Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №4
Фреймворк Spring. Работа с базами данных.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Введение

После знакомства с контейнером Spring пришло время применить полученные знания для создания действующего приложения. Лучше всего начать с требования, которое предъявляется к любому корпоративному приложению, такого как возможность хранения данных. Каждый из нас, вероятно, уже имел дело с базами данных в прошлом. При этом вы знаете, что доступ к данным имеет много препятствий. Нужно настроить фреймворк для доступа к данным, открыть соединения, обработать различные исключения и закрыть соединения. Если в этой последовательности что-то будет сделано неправильно, можно испортить или удалить важные данные. Если вам не приходилось испытывать последствия ошибок при обращении с данными, то поверьте на слово, это весьма неприятно.

Так как мы стремимся избежать неприятностей, обратимся за по мощью к фреймворку Spring. В Spring имеется набор модулей для интеграции с различными технологиями хранения данных. Если для хранения данных используется непосредственно JDBC, iBATIS или фреймворк объектно-реляционного отображения (ORM), такой как Hibernate, Spring избавит вас от рутины при разработке программного кода, реализующего доступ к данным. Вместо возни с низкоуровневым доступом к данным можно положиться на Spring, который выполнит эту работу за вас, и сконцентрироваться на управлении данными в самом приложении.

2 Философия доступа к данным в Spring

Одна из целей Spring – обеспечить возможность разработки приложений, основанных на интерфейсах, в соответствии с принципами объектно-ориентированного программирования. Поддержка доступа к данным в Spring не является исключением.

Аббревиатура DAO обозначает «объект доступа к данным» (Data Access Object), что прекрасно описывает роль DAO в приложении. Объекты доступа к данным являются средствами чтения и записи данных в базу данных. Они должны предоставлять эту функциональность через интерфейс, доступный остальной части приложения.

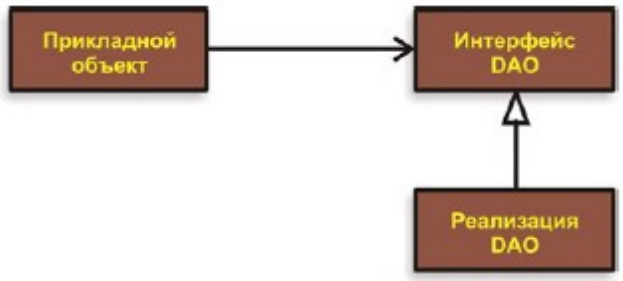

Рисунок 1 — Прикладные объекты не реализуют доступа к данным. Вместо этого они делегируют эти операции объектам доступа к данным.

Рисунок 1 демонстрирует правильный подход к проектированию *слоя доступа к данным*. Как показано на рисунке, прикладные объекты получают доступ к DAO через интерфейсы. Такая организация дает несколько преимуществ. Во-первых, это упрощает тестирование прикладных объектов, так как они не связаны с конкретной реализацией доступа к данным. На самом деле можно создать фиктивные реализации этих интерфейсов доступа к данным, позволяющие проверять прикладные объекты без необходимости подключения к базе данных, что значительно ускоряет процесс тестирования и исключает вероятность ошибок из-за противоречивости данных.

Кроме того, слой доступа к данным обеспечивает независимость от конкретной реализации хранилища. То есть подход на основе использования DAO обеспечивает «изоляцию» выбранного хранилища, предоставляя только ключевые методы доступа к данным через интерфейс. Это повышает гибкость архитектуры приложения и позволяет заменить используемый фреймворк на другой с минимальными воздействием на остальной программный код. Если детали реализации уровня доступа к данным будут просачиваться в другие части приложения, то все приложение окажется тесно связанным с уровнем доступа к данным, что сделает конструкцию приложения более жесткой.

Один из способов, какими Spring может помочь оградить уровень доступа к данным от остальной части приложения, заключается в предоставлении последовательной иерархии исключений, используемых во всех модулях DAO.

3 Знакомство с иерархией исключений доступа к данным в Spring

Напомним, что такое JDBC. Java DataBase Connectivity (JDBC) — платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE.

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

В JDBC и шагу нельзя ступить без необходимости перехватывать исключение SQLException. Это исключение свидетельствует, что произошла какая-то ошибка при обращении к базе данных, но оно содержит слишком мало информации о том, в чем, собственно, заключается ошибка и как ее обрабатывать.

В число распространенных проблем, которые могут вызывать исключение SQLException, входят:

- приложение не может подключиться к базе данных;
- выполняемый запрос имеет синтаксические ошибки;
- таблицы и/или столбцы, упомянутые в запросе, не существуют;

- попытка вставить или обновить значения нарушает ограничения базы данных.

Самый сложный вопрос, связанный с исключением SQLException определить, как правильно его обработать. Как оказывается, многие проблемы, вызвавшие исключение SQLException, не могут быть обработаны в блоке catch. Большинство исключений SQLException означают фатальную ошибку. Если приложение не может подключиться к базе данных, это обычно означает, что приложение не может продолжать работу. Аналогично, если ошибка содержится в запросе, трудно что-то сделать с этим во время выполнения.

Если после появления исключения SQLException ничего нельзя сделать для нормального продолжения работы приложения, зачем тогда его перехватывать?

Даже при наличии алгоритма решения некоторых проблем, связанных с исключением SQLException, вам все равно придется пере- хватить исключение SQLException и разбираться в его свойствах для получения дополнительной информации о характере проблемы. Это обусловлено тем, что исключение SQLException представляет собой «один ответ на все проблемы», связанные с доступом к данным. Вместо того чтобы иметь разные типы исключений для каждой возможной проблемы, используется исключение SQLException, которое возбуждается при любых проблемах доступа к данным.

Некоторые фреймворки доступа к базам данных предлагают более разнообразные иерархии исключений. Hibernate, например, предлагает около двух десятков различных исключений, каждое из которых соответствует определенной проблеме. Это делает возможным написать блоки catch для конкретных исключений, которые предполагается обрабатывать.

