



**СБЕРБАНК ТЕХНОЛОГИИ**

# **Spring JDBC and Spring Transactions**

- Как настроить спринг для получения данных из БД
- Как получать и обновлять данные с помощью Spring JDBC
- Какие преимущества приносит использование Spring-Jdbc?

Добавить в зависимости модуль для работы с JDBC

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-jdbc</artifactId>  
  <version>4.3.2.RELEASE</version>  
</dependency>
```

- JdbcTemplate
- NamedParameterJdbcTemplate
- RowMapper<T>
- SqlParameterSource
- SimpleJdbcXXX classes
  - SimpleJdbcInsert
  - SimpleJdbcCall
- DataAccessException

1. Настроить DataSource для доступа к БД
2. Создать объект JdbcTemplate передав ему настроенный DataSource
3. Использовать JdbcTemplate для доступа к данным
  - Выборка записей
  - Вставка строк
  - Удаление
  - Обновление
  - DDL операции

## @Configuration

```
public class SpringJdbcMain {
```

```
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context =  
            new AnnotationConfigApplicationContext(SpringJdbcMain.class);  
    }
```

## @Bean

```
public DriverManagerDataSource dataSource() {  
    return new DriverManagerDataSource("jdbc:h2:./testdb", "sa", "");  
}
```

@Component

```
public class SpringJdbcTemplateDemo {  
    private JdbcTemplate jdbcTemplate;
```

@Autowired

```
private SpringJdbcTemplateDemo(DataSource dataSource) {  
    jdbcTemplate = new JdbcTemplate(dataSource);  
}
```

```
public void printUsers() {  
    List<Map<String, Object>> list =  
        jdbcTemplate.queryForList("select * from user order by name");  
    for (Map<String, Object> user : list) {  
        System.out.println(user);  
    }  
}
```

```
jdbcTemplate.update("CREATE SEQUENCE IF NOT EXISTS user_seq");
```

```
jdbcTemplate.update(  
    "CREATE TABLE IF NOT EXISTS USER (" +  
        "    id NUMBER (18) PRIMARY KEY, " +  
        "    login VARCHAR(100), " +  
        "    name VARCHAR(200), " +  
        "    last_access timestamp" +  
        "));
```



Так же как и в JDBC можно использовать параметры запроса с помощью «?»»

```
public void printUserWithLogin(String login) {  
    Map<String, Object> user =  
        jdbcTemplate.queryForMap("select * from user where login=?", login);  
    System.out.println(user);  
}
```

Есть ли преимущество параметризованного запроса над конструированным?

