Spring JDBC — это модуль Spring Framework, который предоставляет удобный и простой способ работы с базами данных через JDBC API. Предоставляет доступ к реляционным базам данных без использования всей сложности JPA.

JPA предлагает такие функции, как ленивая загрузка (lazy loading), кеширование и отслеживание изменений (dirty tracking). Эти фичи очень крутые, если они действительно вам нужны. Однако они могут усложнить понимание логики доступа к данным.

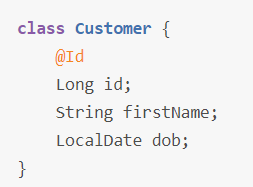
Spring Data JDBC фокусируется на гораздо более простой модели. Не будет кэширования, отслеживания изменений, или ленивой загрузки. SQL запросы будут выполнены тогда и только тогда, когда вы вызываете метод репозитория. Возвращаемый результат будет полностью загружен в память после выполнения метода. Не будет и механизма “сессии” или прокси-объектов для entities. Нет автоматического сохранения сущностей как в jpa, все операции с БД должны вызываться явно.

Spring Data JDBC можно настроить следующей конфигурацией.



1. **@EnableJdbcRepositories** активирует автоматическое создание репозиториев. Для того, чтобы это работало, нужно предоставить несколько дополнительных бинов, для чего и потребуется остальная часть нашего класса конфигурации.
2. Очень важный компонент это **NamedParameterJdbcOperations**, который используется для выполнения запросов к базе.
3. Менеджер транзакций, строго говоря, не обязателен. Но без него не будет и поддержки транзакций.
4. Spring Data JDBC не использует **DataSource** напрямую, но TransactionManager и NamedParameterJdbcOperation требуют его наличия в контексте, поэтому мы и определяем нужный бин.

Теперь определяем Entity (обратите внимание, что @Id импортируется из org.springframework.data.annotation.Id) :



Spring также может реализовать репозиторий за вас. Для этого достаточно объявить интерфейс, унаследованный от **CrudRepository.**



**@Modifying –** Аннотация, указывающая что запрос обновляет или удаляет данные.

**JdbcTemplate –** ключевой компонент Spring JDBC. Выполняет основные операции доступа к бд:

* создание и закрытие соединений
* выполнение операторов и вызовов хранимых процедур
* итерация по ResultSet и возврат результатов.

Он автоматически открывает и закрывает соединение с базой данных, управляет транзакциями и обрабатывает исключения.

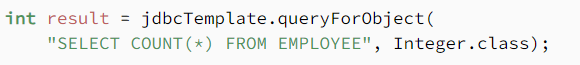
Spring JDBC также предоставляет поддержку транзакций через транзакционный менеджер Spring. Это позволяет управлять транзакциями на уровне приложения и обеспечивает максимальную надежность и целостность данных.

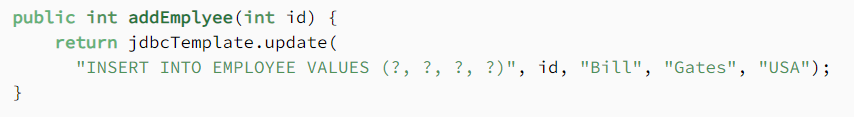
Экземпляры класса JdbcTemplate являются потокозащищёнными. Это значит, что настроив единственный экземпляр класса JdbcTemplate, мы можем затем его использовать для нескольких объектов DAO.

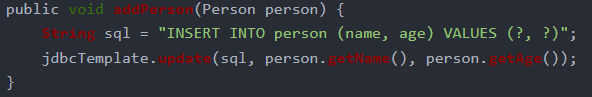
В JDBCTemplate не предусмотрены стандартные средства ORM в отличии от Hibernate или JPA, так что при его использовании нужно самостоятельно писать мапперы, которые будут преобразовывать данные полученные из БД в объекты классов сущностей.

Методы JdbcTemplate:

* **execute**(String sql) – выполняет произвольный SQL-запрос без возвращения результата.
* **query**(String sql, RowMapper<T> rowMapper, Object... args) – выполняет выборку из базы данных с использованием параметризованного SQL-запроса. Результаты выборки мапятся на объекты типа T с помощи RowMapper.
* **update**(String sql, Object... args) – выполняет обновление базы данных (INSERT, UPDATE или DELETE) с использованием параметризованного запросаю Аргументы передаются в запрос в качестве параметров. Возвращает количество затронутых строк.
* **queryForObject**(String sql, Class<T> requiredType, Object... args) – выполняет запрос в бд и возвращает единственный объект типа T. Если запрос вернул несколько строк, будет выброшено исключение.
* **batchUpdate**(String sql, List<Object[]> batchArgs) – выполняет несколько обновлений бд в одной транзакции. Каждое обновление представлено массивом параметров. Возвращает массив int[], представляющий количество затронутых строк для каждого запроса.