Но следует понимать, что исключения, реализованные в Hibernate, являются специфичными для этого фреймворка. А как было сказано выше, желательно было бы изолировать специфику механизма хранения в слое доступа к данным. Если в этом слое возбуждаются исключения Hibernate, то факт использования фреймворка Hibernate вне всяких сомнений просочится в остальные части приложения. Таким образом, или с этим придется смириться, или преобразовывать перехваченные исключения в некие новые, универсальные исключения, возбуждая их повторно.

С одной стороны, иерархия исключений JDBC является слишком общей. Фактически ее трудно даже назвать иерархией. С другой стороны, иерархия исключений в Hibernate является характерной для Hibernate. Нам же необходима иерархия исключений, достаточно информативная, но не связанная непосредственно с конкретным фреймворком доступа к данным.

3.1 Универсальная иерархия исключений в Spring

Модуль Spring JDBC реализует собственную иерархию исключений доступа к данным, позволяющую решить обе проблемы. В отличие от JDBC, в

Spring имеются несколько исключений доступа к данным, каждое из которых описывает причины их возникновения.

Хотя иерархия исключений в Spring гораздо богаче, чем простое исключение SQLException в JDBC, она не связана с каким-либо конкретным решением. Это означает, что фреймворк Spring будет возбуждать одни и те же исключения независимо от выбранного механизма доступа к данным. Это поможет изолировать выбранное решение в слое доступа к данным .

Исключения JDBC	Исключения доступа к данным в Spring
BatchUpdateException DataTruncation SQLException SQLWarning	CannotAcquireLockException CannotSerializeTransactionException CleanupFailureDataAccessException ConcurrencyFailureException DataAccessException DataAccessResourceFailureException DataIntegrityViolationException DataRetrievalFailureException DeadlockLoserDataAccessException EmptyResultDataAccessException IncorrectResultSizeDataAccessException IncorrectUpdateSemanticsDataAccessException InvalidDataAccessApiUsageException InvalidDataAccessResourceUsageException OptimisticLockingFailureException PermissionDeniedDataAccessException PessimisticLockingFailureException TypeMismatchDataAccessException UncategorizedDataAccessException

Из таблицы не очевидно, что все эти исключения наследуют исключение DataAccessException. Особенность исключения DataAccessException состоит в том, что оно является неконтролируемым исключением. Другими словами, нет необходимости перехватывать исключения доступа к данным, возбуждаемые фреймворком Spring (хотя при желании это возможно). Исключение DataAccessException – это лишь частный пример многогранной философии Spring противопоставления контролируемых и неконтролируемых исключений. Фреймворк Spring придерживается позиции, согласно которой многие исключения являются результатом проблем, не разрешимых в блоке catch. Вместо того чтобы вынуждать разработчиков писать блоки обработки исключений (которые часто остаются пустыми), Spring продвигает идею

неконтролируемых исключений. Это позволяет оставить решение о выборе места обработки исключений за разработчиком.

Чтобы воспользоваться преимуществами исключений доступа к данным в Spring, необходимо использовать один из шаблонов доступа к данным, поддерживаемых фреймворком. Посмотрим, как шаблоны Spring могут упростить работу с данными.

4 Шаблоны доступа к данным

4.1 Паттерн “Шаблонный метод”

В паттерне проектирования «Шаблонный метод» абстрактный класс обертывает некоторые заранее выбранные части алгоритма в метод, позволяющий переопределять его части без переписывания. Для выполнения подобных операций в своих приложениях вы можете использовать конкретный класс. Этот паттерн относится к поведенческим паттернам проектирования.

Среди преимуществ использования паттерна «Шаблонный метод»:

- уменьшение объема стереотипного кода в приложении за счет переиспользования кода;
- этот паттерн создает шаблон — способ повторного использования нескольких схожих алгоритмов в целях удовлетворения каких-либо бизнес-требований.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна «Шаблонный метод»

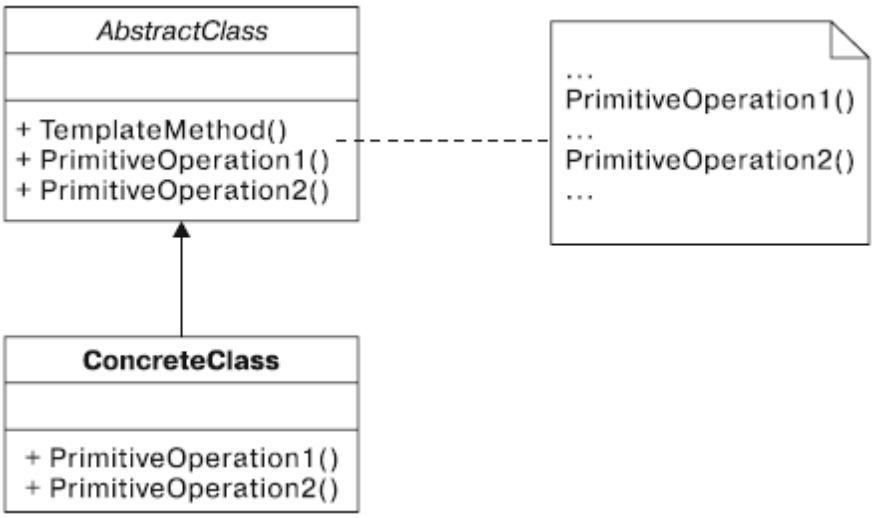

Рисунок 2 – UML-диаграмма для паттерна проектирования «Шаблонный метод»

- AbstractClass (Абстрактный класс). Абстрактный класс, включающий шаблонный метод с описанием каркаса алгоритма.

- ConcreteClass (Конкретный класс). Конкретный подкласс AbstractClass, реализующий операции выполнения простейших шагов, относящихся к конкретному алгоритму.

