# Что такое Maven? Для чего он нужен? Как добавлять в проект библиотеки без него?

Maven – это средство для управления и сборки проектов. Он позволяет разработчикам полностью управлять жизненным циклом проекта.

В случае, когда над проектом работает несколько команд разработчиков, Maven позволяет работать в соответствии с определенными стандартами.

Основной целью Maven является предоставление разработчику:

* Понятной модели для проектов, которая может быть использована повторно и была бы проста в поддержании.
* Плагины, которые могут взаимодействовать с этой моделью.

Структура и содержание проекта Maven указывается в специальном xml-файле, который называется Project Object Model (POM), который является базовым модулем всей системы.

Разработчики не обязаны указывать каждую деталь. Maven обеспечивает поведение проекта по умолчанию. Когда мы создаём проект с использованием этой технологии, то Maven создаёт базовую структуру проекта. Разработчики несут ответственность только за соответствующее размещение файлов.

В таблице указано, как по-умолчанию расположены файлы в проекте:

|  |  |
| --- | --- |
| исходный код | /src/main/java |
| Ресурсы | /src/main/resources |
| Тесты | /src/test |
| Дистрибутив JAR | /target |
| Скомпилированный байт-код | /target/classes |

Для того, чтобы построить проект, Maven предоставляет нам настройки для управления жизненным циклом проекта и его зависимостями. Большая часть его задач поддерживается плагинами Maven.

**Без использования Maven и других сборщиков проектов** (например Ant или Gradle), добавить сторонние библиотеки можно либо добавив библиотеку в LIB идеи, либо через переменную окружения ОС CLASSPATH и в аргументе командной строки -cp при запуске проекта java javac javadoc –cp.

В параметре передаются пути к jar-файлам и корневым директориям с пакетами. Пути разделяют символом “:” в параметре командной строки, или же “;” в переменной окружения. Чтобы включить все файлы директории, разрешается использовать в конце пути символ “\*”.

POM (Project Object Model) является базовым модулем Maven. Это специальный XML-файл, который всегда хранится в базовой директории проекта и называется pom.xml. Файл POM содержит информацию о проекте и различных деталях конфигурации, которые используются Maven для создания проекта.

Этот файл также содержит задачи и плагины. Во время выполнения задач, Maven ищет файл pom.xml в базовой директории проекта. Он читает его и получает необходимую информацию, после чего выполняет задачи.

Среди конфигураций Maven мы можем выделить следующие:

1. зависимости проекта
2. плагины
3. задачи
4. профиль создания
5. версия проекта
6. разработчики
7. список рассылки

Перед тем, как создавать pom.xml нам необходимо прежде всего определить группу проекта (groupId), его имя (artifactId) и его версию. Все это поможет нам унифицировать проект для простой его идентификации в репозитории.

# Как добавить dependency в Maven? Для чего они нужны? Откуда они скачиваются?

# Основные фазы проекта под управлением Maven?

Жизненный цикл сборки в Maven – это чётко определённая последовательность фаз во время выполнения которых должны быть достигнуты определённые цели.

В maven встроены 3 жизненных цикла сборки: clean, default и site.

1. Default цикл обрабатывает развертывание проекта. Цикл по-умолчанию, состоит из 23 фаз, основные фазы проекта:
   1. **validate** - подтверждает, является ли проект корректным и вся ли необходимая информация доступна для завершения процесса сборки.
   2. **compile** - компилирует исходный код проекта
   3. **test** - запускает модульные тесты
   4. **package** - преобразует скомпилированный код и пакет в дистрибутивный формат. Такие как JAR, WAR или EAR.
   5. **verify** - выполняет любые проверки для подтверждения того, что пакет пригоден и отвечает критериям качества
   6. **install** - устанавливает пакет в локальный репозиторий, который может быть использован как зависимость в других локальных проектах.
   7. **deploy** - копирует финальный пакет (архив) в удалённый репозиторий для, того, чтобы сделать его доступным другим разработчикам и проектам.
2. clean цикл - обеспечивает очистку проекта командой maven -clean (удаляется директория сборки target). Состоит из фаз:
   1. pre-clean
   2. clean
   3. post-clean
3. site цикл - используется для создания докладов, документации, развертывания и т.д. Генерирует веб-сайт проекта. Состоит из фаз:
   1. pre-site
   2. site
   3. post-site
   4. site-deploy

# Что такое JDBC? Какие классы/интерфейсы относятся к JDBC?

Java Database Connectivity - это стандартный API для независимого соединения языка программирования Java с различными базами данных.

