2. 데이터 접근 기술 - 스프링 JdbcTemplate

#2.인강/8.스프링 DB 2/강의#

JdbcTemplate 소개와 설정

SQL을 직접 사용하는 경우에 스프링이 제공하는 JdbcTemplate은 아주 좋은 선택지다. JdbcTemplate은 JDBC를 매우 편리하게 사용할 수 있게 도와준다.

장점

- 설정의 편리함
  - JdbcTemplate은 `spring-jdbc` 라이브러리에 포함되어 있는데, 이 라이브러리는 스프링으로 JDBC를 사용할 때 기본으로 사용되는 라이브러리이다. 그리고 별도의 복잡한 설정 없이 바로 사용할 수 있다.
- 반복 문제 해결
  - JdbcTemplate은 템플릿 콜백 패턴을 사용해서, JDBC를 직접 사용할 때 발생하는 대부분의 반복 작업을 대신 처리해준다.
  - 개발자는 SQL을 작성하고, 전달할 파리미터를 정의하고, 응답 값을 매핑하기만 하면 된다.
  - 우리가 생각할 수 있는 대부분의 반복 작업을 대신 처리해준다.
    - 커넥션 획득
    - `statement`를 준비하고 실행
    - 결과를 반복하도록 루프를 실행
    - 커넥션 종료, `statement`, `resultset` 종료
    - 트랜잭션 다루기 위한 커넥션 동기화
    - 예외 발생시 스프링 예외 변환기 실행

단점

- 동적 SQL을 해결하기 어렵다.

이제 직접 JdbcTemplate을 설정하고 적용하면서 이해해보자.

JdbcTemplate 설정

`build.gradle`

```
dependencies {
	implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
	implementation 'org.springframework.boot:spring-boot-starter-web'

	//JdbcTemplate 추가
	implementation 'org.springframework.boot:spring-boot-starter-jdbc'
	//H2 데이터베이스 추가
	runtimeOnly 'com.h2database:h2'
	compileOnly 'org.projectlombok:lombok'
	annotationProcessor 'org.projectlombok:lombok'
	testImplementation 'org.springframework.boot:spring-boot-starter-test'

	//테스트에서 lombok 사용
	testCompileOnly 'org.projectlombok:lombok'
	testAnnotationProcessor 'org.projectlombok:lombok'
}
```

- `org.springframework.boot:spring-boot-starter-jdbc`를 추가하면 JdbcTemplate이 들어있는 `spring-jdbc`가 라이브러리에 포함된다.
- 여기서는 H2 데이터베이스에 접속해야 하기 때문에 H2 데이터베이스의 클라이언트 라이브러리(Jdbc Driver)도 추가하자.
  - `runtimeOnly 'com.h2database:h2'`

추가되는 부분

```
//JdbcTemplate 추가
implementation 'org.springframework.boot:spring-boot-starter-jdbc'
//H2 데이터베이스 추가
runtimeOnly 'com.h2database:h2'
```

JdbcTemplate은 `spring-jdbc` 라이브러리만 추가하면 된다. 별도의 추가 설정 과정은 없다.

주의!
진행하기 전에 먼저 H2 데이터베이스에 `item` 테이블을 생성해야 한다.

```
drop table if exists item CASCADE;
create table item
(
    id        bigint generated by default as identity,
    item_name varchar(10),
    price     integer,
    quantity  integer,
    primary key (id)
);
```

JdbcTemplate 적용1 - 기본

이제부터 본격적으로 JdbcTemplate을 사용해서 메모리에 저장하던 데이터를 데이터베이스에 저장해보자.
`ItemRepository` 인터페이스가 있으니 이 인터페이스를 기반으로 JdbcTemplate을 사용하는 새로운 구현체를 개발하자.

JdbcTemplateItemRepositoryV1

```
package hello.itemservice.repository.jdbctemplate;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import lombok.extern.slf4j.Slf4j;
import org.springframework.dao.EmptyResultDataAccessException;
```

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;

import javax.sql.DataSource;
import java.sql.PreparedStatement;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

/**
 * JdbcTemplate
 */
@Slf4j
@Repository
public class JdbcTemplateItemRepositoryV1 implements ItemRepository {

    private final JdbcTemplate template;

    public JdbcTemplateItemRepositoryV1(DataSource dataSource) {
        this.template = new JdbcTemplate(dataSource);
    }

    @Override
    public Item save(Item item) {
        String sql = "insert into item (item_name, price, quantity) values
(?, ?, ?)";
        KeyHolder keyHolder = new GeneratedKeyHolder();
        template.update(connection -> {
            //자동 증가 키
            PreparedStatement ps = connection.prepareStatement(sql, new
String[]{"id"});
            ps.setString(1, item.getItemName());
            ps.setInt(2, item.getPrice());
            ps.setInt(3, item.getQuantity());
            return ps;
```

```
        }, keyHolder);

        long key = keyHolder.getKey().longValue();
        item.setId(key);
        return item;
    }

    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        String sql = "update item set item_name=?, price=?, quantity=? where
id=?";
        template.update(sql,
                updateParam.getItemName(),
                updateParam.getPrice(),
                updateParam.getQuantity(),
                itemId);
    }