4.2 Шаблоны доступа к данным в Spring

Этот шаблон применяется в Spring для доступа к данным. Независимо от используемых технологий для доступа к данным требуется выполнить определенные шаги. Например, всегда следует создавать подключение к хранилищу данных и освобождать ресурсы по завершении. Это – фиксированные шаги в процессе доступа к данным. Но каждый метод доступа к данным имеет свои отличительные особенности. Мы запрашиваем разные объекты и обновляем данные по-разному. Это – переменные шаги в процессе доступа к данным.

Фреймворк Spring разделяет фиксированные и переменные части процесса доступа к данным на два отдельных класса: шаблоны и обратные вызовы. Шаблоны управляют фиксированной частью процесса, а пользовательский программный код доступа к данным обслуживает обратные вызовы. Зоны ответственности этих двух классов изображены на рисунке 3.

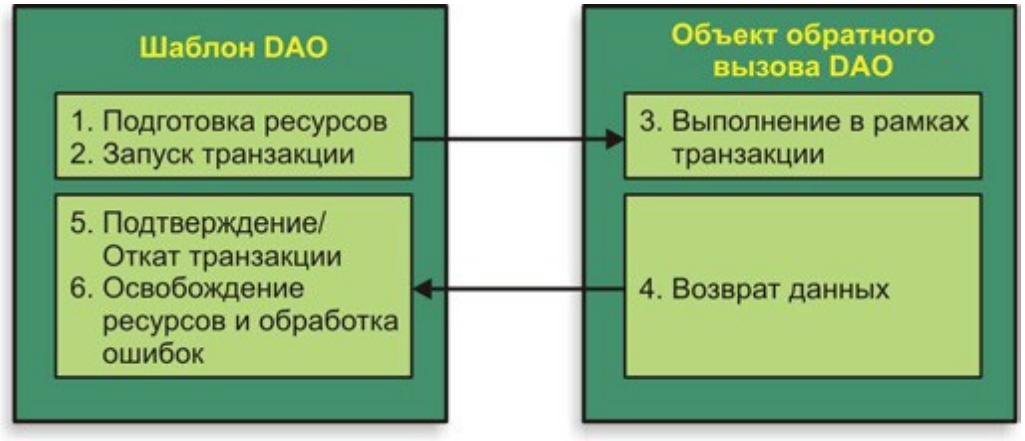

Рисунок 3 – Классы шаблонов DAO в Spring несут ответственностьмза общие функции доступа к данным. Для реализации специфических функций они могут обращаться и к пользовательским объектам обратного вызова

Как показано на рисунке 3, классы шаблонов в Spring заведуют фиксированными частями процесса доступа к данным – управление транзакциями, управление ресурсами и обработка исключений. А конкретные особенности доступа к данным, относящиеся к приложению, – создание отчетности, связывание параметров и передача результатов – обрабатываются реализацией обратного вызова. На практике такое разделение обеспечивает элегантное решение, поскольку вам достаточно побеспокоиться лишь о реализации своей части логики доступа к данным. В Spring имеется несколько шаблонов на выбор, в зависимости от выбранного типа хранилища данных. При

использовании JDBC лучшим выбором будет JdbcTemplate. А если вы предпочитаете один из фреймворков объектно-реляционного отображения, то более подходящим выбором будет HibernateTemplate или JpaTemplate.

Как будет показано далее, для использования шаблона доступа к данным достаточно просто определить его в файле с настройками как обычный компонент в контексте Spring и затем связать с объектом DAO в приложении. Или же можно воспользоваться преимуществом поддержки классов DAO в Spring для еще большего упрощения конфигурации приложения. Прямое связывание шаблонов – это хорошо, но Spring также предоставляет набор удобных базовых классов DAO, которые могут управлять шаблоном без вашего участия. Посмотрим, как действуют эти классы DAO, основанные на шаблонах.

5 Использование классов поддержки DAO (Data Access Objects)

Шаблоны доступа к данным – это только часть модуля доступа к данным в Spring. Каждый шаблон также предоставляет удобные методы, упрощающие доступ к данным, избавляя от необходимости создавать явную реализацию обратного вызова. Кроме того, поверх конструкции «шаблон/обратный вызов» Spring предоставляет классы поддержки DAO для наследования вашими собственными классами DAO. Взаимосвязи между классом шаблона, классом поддержки DAO и вашей реализацией DAO схематически изображены на рисунке 4.

Рисунок 4 – Отношения между прикладным объектом DAO,поддержкой DAO в Spring и классами шаблонов

Далее, когда будут исследованы варианты поддержки доступа к данным в Spring, будет показано, как классы поддержки DAO обеспечивают удобный доступ к поддерживаемым ими классам шаблонов. При создании прикладной реализации DAO можно создать свой класс, унаследовав в нем класс поддержки DAO, и вызвать метод получения шаблона, чтобы иметь прямой доступ к шаблону, лежащему в основе. Например, если прикладной объект DAO наследует класс поддержки JdbcDaoSupport, чтобы получить шаблон доступа JdbcTemplate, достаточно просто вызвать метод getJdbcTemplate().

Плюс, если потребуется получить доступ непосредственно к фреймворку обслуживания хранилища, каждый из классов поддержки DAO обеспечивает

доступ к любым классам, используемым для взаимодействия с базой данных. Например, класс JdbcDaoSupport содержит метод getConnection() для работы непосредственно с JDBC-соединением.

Подобно тому как фреймворк Spring предоставляет несколько реализаций классов шаблонов доступа к данным, он также обеспечивает несколько классов поддержки DAO, по одному для каждого шаблона.

Даже при том, что Spring обеспечивает поддержку лишь нескольких типов хранилищ, нам не хватит времени для изучения всех. Поэтому основное наше внимание будет сосредоточено на наиболее выгодных, вариантах и тех, которые чаще используются на практике .

