Hibernate

Источник: DMdev

**Введение**

Проблемы, которые решает Hibernate:

Несоответствие объектно-ориентированной модели и реляционной

Ручное однотипное написание простейших SQL запросов (для CRUD, например)

Отложенная загрузка данных (это тоже уже реализовано в Hibernate)

Управление транзакциями (сложная логика при исп. JDBC)

и др.

ORM – процесс преобразования объектно-ориентированной модели в реляционной. Hibernate это автоматизирует.

**Session**

Session в Hibernate это что-то вроде оболочки над Connection из JDBC. SessionFactory – это что-то вроде ConnectionPool. Должен создаваться только 1 объект SessionFactory и он должен закрываться при завершении работы с приложением. Session тоже необходимо закрывать.

Конфигурацию ConnectionPool мы задавали в файле properties. По аналогии мы задаём конфигурацию SessionFactory в hibernate.cfg.xml. А потом с помощью объекта Configuration создаём объект SessionFactory. На основе настроек, которые мы прописали в конфиг. файле и кучи разных дефолтов, прописанных в классе Configuration.

Session не autocommit и нужно вызвать beginTransaction(); и endTransaction();

У Hibernate отложенная отправка запросов. Т.е. он максимально отодвигает момент, когда нужно открыть транзакцию и начать общаться с БД. Это для оптимизации. Запрос выполниться только тогда, когда мы сделаем коммит у транзакции или будем закрывать сессию.

Во всех запросах будет использоваться идентификатор (@Id)

Чтобы сохранить сущность в БД, просто вызываем session.save(\*entity\*)

session.update(\*entity\*) выдаст exception, если сущности нет в БД.

session.delete(\*entity\*) не выдаст exception, если сущности нет в БД

session.get(\*entity class\*, идентификатор)

Ещё основные , saveOrUpdate, createQuery

**Entity**

Сущности – то, что мы будем проецировать на таблицы в БД.

Для того, чтобы класс был Hibernate сущностью должы быть соблюдены некоторые правила:

1. Сущность должна быть POGO т.е. все поля private + get и set и к ним.
2. Сущность должна быть не immutable т.е. поля не final, и сам класс не final
3. Должен быть конструктор без параметров

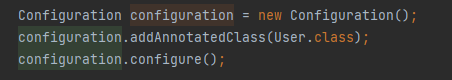
Т.е. чтобы это всё создать придётся написать класс с полями, геттерами, сеттерами, пустым конструктором. Возможно, нам ещё понадобится полный конструктор и hashCode и equals. Это много однотипного кода (boilerplate code). От его написания можно убежать, используя Lombok. Для примера выше используя @Data, @NoArgsConctructor, @AllArgsConstructor.

Чтобы обозначить, что POGO – сущность, нужно аннотировать класс аннотацией из JPA @Entity. Но каждая сущность должна иметь первичный ключ. Для его обозначения используем аннотацию @Id у конкретного поля. Ещё поле @Id должно быть Serializable.

По умолчанию Hibernate берёт названия классов/полей в качестве названий таблиц/колонок в БД. С помощью аннотации @Table можно указать название таблицы, схемы и т.д. с чем будет маппиться.

Наши entity нужно добавить в SessionFactory.

Добавляем через так:



Или прописываем маппинг в xml:



Различия в именовании столбцов можно нивелировать, используя, например:



Или же для каждой колонки отдельно, используя @Column

**Type converters**

По сути, это прослойка между типами Java и типами SQL. В configuration есть список basicTypes – это список всех типов, которые поддерживает Hibernate, и он может преобразовывать тип в типы из пакета java.sql, знает как назначать этот тип в PrepareStatment и получать обратно этот тип из ResultSet. Соответственно, нельзя просто написать свой тип и без написания доп. кода его использовать. Нужно написать свою реализацию для Type с нашим типом либо реализацию для UserType.

Есть вариант попроще. Написать свой Converter, который умеет преобразовывать наш тип в один из классов java.sql и наоборот.

Написали реализацию AttributeConverter и нужно подсказать Hibernate, что для этого поля используй этот конвертер. Для этого помечаем поле @Convert(converter = \*converter\*.class) или объекту configuration указываем configuration.addAttributeConverter(new BirthdayConverter(), true) чтобы он автоматически применял ко всем полям такого типа этот конвертер (думаю, он из джинериков понимает к какому типу какой конвертер).