    @Override
    public Optional<Item> findById(Long id) {
        String sql = "select id, item_name, price, quantity from item where id
= ?";
        try {
            Item item = template.queryForObject(sql, itemRowMapper(), id);
            return Optional.of(item);
        } catch (EmptyResultDataAccessException e) {
            return Optional.empty();
        }
    }

    @Override
    public List<Item> findAll(ItemSearchCond cond) {
        String itemName = cond.getItemName();
        Integer maxPrice = cond.getMaxPrice();

        String sql = "select id, item_name, price, quantity from item";
        //동적 쿼리
        if (StringUtils.hasText(itemName) || maxPrice != null) {
            sql += " where";
```

```
        }

        boolean andFlag = false;
        List<Object> param = new ArrayList<>();
        if (StringUtils.hasText(itemName)) {
            sql += " item_name like concat('%',?,'%')";
            param.add(itemName);
            andFlag = true;
        }

        if (maxPrice != null) {
            if (andFlag) {
                sql += " and";
            }
            sql += " price <= ?";
            param.add(maxPrice);
        }

        log.info("sql={}", sql);
        return template.query(sql, itemRowMapper(), param.toArray());
    }

    private RowMapper<Item> itemRowMapper() {
        return (rs, rowNum) -> {
            Item item = new Item();
            item.setId(rs.getLong("id"));
            item.setItemName(rs.getString("item_name"));
            item.setPrice(rs.getInt("price"));
            item.setQuantity(rs.getInt("quantity"));
            return item;
        };
    }

}
```

기본

- `JdbcTemplateItemRepositoryV1`은 `ItemRepository` 인터페이스를 구현했다.
- `this.template = new JdbcTemplate(dataSource)`

- `JdbcTemplate`은 데이터소스(`dataSource`)가 필요하다.
- `JdbcTemplateItemRepositoryV1()` 생성자를 보면 `dataSource`를 의존 관계 주입 받고 생성자 내부에서 `JdbcTemplate`을 생성한다. 스프링에서는 `JdbcTemplate`을 사용할 때 관례상 이 방법을 많이 사용한다.
- 물론 `JdbcTemplate`을 스프링 빈으로 직접 등록하고 주입받아도 된다.

save()

데이터를 저장한다.

- `template.update()` : 데이터를 변경할 때는 `update()`를 사용하면 된다.
  - `INSERT`, `UPDATE`, `DELETE` SQL에 사용한다.
  - `template.update()`의 반환 값은 `int`인데, 영향 받은 로우 수를 반환한다.
- 데이터를 저장할 때 PK 생성에 `identity`(auto increment) 방식을 사용하기 때문에, PK인 ID 값을 개발자가 직접 지정하는 것이 아니라 비워두고 저장해야 한다. 그러면 데이터베이스가 PK인 ID를 대신 생성해준다.
- 문제는 이렇게 데이터베이스가 대신 생성해주는 PK ID 값은 데이터베이스가 생성하기 때문에, 데이터베이스에 INSERT가 완료 되어야 생성된 PK ID 값을 확인할 수 있다.
- `KeyHolder`와 `connection.prepareStatement(sql, new String[]{"id"})`를 사용해서 `id`를 지정해주면 `INSERT` 쿼리 실행 이후에 데이터베이스에서 생성된 ID 값을 조회할 수 있다.
- 물론 데이터베이스에서 생성된 ID 값을 조회하는 것은 순수 JDBC로도 가능하지만, 코드가 훨씬 더 복잡하다.
- 참고로 뒤에서 설명하겠지만 JdbcTemplate이 제공하는 `SimpleJdbcInsert`라는 훨씬 편리한 기능이 있으므로 대략 이렇게 사용한다 정도로만 알아두면 된다.

update()

데이터를 업데이트 한다.

- `template.update()` : 데이터를 변경할 때는 `update()`를 사용하면 된다.
  - `?`에 바인딩할 파라미터를 순서대로 전달하면 된다.
  - 반환 값은 해당 쿼리의 영향을 받은 로우 수 이다. 여기서는 `where id=?`를 지정했기 때문에 영향 받은 로우수는 최대 1개이다.

findById()

데이터를 하나 조회한다.

- `template.queryForObject()`
  - 결과 로우가 하나일 때 사용한다.
  - `RowMapper`는 데이터베이스의 반환 결과인 `ResultSet`을 객체로 변환한다.
  - 결과가 없으면 `EmptyResultDataAccessException` 예외가 발생한다.
  - 결과가 둘 이상이면 `IncorrectResultSizeDataAccessException` 예외가 발생한다.
- `ItemRepository.findById()` 인터페이스는 결과가 없을 때 `Optional`을 반환해야 한다. 따라서 결과가 없으면 예외를 잡아서 `Optional.empty`를 대신 반환하면 된다.

queryForObject() 인터페이스 정의

```
<T> T queryForObject(String sql, RowMapper<T> rowMapper, Object... args) throws
DataAccessException;
```

findAll()

데이터를 리스트로 조회한다. 그리고 검색 조건으로 적절한 데이터를 찾는다.

- `template.query()`
  - 결과가 하나 이상일 때 사용한다.
  - `RowMapper`는 데이터베이스의 반환 결과인 `ResultSet`을 객체로 변환한다.
  - 결과가 없으면 빈 컬렉션을 반환한다.
  - 동적 쿼리에 대한 부분은 바로 다음에 다룬다.

query() 인터페이스 정의

```
<T> List<T> query(String sql, RowMapper<T> rowMapper, Object... args) throws
DataAccessException;
```

itemRowMapper()

데이터베이스의 조회 결과를 객체로 변환할 때 사용한다.
JDBC를 직접 사용할 때 `ResultSet`를 사용했던 부분을 떠올리면 된다.
차이가 있다면 다음과 같이 JdbcTemplate이 다음과 같은 루프를 돌려주고, 개발자는 `RowMapper`를 구현해서 그 내부 코드만 채운다고 이해하면 된다.