JDBC решает следующие задачи:

* Создание соединения с БД.
* Создание SQL выражений.
* Выполнение SQL – запросов.
* Просмотр и модификация полученных записей.

Если говорить в целом, то JDBC – это библиотека, которая обеспечивает целый набор интерфейсов для доступа к различным БД. Для доступа к каждой конкретной БД необходим специальный JDBC – драйвер, который является адаптером Java – приложения к БД.

JDBC поддерживает как 2-звенную, так и 3-звенную модель работы с БД, но в общем виде, JDBC состоит из двух слоёв.

* **JDBC API**Обеспечивает соединение “приложение – JDBC Manager”.
* **JDBC Driver API**Обеспечивает соединение “JDBC Manager – драйвер”.

JDBC API использует менеджер драйверов и специальные драйверы БД для обеспечения подключения к различным базам данных.

JDBC Manager проверяет соответствие драйвера и конкретной БД. Он поддерживает возможность использования нескольких драйверов одновременно для одновременной работы с несколькими видами БД.

JDBC API состоит из следующих элементов:

* **Менеджер драйверов (Driver Manager)**Этот элемент управляет списком драйверов БД. Каждой запрос на соединение требует соответствующего драйвера. Первое совпадение дает нам соединение.
* **Драйвер (Driver)**Этот элемент отвечает за связь с БД. Работать с ним нам приходится крайне редко. Вместо этого мы чаще используем объекты DriverManager, которые управляют объектами этого типа.
* **Соединение (Connection)**Этот интерфейс обеспечивает нас методами для работы с БД. Все взаимодействия с БД происходят исключительно через Connection.
* **Выражение (Statement)**Для подтверждения SQL-запросов мы используем объекты, созданные с использованием этого интерфейса.
* **Результат (ResultSet)**Экземпляры этого элемента содержат данные, которые были получены в результате выполнения SQL – запроса. Он работает как итератор и “пробегает” по полученным данным.
* **Исключения (SQL Exception)**Этот класс обрабатывает все ошибки, которые могут возникнуть при работе с БД.

**Для того, чтобы установить соединение с БД, нам необходимо выполнить следующие шаги:**

1. Импортировать пакет JDBC в наш Java код.
2. Зарегистрировать JDBC драйвер.
3. Передать информацию для соединения с БД (URL, имя пользователя и пароль).
4. Создать соединение с помощью метода getConnection().

1. Все необходимые классы для работы с JDBC находятся в java.sql.\* или javax.sql.\*

2. После импорта JDBC мы должны зарегистрировать наш драйвер перед тем, как его использовать. Это процесс, с помощью которого файл класса драйвера загружается в память. После этого он может быть использован, как реализация интерфейса JDBC. Существует два способа регистрации драйвера.

**DriverManager.registerDriver()**

В этом случае мы используем статический метод DriverManager.registerDriver() для регистрации драйвера.

**Class.forName()**

Этот способ является наиболее распространённым. Мы используем метод Class.forName() для динамической загрузки класса драйвера в память, после чего происходит его автоматическая регистрация. Данный способ является предпочтительным, так как он обеспечивает конфигурируемую и переносимую регистрацию драйверов.

3. После того, как мы зарегистрировали наш драйвер, мы можем установить соединение, используется метод **DriverManager.getConnection()**.

Этот метод имеет три перегруженных варианта:

* **getConnection (String url)**
* **getConnection (String url, Properties props)**
* **getConnection (String url, String user, String password)**

Как мы видим, каждый из этих методов требует URL нашей БД. Т.е. нам необходимо передать информацию о нашей БД.

4. После того, как мы передали всю необходимую информацию, мы можем создать физическое соединение с БД, объект типа Connection. После завершения работы с соединением необходимо закрыть connection методом close.

# Для чего нужен DriverManager?

Этот элемент управляет списком драйверов БД. Каждой запрос на соединение требует соответствующего драйвера. Первое совпадение дает нам соединение.

# Что такое Statement, PreparedStatement, CallableStatement?

Взаимодействовать с БД мы можем с помощью трёх интерфейсов, которые имплементируются каждым драйвером:

* **Statement**Этот интерфейс используется для доступа к БД для общих целей. Он крайне полезен, когда мы используем статические SQL – выражения во время работы программы. Этот интерфейс не принимает никаких параметров.
* **PreparedStatement**Этот интерфейс используется в случае, когда мы планируем использовать SQL – выражения множество раз. Он принимает параметры во время работы программы.

