SqlRegistry sqlRegistry() {
        EmbeddedDbSqlRegistry sqlRegistry = new EmbeddedDbSqlRegistry();
        sqlRegistry.setDataSource(embeddedDatabase());
        return (SqlRegistry) sqlRegistry;
    }

    @Bean
    public Unmarshaller unmarshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("songi.lab.spring.toby.chap7.jaxb");
        return (Unmarshaller) marshaller;
    }

    @Bean
    public DataSource embeddedDatabase(){
        return new EmbeddedDatabaseBuilder()
                .setName("embeddedDatabase")
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:springbook/user/sqlservice/updatable/sqlRegistrySchema.sql")
                .build();
    }

}
```

스프링 3.1부터는 <tx:annotation-driven/>과 같이 특별한 목적을 위해 만들어진 내부적으로 복잡한 로우 레벨의 빈을 등록해주는 전용 태그에 대응되는 어노테이션을 제공해준다. :: `@EnableTransactionManagement`

7.6.2 빈 스캐닝과 자동 와이어링

- `@Autowired`를 이용한 자동와이어링

  `@Autowired`는 자동와이어링 기법을 이용해 조건에 맞는 빈을 찾아 자동으로 수정자 메소드나 필드에 넣어준다. 컨테이너가 이름이나 타입을 기준으로 주입될 빈을 찾아주기 때문에 빈의 프로퍼티 설정을 직접해주는 자바 코드나 XML양을 대폭 줄일 수 있다.

  컨테이너가 자동으로 주입할 빈을 결정하기 어려운 경우엔 직접 프로퍼티에 주입할 대상을 지정하는 방법을 병행하면 된다.

```
public class UserDaoJdbc implements UserDao {
    private DataSource dataSource;
    @Autowired
    public void setDataSource(DataSource dataSource) {
```

```
        this.dataSource = dataSource;
    }
```

```
    public class UserDaoJdbc implements UserDao {
    @Autowired
    private SqlService sqlService;
.....
    @Bean
    public UserDao userDao() {
        return new UserDaoJdbc();
    }
```

필드에 값을 수정하는 수정자 메소드라도 @Autowired를 필드에 직접 부여했다고 메소드를 생략하면 안되는 경우가 있다 → 스프링과 무관하게 직접 오브젝트를 생성하고 다른 오브젝트를 주입해서 테스트하는 순수한 단위 테스트를 만드는 경우에는 수정자 메소드가 필요하다.

자동와이어링은 적절히 사용하면 DI관련코드를 대폭 줄일수 있어 편리하지만 빈 설정정보를 보고 다른 빈과 의존관계가 어떻게 맺어져 있는지 한눈에 파악하기 힘들다.

- @Component를 이용한 자동 빈 등록

  `@Component`는 스프링이 애노테이션에 담긴 메타정보를 이용하기 시작했을 때 @Autowired와 함께 소개된 대표적인 애노테이션인데, 클래스에 부여된다. `@Component` 가 붙은 클래스는 **빈 스캐너를 통해 자동으로 빈으로 등록**된다.

  빈으로 등록될 후보 클래스에 붙여주는 일종의 마커Marker라고 보면 된다.

  빈 자동등록이 컨테이너가 디폴트로 제공하는 기능이 아니기 때문에 @ComponentScan으로 특정 패키지 아래서만 찾도록 기준이 되는 패키지를 지정해줘야 한다.

  @Resource처럼 빈의 아이디를 기준으로 자동와이어링하거나 XML처럼 명시적으로 빈의 아이디를 지정하는 경우, @Autowired로 찾을 대상이 두 개 이상인 경우 빈의 아이디가 중요할 수 있다 → @Component(”이름부여”)

  @Component를 메타애노테이션으로 갖고 있는 애노테이션도 사용할 수 있다. → 빈 스캔 검색 대상으로 만드는 것 이외 부가적인 용도의 마커로 사용하기 위해서다.

  - 애노테이션 포인트컷을 이용하면 패키지나 클래스 이름 패턴 대신 애노테이션 기준으로 어드바이스 적용 대상을 선별할 수 있다(@Transactional)

  스프링은 DAO기능을 제공하는 클래스에는 @Repository애노테이션을 이용하도록 권장한다.

  자동 빈 등록방식도 장단점이 있다.(2권의 1장 참고)

```
@Service("userService")
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;
```

dataSource와 transactionManager빈은 자동등록기능을 적용할 수 없다. 두개의 빈은 스프링이 제공해준 클래스를 사용하는 것이라 소스코드에 @Component나 @Autowired를 적용할 수 없다. 또 dataSource빈은 프로퍼티에 텍스트 값을 넣어줘야하기 때문에 불가능하다.

7.6.3 컨텍스트 분리와 @Import

- 테스트용 컨텍스트 분리
  - DI 설정정보를 분리하는 방법: DI 설정 클래스를 추가하고 관련된 빈 설정 애노테이션, 필드, 메소드를 옮긴다.

