**Описание**

**Поздравляем с переходом в новый модуль PreProject!**

**В этом модуле ты познакомишься с такими темами:**

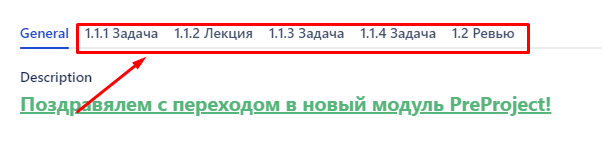
1. JDBC и Hibernate
2. Spring
3. Spring Boot, Spring Security, REST
4. Microservices

**Особенности этого модуля:**

1. Каждая тема сдаётся по отдельности и целиком, тоесть, необходимо решить все задачи и с ними идти к ментору на ревью.
2. **В этом модуле каждый студент сдаёт тему ментору индивидуально, а не группой, как это было на Core0 и Core1!!!**
3. Необходимо завести аккаунт на GitHub, каждая задача должна быть сделана в отдельном репозитории. Ссылки на репозитории необходимо скинуть ментору, которому будете сдавать.

**Это тема JDBC и Hibernate**

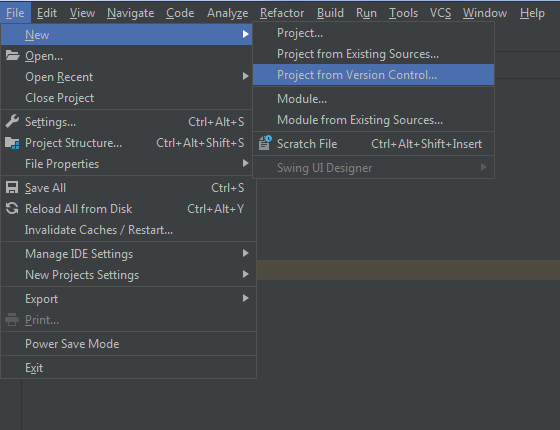
Задачи находятся во вкладках выше👇

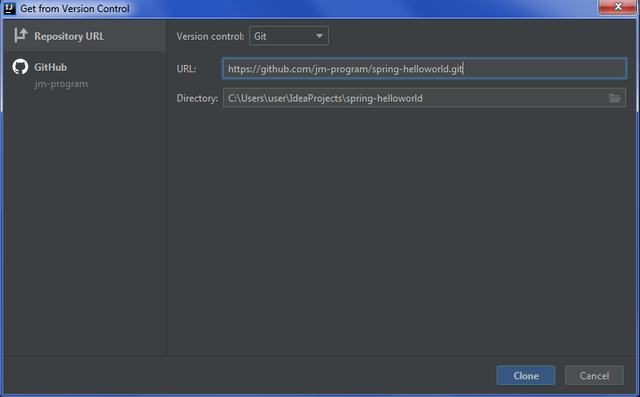


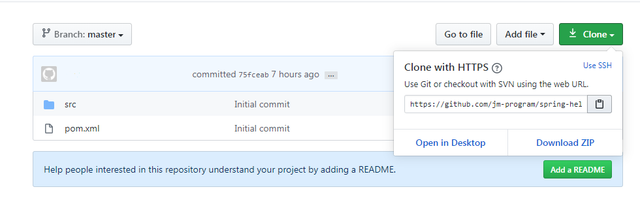
**PP 1.1.1 Задача**

Практическая задача  
**Задание:**

1. Пройти 2 первых блока в приложении [https://learngitbranching.js.org](https://learngitbranching.js.org/) (Введение и Едем дальше).
2. Создать Git репозиторий на GitHub и залить туда пустой проект.
3. Склонируйте себе свой проект, используя IDE. Для этого необходимо выполнить следующие действия:
   1. На верхней панели найдите вкладку **file -> new -> Project from Version Control...**
   2. В открывшемся окне, необходимо вставить в поле для ввода ссылку на репозиторий и нажать кнопку clone. Ссылку на репозиторий можно получить, нажав на кнопку Clone в вашем репозитории на [https://github.com](https://github.com/)







**PP 1.1.2 Лекция**

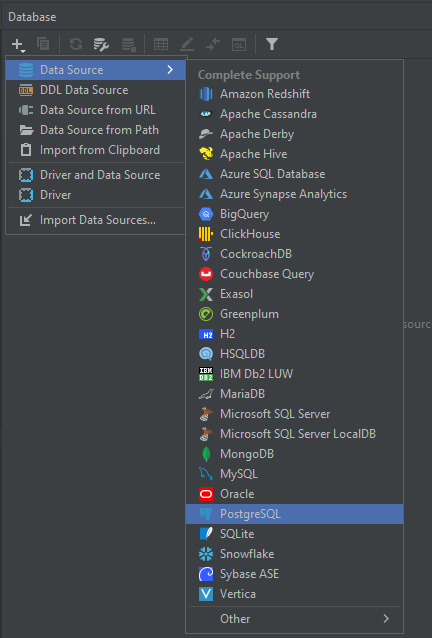
Посмотреть видео <https://www.youtube.com/watch?v=a0umeAEdN5o>

**PP 1.1.3 Задача**

**В этом задании мы познакомимся с такими технологиями**, как:

1. Maven  - [Maven – Welcome to Apache Maven](https://maven.apache.org/)
2. JDBC - <https://o7planning.org/ru/10167/java-jdbc-tutorial>

**Для выполнения задания необходимо:**

* Клонировать/скачать заготовку по [ссылке](https://github.com/VanderDT/Task-1)
* Установить на компьютер pgAdmin и PostgresSQL. Протестировать возможность соединения с базой через идею с помощью встроенной утилиты, как показано на рисунке:
* 

**Описание задачи:**

Необходимо ознакомиться с заготовкой и доработать приложение, которое взаимодействует с базой оперируя пользователем ( класс User ) и проверить свои методы заранее написанными JUnit тестами. По итогу все тесты должны быть пройдены. Разрешается посмотреть реализацию тестов.

Для запуска теста необходимо найти класс в папке test ( показано в предыдущей лекции ) и при нажатии на него правой кнопкой мыши запустить, выбрав Run "Имя класса"

Класс UserHibernateDaoImpl в рамках этой задачи не затрагивается (остаётся пустой)

**User** представляет из себя сущность с полями:

* Long id
* String name
* String lastName
* Byte age

Архитектура приложения создана с опорой на паттерн проектирования MVC (частично, у нас не WEB приложение)

Ознакомиться с паттерном можно [здесь](https://pro-java.ru/patterny-proektirovaniya-java/chto-takoe-pattern-proektirovaniya-mvc-v-java/)

**Требования к классам приложения:**

1. Классы dao/service должны реализовывать соответствующие интерфейсы
2. Класс dao должен иметь конструктор пустой/по умолчанию
3. Все поля должны быть private
4. service переиспользует методы dao
5. Обработка всех исключений, связанных с работой с базой данных должна находиться в dao
6. Класс Util должен содержать логику настройки соединения с базой данных

**Необходимые операции:**

1. Создание таблицы для User(ов) – не должно приводить к исключению, если такая таблица уже существует
2. Удаление таблицы User(ов) – не должно приводить к исключению, если таблицы не существует
3. Очистка содержания таблицы
4. Добавление User в таблицу
5. Удаление User из таблицы ( по id )
6. Получение всех User(ов) из таблицы

**Алгоритм работы приложения**:

*В методе* *main класса Main должны происходить следующие операции:*

1. Создание таблицы User(ов)
2. Добавление 4 User(ов) в таблицу с данными на свой выбор. После каждого добавления должен быть вывод в консоль ( User с именем – name добавлен в базу данных )
3. Получение всех User из базы и вывод в консоль ( должен быть переопределен toString в классе User)
4. Очистка таблицы User(ов)
5. Удаление таблицы

**PP 1.1.4 Задача**

В этом задании мы познакомимся с фреймворком Hibernate и научимся взаимодействовать с помощью него с базой данных.

**Для выполнения задания необходимо:**

1. Ознакомиться с Hibernate. Подробнее узнать о Hibernate можно [Здесь](https://hibernate.org/orm/)
2. Готовая прошлая задача ( работа будет происходить в этом же проекте )

**Описание задачи:**

В прошлой задаче мы познакомились с Maven и JDBC, доработали приложение, взаимодействующее с базой данных.

На этот раз обратим внимание на класс UserHibernateDaoImpl, который реализует тот же интерефейс, что и UserDaoJdbcImpl.

В рамках этой задачи необходимо реализовать взаимодействие с базой данных с помощью Hibernate и дописать методы в классе UserHibernateDaoImpl, проверить свои методы заранее написанными JUnit тестами из заготовки.

**Требования к классам приложения:**

1. UserHibernateDaoImpl должен реализовывать интерефейс UserDao
2. В класс Util должна быть добавлена конфигурация для Hibernate ( рядом с JDBC), без использования xml.
3. Service на этот раз использует реализацию dao через Hibernate
4. Методы создания и удаления таблицы пользователей в классе UserHibernateDaoImpl должны быть реализованы с помощью SQL.

  Алгоритм приложения и операции не меняются в сравнении с предыдущим заданием.

**PP 1.2 Ревью**

1. **Что такое Maven? Для чего он нужен? Как добавлять в проект библиотеки без него?**

Maven - инструмент для сборки проекта и для управления зависимости в приложениях java

Нужен для автомат загружения зависимости+сборка проще

Сборка- это формирование приложения в jar\war файл один и потом можем передать его и запускать где-то в др месте

Предшественником Maven’а был **Ant**, а наследником является **Gradle**.

**Maven** - это инструмент для управления проектами в языке программирования Java. Он предоставляет *средства для сборки проектов, управления зависимостями, а также автоматизации процесса сборки,* тестирования и развертывания приложений.

Основные задачи Maven:

1. **Управление зависимостями**: Maven позволяет автоматически загружать библиотеки и зависимости из центрального репозитория и включать их в проект.
2. **Сборка проекта**: Maven обеспечивает стандартизированный способ сборки проекта, определяя его структуру и управляя процессом сборки.
3. **Управление жизненным циклом проекта**: Maven предоставляет набор стандартных фаз сборки (например, компиляция, упаковка, тестирование, развертывание), которые можно настраивать и расширять в соответствии с требованиями проекта.
4. **Управление конфигурациями проекта**: Maven позволяет хранить конфигурацию проекта в файлах POM (Project Object Model) и обеспечивает ее централизованное управление.

**Способы добавления библиотек** в проект:

1. **Gradle**: Это еще один инструмент управления зависимостями, аналогичный Maven, который позволяет добавлять библиотеки и управлять зависимостями в проекте. Gradle имеет более гибкую систему конфигурации и более простой синтаксис для определения зависимостей.
2. **Вручную**: Вы можете скачать JAR-файлы библиотеки напрямую с веб-сайта разработчика и добавить их в каталог вашего проекта. Затем вам нужно добавить эти JAR-файлы в ваш проект в качестве внешних библиотек. Этот метод менее удобен и более подвержен ошибкам, но иногда может быть полезным для добавления небольших библиотек или временных исправлений.
3. **Системы управления пакетами операционной системы**: Некоторые библиотеки могут быть установлены через систему управления пакетами вашей операционной системы (например, apt в Ubuntu). Это может быть удобным способом добавления библиотек, если они доступны в репозитории вашей операционной системы.