**Запросы пре-компилируеются, может кэшировать запросы.**

* **CallableStatement**Этот интерфейс становится полезным в случае, когда мы хотим получить доступ к различным процедурам БД. Он также может принимать параметры во время работы программы.

Прежде, чем мы сможем использовать экземпляр Statement для выполнения SQL – запросов, нам необходимо создать такой экземпляр. Для этого используется метод **Connection.createStatement()**.

Для работы с БД интерфейс Statement имеет три метода, которые реализуются каждой конкретной реализацией JDBC драйвера:

* **boolean execute (String SQL)**Этот метод возвращает логическое значение **true**, если объект ResultSet может быть получен. В противном случае он возвращает **false**. Он используется для выполнения DDL SQL – запросов или в случаях, когда мы используем динамический SQL.
* **int executeUpdate (String SQL)**Этот метода возвращает количесство столбцов в таблице, на которое повлиял наш SQL – запрос. Мы используем этот метод для выполнения SQL – запросов, когда хотим получить количество задействованных столбцов, например количество данных по определенному запросу.
* **ResultSet executeQuery (String SQL)**Этот метод возвращает нам экземпляр ResultSet. Мы используем этот метод в случаях, когда мы рассчитываем получить множество объектов в результате выполнения нашего SQL – запроса. Например, при получении списка элементов, которые удовлетворяют определенным условиям.

Statement также необходимо закрывать методом close.

# Что такое sql-injection?

SQL инъекция на данный момент является не только широко распространенной уязвимостью, а еще и одной из самых опасных.

Суть уязвимости — выполнение произвольного запроса к базе данных. Запрос может быть любым: на чтение, запись, модификацию и удаление каких-либо записей. Этими угрозами все не ограничивается, так как при определённых обстоятельствах можно добраться и до чтения/записи локальных файлов или даже до выполнения кода! Все зависит от целей, которые преследует злоумышленник, от используемой системы и того, как она сконфигурирована.

Существует несколько типов SQL инъекций:

* Классическая SQL Injection — простая и легкая в эксплуатации. Позволяет злоумышленнику атаковать БД и сразу видеть результат атаки. В последнее время встречается нечасто.
* Error-based SQL Injection — чуть более сложный и затратный по времени тип атаки, позволяющий, на основе выводимых ошибок СУБД, получить информацию о всей БД и хранящиеся в ней данные. Эксплуатируется, если кто-то в спешке забыл отключить вывод ошибок.
* Boolean-based SQL Injection — одна из «слепых» инъекций. Суть атаки сводится к добавлению специального подзапроса в уязвимый параметр, на который БД будет отвечать либо True, либо, неожиданно, False. Атака не позволяет сразу вывести все данные БД «на экран» злоумышленнику, но позволяет, перебирая параметры раз за разом, получить содержимое БД, хотя для этого потребуется временной отрезок соизмеримый с содержимым БД .
* Time-based SQL Injection — следующая из «слепых» инъекций. В данном случае злоумышленник добавляет подзапрос, приводящий к замедлению или паузе работы БД при некоторых условиях. Таким образом, атакующий, сравнивая время ответа на «True» и на «False» запросы, символ за символом может получить все содержимое БД, но времени уйдет на это больше, чем в случае эксплуатации Boolean-based атаки.
* Out-of-band SQL Injection — редкий тип. Атака может быть успешна только при определенных обстоятельствах, например, если сервер БД может генерировать DNS- или HTTP-запросы, что встречается нечасто. Также, как и Blind SQL, позволяет посимвольно собирать информацию о хранящихся там данных.

**«Слив» БД**

Наиболее популярное явление, к которому приводит успешная атака — утечка всей информации, содержащейся в БД. Хорошо, если там не было ничего ценного (кроме фотографий ваших любимых котиков ^\_^), но, в зависимости от обстоятельств, чаще в БД хранится множество весьма критичных данных: учетные записи пользователей (включая пароли), номера телефонов, адреса электронной почты, а нередко номера карт, их сроки действия и прочие сопутствующие сведения.

Чем грозит? В первую очередь, репутационными потерями, ведь мало кто захочет пользоваться сервисом, из которого могут вытащить какую-либо информацию о нём. Ещё хуже, если пароли хэшировались слабыми алгоритмами, что потенциально приведет к их восстановлению. Как вы понимаете, это крайне негативно отразится на всем бизнесе.