```
while(resultSet 이 끝날 때 까지) {
  rowMapper(rs, rowNum)
}
```

JdbcTemplate 적용2 - 동적 쿼리 문제

결과를 검색하는 `findAll()`에서 어려운 부분은 사용자가 검색하는 값에 따라서 실행하는 SQL이

동적으로 달려져야 한다는 점이다.
예를 들어서 다음과 같다.

검색 조건이 없음

```
select id, item_name, price, quantity from item
```

상품명(`itemName`)으로 검색

```
select id, item_name, price, quantity from item
 where item_name like concat('%',?,'%')
```

최대 가격(`maxPrice`)으로 검색

```
select id, item_name, price, quantity from item
 where price <= ?
```

상품명(`itemName`), 최대 가격(`maxPrice`) 둘다 검색

```
select id, item_name, price, quantity from item
 where item_name like concat('%',?,'%')
   and price <= ?
```

결과적으로 4가지 상황에 따른 SQL을 동적으로 생성해야 한다. 동적 쿼리가 언듯 보면 쉬워 보이지만, 막상 개발해보면 생각보다 다양한 상황을 고민해야 한다. 예를 들어서 어떤 경우에는 `where` 를 앞에 넣고 어떤 경우에는 `and` 를 넣어야 하는지 등을 모두 계산해야 한다.
그리고 각 상황에 맞추어 파라미터도 생성해야 한다.
물론 실무에서는 이보다 훨씬 더 복잡한 동적 쿼리들이 사용된다.

참고로 이후에 설명할 MyBatis의 가장 큰 장점은 SQL을 직접 사용할 때 동적 쿼리를 쉽게 작성할 수 있다는 점이다.

JdbcTemplate 적용3 - 구성과 실행

실제 코드가 동작하도록 구성하고 실행해보자.

JdbcTemplateV1Config

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jdbctemplate.JdbcTemplateItemRepositoryV1;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV1;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
@RequiredArgsConstructor
public class JdbcTemplateV1Config {

    private final DataSource dataSource;

    @Bean
    public ItemService itemService() {
        return new ItemServiceV1(itemRepository());
    }

    @Bean
    public ItemRepository itemRepository() {
        return new JdbcTemplateItemRepositoryV1(dataSource);
    }

}
```

- `ItemRepository` 구현체로 `JdbcTemplateItemRepositoryV1` 이 사용되도록 했다. 이제 메모리 저장소가 아니라 실제 DB에 연결하는 JdbcTemplate이 사용된다.

ItemServiceApplication - 변경

```
//@Import(MemoryConfig.class)
@Import(JdbcTemplateV1Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {}
```

데이터베이스 접근 설정

`src/main/resources/application.properties`

```
spring.profiles.active=local
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.username=sa
```

- 이렇게 설정만 하면 스프링 부트가 해당 설정을 사용해서 커넥션 풀과 `DataSource`, 트랜잭션 매니저를 스프링 빈으로 자동 등록한다.
  - (앞에서 학습한 스프링 부트의 자동 리소스 등록 내용을 떠올려보자.)

> **주의!**
> 여기서는 `src/test`가 아니라 `src/main`에 위치한 `application.properties` 파일을 수정해야 한다!

실행

- 실제 DB에 연결해야 하므로 H2 데이터베이스 서버를 먼저 실행하자.
- 앞서 만든 item 테이블이 잘 생성되어 있는지 다시 확인하자.
- `ItemServiceApplication.main()`을 실행해서 애플리케이션 서버를 실행하자.

- 웹 브라우저로 다음에 접속하자: http://localhost:8080
- 실행해보면 잘 동작하는 것을 확인할 수 있다. 그리고 DB에 실제 데이터가 저장되는 것도 확인할 수 있다.
- 참고로 서버를 다시 시작할 때 마다 `TestDataInit`이 실행되기 때문에 `itemA`, `itemB`도 데이터베이스에 계속 추가된다. 메모리와 다르게 서버가 내려가도 데이터베이스는 유지되기 때문이다.

로그 추가

JdbcTemplate이 실행하는 SQL 로그를 확인하려면 `application.properties`에 다음을 추가하면 된다. `main`, `test` 설정이 분리되어 있기 때문에 둘다 확인하려면 두 곳에 모두 추가해야 한다.

```
#jdbcTemplate sql log
logging.level.org.springframework.jdbc=debug
```

JdbcTemplate - 이름 지정 파라미터 1

순서대로 바인딩

JdbcTemplate을 기본으로 사용하면 파라미터를 순서대로 바인딩 한다.
예를 들어서 다음 코드를 보자.

```
String sql = "update item set item_name=?, price=?, quantity=? where id=?";
template.update(sql,
        itemName,
        price,
        quantity,
        itemId);
```

여기서는 `itemName`, `price`, `quantity`가 `SQL에 있는 ?`에 순서대로 바인딩 된다.
따라서 순서만 잘 지키면 문제가 될 것은 없다. 그런데 문제는 변경시점에 발생한다.