Начнем с простого доступа через JDBC, так как это основной способ чтения и записи данных. Далее мы рассмотрим работу с Hibernate и JPA, двумя самыми популярными ORM-решениями на базе POJO. Но обо всем по порядку – большинство вариантов поддержки хранилищ, имеющихся в Spring, будут зависеть от источника данных. Поэтому, прежде чем приступить к созданию шаблонов и объектов DAO, необходимо настроить фреймворк Spring для работы с источником данных, чтобы обеспечить доступ объектов DAO к базе данных.

6 Источник данных JDBC

Простейшие источники данных, которые только можно настроить в Spring – это те, что определены с использованием драйвера JDBC. Spring предлагает на выбор два класса таких источников данных (оба в пакете org.springframework.jdbc.datasource).

DriverManagerDataSource – каждый раз, когда запрашивается соединение, возвращает новое соединение. В отличие от BasicDataSource в DBCP, соединения, предоставляемые DriverManagerDataSource, не объединяются в пул.

SingleConnectionDataSource – каждый раз, когда запрашивается соединение, возвращает одно и то же соединение. Хотя SingleConnectionDataSource и не является пулом в полном смысле этого слова, тем не менее его можно воспринимать как источник данных с пулом, содержащим единственное соединение.

Конфигурация при помощи Java-кода будет выглядеть следующим образом:

```
@Autowired
private Environment env;

@Bean
DataSource dataSource() {
	DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName(env.getProperty("dataSource.driverClassName"));
	dataSource.setUrl(env.getProperty("dataSource.url"));
	dataSource.setUsername(env.getProperty("dataSource.username"));
```

```
        dataSource.setPassword(env.getProperty("dataSource.password"));
        return dataSource;
    }
```

7 Использование JDBC совместно со Spring

Существует множество технологий хранения данных. Hibernate, iBATIS и JPA – лишь некоторые из них. Несмотря на немалое количество вариантов, записывать Java-объекты прямо в базу данных – это уже немного старомодный путь для заработка. Стоп, а как же люди теперь зарабатывают деньги?! А, проверенным дедовским методом – сохраняя данные с помощью старого доброго JDBC.

А почему бы и нет? JDBC не требует владения языком запросов другого фреймворка. Он основан на SQL – языке доступа к данным. Плюс при использовании JDBC можно куда более точно настроить производительность доступа к данным в сравнении с любыми другими технологиями. И JDBC позволяет пользоваться всеми преимуществами конкретных особенностей базы данных, тогда как другие фреймворки могут препятствовать этому или даже запрещать. Более того, JDBC дает возможность работать с данными на гораздо более низком уровне, чем прочие фреймворки доступа к данным, позволяя, например, манипулировать отдельными столбцами в базе данных. Такие широкие возможности доступа к данным могут пригодиться, например, в приложениях создания отчетов, где нет смысла преобразовывать данные в объекты, чтобы затем вновь извлекать эти данные из объектов, преобразуя их практически к исходному виду. Но не все так безоблачно в мире JDBC. К его мощности, гибкости, и простоте прилагаются некоторые недостатки.

Несмотря на то что JDBC API действует в тесном контакте с базой данных, ответственность за управление всем, что касается доступа к базе данных, возлагается на программиста. Сюда входят: управление ресурсами базы данных и обработка исключений. Если вам приходилось писать программы, использующие JDBC для вставки данных в базу, фрагмент, представленный в листинге ниже, не покажется вам чем-то необычным:

```
    private static final String SQL_INSERT_STUDENT =
                    "insert into student (id, first_name, last_name, patronymic,
avg_mark) values (?, ?, ?, ?, ?)";

    private DataSource dataSource;
    public void addStudent(Student student) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try
    {
        conn = dataSource.getConnection(); // Получить соединение
        stmt = conn.prepareStatement(SQL_INSERT_STUDENT); // Создать запрос
        stmt.setInt(1, student.getId()); // Связать параметры
        stmt.setString(2, student.getFirstName());
        stmt.setString(3, student.getLastName());
        stmt.setString(4,student.getPatronymic());
        stmt.setDouble(5, student.getAvgMark());
        stmt.execute(); // Выполнить запрос
```

```
} catch (SQLException e) { // Обработать исключение (как-нибудь)
      // выполнить что-нибудь... хотя... не уверен, что тут можно сделать
} finally {
      try {
            if (stmt != null) { // Освободить ресурсы
                  stmt.close();
            }
            if (conn != null) {
                  conn.close();
            }
      } catch (SQLException e) {
            // Я еще менее уверен, что тут можно сделать
      }
      }
      }
```

Святые угодники! Более 20 строк кода, чтобы вставить простой объект в базу данных. При этом сами операции JDBC выглядят проще некуда. Неужели обязательно писать так много строк, чтобы выполнить простейшую операцию? Конечно же нет. В действительности здесь лишь несколько строк реализуют вставку. Но JDBC требует корректной обработки соединений и запросов, а также возможного исключения SQLException. Что касается исключения SQLException: беда в том, что здесь совершенно не понятно, как его обрабатывать (поскольку неочевидны проблемы, вызвавшие его), но вы вынуждены дважды предусматривать его обработку! Его следует перехватить на случай, если ошибка возникла в момент вставки новой записи, и снова перехватить его при закрытии запроса и соединения. Слишком сложно, особенно если учесть, что некоторые ошибки не могут быть обработаны программно.

8 Работа с шаблонами JDBC

Модуль JDBC в Spring освобождает от необходимости управления ресурсами и обработки исключений. Он дает свободу писать только тот код, который необходим для перемещения данных в базу данных и обратно.

Как говорилось ранее, фреймворк Spring скрывает весь вспомогательный код доступа к данным за классами шаблонов. Для работы с JDBC фреймворк Spring предоставляет три класса шаблонов, на выбор:

JdbcTemplate – самый основной шаблон JDBC, этот класс предоставляет простой доступ к базе данных через JDBC и простые запросы с индексированными параметрами;

NamedParameterJdbcTemplate – этот шаблон JDBC позволяет создавать запросы с именованными параметрами вместо индексированных;

SimpleJdbcTemplate – эта версия шаблона JDBC использует новые возможности Java 5, такие как автоматическая упаковка и распаковка (autoboxing), шаблонные классы (generics) и списки аргументов переменной длины (varargs) для упрощения работы с шаблоном.