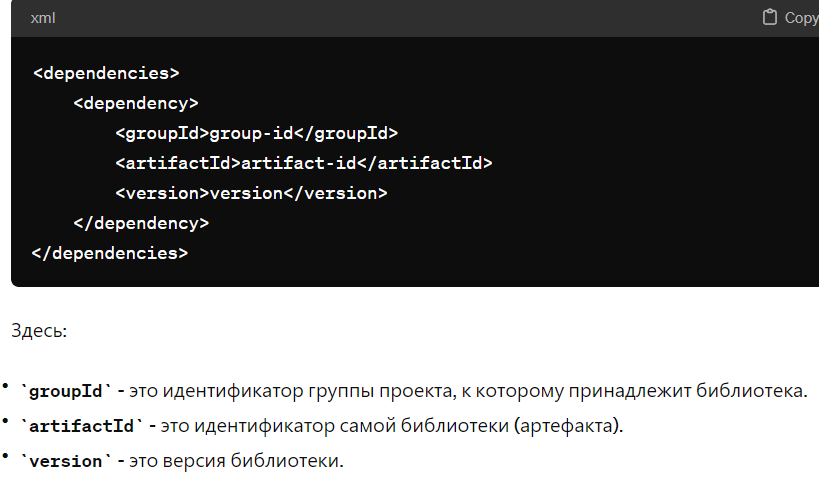
Помимо Maven и Gradle, существуют и другие **инструменты для сборки проектов** в различных языках программирования и средах разработки. Некоторые из них включают:

1. **Ant**: Это инструмент сборки, который изначально был очень популярен в мире Java. Ant использует XML-файлы для определения задач сборки проекта. В настоящее время Maven и Gradle часто предпочтительнее Ant, но иногда он все еще используется, особенно в старых проектах.
2. **Gradle**: Это еще один инструмент управления зависимостями, аналогичный Maven, который позволяет добавлять библиотеки и управлять зависимостями в проекте. Gradle имеет более гибкую систему конфигурации и более простой синтаксис для определения зависимостей.
3. **Как добавить dependency(зависимости) в Maven? Для чего они нужны? Откуда они скачиваются?**

Чтобы добавить зависимости в проект Maven, вы должны отредактировать файл **pom.xml** (Project Object Model), который является основным файлом конфигурации Maven проекта. В **pom.xml** вы определяете зависимости, которые ваш проект использует, и Maven автоматически загружает их из удаленных репозиториев.

Для добавления зависимостей в **pom.xml** выполните следующие шаги:

1. Откройте файл **pom.xml** вашего проекта в текстовом редакторе или IDE.
2. В разделе **<dependencies>** добавьте новый элемент **<dependency>** для каждой зависимости, которую вы хотите добавить. Пример:



1. Сохраните файл **pom.xml**.

После того как вы добавите зависимости в **pom.xml**, Maven автоматически загрузит их из удаленных репозиториев. Maven по умолчанию **использует центральный репозиторий Maven (Maven Central Repository)**, где хранятся тысячи библиотек и зависимостей для Java проектов. Если библиотека не найдена в центральном репозитории, Maven может также загружать зависимости из других удаленных репозиториев, которые могут быть настроены в вашем **pom.xml** или в настройках Maven.

**Причин, по которым зависимости важны:**

1. **Использование внешних библиотек и компонентов**: Многие проекты требуют использования сторонних библиотек, фреймворков или инструментов для решения различных задач. Зависимости позволяют вам указать, какие библиотеки и версии вы хотите использовать в вашем проекте, и автоматически загрузить их при сборке проекта.
2. **Управление версиями**: Зависимости позволяют вам явно указать версии библиотек, которые используются в вашем проекте. Это позволяет избежать проблем с несовместимостью версий и обеспечивает стабильность и надежность вашего приложения.
3. **Упрощение сборки проекта**: Использование зависимостей позволяет автоматизировать процесс сборки проекта. Вместо того чтобы вручную скачивать, добавлять и управлять библиотеками, Maven (или другой инструмент управления зависимостями) делает все это за вас, что экономит время и упрощает разработку.
4. **Совместимость и переносимость**: Использование стандартных инструментов управления зависимостями, таких как Maven или Gradle, обеспечивает совместимость и переносимость вашего проекта. Зависимости указываются в универсальном формате, что делает ваш проект доступным для других разработчиков и обеспечивает совместимость с различными средами разработки и инструментами сборки.
5. **Основные фазы проекта под управлением Maven?**

Проекты под управлением Maven проходят через несколько основных фаз, которые определяют жизненный цикл проекта. В Maven каждая фаза представляет собой набор задач, выполняемых в определенном порядке. Вот основные фазы проекта в Maven:

1. **validate**: В этой фазе Maven проверяет, что проект корректен и все необходимые информационные файлы доступны и правильно настроены. Например, Maven проверяет наличие файла **pom.xml** и его синтаксическую правильность.
2. **compile**: В этой фазе исходный код вашего проекта компилируется в байт-код Java или другой целевой формат, в зависимости от языка программирования.
3. **test**: В этой фазе запускаются тесты JUnit или другие модульные тесты, написанные для вашего проекта. Maven выполняет эти тесты для проверки корректности кода.
4. **package**: В этой фазе Maven упаковывает скомпилированный код, ресурсы и другие файлы проекта в один или несколько артефактов (например, JAR, WAR, или EAR файлы).
5. **verify**: В этой фазе Maven проводит дополнительные проверки качества кода и ресурсов, а также запускает интеграционные тесты для проверки работоспособности приложения в целом.
6. **install**: В этой фазе артефакты вашего проекта (например, JAR файлы) копируются в локальный репозиторий Maven на вашем компьютере. Это позволяет использовать эти артефакты как зависимости в других проектах на вашем компьютере.
7. **deploy**: В этой фазе артефакты вашего проекта публикуются в удаленный репозиторий Maven, что делает их доступными для других разработчиков и проектов.
8. **Что такое JDBC? Какие классы/интерфейсы относятся к JDBC?**

**JDBC - Java DataBase Connectivity —** API для работы с реляционными (зависимыми) БД. Платформенно независимый промышленный стандарт взаимодействия Javaприложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE. Предоставляет методы для получения и обновления данных. Не зависит от конкретного типа базы. Библиотека, которая входит в стандартную библиотеу, содержит: набор классов и интерфейсов для работы с БД (для нас разработчиков api) + интерфейсы баз данных**. JDBC реализует механизмы работы подключений к базе данных, создания запросов и обработки результатов.**

**JDBC (Java Database Connectivity)** - это API (Application Programming Interface) для работы с базами данных из приложений, написанных на языке программирования Java. JDBC предоставляет набор классов и методов, которые позволяют Java-приложениям устанавливать соединение с базой данных, отправлять SQL-запросы к базе данных, получать результаты запросов и управлять транзакциями.

Основные компоненты JDBC:

1. **Драйверы JDBC**:
   * Драйвер JDBC - это программное обеспечение, которое позволяет Java-приложению взаимодействовать с конкретной базой данных. Существует четыре типа драйверов JDBC:
     + JDBC-ODBC мост (JDBC-ODBC Bridge): позволяет Java-приложению взаимодействовать с базой данных через ODBC (Open Database Connectivity).
     + Драйвер типа 1 (Type 1 Driver): использует мост JDBC-ODBC для взаимодействия с базой данных.
     + Драйвер типа 2 (Type 2 Driver): напрямую взаимодействует с API СУБД на уровне клиента.
     + Драйвер типа 3 (Type 3 Driver): взаимодействует с СУБД по сети через промежуточный сервер.
     + Драйвер типа 4 (Type 4 Driver): напрямую взаимодействует с СУБД по протоколу базы данных.
2. **Интерфейсы и классы JDBC**:
   * JDBC предоставляет ряд интерфейсов и классов для работы с базой данных:
     + **Connection**: представляет соединение с базой данных и позволяет выполнять SQL-запросы.
     + **Statement**: используется для выполнения SQL-запросов без параметров.
     + **PreparedStatement**: предварительно скомпилированный SQL-запрос с параметрами.
     + **CallableStatement**: используется для вызова хранимых процедур в базе данных.
     + **ResultSet**: представляет результат SQL-запроса в виде набора строк и столбцов.
3. **URL-адреса базы данных**:
   * Для установки соединения с базой данных JDBC использует URL-адреса базы данных. Формат URL-адреса зависит от конкретного драйвера и СУБД, с которой вы работаете.
4. **Установка соединения и выполнение запросов**:
   * Для установки соединения с базой данных используется метод **DriverManager.getConnection()**. После получения объекта **Connection** можно использовать объекты **Statement**, **PreparedStatement** или **CallableStatement** для выполнения SQL-запросов.
   * После выполнения запроса полученные данные могут быть обработаны с помощью объекта **ResultSet**.
5. **Закрытие ресурсов**:
   * Важно закрывать все ресурсы JDBC (Connection, Statement, ResultSet) после их использования для освобождения ресурсов и предотвращения утечек памяти. Это можно сделать с помощью метода **close()**.

JDBC (Java Database Connectivity) предоставляет ряд классов и интерфейсов для работы с базами данных из приложений Java. Вот основные из них(2 класса + 4 интерфейса):

1. **DriverManager**: Этот **класс** управляет регистрацией драйверов JDBC и предоставляет методы для установки соединения с базой данных.
2. **Connection**: **Интерфейс**, представляющий соединение с базой данных. **Connection** используется для управления соединениями, создания объектов **Statement** и **PreparedStatement** и выполнения транзакций.
3. **Statement**: **Интерфей**с, используемый для выполнения SQL-запросов без параметров. Объекты **Statement** создаются через объект **Connection**.
4. **PreparedStatement**: Подинтерфейс **Statement**, который представляет предварительно скомпилированный SQL-запрос с параметрами.
5. **CallableStatement**: Подинтерфейс **PreparedStatement**, используемый для вызова хранимых процедур в базе данных.
6. **ResultSet**: Интерфейс, представляющий результаты выполнения SQL-запроса в виде набора строк и столбцов.
7. **DatabaseMetaData**: **Интерфейс**, предоставляющий метаданные о базе данных, такие как список таблиц, их структура и свойства.
8. **ResultSetMetaData**: **Интерфейс**, предоставляющий метаданные о результирующем наборе, такие как список столбцов, их типы данных и другие свойства.
9. **SQLException**: Это **класс** исключения, который представляет ошибки, возникающие при взаимодействии с базой данных через JDBC.

Эти классы и интерфейсы предоставляют мощные средства для взаимодействия с базами данных из приложений Java, что делает JDBC важным компонентом при разработке Java-приложений, особенно тех, которые работают с данными.