Однако, может быть ситуация, в которой типа из нашей БД, не будет в java.sql. В таком случае не получится написать конвертер т.к. не во что конвертировать и нужно создать свой тип. Или нам потребуется любая другого рода гибкость, когда мы хотим не просто отображать свой тип на java.sql тип. Например, в posgresql есть тип JSON, а в Hibernate basicTypes его нет. Можно написать свою реализацию для Type или UserType, но это сложно так что есть специальные библиотеки, например HibernateTypes, где можно поискать реализацию Type для необходимого типа.

Когда имеется Type мы должны указать его для поля c помощью @Type(type = \*полный путь к типу\*). 

Но чтобы не указывать полный путь к типу, у Type есть метод getName(). Это псевдоним для полного пути.





Если он не задан у Type или ещё что-то, можно использовать аннотацию @TypeDef(name = “barabJson”, typeClass = JsonType.class). Эту аннотацию можно использовать и для всего пакета.

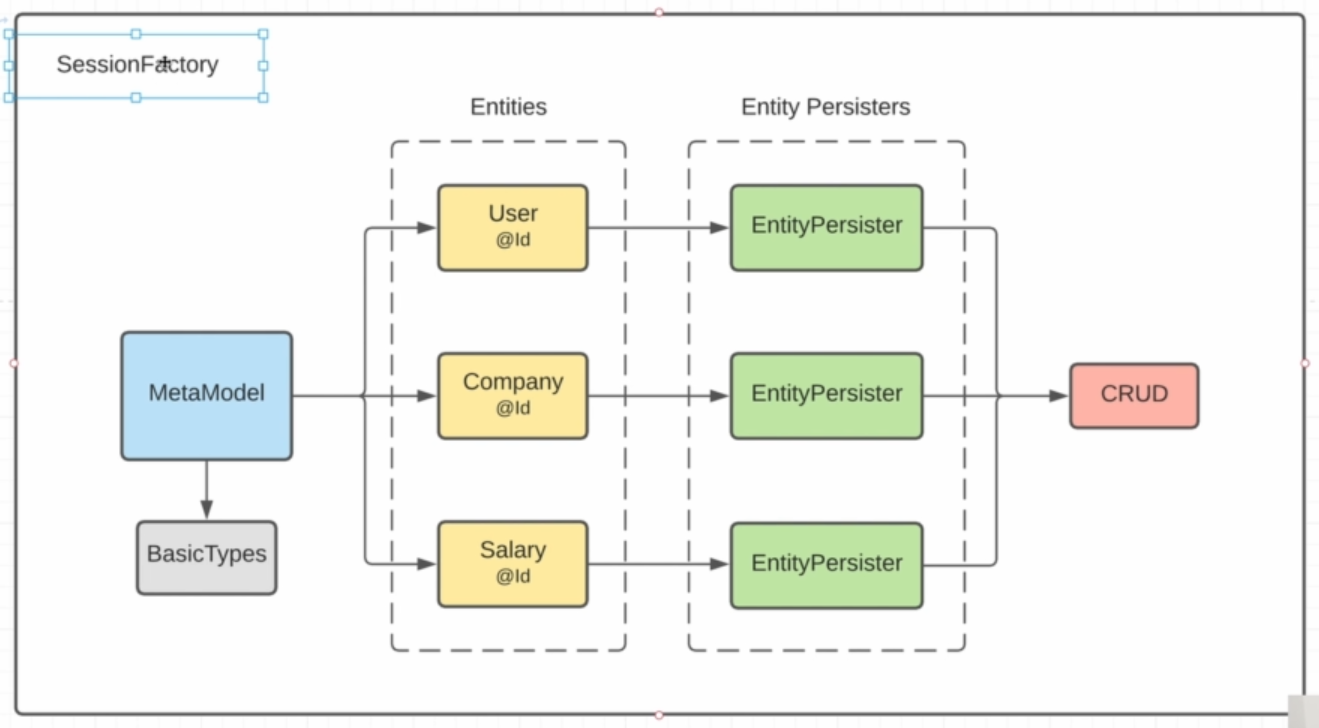
Также нужно зарегистрировать этот Type в Hibernate



**EntityPersister**

Именно эта сущность занимается тем, что загружает, обновляет, удаляет, получает сущность из БД. Т.е. это и есть маппинг между реляционной моделью и объектно ориентированной. Соответственно делает она это, при помощи Type, Converter политик именования, аннотаций и т.д.

Получается следующая модель:



Для каждой из сущностей даётся свой entityPersister, который знает как выполнять соответствующие запросы, например CRUD, для наших сущностей и знает как выполнять преобразования моделей.

**First Level Cache**

У Session есть persistenceContext. Его и называют firstLevelCache. У каждой сессии, свой persistenceContext,.