```
public void printUserWithLoginBAD(String login) {  
    List<Map<String, Object>> user =  
        jdbcTemplate.queryForList("select * from user where login='" + login + "'");  
    System.out.println(user);  
}
```

# ХАКЕР БЫСТРО НАЙДЕТ ПРЕИМУЩЕСТВА :)



**Всегда используй параметризованные запросы или экранируй ненадежные данные**

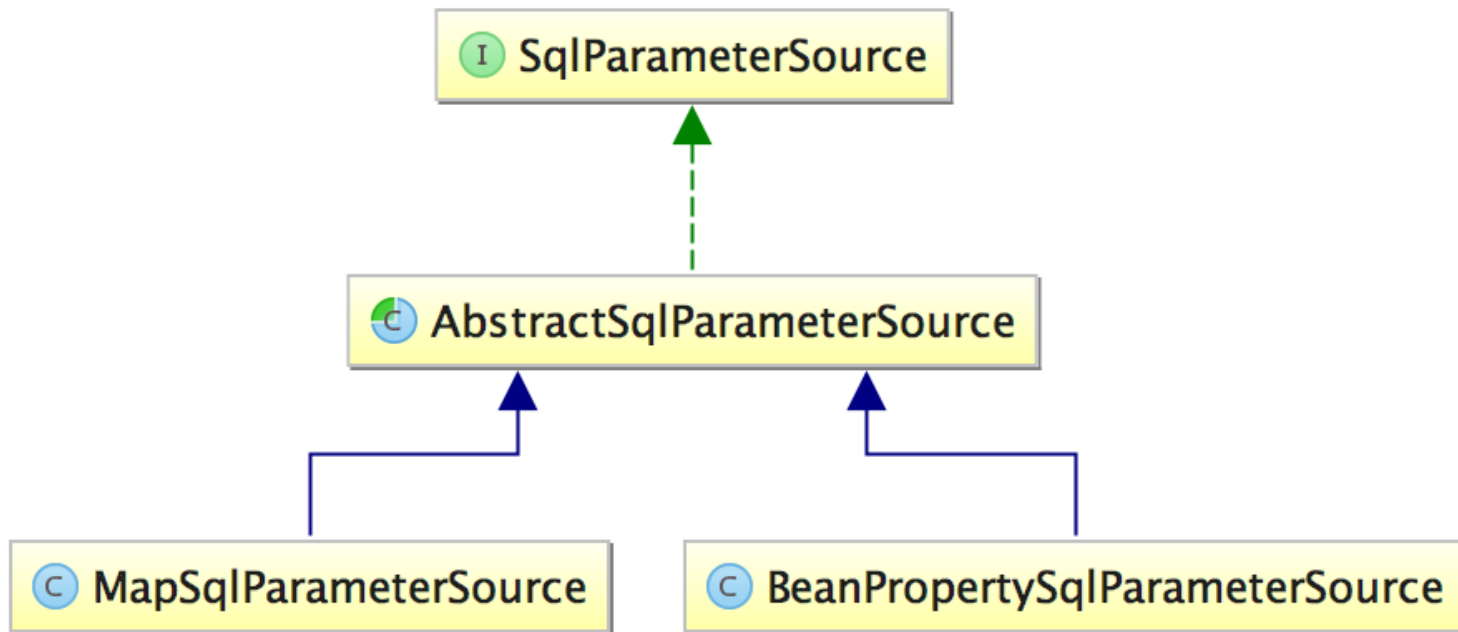


Spring JDBC позволяет указывать именованные параметры в самом запросе.

Для этого используется NamedParameterJdbcTemplate.

```
public void insertUser(User user) {  
    SqlParameterSource params = new BeanPropertySqlParameterSource(user);  
    int result = namedParameterJdbcTemplate.update(  
        "INSERT INTO USER (ID, NAME, login, LAST_ACCESS) " +  
        "VALUES (user_seq.nextval, :name , :login, :lastAccess)", params);  
    System.out.println(result == 1 ? "OK" : "Fail");  
    printUserWithLogin(user.getLogin());  
}
```

Источником именованных параметров может быть Map, свойства объекта











```
public void batchInsertUsers(User... users) {
    int[] ints = jdbcTemplate.batchUpdate("INSERT INTO USER (ID, NAME, login, LAST_ACCESS) " +
        "VALUES (user_seq.nextval, ?, ?, ?)", new BatchPreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement ps, int i) throws SQLException {
            System.out.println(i);
            ps.setString(1, users[i].getName());
            ps.setString(2, users[i].getLogin());
            ps.setTimestamp(3, new Timestamp(users[i].getLastAccess().getTime()));
        }

        @Override
        public int getBatchSize() {
            return users.length;
        }
    });
    int totalUpdates = Arrays.stream(ints).sum();
    System.out.println("total: " + totalUpdates);
}
```

```
public void batchUpdateUsers(List<User> users) {  
    SqlParameterSource[] batchParameters = SqlParameterSourceUtils.createBatch(users.toArray());  
    int[] batchUpdate = namedParameterJdbcTemplate.batchUpdate(  
        "INSERT INTO USER (id, login, NAME, last_access) " +  
        "VALUES (user_seq.nextval, :login, :name, :lastAccess)",  
        batchParameters);  
  
    System.out.println(Arrays.stream(batchUpdate).sum());  
}
```

## SqlParameterSourceUtils

-   createBatch(Map<String, ?>[]): SqlParameterSource[]
-   createBatch(Object[]): SqlParameterSource[]
-   extractCaseInsensitiveParameterNames(SqlParameterSource): Map<String, String>
-   getTypedValue(SqlParameterSource, String): Object

Строки таблиц вида *Map<ColumnName, ColumnValue>* можно конвертировать в типизированные объекты с помощью *RowMapper*-ов