1. **Для чего нужен DriverManager?**

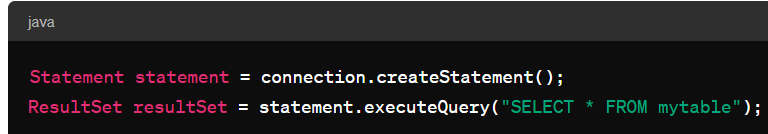
**DriverManager** в JDBC - это класс, который обеспечивает регистрацию драйверов JDBC и управление соединениями с базами данных. Его основные функции включают:

1. **Регистрация драйверов JDBC**: **DriverManager** предоставляет метод **registerDriver()**, который используется для регистрации драйверов JDBC. Этот метод позволяет Java-приложению сообщить **DriverManager**, какие драйверы JDBC доступны для использования.
2. **Установка соединения с базой данных**: **DriverManager** предоставляет метод **getConnection()**, который используется для установки соединения с базой данных. При вызове этого метода **DriverManager** определяет подходящий зарегистрированный драйвер JDBC и вызывает метод **connect()** этого драйвера для установки соединения с базой данных.
3. **Управление пулом соединений**: В некоторых случаях **DriverManager** может управлять пулом соединений, предоставляя возможность повторного использования соединений с базой данных для улучшения производительности и снижения нагрузки на базу данных.
4. **Что такое Statement, PreparedStatement, CallableStatement?**

**Statement**, **PreparedStatement** и **CallableStatement** - это три основных интерфейса в JDBC для выполнения SQL-запросов к базе данных. Каждый из них предназначен для выполнения определенного типа запросов и имеет свои особенности.

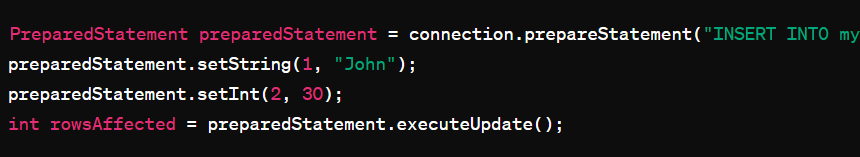
**Statement**:

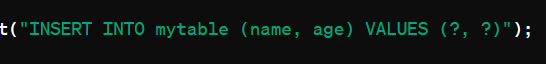
* **Statement** - это интерфейс, который используется для выполнения простых SQL-запросов без параметров. Он подходит для выполнения статических SQL-запросов, которые не требуют передачи параметров. Пример использования **Statement**:



**PreparedStatement**:

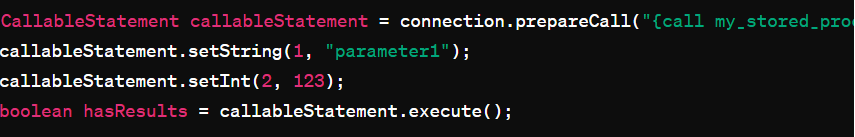
* **PreparedStatement** - это подинтерфейс **Statement**, который представляет предварительно скомпилированный SQL-запрос с параметрами. Он предоставляет улучшенную производительность и безопасность за счет предварительной компиляции запроса. Преимуществом **PreparedStatement** является возможность использования параметров, что позволяет избежать SQL-инъекций и повторно использовать запросы с разными значениями параметров. Пример использования **PreparedStatement**:





**CallableStatement**:

* **CallableStatement** - это подинтерфейс **PreparedStatement**, который используется для вызова хранимых процедур в базе данных. Он позволяет передавать параметры в хранимую процедуру и получать результаты ее выполнения. Пример использования **CallableStatement**:





1. **Что такое sql-injection?**

**SQL-инъекция** (SQL injection) - это **тип атаки** на систему управления базами данных (**СУБД**), при которой злоумышленник **внедряет вредоносный SQL-код в строку запроса, который будет выполнен на сервере баз данных.** Этот вид атаки возникает из-за недостаточной или отсутствующей фильтрации и проверки пользовательского ввода перед его использованием в SQL-запросах.

Принцип работы SQL-инъекции следующий:

1. Злоумышленник ищет **уязвимую точку в веб-приложении**, где можно ввести данные для запроса к базе данных, например, через веб-форму или параметры URL.
2. Злоумышленник **вводит вредоносный SQL-код** в это поле ввода, который *затем передается на сервер баз данных.*
3. Если приложение неправильно обрабатывает пользовательский ввод, вредоносный SQL-код может быть выполнен на сервере баз данных.
4. В результате этого злоумышленник может **получить доступ к конфиденциальной информации**, модифицировать данные в базе данных, выполнять операции управления базой данных и т. д.

Примеры вредоносных действий, которые может выполнить атакующий при успешной SQL-инъекции:

* Получение конфиденциальной информации из базы данных, такой как пароли, логины, персональные данные и т. д.
* Модификация, добавление или удаление данных в базе данных.
* Выполнение административных операций в базе данных, таких как создание или удаление таблиц и пользователей.

Для защиты от SQL-инъекций необходимо правильно фильтровать и проверять пользовательский ввод перед его использованием в SQL-запросах. Использование параметризованных запросов и подготовленных операторов SQL также является эффективным способом предотвращения SQL-инъекций.

1. **Что такое ResultSet? Как с ним работать?**

**ResultSet** - это объект в языке Java, представляющий результат выполнения SQL-запроса к базе данных. **Он представляет собой таблицу данных,** состоящую из строк и столбцов, полученных в результате выполнения запроса.

Когда вы выполняете запрос к базе данных с помощью объекта Statement или PreparedStatement, база данных возвращает результаты в виде ResultSet. *ResultSet содержит набор строк (записей) и столбцов (полей) со значениями.*

Вы можете перемещаться по ResultSet с помощью методов next(), previous(), first(), last() и т. д., чтобы извлекать данные из каждой строки и каждого столбца. **Метод next()** перемещает курсор на следующую строку в ResultSet. Когда курсор достигает конца ResultSet, метод next() возвращает false, и выход из цикла производится.

Каждая строка ResultSet соответствует одной записи из результата запроса, а каждый столбец соответствует определенному полю или столбцу из таблицы в базе данных.

ResultSet также содержит метаданные, такие как информацию о типах и именах столбцов, которые могут быть использованы для дальнейшей обработки результатов запроса.

Чтобы работать с объектом ResultSet в Java, следует использовать его методы для перемещения по строкам и извлечения значений из столбцов. Вот основные шаги по работе с ResultSet:

1. **Перемещение по ResultSet**: Для перемещения курсора по строкам ResultSet используются методы **next()**, **previous()**, **first()**, **last()** и другие. Метод **next()** перемещает курсор на следующую строку. Когда курсор достигает конца ResultSet, метод **next()** возвращает **false**.
2. **Извлечение значений из столбцов**: Для извлечения значений из текущей строки ResultSet используются методы **getInt()**, **getString()**, **getDouble()** и другие в зависимости от типа данных столбца. Например, **getInt()** используется для извлечения целочисленного значения из столбца, а **getString()** - для извлечения строки.
3. **Работа с метаданными**: ResultSet содержит метаданные о результате запроса, такие как имена столбцов и их типы данных. Методы **getMetaData()** позволяют получить объект **ResultSetMetaData**, содержащий информацию о столбцах ResultSet. Это можно использовать для динамической обработки результатов запроса.
4. **Рассказать про паттерн DAO.**

**- это паттер программирования для создания интерфейса для взаимодействия с БД**

Начнем с рассмотрения шаблона DAO (Data Access Object). Как можно понять из названия, этот **шаблон предназначен для доступа к данным.** Он предоставляет абстрактный интерфейс для обращений к БД. *Основная суть работы данного шаблона заключается в том, чтобы можно было выполнять определенные операции не сильно вдаваясь в детали реализации базы данных*.  При использовании шаблона DAO функции для работы c конкретной таблицей хранятся в файле модели. Соответственно, данная модель наследует абстрактный класс, реализующий DAO. Когда будет получен ряд данных в DAO, в результирующем объекте или массиве будут содержаться все поля из БД.

**DAO (Data Access Object)** - это шаблон проектирования, который используется для абстрагирования и управления доступом к данным из различных источников данных, таких как базы данных, файлы, веб-службы и т. д. Он позволяет разделить бизнес-логику приложения от деталей доступа к данным, обеспечивая более гибкую и модульную архитектуру.

Вот основные принципы и составляющие паттерна DAO:

1. **Интерфейс DAO (DAO Interface)**:
   * Определяет методы для выполнения операций с данными, таких как чтение, запись, обновление и удаление.
   * Абстрагирует бизнес-логику от способа доступа к данным, предоставляя единый интерфейс для работы с данными.
   * Может включать методы для выполнения различных запросов к данным в соответствии с требованиями бизнес-логики.
2. **Реализация DAO (DAO Implementation)**:
   * Предоставляет конкретную реализацию методов, определенных в интерфейсе DAO.
   * Взаимодействует с конкретным источником данных, таким как база данных, файлы или веб-службы, для выполнения операций с данными.
   * Может использовать специализированные библиотеки или ORM-фреймворки для работы с различными типами данных и источниками данных.
3. **Бизнес-объекты (Business Objects)**:
   * Представляют объекты данных или модели предметной области, с которыми работает приложение.
   * Используются для передачи данных между бизнес-логикой и слоем доступа к данным.

Преимущества использования паттерна DAO:

* **Разделение ответственностей**: Позволяет разделить бизнес-логику приложения и операции доступа к данным, что делает приложение более модульным и поддерживаемым.
* **Гибкость**: Обеспечивает гибкость замены и изменения источников данных без влияния на бизнес-логику приложения.
* **Тестируемость**: Упрощает тестирование приложения, так как бизнес-логика может быть протестирована независимо от слоя доступа к данным.

Паттерн DAO широко применяется в различных типах приложений, особенно там, где требуется доступ к данным из разных источников или где необходимо обеспечить модульность и расширяемость кода.

1. **Что такое JPA?**

**Java Persistence API** - это спецификация (стандарт, технология), обеспечивающая объектно-реляционное отображение простых JAVA-объектов (Plain Old Java Object - POJO) и предоставляющая универсальный API для сохранения, получения и управления такими объектами.

**JPA (Java Persistence API)** - это стандартный интерфейс программирования приложений Java EE для работы с объектно-реляционным отображением (ORM) в Java-приложениях. Он предоставляет унифицированный способ взаимодействия с базами данных из Java-приложений, обеспечивая абстракцию от деталей конкретной реализации ORM.

Сам JPA не умеет ни сохранять, ни управлять объектами, **JPA только определяет правила игры**: как *должен действовать каждый провайдер* (Hibernate, EclipseLink, OJB, Torque и т.д.), реализующий стандарт JPA. Для этого JPA определяет интерфейсы, которые должны быть реализованы провайдерами. Также JPA определяет правила*, как должны описываться метаданные отображения и как должны работать провайдеры*.

**Каждый провайдер обязан реализовывать всё из JPA**, определяя стандартное получение, сохранение и управление объектами. Помимо этого, провайдеры могут добавлять свои личные классы и интерфейсы, расширяя функционал JPA.

JPA:

* API в пакете javax.persistance (набор интерфейсов EntityManager, Query, EntityTransaction),
* JPQL - объектный язык запросов (запросы выполняются к объектам)
* Metadata (аннотации или xml)

JAVA-код, написанный только с использованием интерфейсов и классов JPA, позволяет разработчику гибко менять одного провайдера на другого. Например, если приложение использует Hibernate как провайдера, то ничего не меняя в коде можно поменять провайдера на любой другой. Но, если мы в коде использовали интерфейсы, классы или аннотации, например, из Hibernate, то поменяв провайдера на EclipseLink, эти интерфейсы, классы или аннотации уже работать не будут.