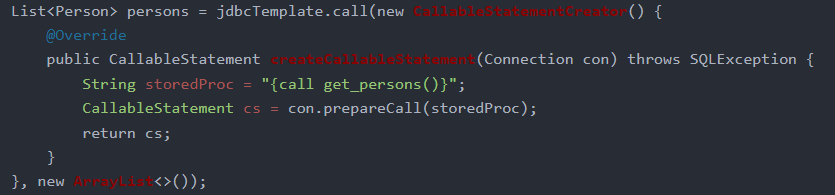




**Работа с хранимыми процедурами**

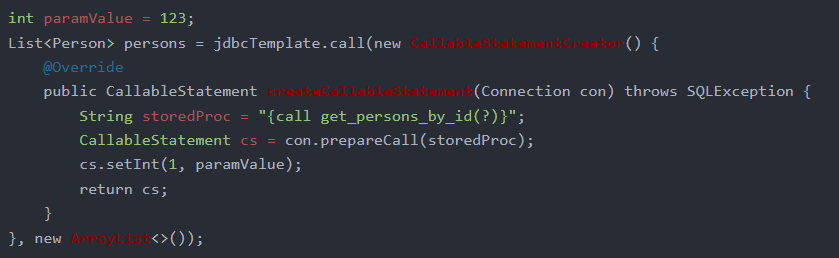
**JdbcTemplate.call**(CallableStatementCreator csc, List<SqlParameter> declaredParameters) **–** метод для выполнения хранимых процедур.

предположим, что у нас есть хранимая процедура get\_persons, которая возвращает список записей из таблицы person. Мы можем вызвать эту процедуру с помощью JdbcTemplate следующим образом:



Интерфейс **CallableStatementCreator** создает **CallableStatement**, который вызывает хранимую процедуру get\_persons. Метод **createCallableStatement**() возвращает объект CallableStatement, который содержит информацию о вызываемой процедуре.

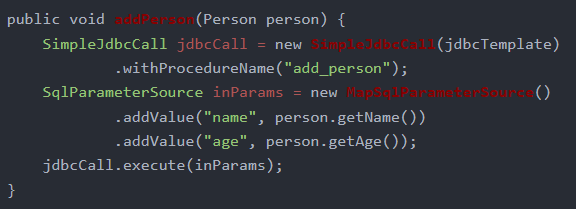
Если хранимая процедура принимает параметры, мы можем передать их в списке параметров вторым аргументом метода call().

****

Мы используем метод **setInt()** объекта CallableStatement для установки значения параметра.

**SimpleJdbcCall** - это класс в Spring JDBC, который предоставляет удобный способ вызова хранимых процедур и функций базы данных.

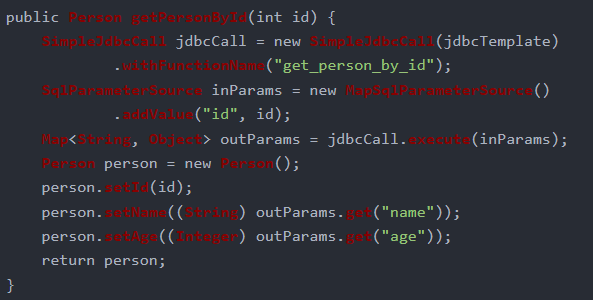
SimpleJdbcCall использует объект JdbcTemplate для выполнения запросов к базе данных. Он позволяет вызывать хранимые процедуры и функции базы данных, передавать параметры и получать результаты.



**withProcedureName() –** метод, в котором указываем имя хранимой процедуры.

объект **SqlParameterSource** содержит значения параметров для хранимой процедуры.

SimpleJdbcCall также позволяет получать результаты из хранимых процедур и функций.