누군가 다음과 같이 SQL 코드의 순서를 변경했다고 가정해보자. (`price`와 `quantity`의 순서를 변경했다.)

```
String sql = "update item set item_name=?, quantity=?, price=? where id=?";
template.update(sql,
        itemName,
        price,
        quantity,
        itemId);
```

이렇게 되면 다음과 같은 순서로 데이터가 바인딩 된다.

`item_name=itemName, quantity=price, price=quantity`

결과적으로 `price`와 `quantity`가 바뀌는 매우 심각한 문제가 발생한다. 이럴일이 없을 것 같지만,
실무에서는 파라미터가 10~20개가 넘어가는 일도 아주 많다. 그래서 미래에 필드를 추가하거나,
수정하면서 이런 문제가 충분히 발생할 수 있다.

버그 중에서 가장 고치기 힘든 버그는 데이터베이스에 데이터가 잘못 들어가는 버그다. 이것은 코드만 고치는 수준이 아니라 데이터베이스의 데이터를 복구해야 하기 때문에 버그를 해결하는데 들어가는 리소스가 어마어마하다.
실제로 수많은 개발자들이 이 문제로 장애를 내고 퇴근하지 못하는 일이 발생한다.

개발을 할 때는 코드를 몇줄 줄이는 편리함도 중요하지만, 모호함을 제거해서 코드를 명확하게 만드는 것이 유지보수 관점에서 매우 중요하다.

이처럼 파라미터를 순서대로 바인딩 하는 것은 편리하기는 하지만, 순서가 맞지 않아서 버그가 발생할 수도 있으므로 주의해서 사용해야 한다.

이름 지정 바인딩

JdbcTemplate은 이런 문제를 보완하기 위해 `NamedParameterJdbcTemplate`라는 이름을 지정해서 파라미터를 바인딩 하는 기능을 제공한다.

지금부터 코드로 알아보자.

JdbcTemplateItemRepositoryV2

```
package hello.itemservice.repository.jdbctemplate;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import lombok.extern.slf4j.Slf4j;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;
```

```java
import javax.sql.DataSource;
import java.util.List;
import java.util.Map;
import java.util.Optional;

/**
 * NamedParameterJdbcTemplate
 * SqlParameterSource
 * - BeanPropertySqlParameterSource
 * - MapSqlParameterSource
 * Map
 *
 * BeanPropertyRowMapper
 *
 */
@Slf4j
@Repository
public class JdbcTemplateItemRepositoryV2 implements ItemRepository {

    private final NamedParameterJdbcTemplate template;

    public JdbcTemplateItemRepositoryV2(DataSource dataSource) {
        this.template = new NamedParameterJdbcTemplate(dataSource);
    }

    @Override
    public Item save(Item item) {
        String sql = "insert into item (item_name, price, quantity) " +
                     "values (:itemName, :price, :quantity)";

        SqlParameterSource param = new BeanPropertySqlParameterSource(item);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        template.update(sql, param, keyHolder);

        Long key = keyHolder.getKey().longValue();
        item.setId(key);
        return item;
    }
```

```
    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        String sql = "update item " +
                "set item_name=:itemName, price=:price, quantity=:quantity " +
                "where id=:id";

        SqlParameterSource param = new MapSqlParameterSource()
                .addValue("itemName", updateParam.getItemName())
                .addValue("price", updateParam.getPrice())
                .addValue("quantity", updateParam.getQuantity())
                .addValue("id", itemId); //이 부분이 별도로 필요하다.
        template.update(sql, param);
    }

    @Override
    public Optional<Item> findById(Long id) {
        String sql = "select id, item_name, price, quantity from item where id
= :id";
        try {
            Map<String, Object> param = Map.of("id", id);
            Item item = template.queryForObject(sql, param, itemRowMapper());
            return Optional.of(item);
        } catch (EmptyResultDataAccessException e) {
            return Optional.empty();
        }
    }

    @Override
    public List<Item> findAll(ItemSearchCond cond) {

        Integer maxPrice = cond.getMaxPrice();
        String itemName = cond.getItemName();

        SqlParameterSource param = new BeanPropertySqlParameterSource(cond);

        String sql = "select id, item_name, price, quantity from item";
        //동적 쿼리
        if (StringUtils.hasText(itemName) || maxPrice != null) {
            sql += " where";
```

```
        }

        boolean andFlag = false;
        if (StringUtils.hasText(itemName)) {
            sql += " item_name like concat('%',:itemName,'%')";
            andFlag = true;
        }

        if (maxPrice != null) {
            if (andFlag) {
                sql += " and";
            }
            sql += " price <= :maxPrice";
        }

        log.info("sql={}", sql);
        return template.query(sql, param, itemRowMapper());
    }

    private RowMapper<Item> itemRowMapper() {
        return BeanPropertyRowMapper.newInstance(Item.class); //camel 변환 지원
    }

}
```