**Deface**

Про последствия deface’а, наверняка, слышал каждый. Некоторые, даже встречали разного рода «акции», выраженные в размещении рекламы на главных страницах зачастую немаленьких и известных организаций, настройку редиректов на «вирусные сайты», замену всей главной страницы каким-нибудь мемом, а иногда даже и угрозами. Такие манипуляции зачастую организованы как раз при помощи SQL Injection-атак.  
Помимо столь грубых способов указать на проблемы безопасности, существуют более утонченные и элегантные. Например, долгое время предлагалась «услуга» по размещению номера телефона заказчика на сайте жертвы. То есть пользователь, заходя на сайт компании А, знакомился с описанием товаров и услуг, но звонил по подмененному номеру уже в компанию Б. И, что тоже немаловажно, такие «модификации» сайта замечаются далеко не сразу.

Отдельно стоит упомянуть популярные в последнее время ICO. Почему? А потому что, если кто-нибудь заменит опубликованный в официальном источнике адрес кошелька, на который собираются средства, эти средства станут собираться уже на счете злоумышленника. Такие атаки были с успехом проведены несколько раз в прошлом году.

**DoS**

Отказ в обслуживании, пожалуй, одно из самых нелюбимых бизнесом событий. Причина этого проста: возникают простои в обслуживании клиентов, которые ведут как к потере репутации, так и к упущенной выгоде. DoS вызывается просто: либо база заполняется «мусорными» записями, либо, что гораздо опаснее, она просто удаляется. Второй случай особенно интересен, если по каким-либо причинам не делались (или не проверялись!) бэкапы.

**Чтение системных файлов**

Атака встречается весьма редко и работает только при некорректной настройке системы. Суть заключается в чтении системных файлов, в которых также указаны различные настройки, учётные записи и прочие чувствительные данные. Обычно подобные атаки применяются для углубления понимания атакуемой системы и намётывания дальнейших действий с целью получения доступа к ней.

**Повышение привилегий**

Некоторых нехороших пользователей по каким-то причинам может не устраивать тот факт, что они не являются администраторами в системах. С помощью SQL инъекции у таких пользователей появляется шанс исправить данную ситуацию. Так, например, в популярной CMS Joomla существовала возможность через инъекцию вытащить Cookie учётной записи администратора. Одним запросом, кстати говоря. А небезызвестная и распространённая MS SQL может предоставить даже несколько способов повысить привилегии (в зависимости от того, как сконфигурирована СУБД).

К чему это может привести? К большим проблемам, если кратко. Последующий сценарий подобной атаки может быть любым: это и все ранее перечисленные угрозы, и дополнительно, все действия, доступные администратору (может отличаться от системы к системе).  
Повышение привилегий — одно из самых неприятных развитий событий, так как злоумышленник с правами администратора может доставить немало хлопот.

**Remote Code Execution (RCE)**

Является, пожалуй, самым опасным вектором атаки, к счастью, нечасто встречающимся. Обычно выполняется с целью получения shell’а и, следовательно, контроля над сервером целиком.

Часто RCE осуществляется уже после атаки Privilege Escalation и из-за слабых настроек прав доступа в системе, но это происходит не всегда так. Для реализации атаки злоумышленник загружает файл-зловред и либо запускает его удалённо, либо сам одним из доступных образом «цепляется» к нему.

Какие последствия могут быть? Любые, включая все ранее перечисленные. Все опять же зависит от целей злоумышленника. Кто-то начинает использовать сервер для майнинга криптовалюты, кто-то может настроить репликацию БД на «свой» сервер, а кто-то просто пойдет дальше и постарается получить управление остальными серверами в локальной сети. Последствия могут быть самыми печальными.

**Как защититься?**

Несмотря на всю серьёзность уязвимости, защита от атаки весьма проста: при разработке приложения необходимо уделить особое внимание фильтрации ввода и настройке прав. Обязательно отключайте вывод ошибок на «рабочей» системе, ни к чему пользователю такая чувствительная информация. И после каких-либо изменений в приложении проводить ASV-сканирования для гарантии. А в идеале, ввести данные действия в Software Development Life Cycle. Он же у вас уже есть, верно?

Используете готовое приложение? Всегда своевременно обновляйтесь. Часто это простое правило может уберечь от плачевных последствий.

Дополнительной мерой является установка Web Application Firewall (WAF) систем. Именно подобный класс решений заточен на борьбу с выше описанными атаками и, надо сказать, весьма хорош.

Хотите убедиться, что ваши системы надежно защищены? Или интересуетесь пилотом WAF? Обращайтесь, мы с радостью проведем аудит или поможем с тестом решения по защите от подобных атак.