8.1 Доступ к данным с использованием JdbcTemplate

Для выполнения своей работы шаблону SimpleJdbcTemplate необходим только компонент, реализующий интерфейс DataSource. Это делает компонент типа SimpleJdbcTemplate достаточно простым в настройке, как показано ниже:

```
@Autowired
public void setDataSource(DataSource dataSource){
	this.jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Фактически компонентом типа DataSource, на который ссылается свойство dataSource, может быть любая реализация javax.sql.DataSource.

Теперь можно внедрить JdbcTemplate в наш объект DAO и использовать его для доступа к базе данных. Например, предположим, что Student DAO основан на JdbcTemplate:

```
@Component
public class StudentJdbcDao {

JdbcTemplate jdbcTemplate;

@Autowired
public void setDataSource(DataSource dataSource){
	this.jdbcTemplate = new JdbcTemplate(dataSource);
	}
}
```

Тогда свойство jdbcTemplate компонента StudentJdbcDAO связывается автоматически с помощью аннотации @Autowired.

Передав в распоряжение нашего объекта DAO шаблон JdbcTemplate, можно существенно упростить метод addStudent() из листинга выше. Новый метод addStudent(), основанный на применении шаблона jdbcTemplate, представлен в листинге ниже.

```
public int insert(Student student){
	return jdbcTemplate.update("insert into student " + "(id, first_name,
			last_name, patronymic, avg_mark) "
			+ "values (?,?,?,?,?)",
			new Object[] {student.getId(), student.getFirstName(),
			 student.getLastName(), student.getPatronymic(),
			 student.getAvgMark()
			});
	}
```

Думаю, вы согласитесь, что эта версия метода insert() выглядит значительно проще (чем addStudent()). Здесь отсутствует программный код, создающий соединение и запрос, а также отсутствует обработка исключений. Нет ничего, кроме чистого кода вставки данных.

Только потому, что вы не видите здесь шаблонного кода, это не означает, что его нет на самом деле. Он просто грамотно спрятан внутри класса шаблона. Когда вызывается метод update(), JdbcTemplate автоматически получает соединение, создает запрос и выполняет его.

Здесь также не видно, как обрабатываются исключения SQLException. Внутри класс JdbcTemplate перехватывает все исключения SQLException и преобразует обобщенное исключение SQLException в одно из более конкретных исключений доступа к данным. Так как в Spring все исключения доступа к данным являются исключениями времени выполнения, их необязательно перехватывать в методе update().

О чтении данных с использованием JdbcTemplate можете ознакомиться самостоятельно.

9 Интеграция Hibernate и Spring

JDBC – это велосипед в мире технологий хранения данных. Он отлично подходит, для чего придуман, и в некоторых ситуациях просто великолепен. Но поскольку наши приложения становятся все более сложными, попробуем сформулировать наши новые требования к хранению данных. Итак, мы должны иметь возможность отображать свойства объектов в столбцы базы данных, и было бы неплохо, если бы SQL-запросы создавались автоматически, без необходимости вручную вводить бесконечные строки из вопросительных знаков. Нам также нужны некоторые, более сложные «фишки».

Отложенная загрузка (lazy loading) – иерархии наших объектов становятся все более сложными, поэтому иногда бывает нежелательно извлекать все дерево наследования немедленно. Отложенная загрузка позволяет извлекать только те данные, которые действительно необходимы.

Полная загрузка (eager fetching) – полная противоположность отложенной загрузке. Полная загрузка позволяет извлечь все дерево объектов в одном запросе. Когда точно известно, что приложению потребуются все объекты

Каскадирование (cascading) – иногда изменения в одной таблице базы данных должны привести к изменениям в других таблицах.

Существуют фреймворки, способные предоставить подобную функциональность. Общее название для такой функциональности – объектно-реляционное отображение (Object-Relational Mapping, ORM). Применение инструмента ORM для слоя, реализующего хранение данных, может буквально спасти от необходимости писать тысячи строк лишнего кода и сэкономить массу времени на его разработку. Это позволит сместить фокус с написания чреватого ошибками SQL-кода на решение прикладных задач самого приложения.

Фреймворк Spring включает поддержку некоторых фреймворков, реализующих хранение данных, включая Hibernate, iBATIS, Java Data Objects (JDO) и Java Persistence API (JPA). Как и в случае с поддержкой JDBC, Spring

обеспечивает поддержку фреймворков ORM, предоставляя точки интеграции для фреймворков, а также некоторые дополнительные услуги, такие как:

- интегрированная поддержка декларативных транзакций;
- прозрачная обработка исключений;
- легковесные классы шаблонов с поддержкой выполнения в многопоточной среде;
- классы поддержки DAO;
- управление ресурсами.

Мы не будем рассматривать все фреймворки ORM, поддерживаемые фреймворком Spring. Это даже хорошо, потому что поддержка различных реализаций ORM в Spring похожа друг на друга. Как только вы приобретаете навык использования любого из фреймворков ORM в Spring, вы легко сможете переключиться на другой. Начнем изучение с обзора интеграции Spring с самым, пожалуй, популярным фреймворком – Hibernate .

Hibernate – это открытый фреймворк ORM, получивший широкую популярность в сообществе разработчиков. Он обеспечивает не только базовые возможности объектно-реляционного отображения, но также и все прочие «фишки», которые принято ожидать от полнофункционального инструмента ORM, такие как отложенная загрузка, полная загрузка и распределенное кеширование. В этом разделе мы сконцентрируемся на особенностях интеграции Spring с Hibernate, не углубляясь в запутанные детали использования фреймворка Hibernate.

9.1 Обзор Hibernate

В предыдущем разделе было показано, как использовать в приложении шаблоны JDBC, предоставляемые фреймворком Spring. Как оказывается, для работы с Hibernate фреймворк Spring предлагает похожий класс шаблона, абстрагирующий использование возможностей фреймворка Hibernate. Исторически для работы с фреймворком Hibernate в приложениях на основе Spring используется класс HibernateTemplate. Подобно своим сородичам из реализации поддержки JDBC, класс HibernateTemplate сам беспокоится обо всех тонкостях взаимодействий с Hibernate, перехватывая исключения, генерируемые этим фреймворком и преобразуя их в неконтролируемые исключения Spring.

Одной из областей ответственности класса HibernateTemplate является управление сеансами Hibernate. Сюда входят: открытие и закрытие сеанса, а также обеспечение уникальности сеанса для каждой транзакции. Без применения класса HibernateTemplate у вас не было бы иного выбора, как загромождать реализацию своих объектов DAO шаблонным кодом управления сеансами.

Недостатком класса HibernateTemplate является его некоторая навязчивость. При использовании класса HibernateTemplate в реализации объектов DAO (непосредственно или через класс поддержки HibernateDaoSupport) класс

реализации DAO оказывается тесно привязанным к Spring API. Для кого-то это может оказаться не очень большой проблемой, но для других такая тесная связь с фреймворком Spring может оказаться нежелательной.

Даже при том, что класс HibernateTemplate все еще остается доступным, он больше не считается лучшим способом взаимодействия с Hibernate. В версии Hibernate 3 появились контекстные сеансы (contextual sessions), посредством которых Hibernate сам осуществляет управление сеансами Session и их распределением по одному на каждую транзакцию. Теперь нет никакой необходимости использовать класс HibernateTemplate, чтобы гарантировать это поведение, что избавляет классы DAO от необходимости писать программный код, накладывающий зависимость от Spring.

Поскольку контекстные сеансы признаны более удачным способом взаимодействия с фреймворком Hibernate, мы сконцентрируемся на них и не будем тратить время на знакомство с классом HibernateTemplate.

9.2 Объявление фабрики сеансов Hibernate

Основным интерфейсом для взаимодействий с Hibernate является интерфейс org.hibernate.Session. Интерфейс Session обеспечивает базовую функциональность доступа к данным, позволяя сохранять, обновлять, удалять и загружать объекты в/из базы данных. Именно через интерфейс Session прикладные объекты DAO будут выполнять все операции с хранилищем данных. Стандартный способ получить ссылку на объект Session – обратиться к реализации интерфейса SessionFactory в Hibernate. Среди всего прочего интерфейс SessionFactory отвечает за открытие, закрытие и управление сеансами Hibernate.

Получить доступ к SessionFactory в приложениях на основе Spring можно через компоненты фабрики сеансов. Эти компоненты реализуют интерфейс FactoryBean фреймворка Spring, который воспроизводит объекты класса SessionFactory при внедрении в свойства типа SessionFactory. Это позволяет настраивать фабрику сеансов Hibernate наряду с другими компонентами в контексте приложения Spring. Что касается настройки компонента фабрики сеансов Hibernate, мы воспользуемся настройкой при помощи Java-кода.