У persistenceContext есть entitiesByKey – map<EntityKey, Object>.

EntityKey содержит:

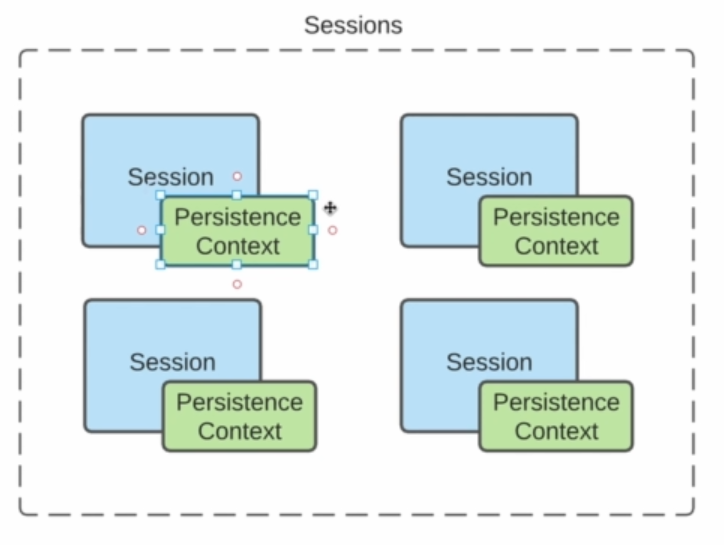
* identifier (entity id)
* hashCode
* EntityPersister

Методы Session get, save и д.т. кладут в persistenceContext объекты, после того, как они сделали запрос к БД. По сути persistenceContext является проекцией на состояние нашей БД. И, например, если мы запрашиваем объект с Id и классом, который уже запрашивали, повторного запроса к бд не будет. Объект выдастся нам из persistenceContext.

Если мы хотим удалить объект из кэша, используем:

* session.evict(\*object\*);
* session.clear();
* И при закрытии сессии объекты тоже удалятся (логично)

Если мы будем изменять полученный от сессии объект (который хранится в persistenceContext), эти изменения Hibernate автоматически отразятся и в БД (сделает update), даже если мы не вызовем update у session. Это понятие DirtySession. Есть метод проверки session.isDirty(). Метод session.flush() можно вызвать руками, чтобы слить изменения в объектах в БД.

Схематично SessionFactory выглядит так:

И у каждого объекта сессии может быть своё состояние сущности. Следовательно, состояние сущности может быть разное, в зависимости от сессии.