기본

- `JdbcTemplateItemRepositoryV2`는 `ItemRepository` 인터페이스를 구현했다.
- `this.template = new NamedParameterJdbcTemplate(dataSource)`
  - `NamedParameterJdbcTemplate`도 내부에 `dataSource`가 필요하다.
  - `JdbcTemplateItemRepositoryV2` 생성자를 보면 의존관계 주입은 `dataSource`를 받고 내부에서 `NamedParameterJdbcTemplate`을 생성해서 가지고 있다. 스프링에서는 `JdbcTemplate` 관련 기능을 사용할 때 관례상 이 방법을 많이 사용한다.
  - 물론 `NamedParameterJdbcTemplate`을 스프링 빈으로 직접 등록하고 주입받아도 된다.

save()

SQL에서 다음과 같이 `?` 대신에 `:파라미터이름`을 받는 것을 확인할 수 있다.

```
insert into item (item_name, price, quantity) " +
```

```
            "values (:itemName, :price, :quantity)"
```

추가로 `NamedParameterJdbcTemplate`은 데이터베이스가 생성해주는 키를 매우 쉽게 조회하는 기능도 제공해준다.

JdbcTemplate - 이름 지정 파라미터 2

이름 지정 파라미터

파라미터를 전달하려면 `Map`처럼 `key`, `value` 데이터 구조를 만들어서 전달해야 한다.
여기서 `key`는 `:파리이터이름`으로 지정한, 파라미터의 이름이고, `value`는 해당 파라미터의 값이 된다.

다음 코드를 보면 이렇게 만든 파라미터(`param`)를 전달하는 것을 확인할 수 있다.
`template.update(sql, param, keyHolder);`

이름 지정 바인딩에서 자주 사용하는 파라미터의 종류는 크게 3가지가 있다.

- `Map`
- `SqlParameterSource`
  - `MapSqlParameterSource`
  - `BeanPropertySqlParameterSource`

1. Map

단순히 `Map`을 사용한다.

`findById()` 코드에서 확인할 수 있다.

```
Map<String, Object> param = Map.of("id", id);
Item item = template.queryForObject(sql, param, itemRowMapper());
```

2. MapSqlParameterSource

`Map`과 유사한데, SQL 타입을 지정할 수 있는 등 SQL에 좀 더 특화된 기능을 제공한다.
`SqlParameterSource` 인터페이스의 구현체이다.
`MapSqlParameterSource`는 메서드 체인을 통해 편리한 사용법도 제공한다.

`update()` 코드에서 확인할 수 있다.

```
SqlParameterSource param = new MapSqlParameterSource()
        .addValue("itemName", updateParam.getItemName())
        .addValue("price", updateParam.getPrice())
        .addValue("quantity", updateParam.getQuantity())
        .addValue("id", itemId); //이 부분이 별도로 필요하다.
template.update(sql, param);
```

3. BeanPropertySqlParameterSource

자바빈 프로퍼티 규약을 통해서 자동으로 파라미터 객체를 생성한다.

예) (`getXxx() -> xxx, getItemName() -> itemName`)

예를 들어서 `getItemName()`, `getPrice()` 가 있으면 다음과 같은 데이터를 자동으로 만들어낸다.

- `key=itemName, value=상품명 값`
- `key=price, value=가격 값`

`SqlParameterSource` 인터페이스의 구현체이다.

`save()`, `findAll()` 코드에서 확인할 수 있다.

```
SqlParameterSource param = new BeanPropertySqlParameterSource(item);
KeyHolder keyHolder = new GeneratedKeyHolder();
template.update(sql, param, keyHolder);
```

- 여기서 보면 `BeanPropertySqlParameterSource` 가 많은 것을 자동화 해주기 때문에 가장 좋아보이지만, `BeanPropertySqlParameterSource` 를 항상 사용할 수 있는 것은 아니다.
- 예를 들어서 `update()` 에서는 SQL에 `:id` 를 바인딩 해야 하는데, `update()` 에서 사용하는 `ItemUpdateDto` 에는 `itemId` 가 없다. 따라서 `BeanPropertySqlParameterSource` 를 사용할 수 없고, 대신에 `MapSqlParameterSource` 를 사용했다.

BeanPropertyRowMapper

이번 코드에서 `V1` 과 비교해서 변화된 부분이 하나 더 있다. 바로 `BeanPropertyRowMapper` 를 사용한 것이다.

JdbcTemplateItemRepositoryV1 - itemRowMapper()

```
private RowMapper<Item> itemRowMapper() {
    return (rs, rowNum) -> {
        Item item = new Item();
        item.setId(rs.getLong("id"));
        item.setItemName(rs.getString("item_name"));
        item.setPrice(rs.getInt("price"));
        item.setQuantity(rs.getInt("quantity"));
        return item;
    };
}
```

JdbcTemplateItemRepositoryV2 - itemRowMapper()

```
private RowMapper<Item> itemRowMapper() {
    return BeanPropertyRowMapper.newInstance(Item.class); //camel 변환 지원
}
```

`BeanPropertyRowMapper`는 `ResultSet`의 결과를 받아서 자바빈 규약에 맞추어 데이터를 변환한다. 예를 들어서 데이터베이스에서 조회한 결과가 `select id, price`라고 하면 다음과 같은 코드를 작성해준다. (실제로는 리플렉션 같은 기능을 사용한다.)

```
Item item = new Item();
item.setId(rs.getLong("id"));
item.setPrice(rs.getInt("price"));
```

데이터베이스에서 조회한 결과 이름을 기반으로 `setId()`, `setPrice()` 처럼 자바빈 프로퍼티 규약에 맞춘 메서드를 호출하는 것이다.