```
interface RowMapper<T> {  
  
    T mapRow(ResultSet rs, int rowNum) throws SQLException;  
  
}
```



```
public User getById(long id) {
    List<User> users = namedParameterJdbcTemplate.query(
        "select ID, LAST_ACCESS, LOGIN, NAME from User where id=:id",
        new MapSqlParameterSource("id", id),
        new RowMapper<User>() {
            @Override
            public User mapRow(ResultSet rs, int rowNum) throws SQLException {
                return new User(rs.getLong(1), rs.getDate(2),
                    rs.getString(3), rs.getString(4));
            }
        });
    if (users.isEmpty()) {
        throw new IllegalArgumentException("No user with id=" + id);
    }
    return users.get(0);
}
```

Добавим таблицу с фотографиями пользователей, которые будем хранить в колонке типа BLOB (Binary long object)

В Spring JDBC управлением LOB структурами занимаются реализации интерфейса LobHandler.

```
@Bean
public LobHandler lobHandler() {
    return new DefaultLobHandler();
}
```

Тогда поле BLOB можно вставить с помощью вспомогательного класса реализующего PreparedStatementCallback интерфейс:

```
public void addPicture(long userId, byte[] photo) {  
    jdbcTemplate.execute("insert into photos (user_id, photo) values (?, ?)",  
        new AbstractLobCreatingPreparedStatementCallback(lobHandler) {  
            @Override  
            protected void setValues(PreparedStatement ps, LobCreator lobCreator)  
                throws SQLException {  
                ps.setLong(1, userId);  
                lobCreator.setBlobAsBytes(ps, 2, photo);  
            }  
        });  
}
```

Загрузка BLOB и CLOB происходит стандартным для JdbcTemplate способом, а для конвертации используется все тот же LobHandler