1. **Что такое ORM?**

*– это технология программирования, которая связывает наши объекты с реляционными данными.*

Object Relation Mapping – ORM, механизма взаимодействия, который представляет собой вспомогательную прослойку между приложением и базой данных, позволяющую реализовать механизмы представления и взаимодействия между БД и собственно кодом. Суть работы ORM заключается в следующем: мы представляем ряды из таблиц в качестве объектов, свойства которых будут соответствовать именам полей из таблиц, а значения этих свойств – значениям из базы данных. Таким образом мы получаем соответствие одна строка в БД – один объект.

На практике реализация ORM осуществляется с помощью шаблонов DAO, Active Record или DataMapper. Также возможно построение гибридов из нескольких шаблонов.

**ORM (Object-relational mapping)** — технология программирования, которая позволяет преобразовывать несовместимые типы моделей в ООП, в частности, между хранилищем данных и объектами программирования. ORM используется для упрощения процесса сохранения объектов в реляционную базу данных и их извлечения, при этом ORM сама заботится о преобразовании данных между двумя несовместимыми состояниями.

**Object Relational Mapping** - это концепция/процесс преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот. Например, в Java это делается с помощью рефлексии и JDBC.

ORM (Object-Relational Mapping) - это техника программирования, которая позволяет преобразовывать данные из объектной модели в реляционную модель (и наоборот), обеспечивая тем самым удобное взаимодействие между объектно-ориентированным приложением и реляционной базой данных.

Основные принципы и цели ORM:

1. **Абстрагирование от базы данных**: ORM позволяет разработчикам работать с объектами и классами в языке программирования, не беспокоясь о деталях хранения данных в базе данных. Это позволяет более эффективно использовать объектно-ориентированные концепции и сосредоточиться на бизнес-логике приложения.
2. **Устранение необходимости в написании SQL-запросов вручную**: ORM обычно предоставляет способы выполнения операций CRUD (Create, Read, Update, Delete) с использованием объектов и методов, вместо написания SQL-запросов вручную. Это сокращает объем кода и упрощает его поддержку.
3. **Уменьшение связанности кода**: ORM позволяет отделить бизнес-логику приложения от деталей работы с базой данных, что уменьшает связанность кода и делает его более гибким и легко поддающимся изменениям.
4. **Улучшение переносимости и совместимости**: ORM-фреймворки обычно обеспечивают абстракцию от специфических особенностей баз данных, что улучшает переносимость приложения между различными СУБД и позволяет использовать один и тот же код на разных платформах.

Примеры популярных ORM-фреймворков в Java включают Hibernate, JPA (Java Persistence API), EclipseLink и MyBatis. Эти фреймворки предоставляют различные уровни абстракции и функциональные возможности для работы с данными в Java-приложениях, облегчая взаимодействие с базами данных и ускоряя процесс разработки.

1. **Что такое Hibernate?**

**Hibernate -** это провайдер, реализующий спецификацию JPA. *Hibernate полностью реализует JPA плюс добавляет функционал в виде своих классов и интерфейсов*, расширяя свои возможности по работе с сущностями и БД.

**Hibernate** - это мощный и популярный ORM-фреймворк для языка Java, который обеспечивает удобный способ взаимодействия с реляционными базами данных. Он предоставляет инструменты и функции для отображения объектов Java на записи в базе данных и управления этими объектами с помощью API и запросов на языке HQL (Hibernate Query Language).

Вот основные возможности и преимущества Hibernate:

1. **Отображение объектно-реляционного соответствия (ORM)**: Hibernate позволяет разработчикам работать с объектами и классами Java, а не с SQL-запросами и таблицами баз данных напрямую. Он обеспечивает прозрачное отображение объектов на записи в базе данных и автоматически выполняет необходимые операции чтения, записи, обновления и удаления данных.
2. **Поддержка различных СУБД**: Hibernate поддерживает большинство распространенных реляционных баз данных, таких как MySQL, PostgreSQL, Oracle, Microsoft SQL Server и другие, что обеспечивает гибкость выбора СУБД для ваших приложений.
3. **Язык запросов HQL**: Hibernate предоставляет собственный язык запросов - HQL (Hibernate Query Language), который абстрагирует от специфических особенностей SQL и позволяет писать запросы на языке Java-подобном синтаксисе, используя объекты и свойства классов вместо таблиц и столбцов.
4. **Кэширование**: Hibernate предоставляет механизм кэширования данных, который позволяет улучшить производительность приложения путем кеширования часто используемых данных в памяти приложения или на уровне базы данных.
5. **Транзакции и управление сеансами**: Hibernate обеспечивает поддержку транзакций и управление сеансами для обеспечения целостности данных и безопасности при взаимодействии с базой данных.
6. **Интеграция с другими технологиями**: Hibernate хорошо интегрируется с другими технологиями и фреймворками Java, такими как Spring Framework, что обеспечивает еще большую гибкость и функциональность при разработке приложений.

Hibernate является одним из самых популярных ORM-фреймворков в мире Java и широко используется в различных типах приложений, от небольших веб-сайтов до крупных корпоративных приложений. Его простота использования, богатые функциональные возможности и активное сообщество разработчиков делают его предпочтительным выбором для работы с данными в Java-приложениях.

1. **В чем разница между JPA и Hibernate? Как связаны все эти понятия?**

**Различия *JPA* *JDBC* и Hibernate:**

**JDBC** является гораздо более низкой (и более старой) спецификацией, чем JPA. JDBC - это API-интерфейс для взаимодействия с базой данных с использованием чистого SQL - отправки запросов и получения результатов. Он не имеет понятия об объектах или иерархиях. При использовании JDBC вам необходимо преобразовать набор результатов в объекты Java.

А **в JPA** (который использует JDBC снизу) вы также указываете эти детали метаданных базы данных, но с использованием аннотаций Java. Таким образом, JPA создает запросы на обновление для вас и управляет объектами, которые вы искали или создали / обновили (это также делает больше). Основной интерфейс **EntityManager**

**Hibernate** одна из самых популярных открытых реализаций последней версии спецификации. То есть JPA только описывает правила и API, а Hibernate реализует эти описания. Основной класс – **Session**

JPA (Java Persistence API) и Hibernate - это два различных, но связанных понятия в контексте работы с базами данных в Java.

1. **JPA (Java Persistence API)**:
   * JPA - это стандарт Java EE для работы с объектно-реляционным отображением (ORM).
   * JPA предоставляет набор интерфейсов и аннотаций для работы с базами данных из Java-приложений.
   * Он определяет стандартный способ описания сущностей, их отношений и запросов к данным.
   * JPA не предоставляет реализацию; он определяет только интерфейсы и структуру.
   * Реализации JPA могут быть предоставлены различными поставщиками ORM.
2. **Hibernate**:
   * Hibernate - это одна из реализаций спецификации JPA.
   * Он является ORM-фреймворком для Java, который обеспечивает отображение объектов Java на таблицы базы данных и обратно.
   * Hibernate предоставляет реализацию интерфейсов JPA и дополнительные функциональные возможности.
   * Это мощный и гибкий инструмент, который предоставляет богатые функции для работы с базами данных.

Таким образом, JPA определяет стандартные интерфейсы и аннотации для работы с базами данных в Java, в то время как Hibernate предоставляет конкретную реализацию этого стандарта, а также расширенные функции и возможности. Hibernate можно рассматривать как один из вариантов использования JPA в вашем приложении, а JPA - как более общий стандарт, который можно реализовать различными способами.

1. **Какие классы/интерфейсы относятся к JPA/Hibernate?**

JPA (Java Persistence API) и Hibernate предоставляют ряд классов и интерфейсов для работы с базами данных и объектно-реляционным отображением (ORM). Вот основные классы и интерфейсы, относящиеся к JPA и Hibernate:

1. **EntityManager**: Интерфейс **EntityManager** представляет собой центральную точку управления сущностями в JPA. Он используется для выполнения операций над сущностями, таких как *сохранение, обновление, удаление и выполнение запросов.*
2. **EntityManagerFactory**: Интерфейс **EntityManagerFactory** представляет собой фабрику **EntityManager**. Он используется для создания экземпляров **EntityManager**.
3. **EntityTransaction**: Интерфейс **EntityTransaction** используется для управления транзакциями в JPA. Он предоставляет методы для начала, фиксации и отката транзакции.
4. **CriteriaQuery**: Интерфейс **CriteriaQuery** используется для создания динамических запросов к базе данных с использованием критериев. Он позволяет строить запросы на основе типобезопасных критериев, представленных в виде Java-объектов.
5. **CriteriaBuilder**: Интерфейс **CriteriaBuilder** используется вместе с **CriteriaQuery** для построения критериев запроса. Он предоставляет методы для создания предикатов, выражений, функций и других компонентов запроса.
6. **Query**: Интерфейс **Query** представляет собой запрос к базе данных в JPA. Он используется для выполнения запросов JPQL (Java Persistence Query Language) и нативных SQL-запросов.
7. **Session**: В Hibernate класс **Session** представляет собой основной механизм взаимодействия с базой данных. Он аналогичен **EntityManager** в JPA и используется для выполнения операций над объектами и управления транзакциями.
8. **SessionFactory**: Класс **SessionFactory** представляет собой фабрику **Session** в Hibernate. Он используется для создания экземпляров **Session**.
9. **Transaction**: Класс **Transaction** в Hibernate аналогичен **EntityTransaction** в JPA. Он используется для управления транзакциями и предоставляет методы для начала, фиксации и отката транзакции.

Эти классы и интерфейсы обеспечивают основные средства для работы с базами данных и ORM в Java с использованием JPA и Hibernate. Они позволяют выполнять различные операции с объектами и управлять транзакциями для обеспечения целостности данных.

**Запрос с использованием Criteria API** - это способ формирования запросов к базе данных в Java с использованием Criteria API, предоставляемого Java Persistence API (JPA). Criteria API позволяет создавать запросы к базе данных с помощью объектной модели, что делает их более типобезопасными и удобными для работы в Java коде, по сравнению с использованием строковых запросов (JPQL или SQL).