별칭

그런데 `select item_name`의 경우 `setItem_name()`이라는 메서드가 없기 때문에 골치가 아프다. 이런 경우 개발자가 조회 SQL을 다음과 같이 고치면 된다.

`select item_name as itemName`

별칭 `as`를 사용해서 SQL 조회 결과의 이름을 변경하는 것이다. 실제로 이 방법은 자주 사용된다. 특히 데이터베이스 컬럼 이름과 객체 이름이 완전히 다를 때 문제를 해결할 수 있다. 예를 들어서

데이터베이스에는 `member_name` 이라고 되어 있는데 객체에 `username` 이라고 되어 있다면 다음과 같이 해결할 수 있다.
`select member_name as username`

이렇게 데이터베이스 컬럼 이름과 객체의 이름이 다를 때 별칭(`as`)을 사용해서 문제를 많이 해결한다.
`JdbcTemplate` 은 물론이고, `MyBatis` 같은 기술에서도 자주 사용된다.

관례의 불일치
자바 객체는 카멜(`camelCase`) 표기법을 사용한다. `itemName` 처럼 중간에 낙타 봉이 올라와 있는 표기법이다.
반면에 관계형 데이터베이스에서는 주로 언더스코어를 사용하는 `snake_case` 표기법을 사용한다.
`item_name` 처럼 중간에 언더스코어를 사용하는 표기법이다.

이 부분을 관례로 많이 사용하다 보니 `BeanPropertyRowMapper` 는 언더스코어 표기법을 카멜로 자동 변환해준다.
따라서 `select item_name` 으로 조회해도 `setItemName()` 에 문제 없이 값이 들어간다.

정리하면 `snake_case` 는 자동으로 해결되니 그냥 두면 되고, 컬럼 이름과 객체 이름이 완전히 다른 경우에는 조회 SQL에서 별칭을 사용하면 된다.

JdbcTemplate - 이름 지정 파라미터 3

이제 이름 지정 파라미터를 사용하도록 구성하고 실행해보자.

JdbcTemplateV2Config

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jdbctemplate.JdbcTemplateItemRepositoryV2;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV1;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import javax.sql.DataSource;

@Configuration
@RequiredArgsConstructor
public class JdbcTemplateV2Config {

    private final DataSource dataSource;

    @Bean
    public ItemService itemService() {
        return new ItemServiceV1(itemRepository());
    }

    @Bean
    public ItemRepository itemRepository() {
        return new JdbcTemplateItemRepositoryV2(dataSource);
    }

}
```

- 앞서 개발한 `JdbcTemplateItemRepositoryV2`를 사용하도록 스프링 빈에 등록한다.

ItemServiceApplication - 변경

```
package hello.itemservice;

import hello.itemservice.config.*;
import hello.itemservice.repository.ItemRepository;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Profile;


@Slf4j
//@Import(MemoryConfig.class)
//@Import(JdbcTemplateV1Config.class)
```

```
@Import(JdbcTemplateV2Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {

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

  @Bean
  @Profile("local")
  public TestDataInit testDataInit(ItemRepository itemRepository) {
      return new TestDataInit(itemRepository);
  }

}
```

- `JdbcTemplateV2Config.class`를 설정으로 사용하도록 변경되었다.
  - `@Import(JdbcTemplateV1Config.class)` ➡ `@Import(JdbcTemplateV2Config.class)`

실행

http://localhost:8080

- 기능이 잘 동작하는지 확인해보자.

JdbcTemplate - SimpleJdbcInsert

JdbcTemplate은 INSERT SQL를 직접 작성하지 않아도 되도록 `SimpleJdbcInsert`라는 편리한 기능을 제공한다.

JdbcTemplateItemRepositoryV3

```
package hello.itemservice.repository.jdbctemplate;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
```

```java
import hello.itemservice.repository.ItemUpdateDto;
import lombok.extern.slf4j.Slf4j;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;

import javax.sql.DataSource;
import java.util.List;
import java.util.Map;
import java.util.Optional;

/**
 * SimpleJdbcInsert
 */
@Slf4j
@Repository
public class JdbcTemplateItemRepositoryV3 implements ItemRepository {

    private final NamedParameterJdbcTemplate template;
    private final SimpleJdbcInsert jdbcInsert;

    public JdbcTemplateItemRepositoryV3(DataSource dataSource) {
        this.template = new NamedParameterJdbcTemplate(dataSource);
        this.jdbcInsert = new SimpleJdbcInsert(dataSource)
                .withTableName("item")
                .usingGeneratedKeyColumns("id");
//                .usingColumns("item_name", "price", "quantity"); //생략 가능
    }

    @Override
    public Item save(Item item) {
        SqlParameterSource param = new BeanPropertySqlParameterSource(item);
```

```
        Number key = jdbcInsert.executeAndReturnKey(param);
        item.setId(key.longValue());
        return item;
    }

    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        String sql = "update item " +
                "set item_name=:itemName, price=:price, quantity=:quantity " +
                "where id=:id";