```
public byte[] getPicture(long userId) {  
    return jdbcTemplate.queryForObject("select photo from photos where user_id = ?",  
        new RowMapper<byte[]>() {  
            @Override  
            public byte[] mapRow(ResultSet rs, int rowNum) throws SQLException {  
                return lobHandler.getBlobAsBytes(rs, 1);  
            }  
        },  
        userId);  
}
```

В разных базах данных могут возникать свои исключительные ситуации, которые на языке JDBC проявляются в виде SQLException. Для более точного разделения этих ситуаций SpringJdbc вводит собственную систему исключений.

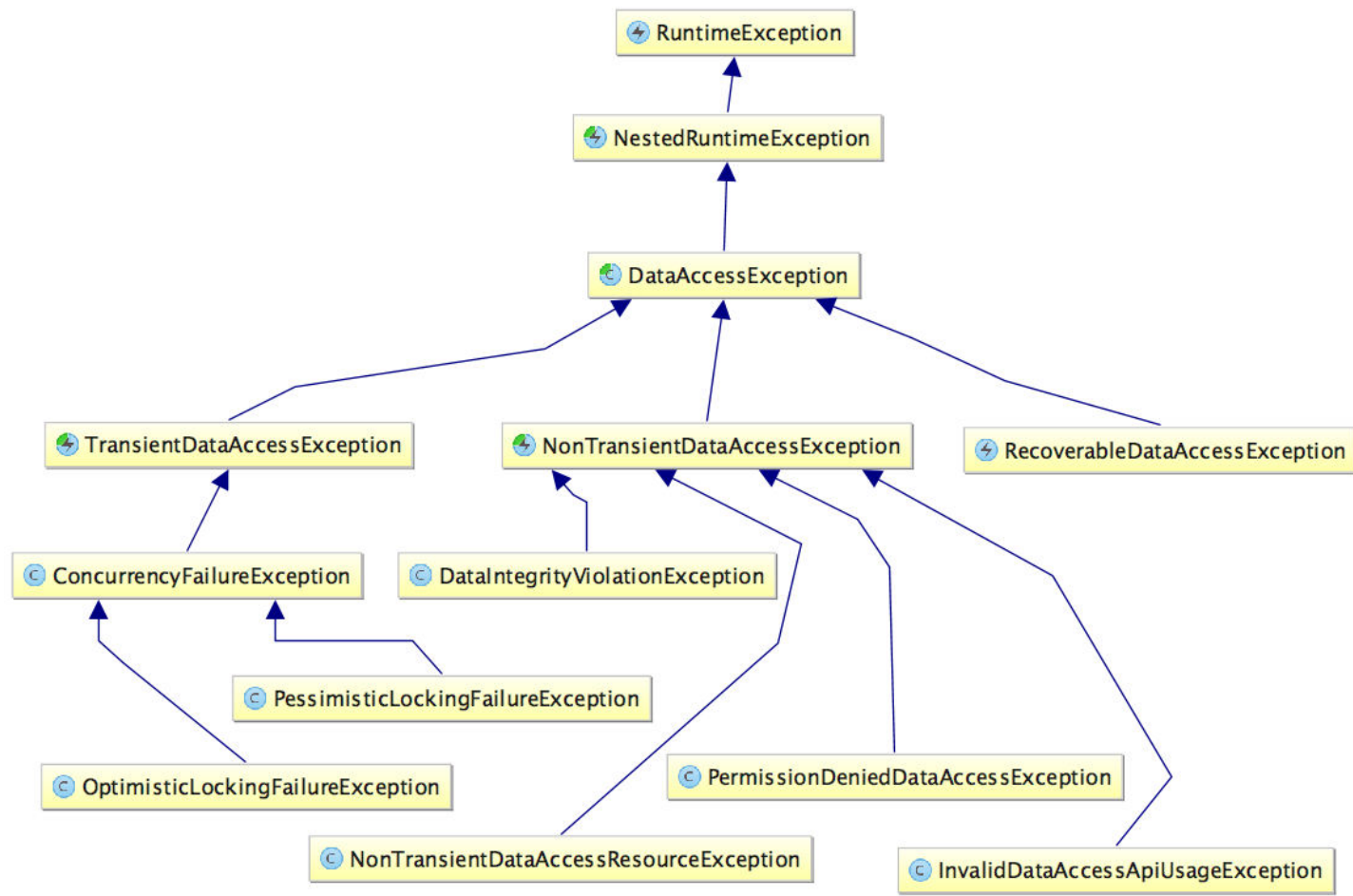
Чтобы встроиться в модель исключений Spring надо зарегистрировать реализацию интерфейса SQLExceptionTranslator

```
interface SQLExceptionTranslator {  
    DataAccessException translate(String task, String sql, SQLException ex);  
}
```

SQLExceptionTranslator входит в SPRING и используется по умолчанию

```
public class CustomSQLErrorCodeTranslator extends SQLErrorCodeSQLExceptionTranslator {  
    @Override  
    protected DataAccessException customTranslate  
        (String task, String sql, SQLException sqlException) {  
        if ("ORA:20666".equals(sqlException.getSQLState())) {  
            return new PermissionDeniedDataAccessException("User blocked", sqlException);  
        }  
        return null;  
    }  
}
```

```
// ...  
CustomSQLErrorCodeTranslator myErrorCodeTranslator = new CustomSQLErrorCodeTranslator();  
jdbcTemplate.setExceptionTranslator(myErrorCodeTranslator);  
//...
```



Это еще не все?



@Component

```
class SimpleJdbcInsertDemo {
```

```
    private final SimpleJdbcInsert simpleJdbcInsert;
```

@Autowired

```
public SimpleJdbcInsertDemo(DataSource dataSource) {  
    simpleJdbcInsert = new SimpleJdbcInsert(dataSource)  
        .withTableName("products")  
        .usingGeneratedKeyColumns("ID");  
}
```

```
long insert(Product product) {  
    BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(product);  
    Number result = simpleJdbcInsert.executeAndReturnKey(params);  
    System.out.println("new product id: " + result);  
    return result.longValue();  
}
```

Рассмотреть самостоятельно возможности и способы применения классов

SimpleJdbcCall

MappingSqlQuery

SqlUpdate

StoredProcedure

Допустим у нас есть таблица счетов