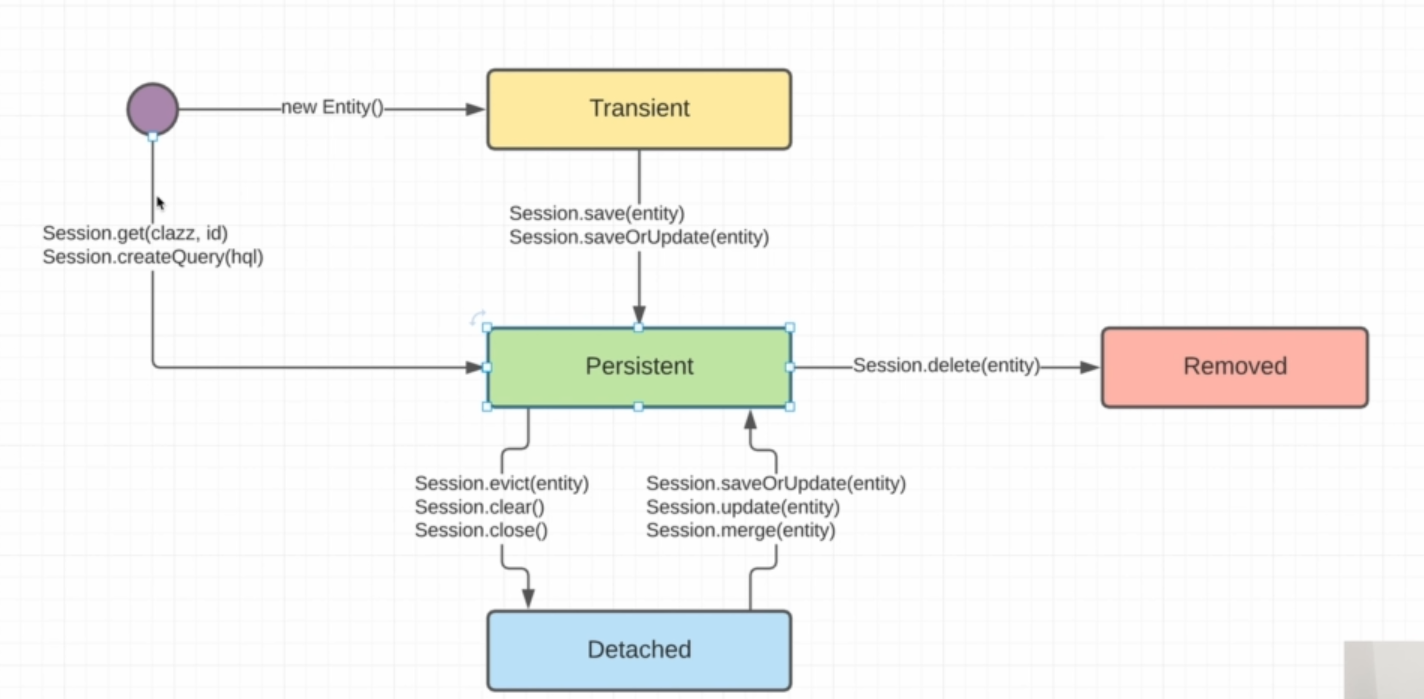
**Entity Lifecycle**

У каждой сущности есть 4 основных состояния:

* Transient (ни с какой сессией не проассоциирована)
* Persistent (сущность проассоциирована с каким-либо PersistentContext)
* Detached (все методы из Detached при переводе в Persistent вызывают метод get)
* Removed (При session.delete сущность удаляется из БД и из PersistentContext)

Также когда мы делали update() объект помещался в PersistentContext, но это потому что выполняется неявно метод get(), чтобы получить сущность из БД и проассоциировать эту сущность с PersistentContext и далее уже выполнить метод update.

Схематично это выглядит так:



Соответственно по отношению к разным сессиям одна и та же сущность может быть в разных состояниях.

session.refresh(\*entity\*) – делает select запрос к БД и перезаписывает все поля в entity на значения, записанные в БД, для этого entity. Переводит entity в состояние Persistent.

session.merge(\*entity\*) – обновляет информацию в БД на основе переданного объекта. Merge делает select запрос, получает объект (соответственно этот объект попадает в PersistentContext) потом перезаписывает поля этого объекта на такие же значения, как и в user. возвращает результат – обновлённый объект. Он имеет состояние Persistent. Merge не меняет передаваемый ему объект. Если в метод merge() передать объект уже присоединенный к сессии (со статусом Persistent), то ничего не произойдет – метод просто вернет этот же объект.

**JPA**

Java Persistence API (JPA) – спецификация Java, которая представляет набор интерфейсов/аннотаций для возможности сохранять в удобном виде Java объекты в БД и, наоборот, извлекать информацию из БД в виде java объектов(ORM).

Hibernate – (ORM framework) – это одна из самых распространённых JPA реализаций.

В Hibernate разработчики решили дать свои, более удобные для них имена методов, однако приходится придерживаться спецификации JPA т.к. Hibernate это реализация и он обязан реализовывать эти методы интерфейсов (javax.persistence).

Примеры:

* get – find
* evict – detach
* save – persist
* delete – remove

Все эти методы работают аналогично. Некоторые из них даже реализованы просто через вызов других.

**Логирование (Logging)**

Без систем логирования, если что-то пойдёт не так, мы никак не узнаем об этом. А решить такую проблему почти нереально.

Логирование:

Console - выводить в консоль. Самый простой вариант, но при работе с удал. сервером не прокатит)

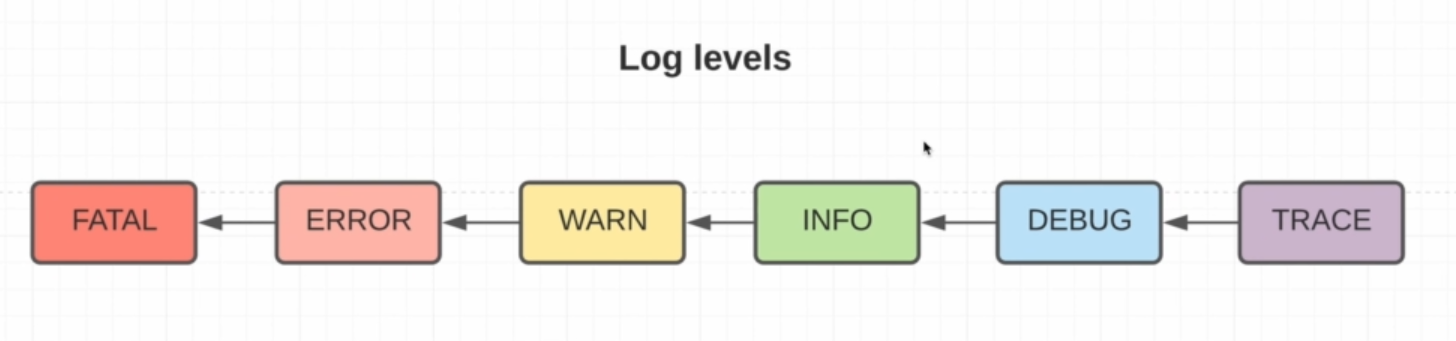
File - писать ошибки, stack trice и т.д. в файл. Очень большие файлы, неудобно.