```
	@Bean
	LocalSessionFactoryBean sessionFactory() {
		LocalSessionFactoryBean localSessionFactoryBean = new
LocalSessionFactoryBean();
		localSessionFactoryBean.setDataSource(dataSource());
		localSessionFactoryBean.setPackagesToScan("entity");
		Properties properties = new Properties();
		properties.setProperty("hibernate.dialect",
env.getProperty("hibernate.dialect"));
		localSessionFactoryBean.setHibernateProperties(properties);
		return localSessionFactoryBean;
	}
```

Конфигурация компонента LocalSessionFactoryBean включает настройку трех свойств. В свойство dataSource внедряется ссылка на компонент DataSource. Воспользуемся свойством PackagesToScan , чтобы сообщить фреймворку Spring один или более пакетов, где следует искать классы объектов предметной области, аннотированные для сохранения с помощью Hibernate. В их число входят классы, отмеченные аннотациями JPA @Entity или @MappedSuperclass , и собственной аннотацией Hibernate – @Entity. Наконец, свойство hibernateProperties позволяет определить мелкие детали поведения Hibernate. В данном случае фреймворку Hibernate сообщается, что он будет взаимодействовать с базой данных PostgreSQL и для создания SQL-запросов должен использовать диалект PostgresDialect.

9.3 Создание классов для работы с Hibernate, независимых от Spring

Как отмечалось выше, без контекстных сеансов гарантировать создание единственного сеанса для каждой транзакции можно с помощью шаблона для работы с Hibernate в Spring. Но теперь, когда управление сеансами берет на себя сам фреймворк Hibernate, нет необходимости использовать класс шаблона. Это означает возможность непосредственного связывания сеанса Hibernate с прикладными классами DAO.

```
	@Repository
	@Transactional
	public class HibernateStudentDao {

	private SessionFactory sessionFactory;

	@Autowired
	public HibernateStudentDao(SessionFactory sessionFactory){
		this.sessionFactory = sessionFactory;
	}

	private Session currentSession() {                          // Извлекает текущий
		return sessionFactory.getCurrentSession();  // сеанс из фабрики
	}                                                                                    //
SessionFactory
	public void addStudent(Student student) {
		currentSession().save(student);                 // Использует текущий сеанс
	}

	public Student getStudentById(long id) {              // Использует текущий сеанс
		return (Student) currentSession().get(Student.class, id);
	}
	public void saveStudent(Student student) {
		currentSession().update(student);                      // Использует текущий
сеанс
	}