```
public void createSchema() {  
    jdbcTemplate.update(  
        "CREATE TABLE if not exists Account(" +  
            "id bigint IDENTITY, " +  
            "pac VARCHAR (20), " +  
            "balance bigint" +  
            ")");  
}
```

Теперь мы пытаемся выполнить бизнес операцию по переводу средств с одного счета на другой

```
void transfer(int amount, String accountFrom, String accountTo) {  
    accountDao.withdrawal(accountFrom, amount);  
    if (amount > 100) {  
        throw new IllegalStateException();  
    }  
    accountDao.deposit(accountTo, amount);  
}
```

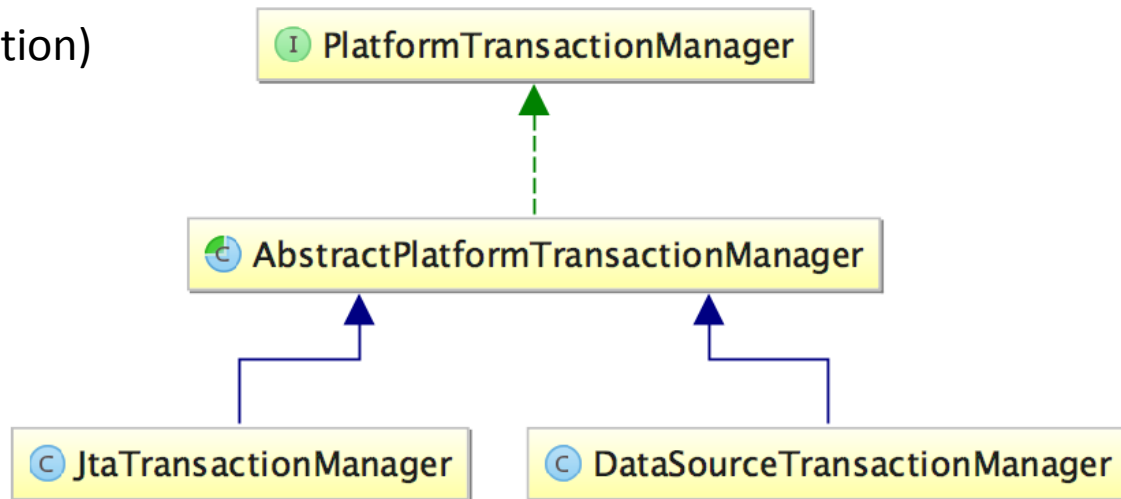
Spring поддерживает следующие модели управления транзакциями

- Программное управление транзакцией
- Декларативное управление транзакцией

Spring скрывает особенности работы с сервисом транзакций (JTS) через абстракцию PlatformTransactionManager.

В зависимости от задач менеджер может быть настроен для работы

- через JDBC connection (Local transactions)
- через JTA (Global transaction)



```
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

PlatformTransactionManager имеет простой интерфейс

```
interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

При запросе менеджера объекта транзакции можно указать следующие свойства желаемой

Свойство	Описание
Isolation	уровень изоляции транзакции — см JDBC IsolationLevels
Propagation	поведение при нескольких открытых уже транзакциях
Timeout	время транзакции после которого будет откат
Read-only	транзакция используется только для чтения

Propagation	Описание
REQUIRED	транзакция требуется, если она еще не связана с потоком то необходимо создать
REQUIRES_NEW	требуется новая вложенная транзакция - текущая, если есть, должна приостановиться на время выполнения новой)
MANDATORY	ожидается что транзакция уже открыта - если нет, то должна выброситься ошибка
etc...	