Database – можно доставать обычным запросом Error или Stack Trice для этих Error или Warn – удобно.

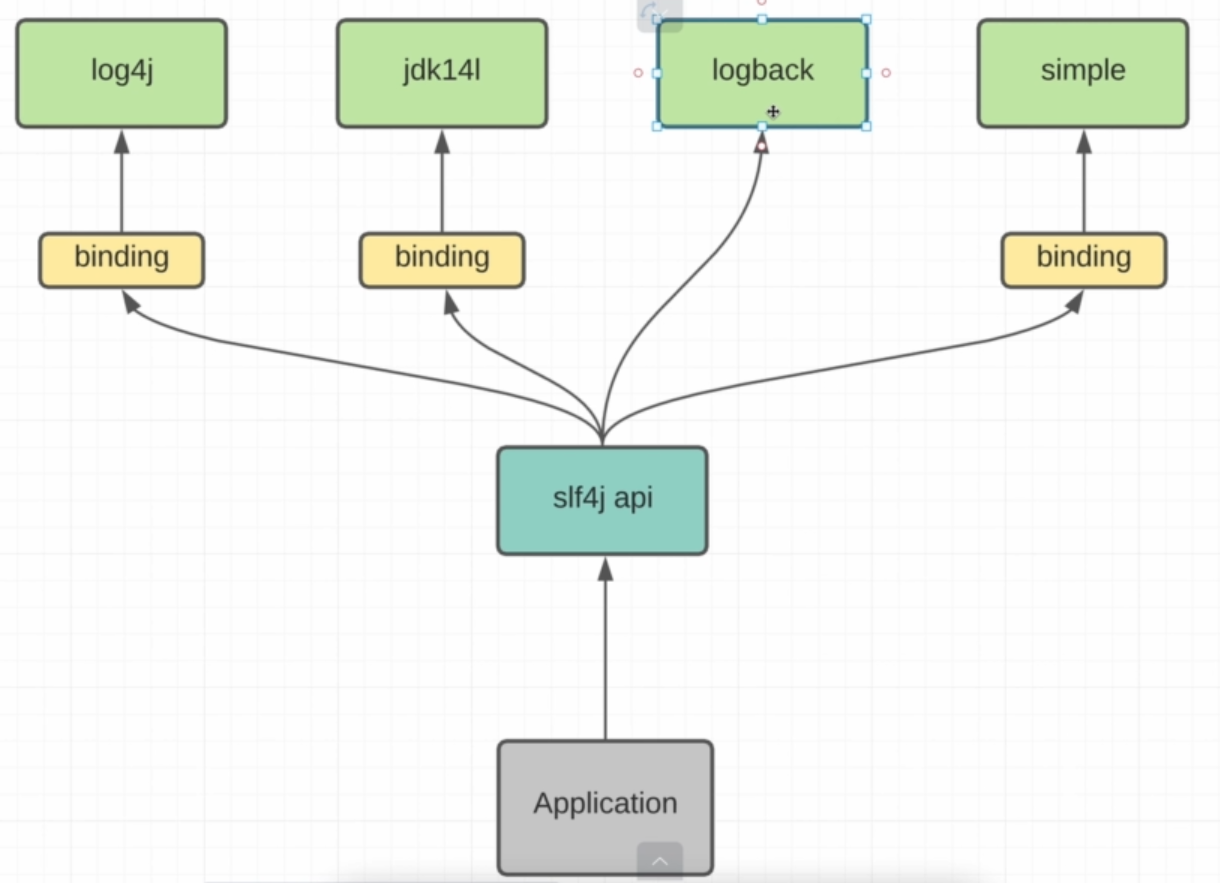
TCP/IP – в реальных приложениях отправляют сообщения отправляют УД, который уже занимается как анализировать логи или метрики.