# Что такое ResultSet? Как с ним работать?

Интерфейс java.sql.ResultSet представляет собой множетсво результатов, запроса в БД.

Экземпляр ResultSet имеет указатель, который указывает на текущую строку в полученном множестве.

Все методы интерфейса java.sql.ResultSet мы можем разделить на три большие группы:

* **Методы получения данных.**Эти методы используются для просмотра данных конкетной записи, на которую ссылается указатель.
* **Методы изменения данных.**  
  Методы этой группы используются для изменения данных текущей записи. Эти изменения передаются в используемую БД.
* **Методы навигации.**  
  Эти методы используются для перемещения указателя.

Курсор двигается на основе свойств ResultSet. Эти свойства указываются при создании экземпляра ResultSet.

Для орпедления этих свойтсв используются следующие методы:

* createStatement (int RSType, int RSConcurrency);
* prepareStatement (String SQL, int RSType, int RSConcurrency);
* prepareCall (String SQL, int RSType, int RSConcurrency);

Аргумент RSType определяет тип ResultSet, а второй – определяет, используется ли данный экземпляр ResultSet только для чтения, или для чтения и изменения также.

**Типы ResultSet**

Возможные типы ResultSet приведены ниже. Тип TYPE\_FORWARD\_ONLY используется по умолчанию.

Рассмотрим эти типы:

* **ResultSet.TYPE-FORWARD\_ONLY**Указатель двигается только вперёд по множеству полученных результатов.
* **ResultSet.TYPE\_SCROLL\_INTENSIVE**Указатель может двигаться вперёд и назад и не чуствителен к изменениям в БД, которые сделаны другими пользователями после того, как ResultSet был создан.
* **ResultSet.TYPE\_SCROLL\_SENSITIVE**Указатель может двигаться вперёд и назад и чувствителен к изменениям в БД, которые сделаны другими пользователями после того, как ResultSet был создан.

# Рассказать про паттерн DAO.

Data access object - паттерн, используемый для абстрагирования и инкапсулирования доступа к источнику данных. DAO слой управляет соединением с базой данных для получения и записи данных. Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов. Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO выполняет функцию адаптера между компонентом и источником данных.

**Облегчает миграцию -** Уровень объектов DAO облегчает приложению миграцию на другую реализацию БД. Бизнес-объекты не знают о деталях реализации используемых данных. Следовательно, процесс миграции требует изменений только в уровне DAO. Более того, при использовании стратегии генератора можно предоставить конкретную реализацию генератора для каждой реализации хранилища данных. В этом случае миграция на другую реализацию хранилища означает предоставление приложению новой реализации генератора.

**Централизует весь доступ к данным в отдельном уровне -** Поскольку все операции доступа к данным реализованы в объектах DAO, отдельный уровень доступа к данным может рассматриваться как уровень, изолирующий остальную часть приложения от реализации доступа к данным. Такая централизация облегчает поддержку и управление приложением.

# Что такое JPA?

JPA (Java Persistence API) это спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде. Сама Java не содержит реализации JPA, однако есть существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения java объектов в базы данных (ORM систем), но один из самых популярных в Java мире.

Спецификация JPA определяет следующее:

1. способ определения метаданных отображений – как хранимые классы и их  
   свойства соотносятся со схемой базы данных. JPA широко использует Java- аннотации в классах предметной модели, но вы можете определять отображения при помощи XML;
2. API для основных CRUD-операций, производимых над экземплярами хра- нимых классов; наиболее известен класс javax.persistence.EntityManager, используемый для сохранения и загрузки данных;
3. язык и API для создания запросов, использующих классы и их свойства. Этот язык называется Java Persistence Query Language (JPQL) и очень похож на SQL. Стандартизированный API позволяет программно создавать *запросы с критериями* без работы со строковыми значениями;
4. порядок взаимодействия механизма хранения с транзакционными сущно- стями для сравнения состояний объектов (dirty checking), извлечения ассо- циаций и выполнения прочих оптимизаций. Кроме того, в последней специ- фикации JPA рассмотрены основные стратегии кэширования.

Hibernate реализует JPA и поддерживает все стандартизированные отображения, запросы и программные интерфейсы.

# Что такое ORM?

Вкратце, объектно-реляционное отображение – это техника программирования, которая помогает обеспечивать преобразования (двустороннего) данных при их обмене между реляционной БД и Java (ЯП).

# Что такое Hibernate?

Фреймворк, предназначенный для работы с базами данных. Работает по принципу ORM (Object-relational mapping) - Объектно-реляционное связывание или работа с таблицами БД как с объектами (​​создание связи между объектом и реляционной сущностью).