```
void doTransactionalWork(int amount, String accountFrom, String accountTo) {  
    TransactionStatus transactionStatus =  
        transactionManager.getTransaction(new DefaultTransactionDefinition(PROPAGATION_REQUIRED));  
    try {  
        transfer(amount, accountFrom, accountTo);  
        transactionManager.commit(transactionStatus);  
    } catch (Exception e) {  
        System.out.println("rolling back");  
        transactionManager.rollback(transactionStatus);  
    }  
}
```

Иногда бывает полезно узнать статус выполнения транзакции

```
interface TransactionStatus extends SavepointManager {  
  
    boolean isNewTransaction();  
  
    boolean hasSavepoint();  
  
    void setRollbackOnly();  
  
    boolean isRollbackOnly();  
  
    void flush();  
  
    boolean isCompleted();  
  
}
```

Использование PlatformTransactionManager вынуждает вручную вызывать методы связанные с началом и завершением транзакции (getTransaction, commit, rollback). Это может привести к ошибкам программиста!

Поэтому в арсенале Spring есть шаблонный метод работы с транзакциями — класс TransactionTemplate.

```
@Bean
public TransactionTemplate transactionTemplate(PlatformTransactionManager transactionManager) {
    DefaultTransactionDefinition definition =
        new DefaultTransactionDefinition(TransactionDefinition.PROPGATION_REQUIRED);
    definition.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
    return new TransactionTemplate(transactionManager, definition);
}
```

```
public void doTransactionalWork(final int amount, final String accountFrom, final String accountTo) {  
    transactionTemplate.execute(new TransactionCallbackWithoutResult() {  
        @Override  
        protected void doInTransactionWithoutResult(TransactionStatus status) {  
            transfer(amount, accountFrom, accountTo);  
        }  
    });  
}
```

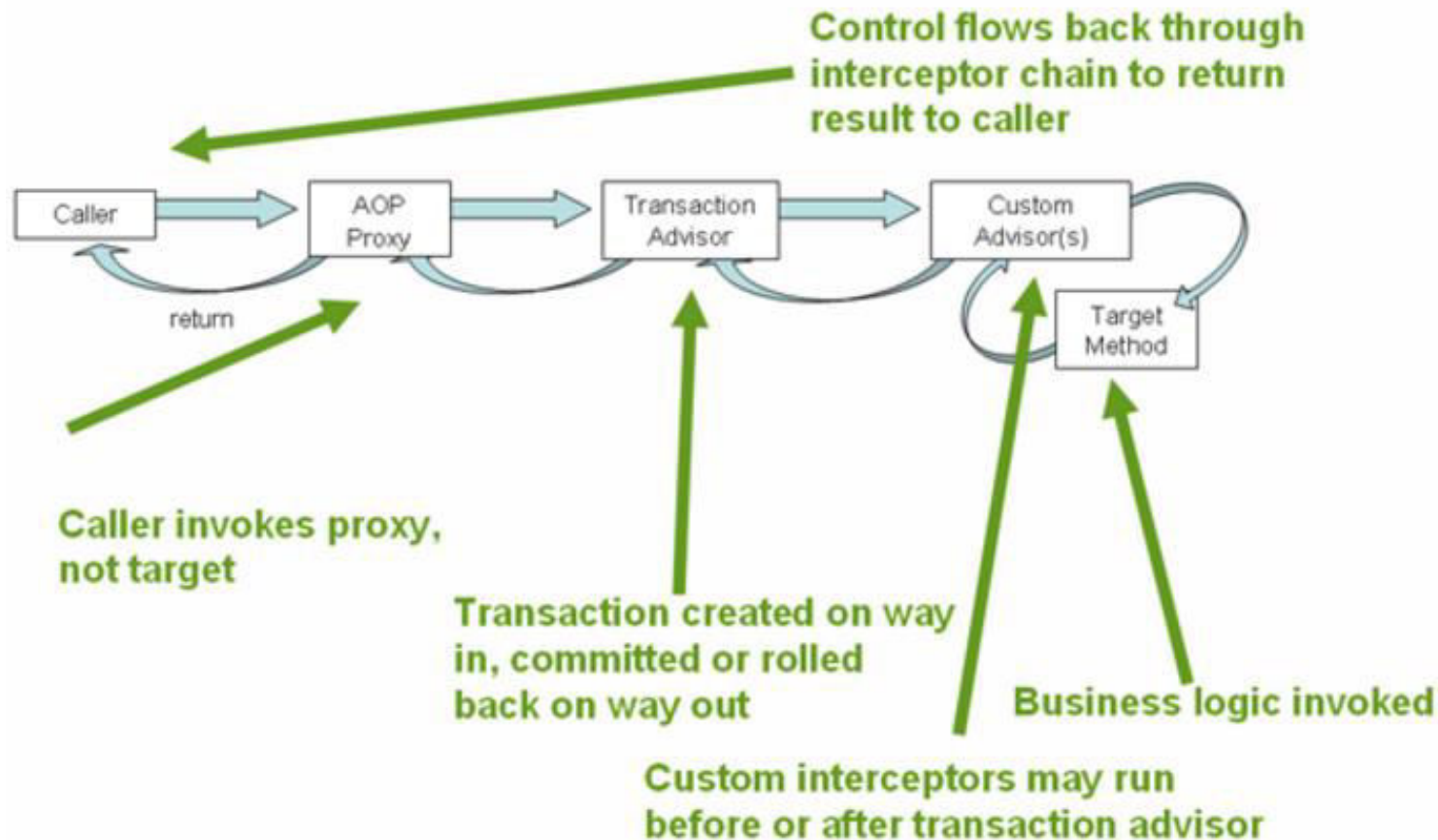
Для небольших проектов ручное управление транзакциями вполне подходит, однако в проектах, где число бизнес операций выше 10-15 код становится трудно управляемым — в случае изменений правил работы с транзакциями потребуется во всех местах править.

Поэтому в проектах с большим числом операций с БД чаще используется декларативный подход.

Декларативность осуществляется с помощью Spring-AOP

- транзакционные методы **декларируются** аннотациями или pointcut
- Spring создает TransactionProxy, который перед вызовом метода регистрирует начало транзакции а по завершении метода коммитит или откатывает транзакцию

# TRANSACTIONAL PROXY IN ACTION



@Configuration

@EnableTransactionManagement



@ComponentScan

public class SpringJdbcMain {

@Bean