        SqlParameterSource param = new MapSqlParameterSource()
                .addValue("itemName", updateParam.getItemName())
                .addValue("price", updateParam.getPrice())
                .addValue("quantity", updateParam.getQuantity())
                .addValue("id", itemId);
        template.update(sql, param);
    }

    @Override
    public Optional<Item> findById(Long id) {
        String sql = "select id, item_name, price, quantity from item where id
= :id";
        try {
            Map<String, Object> param = Map.of("id", id);
            Item item = template.queryForObject(sql, param, itemRowMapper());
            return Optional.of(item);
        } catch (EmptyResultDataAccessException e) {
            return Optional.empty();
        }
    }

    @Override
    public List<Item> findAll(ItemSearchCond cond) {

        Integer maxPrice = cond.getMaxPrice();
        String itemName = cond.getItemName();

        SqlParameterSource param = new BeanPropertySqlParameterSource(cond);
```

```
        String sql = "select id, item_name, price, quantity from item";
        //동적 쿼리
        if (StringUtils.hasText(itemName) || maxPrice != null) {
            sql += " where";
        }

        boolean andFlag = false;
        if (StringUtils.hasText(itemName)) {
            sql += " item_name like concat('%',:itemName,'%')";
            andFlag = true;
        }

        if (maxPrice != null) {
            if (andFlag) {
                sql += " and";
            }
            sql += " price <= :maxPrice";
        }

        log.info("sql={}", sql);
        return template.query(sql, param, itemRowMapper());
    }

    private RowMapper<Item> itemRowMapper() {
        return BeanPropertyRowMapper.newInstance(Item.class);
    }

}
```

기본

- `JdbcTemplateItemRepositoryV3`은 `ItemRepository` 인터페이스를 구현했다.
- `this.jdbcInsert = new SimpleJdbcInsert(dataSource)` : 생성자를 보면 의존관계 주입은 `dataSource`를 받고 내부에서 `SimpleJdbcInsert`을 생성해서 가지고 있다. 스프링에서는 `JdbcTemplate` 관련 기능을 사용할 때 관례상 이 방법을 많이 사용한다.
  - 물론 `SimpleJdbcInsert`을 스프링 빈으로 직접 등록하고 주입받아도 된다.

SimpleJdbcInsert

```
this.jdbcInsert = new SimpleJdbcInsert(dataSource)
                .withTableName("item")
                .usingGeneratedKeyColumns("id");
//                .usingColumns("item_name", "price", "quantity"); //생략 가능
```

- `withTableName` : 데이터를 저장할 테이블 명을 지정한다.
- `usingGeneratedKeyColumns` : `key` 를 생성하는 PK 컬럼 명을 지정한다.
- `usingColumns` : INSERT SQL에 사용할 컬럼을 지정한다. 특정 값만 저장하고 싶을 때 사용한다. 생략할 수 있다.

`SimpleJdbcInsert` 는 생성 시점에 데이터베이스 테이블의 메타 데이터를 조회한다. 따라서 어떤 컬럼이 있는지 확인 할 수 있으므로 `usingColumns` 을 생략할 수 있다. 만약 특정 컬럼만 지정해서 저장하고 싶다면 `usingColumns` 를 사용하면 된다.

애플리케이션을 실행해보면 `SimpleJdbcInsert` 이 어떤 INSERT SQL을 만들어서 사용하는지 로그로 확인할 수 있다.

```
DEBUG 39424 --- [           main] o.s.jdbc.core.simple.SimpleJdbcInsert    :
Compiled insert object: insert string is [INSERT INTO item (ITEM_NAME, PRICE,
QUANTITY) VALUES(?, ?, ?)]
```

save()

`jdbcInsert.executeAndReturnKey(param)` 을 사용해서 INSERT SQL을 실행하고, 생성된 키 값도 매우 편리하게 조회할 수 있다.

```
public Item save(Item item) {
    SqlParameterSource param = new BeanPropertySqlParameterSource(item);
    Number key = jdbcInsert.executeAndReturnKey(param);
    item.setId(key.longValue());
    return item;
}
```

나머지는 코드 부분은 기존과 같다.

JdbcTemplateV3Config

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jdbctemplate.JdbcTemplateItemRepositoryV3;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV1;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
@RequiredArgsConstructor
public class JdbcTemplateV3Config {

    private final DataSource dataSource;

    @Bean
    public ItemService itemService() {
        return new ItemServiceV1(itemRepository());
    }

    @Bean
    public ItemRepository itemRepository() {
        return new JdbcTemplateItemRepositoryV3(dataSource);
    }

}
```

- 앞서 개발한 `JdbcTemplateItemRepositoryV3`를 사용하도록 스프링 빈에 등록한다.

ItemServiceApplication - 변경

```
@Slf4j
//@Import(MemoryConfig.class)
```

```
//@Import(JdbcTemplateV1Config.class)
//@Import(JdbcTemplateV2Config.class)
@Import(JdbcTemplateV3Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {}
```

- `JdbcTemplateV3Config.class`를 설정으로 사용하도록 변경되었다.
  - `@Import(JdbcTemplateV2Config.class)` ➡ `@Import(JdbcTemplateV3Config.class)`

실행

http://localhost:8080

- 잘 동작하는지 확인해보자.

JdbcTemplate 기능 정리

JdbcTemplate의 기능을 간단히 정리해보자.