```
public class TestUserService implements UserService {
```

```
    @Autowired
    private UserDao userDao;
```

테스트용으로 특별히 만든 빈은 설정정보에 내용이 드러나 있는 편이 좋다.

- @Import
  - DAO에서는 sqlService타입의 빈을 DI 받을 수 있기만 하면 되지 구체적인 구현 방법을 알 필요가 없다.→ sql 관련 빈들을 분리하면 좋다.
  - 테스트용 설정 정보는 애플리케이션 핵심 설정 정보와 깔끔하게 분리되는 편이 낫다. SQL 서비스 관련 빈설정은 별도 클래스로 분리했지만 항상 필요한 정보기 때문에 AppContext와 긴밀하게 연결해주는게 좋다. → `@Import` 로 SqlServiceContext는 AppContext에 포함되는 보조 설정 정보로 사용하게 한다.

```
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "songi.lab.spring.toby.chap7")
@Import(SqlServiceContext.class)
public class AppContext {
```

7.6.4 프로파일

테스트 환경과 운영환경에서 각기 다른 빈 정의가 필요한 경우가 있다. → 운영환경에서는 반드시 필요하지만 테스트 실행 중에는 배제돼야 하는 빈 설정을 별도의 설정 클래스를 만들어 따로 관리할 필요가 있다.

- @Profile과 @ActiveProfiles

  스프링 3.1은 환경에 따라서 빈 설정 정보가 달라져야 하는 경우에 파일을 여러개 쪼개고 조합하는 번거로운 방법대신 간단히 설정 정보를 구성할 수 있는 방법을 제공한다. → 실행환경에따라 빈 구성이 달라지는 내용을 프로파일로 정의해서 만들어두고 실행 시점에 어떤 프로파일의 빈 설정을 사용할 지 지정한다.

  프로파일은 간단한 이름과 빈 설정으로 구성되며, 설정 클래스 단위로 지정한다.

```
@Configuration
@Profile("test")
public class TestAppContext {
```

  활성 프로파일이란 스프링 컨테이너를 실행할 때 추가로 지정해주는 속성인데 테스트에 활성 프로파일을 지정해야한다.

```
@ActiveProfiles("test")
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = AppContext.class)
public class UserDaoTest {
```

- 컨테이너의 빈 등록 정보 확인

```
@Autowired
    DefaultListableBeanFactory bf;

    @Test
    void beans() {
        for (String n : bf.getBeanDefinitionNames()) {
            System.out.println(n + " \t " + bf.getBean(n).getClass().getName());
        }
    }
```

- 중첩 클래스를 이용한 프로파일 적용

클래스가 많아지니 전체 구성을 살펴보기 어렵다→ 설정정보를 하나의 파일로 모아본다.⇒ 프로파일이 지정된 독립된 설정 클래스의 구조는 그대로 유지한 채 소스코드의 위치만 통합한다.→ 스태틱 클래스로 만든다.

▼ code

```
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "songi.lab.spring.toby.chap7")
@Import(SqlServiceContext.class)
public class AppContext {
    /**
     * DB 연결과 트랜잭션
     */
    @Bean
    public DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();

        dataSource.setDriverClass(Driver.class);
        dataSource.setUrl("jdbc:mysql://localhost:/springbook?characterEncoding=UTF=8");
        dataSource.setUsername("spring");
        dataSource.setPassword("book");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager tm = new DataSourceTransactionManager();
        tm.setDataSource(dataSource());
        return tm;
    }

    /**
     * 애플리케이션 로직 & 테스트
     */

    @Autowired
    SqlService sqlService;

    //    @Bean
//    public UserDao userDao() {
//        return new UserDaoJdbc();
//    }
    @Autowired
    UserDao userDao;

    @Bean
    public UserService userService() {
        UserServiceImpl service = new UserServiceImpl();
        service.setUserDao(userDao);
        return service;
    }

    @Configuration
    @Profile("production")
    public static class ProductionAppContext {

    }

    @Configuration
    @Profile("test")
    public static class TestAppContext {
        @Autowired
        UserDao userDao;

        @Bean
        public UserService testUserService(){
            return new TestUserService();
        }
    }
}
```

7.6.5 프로퍼티 소스

AppContext에는 테스트 환경에 종속되는 정보가 남아있다.→dataSource의 DB연결 정보다. DB연결 정보는 환경에 따라 다르게 설정될 수 있어야 한다. → 빌드 작업이 따로 필요 없는 XML이나 프로퍼티 파일 같은 텍스트 파일에 저장해두는 편이 낫다.

- @PropertySource

  스프링3.1은 빈 설정 작업에 필요한 프로퍼티 정보를 컨테이너가 관리하고 제공해준다. 컨테이너가 프로퍼티 값을 가져오는 대상을 프로퍼티 소스라고 한다. 프로퍼티 소스 등록에는 `@PropertySource` 를 이용한다.

  @PropertySource로 등록한 리소스로부터 가져오는 프로퍼티 값은 컨테이너가 관리하는 Environment타입의 환경 오브젝트에 저장된다. 환경 오브젝트는 @Autowired로 필드 주입 받을 수 있다.

  ▼ code