	}
```

Относительно фрагмента в листинге выше следует сделать несколько замечаний. Во-первых, обратите внимание на использование аннотации Spring @Autowired , вынуждающей Spring автоматически внедрить SessionFactory в

свойство sessionFactory объекта HibernateStudentDao. Этот объект SessionFactory используется затем в методе currentSession(), чтобы получить ссылку на сеанс для текущей транзакции.

Отметьте также, что класс отмечен аннотацией @Repository. Это позволяет достичь двух целей. Во-первых, @Repository – еще одна стереотипная аннотация в Spring, которая, кроме всего прочего, обнаруживается элементом конфигурации Spring @ComponentScan. Это означает, что при наличии настроенного элемента @ComponentScan, не требуется явно объявлять компонент HibernateStudentDao.

Для справки покажем конфигурацию с помощью Java-класса, которая оказывается достаточно сложной:

```
@Configuration
@EnableTransactionManagement
@ComponentScan("ru.jdbctest")
@PropertySource("classpath:application.properties")
public class SpringConfig {

@Autowired
private Environment env;

@Bean
DataSource dataSource() {
	DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName(env.getProperty("dataSource.driverClassName"));
	dataSource.setUrl(env.getProperty("dataSource.url"));
	dataSource.setUsername(env.getProperty("dataSource.username"));
	dataSource.setPassword(env.getProperty("dataSource.password"));
	return dataSource;
}

@Bean
public StudentJdbcDao studentJdbcDao() {
	return new StudentJdbcDao();
}

@Bean
LocalSessionFactoryBean sessionFactory() {
	LocalSessionFactoryBean localSessionFactoryBean = new
LocalSessionFactoryBean();
	localSessionFactoryBean.setDataSource(dataSource());
	localSessionFactoryBean.setPackagesToScan("entity");
	Properties properties = new Properties();
	properties.setProperty("hibernate.dialect",
env.getProperty("hibernate.dialect"));
	localSessionFactoryBean.setHibernateProperties(properties);
	return localSessionFactoryBean;
}

 @Bean
 @Autowired
 public HibernateTransactionManager transactionManager(
     SessionFactory sessionFactory) {
      HibernateTransactionManager txManager
```

```
        = new HibernateTransactionManager();
        txManager.setSessionFactory(sessionFactory);

        return txManager;
    }

}
```

Оставим её пока без комментариев, потому что нас больше интересует JPA, нежели Hibernate. Оставим его как переходный этап к JPA.

10 Spring и Java Persistence API

С самого начала спецификация EJB включала понятие компонентов-сущностей (entity beans). В терминах EJB *компонент-сущность* представляет собой тип EJB, описывающий прикладные объекты, хранимые в реляционной базе данных. Компоненты-сущности претерпели несколько этапов развития на протяжении последних лет, включая появление компонентов-сущностей, которые *сами управляют своим сохранением* (bean-managed persistence, BMP), и ком понентов-сущностей, *сохранением которых управляет контейнер* (container-managed persistence, CMP).

Компоненты-сущности пережили взлет и падение популярности EJB. В последние годы разработчики все чаще отказываются от тяжеловесной платформы EJB, отдавая предпочтение реализациям на основе простых POJO. Это явилось серьезной проблемой для организации Java Community Process, формировавшей новую спецификацию EJB на основе POJO. В итоге появилась спецификация JSR-220, также известная как *EJB 3*.

На руинах спецификации компонентов-сущностей EJB 2 сформировался новый стандарт хранения данных в Java – Java Persistence API (JPA). JPA представляет собой механизм хранения данных на основе POJO, заимствующий идеи Hibernate и *Java Data Objects* (JDO) и дополняющий их аннотациями Java 5.

С выходом версии Spring 2.0 состоялась премьера интеграции Spring с JPA. По иронии, многие обвиняют (или приписывают) Spring в упадке EJB. Но теперь, после появления поддержки JPA, многие разработчики рекомендуют использовать в приложениях на основе Spring для работы с базами данных именно JPA. В действительности многие считают, что связка Spring-JPA – это мечта для разработки на основе POJO.

Первый шаг к использованию JPA в Spring заключается в на стройке компонента фабрики диспетчера сущностей в контексте приложения.

10.1 Настройка фабрики диспетчера сущностей

Если говорить в двух словах, приложения на основе JPA используют реализацию EntityManagerFactory для получения экземпляра EntityManager. Спецификация JPA определяет два вида диспетчеров сущностей (entity managers).

Управляемые приложением – эти диспетчеры сущностей создаются, когда приложение непосредственно запрашивает у фабрики диспетчеров сущностей. За создание и уничтожение диспетчеров сущностей, управляемых приложением, а также за их использование в транзакциях отвечает само приложение. Этот тип диспетчеров сущностей в большей степени подходит для использования в автономных приложениях, выполняющихся вне контейнера Java EE.

Управляемые контейнером – эти диспетчеры сущностей создаются и управляются контейнером Java EE. Приложение никак не взаимодействует с фабрикой диспетчеров сущностей. Вместо этого диспетчеры сущностей приобретаются приложением посредством внедрения или из JNDI. За настройку фабрик диспетчеров сущностей отвечает контейнер.

Оба типа диспетчеров сущностей реализуют один и тот же интерфейс EntityManager . Но основное отличие заключается не в EntityManager как таковом, а скорее в том, как диспетчер создается и управляется. Диспетчеры сущностей, управляемые приложением, создаются объектом EntityManagerFactory, полученным вызовом метода createEntityManagerFactory() объекта PersistenceProvider, а диспетчеры сущностей, управляемые контейнером, – вызовом метода createContainerEntityManagerFactory().

Так что же все это значит для разработчиков приложений на основе Spring, желающих использовать JPA? В сущности, не так много. Независимо от способа создания EntityManagerFactory, ответственность за управление диспетчерами сущностей будет нести Spring. При использовании диспетчеров сущностей, управляемых приложением, Spring будет играть роль приложения и обеспечит прозрачное управление диспетчерами от имени приложения. В случае использования диспетчеров сущностей, управляемых контейнером, Spring будет играть роль контейнера.

Каждая фабрика диспетчеров сущностей создается соответствующим компонентом Spring:

LocalEntityManagerFactoryBean создает фабрики EntityManager Factory, управляемые приложением;

LocalContainerEntityManagerFactoryBean создает фабрики EntityManagerFactory, управляемые контейнером.

Важно отметить, что выбор между фабриками, управляемыми приложением и контейнером, полностью прозрачен для приложений на основе Spring. Класс JpaTemplate фреймворка Spring полностью скрывает все сложности использования любой из форм EntityManagerFactory, позволяя сосредоточиться в прикладном коде на его основной цели – реализации доступа к данным.

Единственное существенное отличие между ними состоит в том, как они настраиваются в контексте приложения Spring. Рассмотрим для начала, как настроить в Spring компонент LocalEntityManagerFactoryBean фабрики диспетчеров сущностей, управляемых приложением. А затем проделаем то же с

фабрикой диспетчеров сущностей, управляемых контейнером LocalContainerEntityManagerFactoryBean.

10.2 Настройка механизма JPA, управляемого контейнером

При выполнении в пределах контейнера EntityManagerFactory может быть создан на основе информации, предоставляемой контейнером, в данном случае – контейнером Spring. Например, следующее объявление элемента @Bean демонстрирует, как настроить в Spring механизм JPA, управляемый контейнером, используя LocalContainerEntityManagerFactoryBean.

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
@ComponentScan("ru.jdbctest")
@PropertySource("classpath:application.properties")
public class SpringConfig {

@Autowired
private Environment env;

@Bean
DataSource dataSource() {
	DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName(env.getProperty("dataSource.driverClassName"));
	dataSource.setUrl(env.getProperty("dataSource.url"));
	dataSource.setUsername(env.getProperty("dataSource.username"));
	dataSource.setPassword(env.getProperty("dataSource.password"));
	return dataSource;
}

@Bean
public EntityManagerFactory entityManagerFactory() {
	HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("ru.jdbctest");
    factory.setDataSource(dataSource());
    factory.afterPropertiesSet();

    return factory.getObject();
}

 @Bean
  public PlatformTransactionManager transactionManager() {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory());
    return txManager;
  }

}
```