Используем slf4j api и выбираем для него конкретную реализацию. binding – адаптер, чтобы использовать систему логирования через slf4j (отдельная зависимость из mvn. Но он имеет транзитивную зависимость от slf4j, так что его можно отдельно не подключать).

Когда мы генерируем сообщение для лога, мы присваиваем сообщению соответствующий уровень.

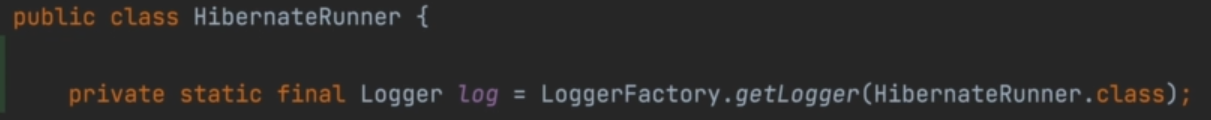


Когда мы подключаем систему логирования, мы должны выставить уровень, который хотим отображать.



Каждый из логеров конфигурируется соответствующим файлом. Для log4g должен быть log4g.xml. Основное что мы там описываем – appender и logger.

Appender берет лог сообщения и отправляет его туда, куда мы укажем. А формат этого сообщения мы задаём с помощью layout. root – это сам логгер. Logger может быть несколько, как и Appender, но его суть в том, что мы задаём соответствие между log level с которого мы начинаем логировать сообщения и аппендером куда мы будем отправлять сообщения. Можно указывать несколько аппендеров логгеру.

Каждый логгер нужно создавать в отдельном классе и при создании указывать этот класс, чтобы в логах была информация о классе, в котором произошла ошибка. Есть @SLF4J в Lombok, которая сгенерирует такую же строчку (есть и конкретные реализации @LOG4J).

далее используем log.error(), log.trace() и т.д.

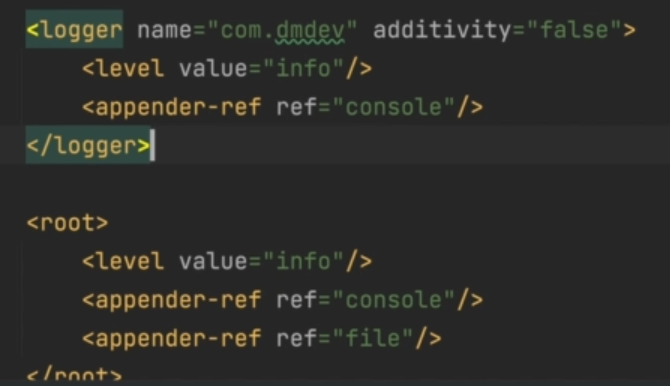
Хорошим правилом является не конкатенация строк, а передавать объекты отдельными аргументами. Сколько {}, столько и объектов должно быть.



Ошибки можно записать в лог и прокинуть дальше, чтобы не зажевало приложение.



Аппендеров может быть сколько угодно, как и логгеров. Также один логгер может использовать несколько аппендеров. Логгер <root> главный и он обязательно должен быть. Можно создавать свои логгеры через <logger>. name логгера определяется по пакету, который мы хотим отслеживать. Например, name = com.barabanov. И тогда все лог – сообщения, которые есть в этом пакете, будут попадать в этот логгер. Но если и root и наш логгер используют один аппендер, сообщения будут дублироваться. Чтобы этого избежать используем параметр additivity=false.

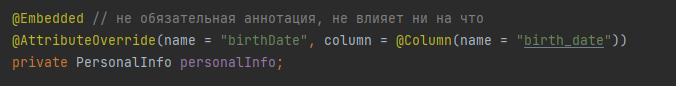


Но обычно много аппендеров и 1-2 логгера.

**Embedded components**

Встраиваемые компоненты. Это способ выделить из сущности какой-то компонент в отдельный объект так, чтобы и этот компонент, и сущность были сопоставлены с одной таблицей. Пусть у нас будут поля, отвечающие за персональные данные у User. Для удобства, возможности повторного переиспользования и т.д. мы вынесем их в класс PersonalInfo. Пометим его аннотацией @Embeddable. К классу встраимого компонента применяются все те же требования, что и к сущности.

Если имя поля у Entity отличалось от имени столбца мы использовали аннотацию к полю @Column(name = “”). Если отличается имя поля у встраиваемого компонента от солонки в таблице для этого entity, то используем аннотацию @AttributeOverride. Эта аннотация repeatable так что можно переименовать все несоответствующие поля.