Основные преимущества использования Criteria API:

1. **Типобезопасность**: Запросы формируются с использованием объектной модели, что делает их более типобезопасными. Ошибки в запросах могут быть выявлены на этапе компиляции.
2. **Удобство**: Criteria API предоставляет удобные методы для создания запросов, такие как **CriteriaBuilder**, **CriteriaQuery**, **Root**, **Predicate** и другие. Эти методы позволяют динамически создавать запросы и применять различные условия и фильтры.
3. **Поддержка ORM**: Criteria API является частью Java Persistence API (JPA) и легко интегрируется с ORM (Object-Relational Mapping) фреймворками, такими как Hibernate, EclipseLink и др.
4. **Основные аннотации Hibernate, рассказать.**

Hibernate предоставляет ряд аннотаций для управления отображением объектов Java на таблицы в базе данных и их взаимодействия с ORM-фреймворком. Вот основные аннотации Hibernate:

1. **@Entity**: Аннотация **@Entity** используется **для указания класса как сущности**, которая будет сохранена в базе данных. *Каждый экземпляр этого класса представляет собой запись в таблице базы данных.* Эта аннотация обычно применяется к классу.
2. **@Table**: Аннотация **@Table** используется **для указания дополнительных параметров таблицы базы данных, связанной с сущностью.** Например, вы можете использовать эту аннотацию, чтобы указать имя таблицы, к которой должна быть отображена сущность, а также другие параметры таблицы, такие как схема, уникальные ограничения и т. д.
3. **@Id**: Аннотация **@Id** используется для указания поля или метода, который является первичным ключом сущности. Первичный ключ уникально идентифицирует каждую запись в таблице.
4. **@GeneratedValue**: Аннотация **@GeneratedValue** указывает способ генерации значений для первичного ключа. Это может быть автоматическая генерация (например, AUTO, IDENTITY, SEQUENCE) или определенное значение, заданное пользователем.
5. **@Column**: Аннотация **@Column** используется для настройки свойств столбца базы данных, соответствующего полю сущности. Например, вы можете указать имя столбца, его тип данных, длину, нулевое или не нулевое значение и т. д.
6. **@OneToMany** и **@ManyToOne**: Эти аннотации используются для описания отношений между сущностями. **@OneToMany** указывает на однонаправленное или двунаправленное отношение "один ко многим", в то время как **@ManyToOne** указывает на однонаправленное отношение "многие к одному".
7. **@JoinColumn**: Аннотация **@JoinColumn** используется для настройки свойств соединительного столбца для ассоциации между таблицами. Она обычно используется с аннотациями **@OneToMany** и **@ManyToOne**.
8. **@Transient**: Аннотация **@Transient** используется для указания поля или метода, которые не должны быть отображены в базе данных. Например, вы можете использовать это для временных или вычисляемых полей.

Эти аннотации предоставляют мощные средства для управления отображением объектов Java на структуры данных в базе данных с помощью Hibernate. Они обеспечивают гибкость и контроль над способом отображения и хранения данных в вашем приложении.

1. **Чем HQL отличается от SQL?**

**SQL** – работает конкретно с базой данных(мы оперируем названиями таблиц , названиями колонок и тд), с таблицей, колонками.

**HQL** -Надстройка, работает с классами, с полями в классах(оперируем названиями java класса), + селект можно не писать.

HQL (Hibernate Query Language) и SQL (Structured Query Language) - это два различных языка запросов, используемых для взаимодействия с базами данных. Вот основные различия между ними:

1. **Уровень абстракции**: SQL является языком запросов, специфичным для баз данных, который используется для работы с данными в реляционных базах данных. HQL, с другой стороны, является языком запросов, специфичным для ORM-фреймворка Hibernate, который используется для работы с объектами в Java и их отображения на таблицы базы данных.
2. **Синтаксис**: Хотя оба языка используются для выполнения запросов к базе данных, их синтаксис существенно отличается. SQL использует набор команд и операторов, таких как SELECT, INSERT, UPDATE, DELETE, WHERE и т. д., для выполнения различных операций над данными в базе данных. HQL, с другой стороны, использует объектно-ориентированный подход и оперирует объектами и их свойствами, а не таблицами и столбцами.
3. **Типы данных**: SQL оперирует стандартными типами данных баз данных, такими как INTEGER, VARCHAR, DATE и другими, которые определяются спецификацией базы данных. HQL оперирует объектами Java и их свойствами, а типы данных определяются Java классами.
4. **Поддержка ORM-функций**: HQL предоставляет ORM-функции и конструкции, такие как JOIN, FETCH, HAVING и другие, которые позволяют работать с объектами и их ассоциациями, как если бы они были в памяти, в то время как SQL работает непосредственно с данными в базе данных.
5. **Портабельность**: HQL обеспечивает более высокую портабельность кода, поскольку запросы HQL абстрагируются от специфических деталей баз данных и могут быть использованы с различными базами данных без изменения синтаксиса. SQL запросы, напротив, могут быть более зависимы от конкретной базы данных и могут требовать изменений при переходе от одной базы данных к другой.
6. **Что такое Query? Как передать в объект Query параметры?**

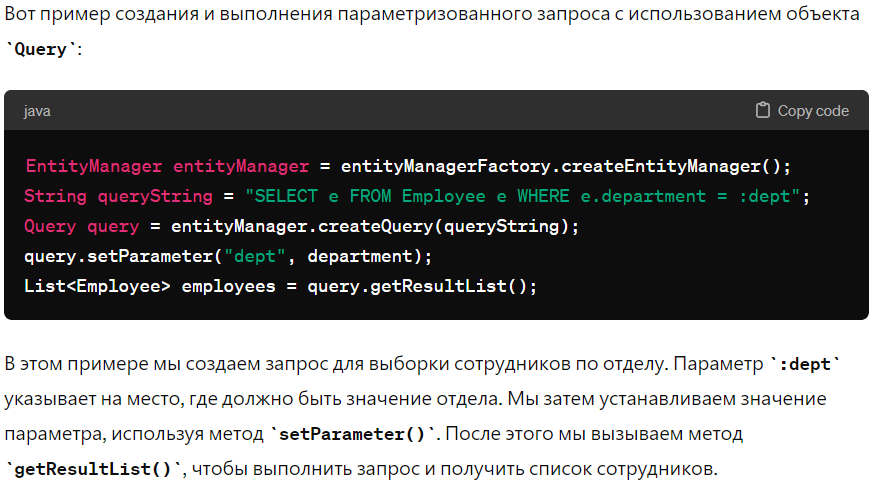
Бывают: безымянные (когда знаки вопроса ставим), и именованные (name, lastName, age)

Query – запрос который делаем с помощью hql он может быть с параметрами(безымянные или именнованные(двоеточие и название перменной)

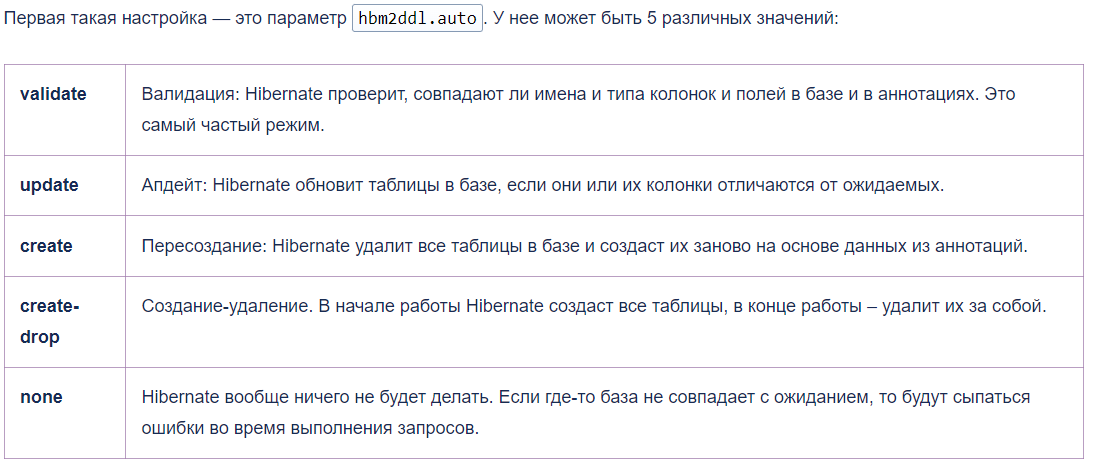
Объект **Query** в контексте JPA (Java Persistence API) представляет собой запрос к базе данных для извлечения данных или выполнения действий с сущностями. Он используется для создания и выполнения запросов на выборку, обновление, удаление и другие операции с объектами сущностей.

Вы можете создать объект **Query**, используя метод **createQuery()** сессии или менеджера сущностей. Затем вы можете выполнить этот запрос с помощью методов, таких как **getResultList()**, **getSingleResult()**, **executeUpdate()** и других, в зависимости от типа запроса.

Для передачи параметров в объект **Query** вы можете использовать параметризованные запросы. В JPA параметры обозначаются с помощью специального синтаксиса **:name**, где **name** - это имя параметра. Затем вы можете установить значения параметров, используя метод **setParameter()** или его перегруженные версии.

****

1. **Какие можно устанавливать параметры в hbm2ddl, рассказать про каждый из них.**



**hbm2ddl** - это инструмент, предоставляемый Hibernate для автоматической генерации схемы базы данных на основе сопоставления объектов (mapping) с помощью файлов маппинга (**.hbm.xml**) или аннотаций. При запуске **hbm2ddl** можно указать различные параметры, чтобы настроить процесс генерации схемы. Вот некоторые из них:

1. **hibernate.hbm2ddl.auto**:
   * Этот параметр определяет режим работы **hbm2ddl**. Он может принимать различные значения:
     + **validate**: Проверяет соответствие схемы базы данных маппингу Hibernate, но не вносит изменений в базу данных.
     + **update**: Обновляет схему базы данных, чтобы она соответствовала маппингу Hibernate. Это может включать создание новых таблиц, добавление новых столбцов и т. д.
     + **create**: Создает схему базы данных с нуля на основе маппинга Hibernate. Существующие таблицы будут удалены.
     + **create-drop**: Создает схему базы данных и удаляет ее после завершения сеанса работы с Hibernate.
2. **hibernate.hbm2ddl.import\_files**:
   * Этот параметр позволяет указать один или несколько SQL-файлов, которые будут выполнены после создания или обновления схемы базы данных. Это может быть полезно, например, для внесения начальных данных или настройки индексов после генерации схемы.
3. **hibernate.hbm2ddl.delimiter**:
   * Определяет разделитель для SQL-скриптов, указанных в параметре **hibernate.hbm2ddl.import\_files**.
4. **hibernate.hbm2ddl.charset\_name**:
   * Устанавливает кодировку, которая будет использоваться при чтении SQL-файлов, указанных в параметре **hibernate.hbm2ddl.import\_files**.
5. **hibernate.hbm2ddl.format\_sql**:
   * Указывает, следует ли форматировать генерируемый SQL-код для улучшения читаемости.

Это лишь некоторые из параметров, которые можно установить для настройки работы **hbm2ddl** в Hibernate. Подробную информацию о других параметрах можно найти в документации Hibernate.

1. **Требования JPA к Entity-классам? Не менее пяти.**

* Поле с id
* Аннотация
* Конструктор с пустыми параметрами
* Не должен быть final этот класс, тк прокси объекты используются с помощью наследования, а от final нельзя наследоваться
* Поля должны быть приватными + обращение через геттеры и сеттеры

Java Persistence API (JPA) устанавливает **некоторые требования к Entity-классам** для их использования в ORM (Object-Relational Mapping). Вот основные требования JPA к Entity-классам:

1. **Аннотация @Entity**: Класс, который должен быть сопоставлен с таблицей базы данных, должен быть аннотирован аннотацией **@Entity**.
2. **Уникальный идентификатор**: Entity-класс должен иметь уникальный идентификатор (Primary Key), который обозначается аннотацией **@Id**. Это может быть поле или набор полей, образующих составной ключ.
3. **Аннотация @GeneratedValue**: Если значение идентификатора должно генерироваться автоматически (например, с использованием автоинкремента или последовательности), то идентифицирующее поле должно быть аннотировано аннотацией **@GeneratedValue**.
4. **Отсутствие конструкторов без аргументов**: Entity-класс должен иметь конструктор с аргументами, который используется для создания новых экземпляров класса.
5. **Геттеры и сеттеры**: Entity-класс должен предоставлять геттеры и сеттеры для доступа к своим полям. Это не является строгим требованием JPA, но рекомендуется для соблюдения принципов инкапсуляции и безопасности.
6. **Правильное переопределение методов equals() и hashCode()**: Entity-классы должны правильно переопределять методы **equals()** и **hashCode()**, чтобы обеспечить корректное сравнение и хэширование объектов.
7. **Индексирование**: Поля, которые часто используются в запросах для фильтрации или сортировки, могут быть проиндексированы для повышения производительности запросов.
8. **Соответствие JavaBeans**: Entity-классы должны следовать принципам JavaBeans, что включает в себя применение private полей и публичных геттеров и сеттеров для доступа к ним.