Java Persistence API - спецификация, в которой описаны правила и API для реализации принципов ORM в Java.

Hibernate - конкретная реализация JPA, слой (фреймворк) между БД и приложением. С помощью hibernate, не нужно создавать прямые SQL запросы к БД через JDBC

Имеются другие реализации, такие как:

1. OpenJPA(Apache)
2. TopLink(Oracle)
3. EclipseLink

Этапы работы с Hibernate

1. Создается проект с использованием любого удобного сборщика проектов (Maven, Gradle etc.) и в файле сборщика указываются зависимости (dependencies). Для работы с Hibernate требуется как минимум, JDBC Driver (например mysql-connector-java при использовании mysql) и hibernate-core.
2. Cоздание hibernate.cfg.xml файла или properties.xml (JPA направленный), в котором указываются необходимые параметры такие как:
   1. JDBC Driver
   2. Информация для подключения к ДБ (url, username, password)
   3. Используемый диалект.
   4. Классы для маппинга.
   5. Множество других параметров работы с БД.
3. Описание связей между POJO (Plain old java object - старый добрый java-объект) и таблицей БД для создания persistent object. Persistent object - POJO, который представляет таблицу из БД. Маппинг может быть реализован 2 способами:
   1. С помощью mapping xml-файла.
   2. С помощью аннотаций в POJO классе.
4. Создается сессия Session методом openSession(), из сессии получается транзакция Transaction методом beginTransaction() и внутри транзакции выполняется запрос к базе данных. Метод commit класса Transaction позволяет завершает текущую транзакцию, записывая данные в базу данных. Метод rollback необходим для отката действий во время транзакции в случае возникновения сбоя во время транзакции, обычно rollback делают в блоке catch при возникновении ошибки в ходе транзакции.
5. Обработка результатов происходит так же внутри транзакции.

# В чем разница между JPA и Hibernate? Как связаны все эти понятия?

Hibernate одна из самых популярных открытых реализаций последней версии спецификации (JPA 2.1). Даже скорее самая популярная, почти стандарт де-факто. То есть JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

# Какие классы/интерфейсы относятся к JPA/Hibernate?

1. **SessionFactory (org.hibernate.SessionFactory)** – неизменяемый потокобезопасный объект с компилированным маппингом для одной базы данных. Необходимо инициализировать SessionFactory всего один раз. Экземпляр SessionFactory используется для получения объектов Session, которые используются для операций с базами данных. Самый тяжелый объект.
2. **Session (org.hibernate.Session)** – однопоточный короткоживущий объект, который предоставляет связь между объектами приложения и базой данных. Он оборачивает JDBC java.sql.Connection и работает как фабрика для org.hibernate.Transaction. Разработчик должен открывать сессию по необходимости и закрывать ее сразу после использования. Экземпляр Session является интерфейсом между кодом в java приложении и hibernate framework и предоставляет методы для операций CRUD.
3. **Transaction (org.hibernate.Transaction)** – однопоточный короткоживущий объект, используемый для атомарных операций. Рабочая единица работы с БД. Обрабатывается менеджером транзакций.

# Основные аннотации Hibernate, рассказать.

В Hibernate предусмотрена возможность конфигурирования приложения с помощью аннотаций.

Аннотации являются мощным инструментом для предоставления метаданных, а также намного нагляднее при чтении нашего кода другим разработчиком.

**@Entity**

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

**@Table**

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (map) данный класс. Аннотация **@Table** имеет различные атрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблице БД*.

**@id**

С помощью аннотации **@Id** мы указываем *первичный ключ (Primary Key)* данного класса.

**@GeneratedValue**

Эта аннотация используется вместе с аннотацией **@Id** и определяет такие параметры, как **strategy** и **generator**.

**@Column**

Аннотация **@Column** определяет к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса).

**@Many-to-many**

**@Many-to-one**

**@One-to-Many**

**@One-to-one**

Аннотации, определяющие взаимодействие между таблицами в базе данных.

# Чем HQL отличается от SQL?

HQL (Hibernate Query Language) – это объектно-ориентированный язык запросов, который крайне похож на SQL.

Отличие между HQL и SQL состоит в том, что SQL работает с таблицами в базе данных и их столбцами, а HQL – с сохраняемыми объектами (Persistent Objects) и их полями (атрибутами класса).

Hibernate транслирует HQL – запросы в понятные для БД SQL – запросы, которые и выполняют необходимые нам действия в БД.