주요 기능

JdbcTemplate이 제공하는 주요 기능은 다음과 같다.

- `JdbcTemplate`
  - 순서 기반 파라미터 바인딩을 지원한다.
- `NamedParameterJdbcTemplate`
  - 이름 기반 파라미터 바인딩을 지원한다. (권장)
- `SimpleJdbcInsert`
  - INSERT SQL을 편리하게 사용할 수 있다.
- `SimpleJdbcCall`
  - 스토어드 프로시저를 편리하게 호출할 수 있다.

> **참고**
>
> 스토어드 프로시저를 사용하기 위한 `SimpleJdbcCall`에 대한 자세한 내용은 다음 스프링 공식 메뉴얼을 참고하자.
>
> https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc-simple-jdbc-call-1

JdbcTemplate 사용법 정리

JdbcTemplate에 대한 사용법은 스프링 공식 메뉴얼에 자세히 소개되어 있다. 여기서는 스프링 공식 메뉴얼이 제공하는 예제를 통해 JdbcTemplate의 기능을 간단히 정리해보자.

> **참고**
> 스프링 JdbcTemplate 사용 방법 공식 메뉴얼
> https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc-JdbcTemplate

조회

단건 조회 - 숫자 조회

```
int rowCount = jdbcTemplate.queryForObject("select count(*) from t_actor",
Integer.class);
```

하나의 로우를 조회할 때는 `queryForObject()`를 사용하면 된다. 지금처럼 조회 대상이 객체가 아니라 단순 데이터 하나라면 타입을 `Integer.class`, `String.class`와 같이 지정해주면 된다.

단건 조회 - 숫자 조회, 파라미터 바인딩

```
int countOfActorsNamedJoe = jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Integer.class,
"Joe");
```

숫자 하나와 파라미터 바인딩 예시이다.

단건 조회 - 문자 조회

```
String lastName = jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        String.class, 1212L);
```

문자 하나와 파라미터 바인딩 예시이다.

단건 조회 - 객체 조회

```
Actor actor = jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        (resultSet, rowNum) -> {
            Actor newActor = new Actor();
            newActor.setFirstName(resultSet.getString("first_name"));
            newActor.setLastName(resultSet.getString("last_name"));
            return newActor;
        },
        1212L);
```

객체 하나를 조회한다. 결과를 객체로 매핑해야 하므로 `RowMapper` 를 사용해야 한다. 여기서는 람다를 사용했다.

목록 조회 - 객체

```
List<Actor> actors = jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        (resultSet, rowNum) -> {
            Actor actor = new Actor();
            actor.setFirstName(resultSet.getString("first_name"));
            actor.setLastName(resultSet.getString("last_name"));
            return actor;
        });
```

여러 로우를 조회할 때는 `query()` 를 사용하면 된다. 결과를 리스트로 반환한다.
결과를 객체로 매핑해야 하므로 `RowMapper` 를 사용해야 한다. 여기서는 람다를 사용했다.

목록 조회 - 객체

```
private final RowMapper<Actor> actorRowMapper = (resultSet, rowNum) -> {
    Actor actor = new Actor();
    actor.setFirstName(resultSet.getString("first_name"));
    actor.setLastName(resultSet.getString("last_name"));
    return actor;
};

public List<Actor> findAllActors() {
    return this.jdbcTemplate.query("select first_name, last_name from t_actor",
actorRowMapper);
```

```
}
```

여러 로우를 조회할 때는 `query()` 를 사용하면 된다. 결과를 리스트로 반환한다.
여기서는 `RowMapper` 를 분리했다. 이렇게 하면 여러 곳에서 재사용 할 수 있다.

변경(INSERT, UPDATE, DELETE)

데이터를 변경할 때는 `jdbcTemplate.update()` 를 사용하면 된다. 참고로 `int` 반환값을 반환하는데, SQL 실행 결과에 영향받은 로우 수를 반환한다.

등록

```
jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

수정

```
jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

삭제

```
jdbcTemplate.update(
        "delete from t_actor where id = ?",
        Long.valueOf(actorId));
```

기타 기능

임의의 SQL을 실행할 때는 `execute()` 를 사용하면 된다. 테이블을 생성하는 DDL에 사용할 수 있다.

DDL

```
jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

스토어드 프로시저 호출

```
jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        Long.valueOf(unionId));
```

정리

실무에서 가장 간단하고 실용적인 방법으로 SQL을 사용하려면 JdbcTemplate을 사용하면 된다.
JPA와 같은 ORM 기술을 사용하면서 동시에 SQL을 직접 작성해야 할 때가 있는데, 그때도 JdbcTemplate을 함께 사용하면 된다.
그런데 JdbcTemplate의 최대 단점이 있는데, 바로 동적 쿼리 문제를 해결하지 못한다는 점이다. 그리고 SQL을 자바 코드로 작성하기 때문에 SQL 라인이 코드를 넘어갈 때 마다 문자 더하기를 해주어야 하는 단점도 있다.

동적 쿼리 문제를 해결하면서 동시에 SQL도 편리하게 작성할 수 있게 도와주는 기술이 바로 MyBatis 이다.

> **참고**
> JOOQ라는 기술도 동적쿼리 문제를 편리하게 해결해주지만 사용자가 많지 않아서 강의에서 다루지는 않는다.