Соблюдение этих требований позволяет JPA корректно работать с Entity-классами и проводить маппинг объектов Java на записи базы данных и обратно.

**Требования к Entity классу в JPA**

1. Entity класс должен быть помечен аннотацией **@Entity** или описан в XML файле конфигурации JPA.

2. Entity класс должен содержать **public или protected конструктор без аргументов** (он также может иметь конструкторы с аргументами).

3. Entity класс должен быть **классом верхнего уровня** (top-level class).

4**. Перечисление** [enum] или **интерфейс** [interface] **не могут быть определены как сущность** [Entity].

5**. Entity класс не может быть финальным классом** (final class). Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables).

6. Если объект Entity класса будет передаваться по значению как отделённый от контекста персистентности объект (detached object), например через удаленный интерфейс (through a remote interface), то он также должен реализовывать 12 интерфейс Serializable (чтобы объекты которые достаются из базы могли сохраняться в кэше).

7. Как обычный так и абстрактный класс может быть Entity. Entities могут наследоваться как от не Entity классов, так и от Entity классов. А не Entity классы могут наследоваться от Entity классов.

8. **Поля Entity класса должны быть объявлены private, protected или package-private**, быть напрямую доступными только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Другие классы должны обращаться только к специальным методам Entity класса, предоставляющим доступ к этим полям (**getter/setter-методам** или другим методам бизнес-логики в Entity классе).

9. **Entity класс должен содержать первичный ключ**, то есть атрибут или группу атрибутов, которые уникально определяют запись этого Entity класса в базе данных.

1. **Жизненный цикл Entity в Hibernate? Рассказать.**

Жизненный цикл сущности (Entity) в Hibernate состоит из нескольких состояний, которые определяют, как сущность взаимодействует с базой данных. Вот основные состояния жизненного цикла сущности в Hibernate(РАССКАЗЫВАЕМ ПЕРВЫЕ 4 СТАДИИ):

1. **Transient (Переходное)**: Сущность считается переходной, если она только что создана с помощью оператора **new**, но еще не связана с сессией Hibernate и не имеет идентификатора. Переходная сущность не отображается в базе данных.
2. **Persistent (Постоянное)**: Сущность считается постоянной, когда она связана с текущей сессией Hibernate и имеет идентификатор. При сохранении переходной сущности в базу данных с помощью метода **save()** или **persist()**, она становится постоянной.
3. **Detached (Отсоединенное)**: Сущность становится отсоединенной, когда ее связь с сессией Hibernate прекращается. Это происходит после завершения транзакции, закрытия сессии или вызова метода **evict()** или **clear()** для отсоединения сущности от сессии.
4. **Removed (Удаленное)**: Сущность считается удаленной, когда она была связана с сессией Hibernate, но затем была удалена из базы данных с помощью метода **delete()** или **remove()**. После удаления сущности она остается в состоянии удаленной до завершения транзакции.
5. **Reattached (Прикрепленное)**: После отсоединения сущности и повторного подключения к сессии Hibernate она становится прикрепленной. Это происходит, например, при вызове метода **update()** или **merge()**, чтобы вновь привязать сущность к сессии и обновить ее состояние в базе данных.

Эти состояния определяют, как Hibernate управляет жизненным циклом сущности и какие операции можно выполнять с сущностями в каждом из состояний. Понимание этих состояний поможет в правильном использовании Hibernate и предотвращении ошибок при работе с сущностями.

**Методом рулбэк бы можем откатывать транзакцию (поместим в блоке catch)**

**Нельзя создать таблицу с User? Уточнить до конца**

**Дополнительная информация(транзакция, кэширование):**

1. **Транзакция** – это группа действий, которые должны быть выполнены только все вместе. Если какое-либо действие не выполнилось или выполнилось с ошибкой, то все остальные действия должны быть отменены.

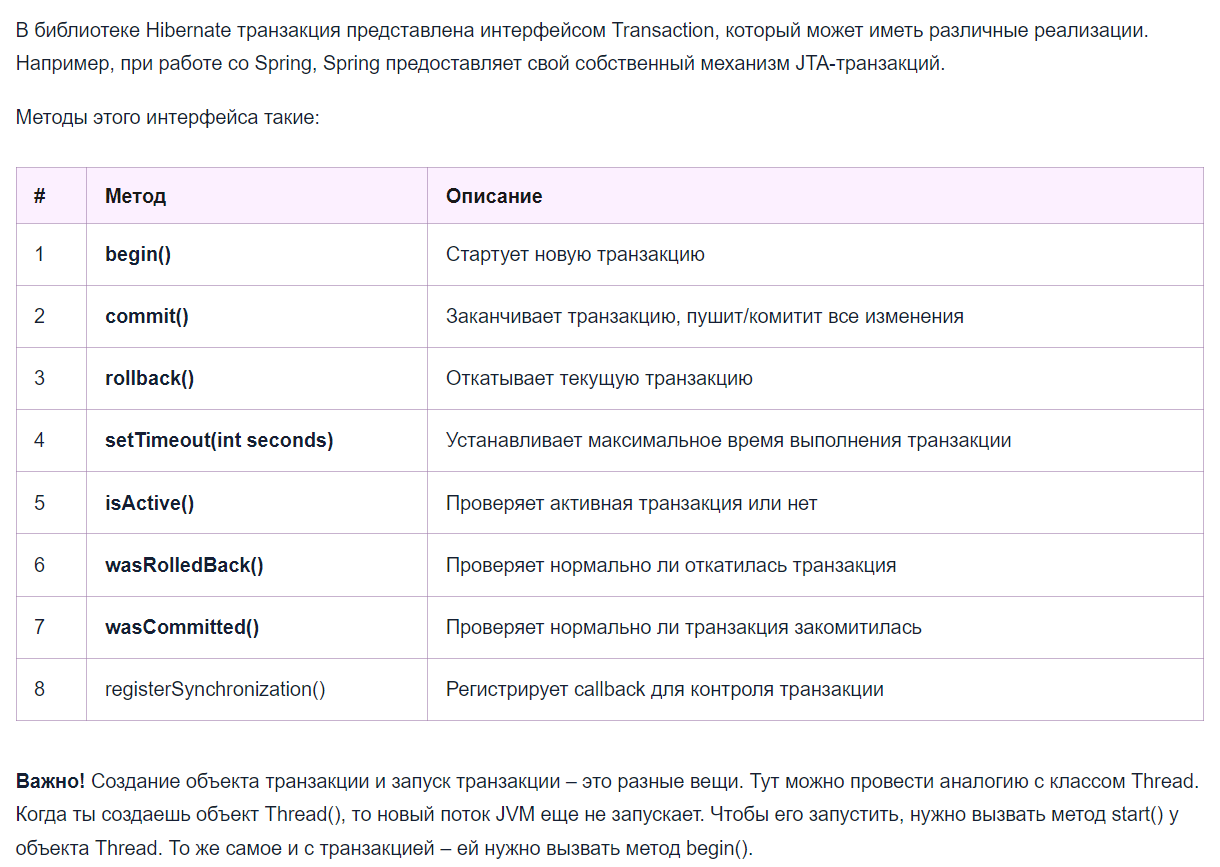
Hibernate умеет работать с двумя видами транзакций:

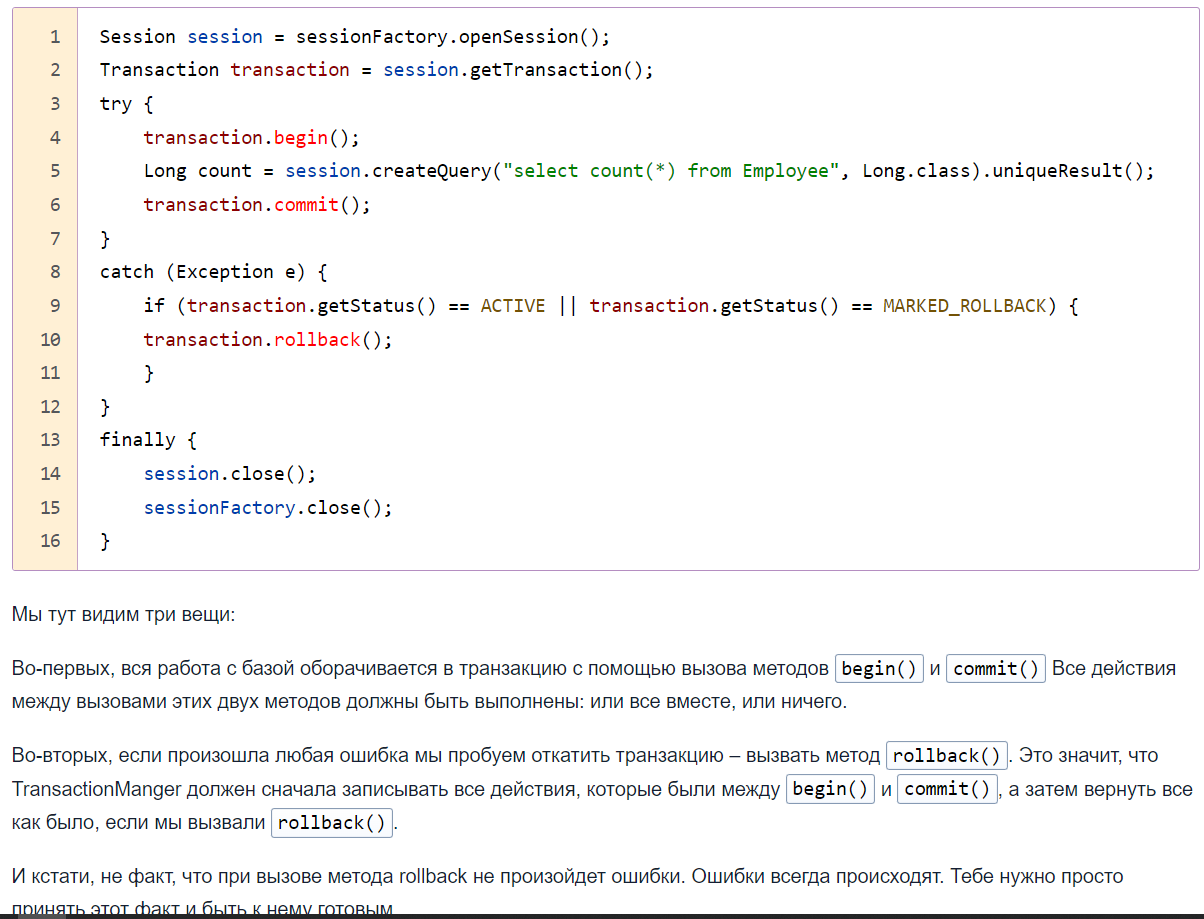
* **JDBC**
* **JTA**

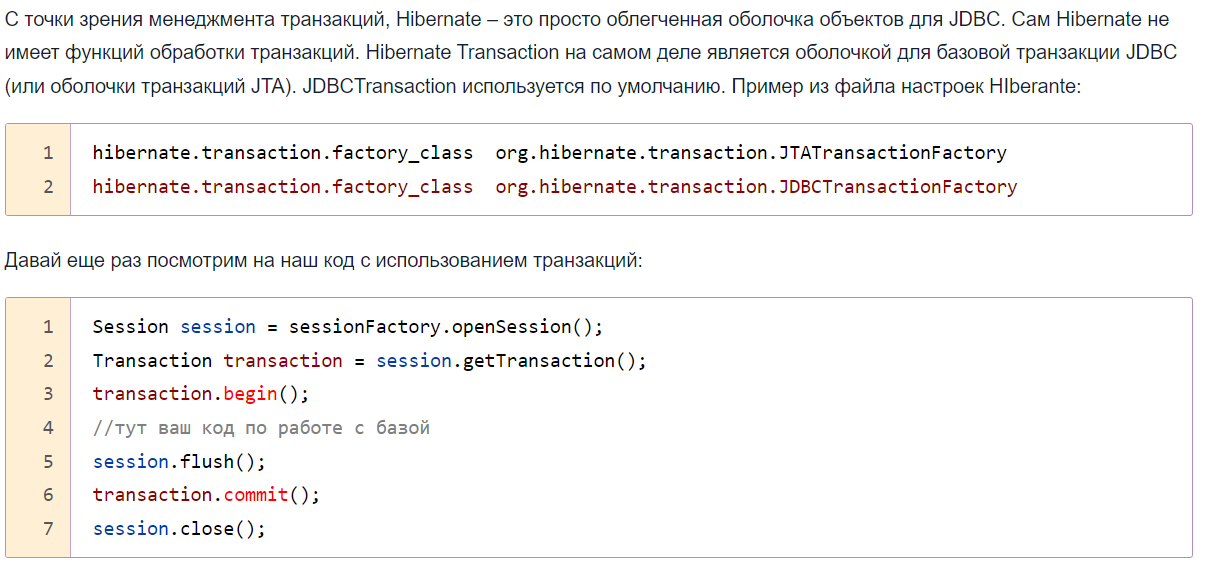
**JDBC-транзакция,** – это фактически транзакция базы данных. Она привязана к работе с базой данных, к JDBC-соединению. И следит за тем, чтобы действия при работе с базой данных выполнялись как надо: или все или ничего.

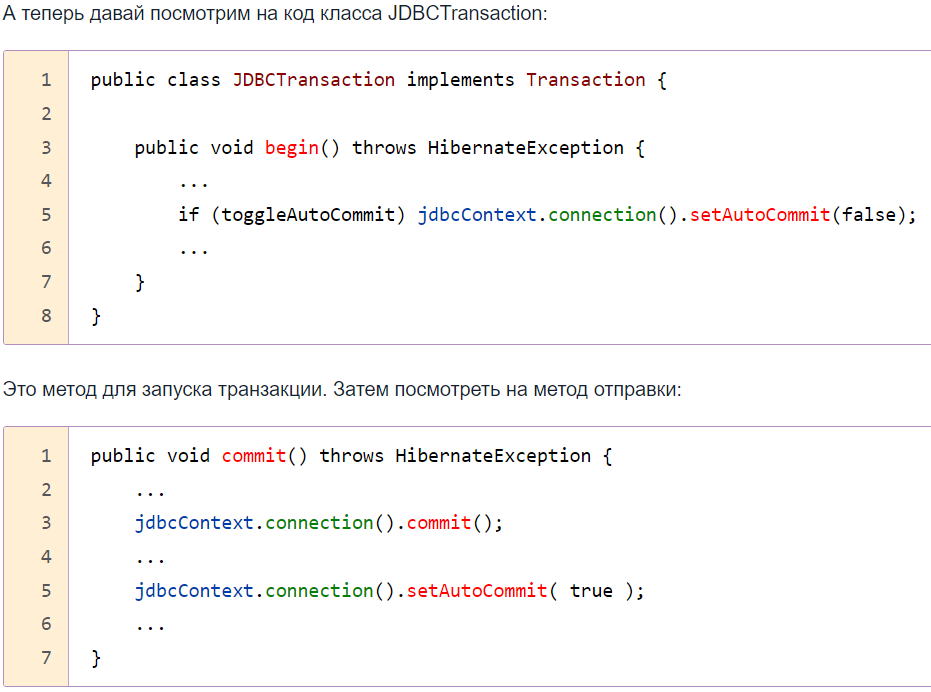
**JTA – транзакция** – это транзакция уровня приложения. Она не привязана ни к какой базе данных. Ее задача следить, чтобы некоторые действия выполнялись: или все, или ничего.

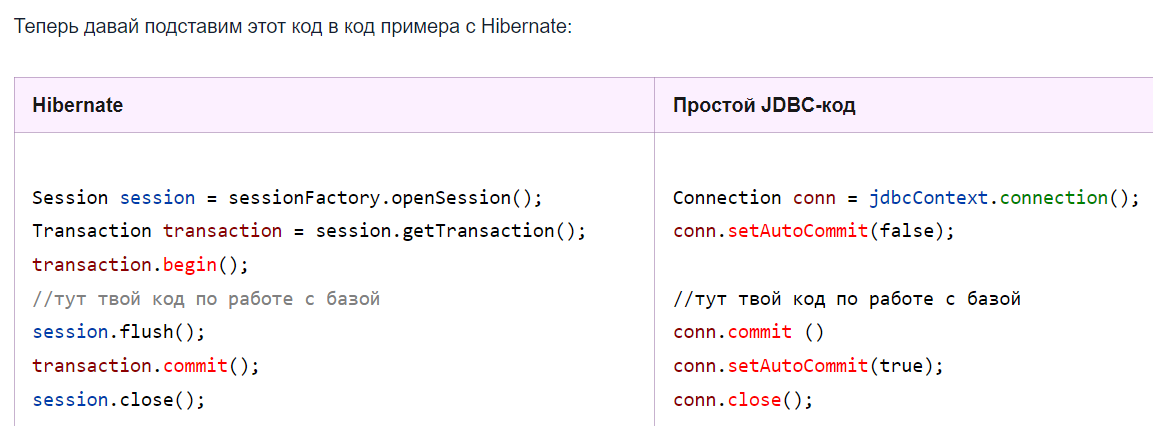
Например, ты можешь записывать данные в несколько разных баз данных в рамках одной JTA-транзакции. Тогда если произойдет ошибка, то JTA-транзакция должна будет откатить изменения во всех базах данных. Даже те, которые были выполнены успешно с точки зрения конкретной базы данных.











1. **Кеширование данных в Hibernate**

Довольно часто в java приложениях с целью снижения нагрузки на БД используют кеш. Не много людей реально понимают как работает кеш под капотом, добавить просто аннотацию не всегда достаточно, нужно понимать как работает система.

Прежде всего Hibernate cache — это 3 уровня кеширования:

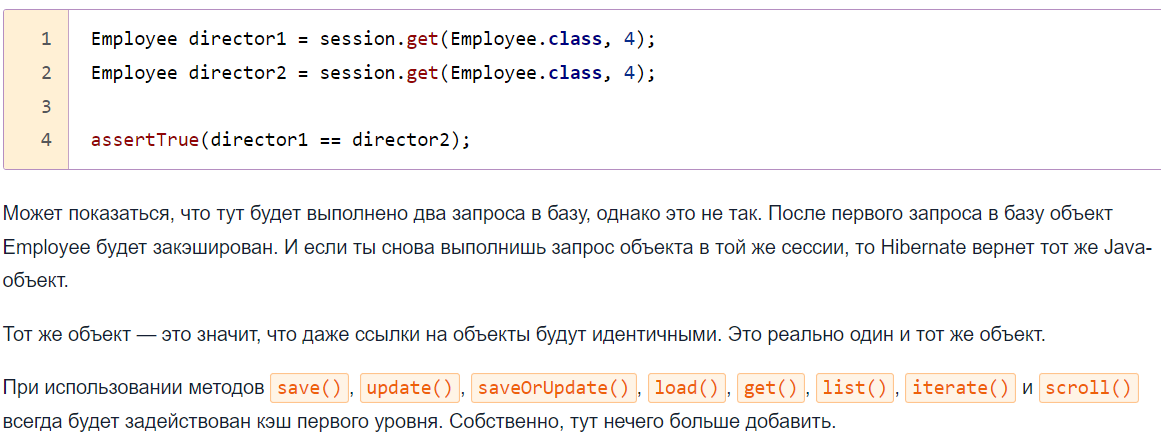
* **Кеш первого уровня (First-level cache);**
* **Кеш второго уровня (Second-level cache);**
* **Кеш запросов (Query cache);**

**Кеш первого уровня**

Кеш первого уровня **всегда привязан к объекту сессии**. Hibernate всегда по умолчанию использует этот кеш и его нельзя отключить.

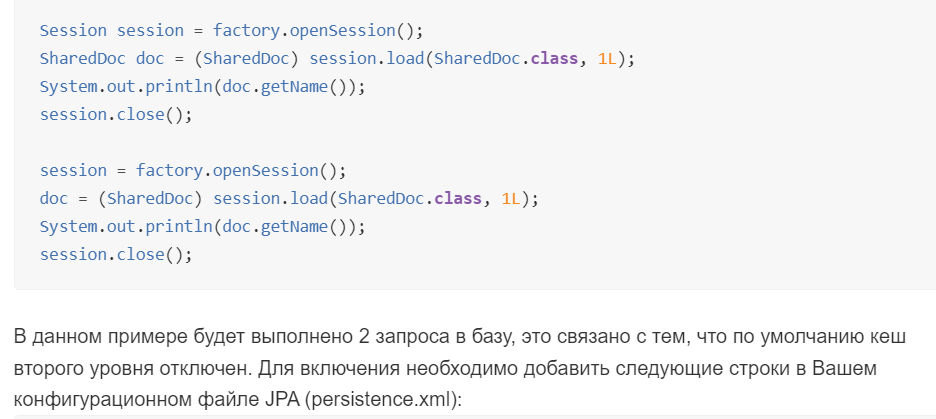


Возможно, Вы ожидаете, что будет выполнено 2 запроса в БД? Это не так. В этом примере будет выполнен 1 запрос в базу, несмотря на то, что делается 2 вызова load(), так как эти вызовы происходят в контексте одной сессии. Во время второй попытки загрузить план с тем же идентификатором будет использован кеш сессии.  
Один важный момент — при использовании метода load() Hibernate не выгружает из БД данные до тех пор пока они не потребуются. Иными словами — в момент, когда осуществляется первый вызов load, мы получаем прокси объект или сами данные в случае, если данные уже были в кеше сессии. Поэтому в коде присутствует getName() чтобы 100% вытянуть данные из БД. Тут также открывается прекрасная возможность для потенциальной оптимизации. В случае прокси объекта мы можем связать два объекта не делая запрос в базу, в отличии от метода get(). При использовании методов save(), update(), saveOrUpdate(), load(), get(), list(), iterate(), scroll() всегда будет задействован кеш первого уровня. Собственно, тут нечего больше добавить.