Мы также имеем возможность использовать обычные SQL – запросы в Hibernate используя Native SQL, но использование HQL является более предпочтительным.

**Основные ключевые слова языка HQL:**

**FROM**

Если мы хотим загрузить в память наши сохраняемые объекты, то мы будем использовать ключевое слово **FROM**.

**INSERT**

Мы используем ключевое слово **INSERT**, в том случае, если хотим добавить запись в таблицу нашей БД.

**UPDATE**

Ключевое слово UPDATE используется для обновления одного или нескольких полей объекта.

**DELETE**

Это ключевое слово используется для удаления одного или нескольких объектов.

**SELECT**

Если мы хотим получить запись из таблицы нашей БД, то мы должны использовать ключевое слово **SELECT**. Пример использования:

Query query = session.createQuery("SELECT D.lastName FROM Developer D");

List developers = query.list();

**AS**

В предыдущем примере мы использовали запись формы Developer D. С использованием опционального ключевого слова **AS** это будет выглядеть так:

Query query = session.createQuery("FROM Developer AS D");

List developers = query.list();

**WHERE**

В том случае, если мы хотим получить объекты, которые соответствуют определенным параметрам, то мы должны использовать ключевое слово **WHERE**.

**ORDER** **BY**

Для того, чтобы отсортировать список объектов, полученных в результате запроса, мы должны применить ключевое слово **ORDER BY**. Нам необходимо указать параметр, по которому список будет отсортирован и тип сортировки – по возрастанию (ASC) или по убывани (DESC).

**GROUP BY**

С помощью ключевого слова **GROUP BY** мы можем группировать данные, полученные из БД по какому-либо признаку.

**17. Что такое Query? Как передать в объект Query параметры?**

Query - объектно-ориентированная реализация запроса hibernate. Экземпляр Query получается путем вызова Session.createQuery();

Запросы выполняются путем вызова list (), scroll (), iterate () или executeUpdate(). Запрос может быть повторно выполнен при последующих вызовах. Однако его продолжительность жизни ограничена продолжительностью жизни объекта Session, который его создал.

Параметры в объект Query можно передать через сеттеры setParameter();

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

**hibernate.hbm2ddl.auto** - свойство, обеспечивающее проверку или экспорт ddl схемы при создании SessionFactory

**create -** При запуске приложения будет создавать схему заново. Все данные которые были в старой схеме потеряются.

**update -** При запуске приложения будет осуществляться сверка имеющейся в БД схемы с классами приложения. Если были изменения, то схема обновится, а данные останутся.

**create-drop -** Аналогично create, только после завершения схема с данными будет удаляться. Удалит схему, когда SessionFactory закрывается явно, обычно, когда приложение остановлено.

**validate -** Будет проверять соответствие схемы и классов, если соответствия нет, то выбросится исключение.

**None -** Никаких действий не будет

# Требования JPA к Entity-классам? Не менее пяти.

1. Класс должен быть помечен аннотацией @Entity из javax.persistence.\* или отображаться как класс для маппинга в xml файле.
2. Класс сущности должен иметь публичный или защищенный конструктор без аргументов. Он также может иметь и дополнительные конструкторы с аргументами.
3. Класс сущности должен быть верхнего уровня (top-level), т.е. не вложенным
4. Не должен быть интерфейсом или перечислением(Enum)
5. Класс не должен быть final. Также поля и методы не должны быть final.
6. Должен реализовывать интерфейс Serializable, если экземпляр класса используется удаленно как отдельный объект.
7. Класс может быть абстрактным. Entity класс может наследовать non-entity, также как и non-entity может наследовать entity класс.
8. Должен содержать первичный ключ, т.е. атрибут или группу атрибутов, которые определяют уникальность entity класса в БД.

# В чем разница в требованиях к Entity в Hibernate, от требований к Entity, указанных в спецификации JPA (см. вопрос 10)?