**Primary key**

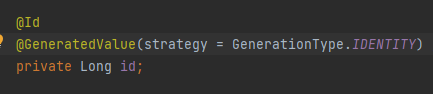
Натуральный первичный ключ:



В реальных приложениях не так часто встречается. Производительность Hibernate лучше с синтетическими (авто генерируемыми) первичными ключами. Также бизнес-логику легче менять с синтетическими ключами.

**IDENTITY**

При управлении генерацией значения id самой БД (используя BIGSERIAL) не достаточно просто @Id пометить поле. Нужно также указать, чтобы Hibernate не вставлял значение в эту колонку т.к. БД сама вставит. Для этого ставим аннотацию @GeneratedValue(strategy = GenerationType.IDENTITY)

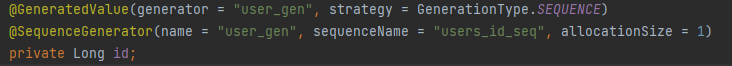


При saveOrUpdate и т.д. будет выполняться сразу insert (но выполниться update, если будет DirtySession).

**SEQUANCE**

Другой вариант – создать sequence. Они есть не во всех СУБД. В MySQL их нет. При вставке используется функция для получения нового значения из этого sequence. В первом варианте тоже используются sequence, только это делает СУБД сама, без нашего участия. Но мы можем и сами создать sequence и указать Hibernate какой sequence использовать для генерации id.

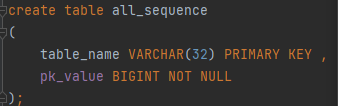
Пример: создали sequence с именем users\_id\_seq, Использовали аннотацию @SequanceGenerator чтобы познакомить Hibernate с этим sequence. В @GeneratedValue указали какой SequanceGenerator используем и указали стратегию sequence. Тут тип данных id в таблице будет уже просто INT или что-то такое. Hibernate будет сначала делать запрос к этому sequence, а потом вставлять значение, полученное из sequence в поле id таблицы сам, а не БД будет его подставлять как в первом варианте.



При использовании, например, saveOrUpdate будет выполняться запрос к sequence и устанавливаться id сущности т.к. после вызова этого метода у сущность будет Persistence, а сущность не может быть Persistence без id. Несмотря на это, сам insert сущности, как и раньше будет выполняться отложено, как и раньше (flush, коммит транзакции).

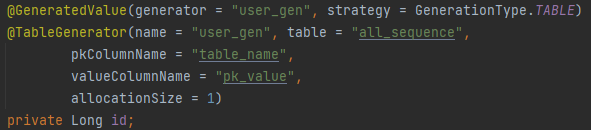
**TABLE**

Такая стратегия самая редкая. Она используется только если хотим сделать тоже самое, что и в **SEQUANCE,** но sequence нет в СУБД. При использовании этой стратегии мы должны предоставить такую таблицу:



В ней для всех сущностей будет храниться текущее значение их первичного ключа.

Познакомили Hibernate с таблицей для хранения счётчиков ключей и показали какие колонки для чего использовать, используя @TableGenerator. Задали стратегию Table



В этой стратегии тоже будет при saveOrUpdate и т.д. будет только с id работа, а сам insert отложенный. Только в этой стратегии Hibernate будет ещё и сам insert / update делать в таблицу с ключами при работе с id.

Эту стратегию использовали до создания sequence и до создания авто генерируемых ключей.

**Embedded ID**

EmbededId – можно использовать Embedded компонент как ID, если Primary key в БД является составным. Для этого помечаем сущность не как @Embedded, а как @EmbeddedId. Также для этого сущность должна быть Serializable. У session будут использоваться всё те же методы и также передаваться ключ в get, однако работать с такими ключами гораздо сложнее и медленнее. Рекомендуется использовать всегда синтетические ключи и стратегию INDENTITY.

**Оставшиеся базовые аннотации при исп. Hibernate**

@Transient – если не хотим, чтобы это поле сохранялось в БД и извлекалось из неё. (но в сущности, на практике лучше хранить только те поля, которые мапятся на поле в БД)

@Temporal(TemporalType.\*\*) - до Date time api (LocalDateTime, LocalTime, LocalDate) был только Date в java, а в SQL больше типов даты было. С помощью этой аннотации можно указать на какой конкретно тип в SQL мапить. Но не стоит вообще использовать Date, т.к. есть Local…

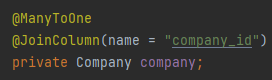
@ColomnTransformer – можно переопределить read, write там выполнять какие-то функции СУБД в БД, например, к значению перед чтением / записью.