```
public DriverManagerDataSource dataSource() {  
    return new DriverManagerDataSource("jdbc:h2:./testdb", "sa", "");  
}
```

@Bean

```
public PlatformTransactionManager transactionManager(DataSource dataSource) {  
    return new DataSourceTransactionManager(dataSource);  
}
```

```
@Transactional(readOnly = true)
public void showAccounts() {
    System.out.println("----- B A L A N C E -----");
    accountDao.showAccounts()
        .stream()
        .forEach(System.out::println);
    System.out.println("----- B A L A N C E -----");
}
```

```
@Transactional
public void transfer(int amount, String accountFrom, String accountTo) {
    accountDao.withdrawal(accountFrom, amount);
    if (amount > 100) {
        throw new IllegalStateException();
    }
    accountDao.deposit(accountTo, amount);
}
```



Свойство	Default	Описание
value	`transactionManager`	имя менеджера транзакции
propagation	PROPAGATION_REQUIRED	Применяемая стратегия наложения или создания транзакции
isolation	ISOLATION_DEFAULT	Уровень изоляции
readOnly	false	только для чтения
timeout	TIMEOUT_DEFAULT	таймоут транзакции после которого — откат
rollbackFor	[RuntimeException]	Список исключений, которые приводят к откату транзакции
noRollbackFor	[]	Список исключений, которые не должны приводить к откату транзакции

Так как для осуществления декларативной транзакции используется SpringAOP, то необходимо, чтобы @Transactional методы были **public**.

Аннотировать можно весь класс - тогда все публичные методы будут транзакционными. При этом аннотации на методах перекрывают определения на классе — более специфичная область применения.

Транзакционными будут те вызовы методов, которые вызваны на АОП-прокси, вызов метода на **this** уже проходит мимо прокси и в декларативной модели участвовать уже не может

@EnableTransactionManagement объявляет только о необходимости искать @Transactional методы только в бинах, объявленных в текущем Spring контексте.

- <http://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/html/jdbc.html>
- <http://www.baeldung.com/spring-jdbc-jdbctemplate>
- <http://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/html/transaction.html>

Разработать консольное приложение для хранения рецептов.

Функциональность:

- Поиск рецепта по имени или части имени блюда
- Добавление рецепта - рецепт состоит из множества ингредиентов и их количественного состава
- Удаление блюда