Здесь в свойство dataSource внедряется ссылка на источник данных, настраиваемый в конфигурации Spring. В данном случае можно использовать любую реализацию javax.sql.DataSource, например реализацию, настроенную в разделе 6. Свойство jpaVendorAdapter может быть использовано для настройки особенностей конкретной реализации JPA. В состав Spring входят несколько классов адаптеров JPA:

EclipseLinkJpaVendorAdapter;

HibernateJpaVendorAdapter;

OpenJpaVendorAdapter;

TopLinkJpaVendorAdapter.

В данном случае использована JPA-реализация для Hibernate, поэтому настройки выполняются с применением HibernateJpaVendorAdapter.

После чего создается *диспетчер транзакций.*

Тема управления транзакциями достаточно объемная и сложная, она не рассматривается в рамках данного курса, дополнительно об этом слушатель может прочитать в литературе. Используем код создания диспетчера транзакций `PlatformTransactionManager` как есть.

10.3 Репозиторий Spring Data

Основное понятие в Spring Data — это репозиторий. Это несколько интерфейсов которые используют JPA Entity для взаимодействия с ней. Так например интерфейс

public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID>

обеспечивает основные операции по поиску, сохранения, удалению данных (CRUD операции) и другие операции.

Есть и другие абстракции, например PagingAndSortingRepository.

Если того перечня что предоставляет интерфейс не достаточно для взаимодействия с сущностью, то можно прямо расширить базовый интерфейс для своей сущности, дополнить его своими методами запросов и выполнять операции.

Наша сущность теперь выглядит следующим образом:

```
@Entity
public class Student {

@Id
private int id;

@Column(name = "first_name")
private String firstName;

@Column(name = "last_name")
private String lastName;

private String patronymic;
```

```
	@Column(name = "avg_mark")
		private double avgMark;
```

Наследуемся от одного из интерфейсов Spring Data, например от JpaRepository (подробнее https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html)

```
	@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {
	@Query("select s from Student s where s.id = ?1")
	Student findById(int id);
	}
```

Здесь мы явным образом указали запрос через аннотацию @Query. В запрос мы можем передать параметры в том же порядке, как они представлены в методе. Т.е. в запрос вместо ?1 будет передан первый параметр метода findById.

Теперь можем получить интерфейсную переменную из контекста

```
	StudentRepository rep = context.getBean("studentRepository", StudentRepository.class);
```

И получить экземпляр Student из БД:

```
	Student s = rep.findById(1);
```

Но подобные операции, как взятие записи по id уже реализованы интерфейсом CrudRepository, реализованные методы вы можете посмотреть здесь:

https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

Тогда переписываем репозиторий следующим образом:

```
	@Repository
	public interface StudentRepository extends CrudRepository<Student, Integer> {

	}
```

Получаем интерфейсную переменную из контекста и вызываем тот же самый метод, но который уже реализован внутри интерфейса за нас:

```
	StudentRepository rep = context.getBean("studentRepository", StudentRepository.class);
	Optional<Student> s = rep.findById(1);
```

10.4 Query methods

Запросы к сущности можно строить прямо из имени метода. Для этого используется механизм префиксов find…By, read…By, query…By, count…By, и get…By, далее от префикса метода начинает разбор остальной части. Вводное

предложение может содержать дополнительные выражения, например, Distinct. Далее первый By действует как разделитель, чтобы указать начало фактических критериев. Можно определить условия для свойств сущностей и объединить их с помощью And и Or.

Например найти всех студентов у которых средняя оценка выше определенного значения:

```
List<Student> findByAvgMarkGreaterThan(double avgMark);
```

Достаточно только определить подобным образом метод, без имплементации и Spring подготовит запрос к сущности.

В документации (https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html) определен весь перечень, и правила написания метода. В качестве результата могут быть сущность T, Optional, List, Stream.