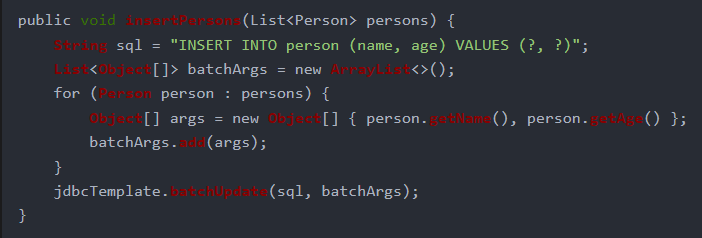


Метод execute() возвращает Map, который содержит результаты выполнения хранимой функции. Метод batchUpdate() может также возвращать массив целых чисел, которые представляют количество затронутых строк каждым SQL-запросом в batch-запросе.

**Batch-запросы**

Batch-запросы в Spring JDBC позволяют выполнить несколько SQL-запросов одновременно в рамках одной транзакции. Это может увеличить производительность при выполнении большого количества запросов, таких как вставка, обновление или удаление данных.

Выполнить batch-запрос можно методом **JdbcTemplate.** **batchUpdate()**, который в простейшем случае принимает массив строк SQL-запросов и массив объектов параметров для каждого запроса.



**Управление транзакциями**

**DataSourceTransactionManager -** это менеджер транзакций, который работает с объектом DataSource, чтобы управлять транзакциями в базе данных. Он автоматически управляет транзакциями при выполнении операций через JdbcTemplate.

Мы помечаем метод аннотацией **@Transactional,** чтобы обеспечить выполнение операции в рамках транзакции.

При вызове метода **updatePerson() Spring** автоматически создаст транзакцию, выполнит операцию обновления и закроет транзакцию. Если во время выполнения операции возникнет исключение, то транзакция будет откатана.

В некоторых случаях может понадобиться выполнить ручное управление транзакциями. Например, если нужно выполнить несколько операций в рамках одной транзакции, которые не могут быть выполнены с помощью одного объекта JdbcTemplate, или если нужно выполнить операции в базе данных в одной транзакции вне контекста Spring.

**PlatformTransactionManager** - это интерфейс, который определяет методы для управления транзакциями в Spring.

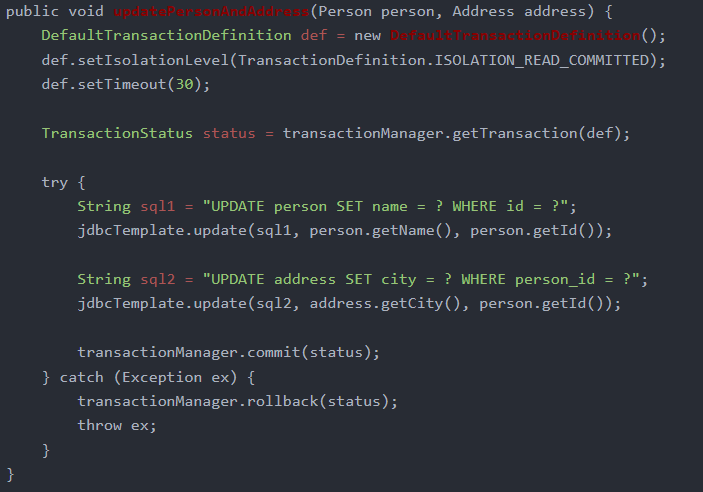
**TransactionDefinition** - это объект, который определяет параметры транзакции, такие как уровень изоляции, тайм-аут и т.д.

**TransactionStatus** - это объект, который представляет текущее состояние транзакции, такое как флаги rollbackOnly и completed.

**TransactionCallback** - это объект, который определяет операции, которые нужно выполнить в рамках транзакции.

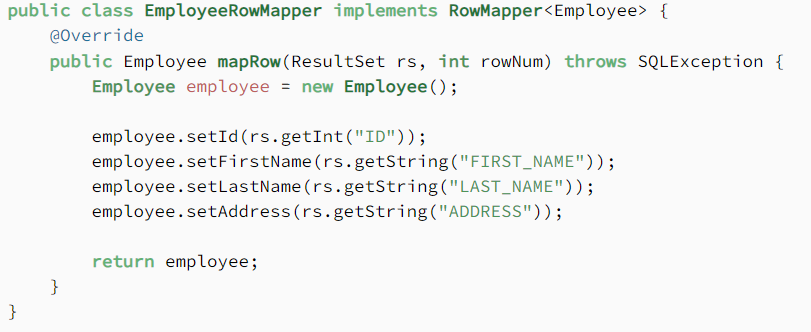
Вызываем у TransactionManager метод **getTransaction()** который возвращает TransactionStatus. В качестве параметра передаем TransactionDefinition. Чтобы зафиксировать транзакцию, вызываем у transactionManager метод **commit()**, чтобы откатить **rollback().**