```
@Autowired
    Environment env;

    /**
     * DB 연결과 트랜잭션
     */
    @Bean
    public DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
        try {
            dataSource.setDriverClass((Class<? extends java.sql.Driver>)
                    Class.forName(env.getProperty("db.driverClass")));
        } catch (ClassNotFoundException e) {
            throw new RuntimeException();
        }
        dataSource.setUrl(env.getProperty("db.url"));
        dataSource.setUsername(env.getProperty("db.username"));
        dataSource.setPassword(env.getProperty("db.password"));

        return dataSource;
    }
```

- PropertySourcePlaceholderConfigurer

  Environment오브젝트 대신 프로퍼티 값을 직접 DI받는 방법도 가능하다. @Value는 값을 주입받을 때 사용한다. 프로퍼티 소스로부터 값을 주입받을 수 있게 치환자(placeholder)를 이용한다.

```
@Bean
    public static PropertySourcesPlaceholderConfigurer placeholderConfigurer(){
        return new PropertySourcesPlaceholderConfigurer();
    }
```

  @Value를 이용하면 driverClass처럼 문자열을 그대로 사용하지 않고 타입변환이 필요한 프로퍼티를 스프링이 알아서 처리해준다는 장점이 있다.

7.6.6 빈 설정의 재사용과 @Enable*

- 빈 설정자

  SQL 서비스를 재사용가능한 독립적인 모듈로 만들려면 SQL매핑 파일위치 지정하는 부분이 문제가 된다. → SQL 매핑 리소스는 빈 클래스 외부에서 설정할 수 있어야 한다.

```
    @Bean
    public SqlMapConfig sqlMapConfig(){
        return new UserSqlMapConfig();
    }

public class UserSqlMapConfig implements SqlMapConfig {
```

```
    @Override
    public Resource getSqlMapResource() {
        return new ClassPathResource("sqlmap.xml", UserDao.class);
    }
}

    @Autowired
    SqlMapConfig sqlMapConfig;

    @Bean
    public SqlService sqlService() {
        OxmSqlService sqlService = new OxmSqlService();
        sqlService.setUnmarshaller(unmarshaller());
        sqlService.setSqlRegistry(sqlRegistry());
        sqlService.setSqlmap(this.sqlMapConfig.getSqlMapResource());
        return sqlService;
    }
```

@Configuration클래스도 빈스캐너를 통해 자동등록되게 만들 수 있다. AppContext가 직접 SqlMapConfig 인터페이스를 구현하게 만들면 컨테이너에 의해 빈 오브젝트로 만들어지고 SqlServiceContext는 SqlMapConfig 타입 빈을 주입 받으려고 할 것이다. AppContext가 SqlMapConfig를 구현하게 했으니 AppContext로 만들어진 오브젝트는 SqlServiceContext에 주입되어 사용된다.

- @Enable* 애노테이션

  @Import도 다른 이름의 애노테이션으로 대체 가능하다.

```
@Import(value = SqlServiceContext.class)
public @interface EnableSqlService {
}
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "songi.lab.spring.toby.chap7")
@EnableSqlService
@PropertySource("/database.properties")
public class AppContext implements SqlMapConfig{
```

7.7 정리

1. SQL 처럼 변경될 수 있는 텍스트로 된 정보는 외부 리소스레 담아두고 가져오게 만들면 편리하다.
2. 성격이 다른 코드가 한데 섞여 있는 클래스라면 먼저 인터페이스를 정의해서 코드를 각 인터페이스별로 분리하는게 좋다. 다른 인터페이스에 속한 기능은 인터페이스를 통해 접근하게 만들고, 간단히 자기참조 빈으로 의존관계를 만들어 검증한다. 검증을 마쳤으면 어예 클래스를 분리해도 좋다.
3. 자주 사용되는 의존 오브젝트는 디폴트로 미리 정의해두면 편리하다.
4. XML과 오브젝트 매핑은 스프링의 OXM 추상화 기능을 활용한다.
5. 특정 의존 오브젝트를 고정시켜 기능을 특화하려면 멤버 클래스로 만드는 것이 편리하다. 기존에 만들어진 기능과 중복되는 부분은 위임을 통해 중복을 제거하는 게 좋다.
6. 외부의 파일이나 리소스를 사용하는 코드에서는 스프링의 리소스 추상화와 리소스 로더를 사용한다.
7. DI를 의식하면서 코드를 작성하면 객체지향 설계에 도움이 된다.
8. DI에는 인터페이스를 사용한다. 인터페이스를 사용하면 인터페이스 분리 원칙을 잘 지키는데도 도움이 된다.
9. 클라이언트에 따라서 인터페이스를 분리할 때, 새로운 인터페이스를 만드는 방법과 이넡페이스를 상속하는 방법 두가지를 사용할 수 있다.
10. 애플리케이션에 내장하는 DB를 사용할 때는 스프링의 내장형 DB추상화 기능과 전용 태그를 사용하면 편리하다.