**Кеш второго уровня**

Если кеш первого уровня привязан к объекту сессии, то кеш второго уровня **привязан к объекту-фабрике сессий (Session Factory object)**. Что как бы подразумевает, что видимость этого кеша гораздо шире кеша первого уровня.



*<property name="hibernate.cache.provider\_class" value="net.sf.ehcache.hibernate.SingletonEhCacheProvider"/>*

//или в более старых версиях

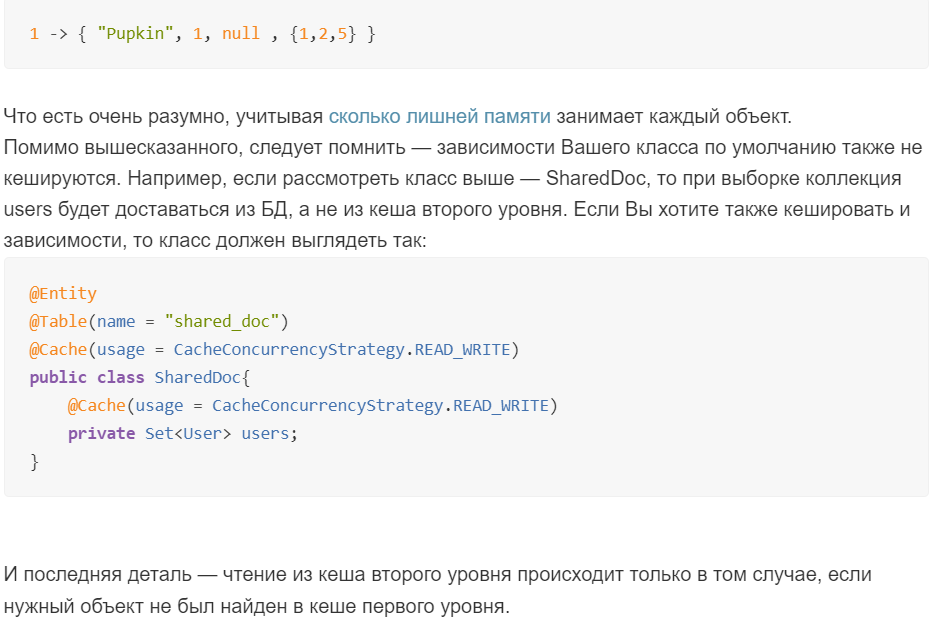
*//<property name="hibernate.cache.provider\_class" value="org.hibernate.cache.EhCacheProvider"/>  
<property name="hibernate.cache.use\_second\_level\_cache" value="true"/>*

Обратите внимание на первую строку. На самом деле, хибернейт сам не реализует кеширование как таковое. А лишь предоставляет структуру для его реализации, поэтому подключить можно любую реализацию, которая соответствует спецификации нашего ORM фреймворка. Из популярных реализаций можна выделить [следующие](http://docs.jboss.org/hibernate/core/3.3/reference/en/html/performance.html#performance-cache):

* EHCache
* OSCache
* SwarmCache
* JBoss TreeCache

Помимо всего этого, вероятней всего, Вам также понадобится отдельно настроить и саму реализацию кеша. В случае с EHCache это нужно сделать в файле [ehcache.xml](http://ehcache.org/ehcache.xml). Ну и в завершение еще нужно указать самому хибернейту, что именно кешировать.



Только после всех этих манипуляций кеш второго уровня будет включен и в примере выше будет выполнен только 1 запрос в базу.  
Еще одна важная деталь про кеш второго уровня про которую стоило бы упомянуть — **хибернейт** не хранит сами объекты Ваших классов. Он **хранит информацию в виде массивов строк, чисел и т. д.** И идентификатор объекта выступает указателем на эту информацию. Концептуально это нечто вроде Map, в которой id объекта — ключ, а массивы данных — значение.   


У Hibernate есть 4 возможных стратегии доступа для кэшируемой сущности, если к ней обращаются из разных потоков:

* read-only
* read-write
* nonstrict-read-write
* transactional

**Только для чтения** (read-only). Стратегия параллелизма, подходящая для данных, которая никогда не изменяется. Hibernate будет просто хранить эти объекты у себя в памяти. Используй его только для справочных данных.

В базах данных хранится очень много информации, которая никогда не меняется. Например, таблица хранит список событий, которые только добавляются, но никогда не изменяются и не удаляются. Если тебе нужно работать с этой таблицей через Hibernate, то тебе подойдет стратегия кэширования read-only.

**Чтение-запись** (read-write). Используй эту стратегию для данных, предназначенных главным образом для чтения. Однако Hibernate будет отслеживать попытки изменения этих данных, хотя и рассчитывает, что они будут очень редкими.

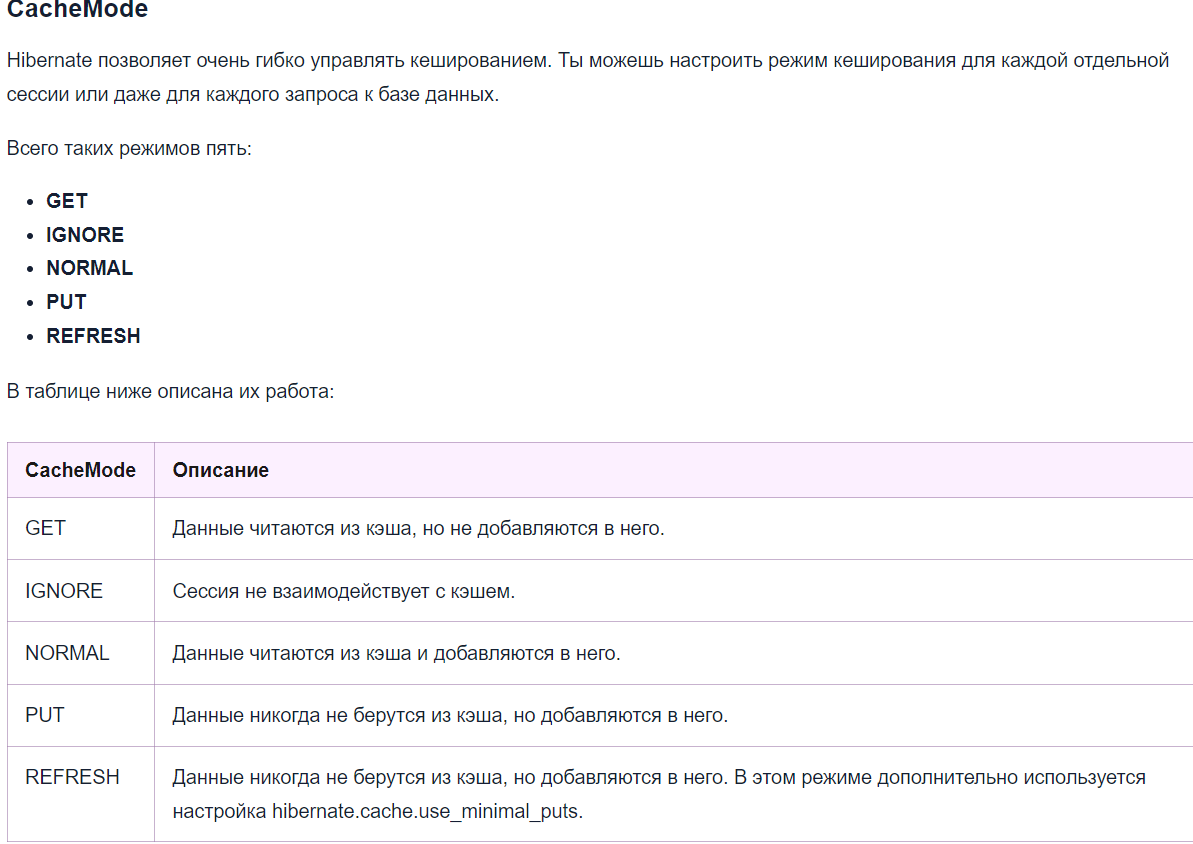
Кешировать нужно в основном те объекты, которые редко меняются и часто читаются/запрашиваются. Если у тебя есть такие объекты, то для них нужно использовать стратегию read-write.

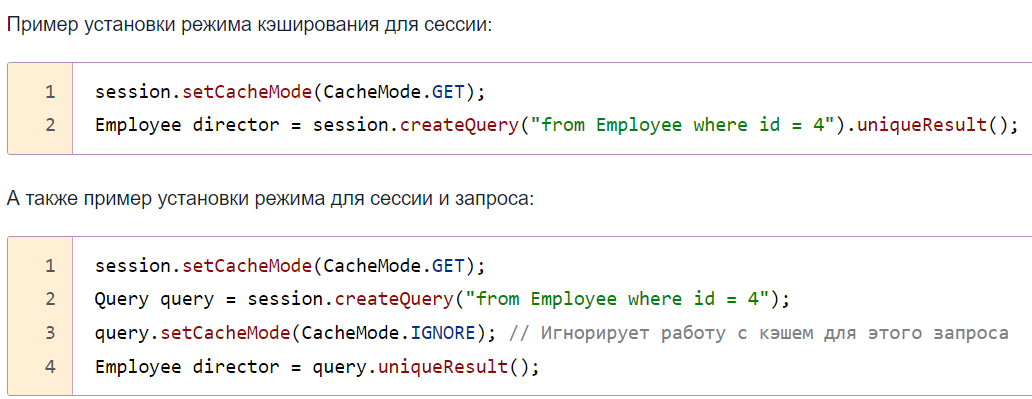
**Nonstrict-read-write**. Эта стратегия не гарантирует согласованности между кэшем и базой данных. Используй эту стратегию, если данные почти никогда не изменяются и небольшая вероятность устаревших данных не является критической проблемой.

В отличии от стратегии read-write, эта стратегия подразумевает, что изменяемые данные не лочатся на чтение. Это может привести к тому, что объект изменился в одном месте, а в другом кто-то читает его старую версию.

Например, пользователь изменил свой комментарий, но остальные пользователи еще какое-то время видят его старую версию. Если для тебя это не является проблемой, тогда используй стратегию nonstrict-read-write.

**Транзакционный**. Используй эту стратегию для данных, предназначенных главным образом для чтения, где важно предотвратить устаревшие данные в параллельных транзакциях в редких случаях обновления.





**Кеш запросов**



Кеш запросов похож на кеш второго уровня. Но в отличии от него — ключом к данным кеша выступает не идентификатор объекта, а совокупность параметров запроса. А сами данные — это идентификаторы объектов соответствующих критериям запроса. Таким образом, этот кеш рационально использовать с кешем второго уровня.