Пример ручной обработки транзакции:



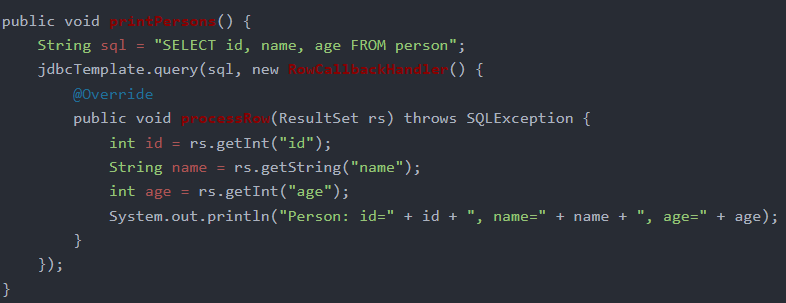
**RowMapper –** интерфейс в Spring JDBC, который используется для маппинга строк из результата выполнения SQL-запроса на объекты Java.

Определяет единственный метод **mapRow()**, который принимает ResultSet SQL-запроса и индекс строки в результате. Метод возвращает замапленный Java объект.



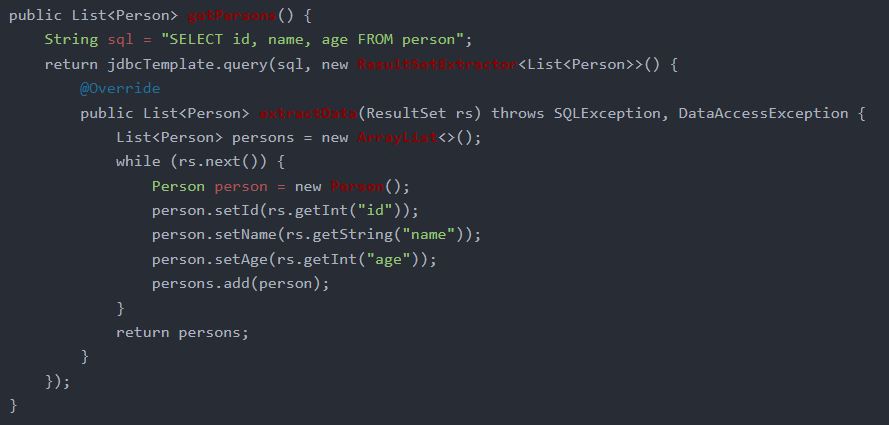
**RowCallbackHandler –** интерфейс, который используется для обработки каждой строки из результата выполнения SQL запроса.

Определяет единственный метод **processRow()**, который принимает текущую строку из результата выполнения. Метод не возвращает значения, а просто обрабатывает текущую строку.



**ResultSetExtractor –** интерфейс, который используется для извлечения результата выполнения SQL-запроса из ResultSet. Он позволяет получить объект Java или коллекцию объектов Java из ResultSet.

Определяет единственный метод extractData(), который принимает ResultSet и возвращает объект Java или коллекцию объектов Java.



Этот интерфейс в основном используется в самой системе JDBC. RowMapper обычно является более простым выбором для обработки ResultSet. В ResultSetExtractor мы должны сами итерироваться по Resultset.

Основная разница между RowMapper и ResultSetExtractor заключается в том, что RowMapper используется для маппинга каждой строки из результата выполнения SQL-запроса на отдельный объект Java, тогда как ResultSetExtractor используется для извлечения всего результата выполнения SQL-запроса в один объект Java или коллекцию объектов Java.

**JdbcTemplate** перехватывает исключения JDBC и транслирует их в исключения типа **DataAccessException**.

Некоторые из наиболее часто используемых исключений, генерируемых JdbcTemplate:

- **BadSqlGrammarException** - возникает при синтаксических ошибках в SQL-запросе.

- **DataAccessException** - общее исключение для ошибок доступа к данным.

- **DataIntegrityViolationException** - возникает при нарушении ограничений целостности данных, таких как уникальность или ссылочная целостность.

- **DuplicateKeyException** - возникает при попытке добавления записи с уже существующим первичным ключом.

- **IncorrectResultSizeDataAccessException** - возникает, когда результат SQL-запроса не соответствует ожидаемому размеру.

- **UncategorizedSQLException** - возникает, когда JDBC генерирует исключение, которое не может быть классифицировано как одно из других исключений.