1) Конструктор без аргументов не обязан быть public или protected, рекомендуется чтобы он был хотя бы package видимости, однако это только рекомендация, если настройки безопасности Java позволяют доступ к приватным полям, то он может быть приватным,  
2) JPA категорически требует не использовать final классы, Hibernate лишь рекомендует не использовать такие классы чтобы он мог создавать прокси для ленивой загрузки, однако позволяет либо выключить прокси [Proxy](https://habrahabr.ru/users/proxy/)(lazy=false), либо использовать в качестве прокси интерфейс, содержащий все методы маппинга для данного класса (аннотацией [Proxy](https://habrahabr.ru/users/proxy/)(proxyClass=интерфейс.class) )

# Жизненный цикл Entity в Hibernate? Рассказать.

У Entity объекта существует четыре статуса жизненного цикла: transient, persistent, detached, или removed. Их описание:

**Transient** - состояние, при котором объект никогда не был связан с какой-либо сессией и не является персистентным(не имеет представления в БД). Этот объект находится во временном состоянии. Объект в этом состоянии может стать персистентным при вызове метода save(), persist() или saveOrUpdate(). Persistent объект может перейти в transient состояние после вызова метода delete().

**Persistent** - когда объект связан с уникальной сессией он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load() находится в состоянии persistent.

**Detached** - если объект был персистентным, но сейчас не связан с какой-либо сессией, то он находится в отвязанном (detached) состоянии. Такой объект можно сделать персистентным используя методы update(), saveOrUpdate(), lock() или replicate(). Состояния transient или detached также могут перейти в состояние persistent как новый объект персистентности после вызова метода merge().

**Removed** - объект создан, управляется JPA, но будет удален после commit'a транзакции.

# Транзакции.

Транзакция это набор операций, которые могут быть либо целиком и успешно выполнены, либо полностью не выполнены.

Транзакции в базах данных соответствуют свойствам ACID:

* Атомарность — транзакция может быть либо целиком выполнена, либо целиком отменена.
* Согласованность — состояние данных должно быть логически согласованным после выполнения (или отмены) транзации
* Изолированность — в процессе работы транзакции другие выполняющиеся в это время транзакции не влияют на неё.
* наДёжность — что-бы не произошло, транзакция останется атомарной.

Если вдруг транзакции в вашей базе не соответствуют этим принципам, то это плохие, негодные транзакции и они не заслуживают чести носить это имя.

Использование транзакций в дикой природе проще показать на примере: положим у нас есть система, в которой клиенты размещают какие-то заказы. За каждый заказ со счёта клиента снимается какая-то сумма. Таким образом, одна логическая единица работы разбивается на три операции в базе данных:

* Добавить заказ.
* Получить текущее значение счёта клиента.
* Уменьшить счёт клиента и обновить значение в базе.

Оборачивание этих трёх операций в одну транзакцию гарантирует нам, что либо мы добавим заказ и спишем деньги, либо ничего не произойдёт вообще. Третьего не дано.

# Транзакции в [Hibernate](https://easyjava.ru/tag/hibernate/" \o "Posts tagged with hibernate)

Hibernate построен поверх [JDBC API](http://easyjava.ru/data/jdbc/) и реализует модель транзакций JDBC. Если быть точным, Hibernate способен работать или с JDBC транзакциями или с JTA транзакциями.О JTA — Java Transaction API я напишу как-нибудь позже, а пока сосредоточимся на JDBC транзакциях, тем более что с точки зрения использования их отличий не так и много.

Транзакцию можно начать вызовом beginTransaction() объекта Session, либо запросить у Session связанный с ней объект Transaction и позвать у последнего метод begin(). С объектом Session всегда связан ровно один объект Transaction, доступ к которому может быть получен вызовом getTransaction()

|  |  |
| --- | --- |
| 1  2 | Session session = sessionFactory.openSession();  Transaction t=session.getTransaction(); |

Методов для подтверждения или отката транзакции у объекта Session нет, необходимо всегда обращаться к объекту Transaction

# Операции над транзакциями

У объекта Transaction есть ещё несколько методов, кроме commit() и rollback(), которые позволяют тонко управлять поведением транзакции. Метод isActive() позволяет проверить, есть ли в рамках объекта Transaction управляему им транзакция. Очевидно, что такая транзакция существует в промежутке времени между вызовами begin() и commit()/rollback().

Метод setRollbackOnly() помечает транзакцию как откаченную в будущем. В отличие от rollback() этот метод не закрывает транзакцию и все последующие запросы к базе будут продолжать выполняться в рамках той же самой транзакции, но завершить эту транзакцию можно будет только откатом и вызовом rollback(). Вызов commit() на такой транзакции выбросит исключение. Проверить состояние транзакции можно вызовом getRollbackOnly().

# Что такое конфигурационный файл Hibernate?

Файл конфигурации Hibernate содержит в себе данные о базе данных и необходим для инициализации SessionFactory. В .xml файле необходимо указать вендора базы данных или JNDI ресурсы, а так же информацию об используемом диалекте, что поможет hibernate выбрать режим работы с конкретной базой данных.