JdbcTemplate является потокобезопасным и может использоваться в многопоточных приложениях без дополнительной синхронизации. Однако, если несколько потоков используют один и тот же экземпляр JdbcTemplate для доступа к базе данных, то может возникнуть ситуация гонки (race condition) при использовании общих ресурсов JDBC, таких как соединения с базой данных. В этом случае, рекомендуется использовать отдельные экземпляры JdbcTemplate для каждого потока или использовать пул соединений (connection pool), который может автоматически управлять доступом к соединениям с базой данных.

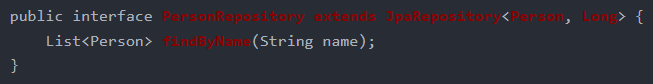
**NamedParameterJdbcTemplate –** обертка над JdbcTemplate, в которой параметры можно задавать именем, а не знаком вопроса. Под капотом он заменяет именованные параметры на JDBC ? и делегирует работу JdbcTemplate.

**SPRING REPOSITORIES**

**Репозитории** Spring Data JPA **-** это интерфейсы, которые вы можете определить для доступа к данным.

Репозиторий - это интерфейс, который определяет методы для работы с данными в базе данных. Spring Data автоматически генерирует реализацию этих методов на основе сигнатуры методов.

Репозитории в Spring Data могут быть созданы путем создания интерфейса, который наследует от одного из интерфейсов-маркеров Spring Data (например, **CrudRepository**, **JpaRepository**, **MongoRepository** и т.д.), и определения методов для выполнения операций над данными.



**CrudRepository –** предоставляет базовые CRUD операции:

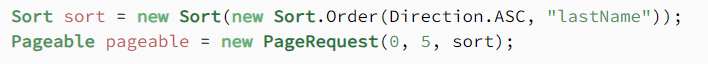
* **save**(S entity) – сохранить сущность (создать новую, либо обновить если она уже есть в БД);
* **findOne**(ID primaryKey) – получить сущность по первичному ключу;
* **findAll() –** получить итерируемый объект всех сущностей;
* **count() –** получить количество сущностей;
* **delete(T entity) –** удалить сущность из бд;
* **exists(ID primaryKey) –** проверить, есть ли в БД сущность с указанным идентификатором;

**PagingAndSortingRepository –** наследуется от CrudRepository. Предоставляет возможности постраничного запроса и сортировки. Объявляет следующие методы:

* **findAll**(Pageable pageable) – получить страницу из бд, основываясь на объекте **Pageable**;
* **findAll**(Sort sort) – получить отсортированный Iterable.

**Pageable** содержит следующие свойства

* Размер страницы
* Номер текущей страницы
* Сортировка



**JpaRepository -** наследуется от PagingAndSortingRepository. Предоставляет методы, связанные с JPA. Такие как flush кэша PersistenceContext и Batch удаление записей. Объявляет следующие методы:

* **flush() –** отправить все изменения, отслеживаемые контекстом в БД.
* **saveAndFlush(T entity) –** сохранить сущность и немедленно отправить изменения в БД.
* **deleteInBatch(Iterable<T> entities) –** удалить набор сущностей одним пакетом.

Для более сложных запросов вы можете аннотировать ваш метод аннотацией Query.

**@Query(“sql string”) –** позволяет указать какой SQL запрос должен быть выполнен с помощью JPQL.

JPQL (Java Persistence Query Language) - это язык запросов, который используется для выполнения запросов к объектам базы данных в Java Persistence API (JPA). JPQL является объектно-ориентированным языком запросов, который позволяет работать с объектами и атрибутами сущностей, а не с таблицами и столбцами, как в SQL.

Можно использовать различные ключевые слова, такие как SELECT, FROM, WHERE, ORDER BY, GROUP BY, JOIN и др., для создания более сложных запросов.

JPQL также поддерживает использование параметров, которые могут быть переданы в запрос во время выполнения. Например:

SELECT c FROM Customer c WHERE c.name = :name

**SPRING TRANSACTIONAL**

**Spring** **Transactional** - это механизм управления транзакциями в Spring Framework. Он предоставляет аннотацию @Transactional, которая позволяет управлять транзакциями в методах сервисов и DAO-классах.

Аннотация **@Transactional** может быть применена к методу или классу и указывает, что метод должен выполняться в рамках транзакции.

Spring Transactional позволяет управлять транзакциями на уровне методов, что делает код более удобочитаемым и позволяет сосредоточиться на бизнес-логике приложения.