@Access(AccessType.\*\*) – есть Field(по умолчанию) и Property. Означает, что Hibernate будет исп. Reflection API для получения доступа к нашим полям. Раньше писали все get() set() вручную и аннотации @Column и т.д. применялись именно к методам, а не к полям. И Hibernate использовал эти методы, а не Reflection API. Но аннотировать поля меньше загромождает код, так удобнее и предпочтительнее.

**Mapping Entity Associations**

**Many to one**

Мы хотим встроить Одну сущность в другую, а не просто id этой сущности. Например, у User поле Company в которой он работает. Для этого записываем поле Company и помечаем его аннотациями @JoinColumn(name = \*имя колонки в таблицу user, которое используется для маппинга\*). По умолчанию это name = названиесущности\_названиепервичного ключа т.е. company\_id.



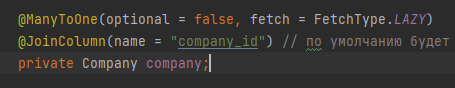
Также нужно пометить тип взаимоотношений у сущностей. Много пользователей у одной компании – ManyToOne. Hibernate обрабатывает это с помощью специального Type – ManyToOneType.

**INNER JOIN**

По умолчанию при таких параметрах происходит users LEFT OUTHER JOIN company. Но возможно мы захотим делать INNER JOIN т.к. он быстрее, а разницы с LEFT OUTHER не будет т.к. у company\_id есть ограничение NOT NULL. Без такого ограничения параметр optional должен быть true (иначе при INNER отбросятся user у которых company null, а это вряд ли то, что нам нужно). Итак, если есть CONSTRAINT NOT NULL, то для улучшения производительности пишем optional = false. И Hibernate будет делать INNER JOIN.

**Отдельный запрос на получение вложенной сущности**

Чтобы был отдельный SELECT на получение Company, а не JOIN, устанавливаем параметр fetch как LAZY. По умолчанию для не коллекций используется EAGER т.е. запрашивается сразу (с JOIN), а для коллекций LAZY т.е. запрос к БД происходит только при первом обращении к полю этой сущности.



**Hibernate proxy**

Существуют 2 типа прокси объектов:

* через интерфейсы ()
* через наследование

Реализация proxy через наследование на примере класса Company:

Наследуемся от Company. Благодаря этому шагу мы сможем передавать объект proxy туда, где требуется Company, но из-за этого у proxy будут поля id и name, унаследованные от Company всегда null.

Реализуем интерфейсы HibernateProxy, ProxyConfiguration для работы нашего объекта как proxy в Hibernate.

В общем работает это так:

У прокси, помимо полей от Company есть поле ByteBuddyInterceptor. Он хранит класс, который потом нужно будет вернуть, идентификатор, по которому можно будет запросить потом Company, Сессию в которой он будет это запрашивать и некоторые другие.

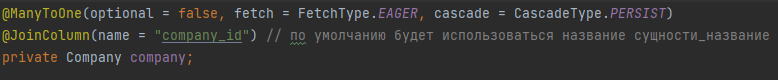
При запросе, например id, объект Component всё ещё не будет проинициализирован т.к. id объекта знает и ByteBuddyInterceptor. А при запросе уже getName() будет инициализация сущности внутри т.к. name ByteBuddyInterceptor уже не хранит. Внутри есть флаг, который сигнализирует при следующем запросе к другому полю что объект уже проинициализирован и ему просто перенаправляются запросы. Если при использовании proxy закрыть сессию, на которую в этом прокси хранится ссылка для запроса к БД будет LazyInitializationException;

Есть несколько утилитных методов для извлечения target из proxy. Hibernate.unproxy(\*proxy или просто объект\*)

**Cascade types**

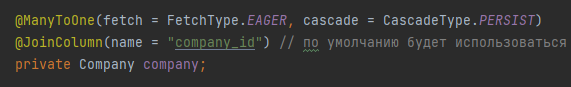
Это параметр у аннотаций @ManyToMany и т.д. Если задать, например cascade = CascadeType.DETACH, то при переводе сущности в состояние detach, сущности связанные с ним будут каскадно тоже переводиться в это состояние. Т.е. сделали session.delete(user) удалится и company на которую в нём хранится ссылка.

То, что написано внизу работать не будет.

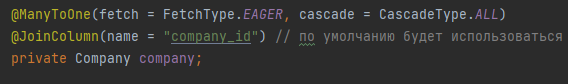


optional = false т.е. companyId не может быть null, но при вызове save(user) сначала сохраниться user, а потом company т.е. при сохранении user company\_id будет null.

Даже это работать не будет. По той же причине. Запись в company на которую ссылается запись в users уже должна быть, когда записывается в users.



Но вот это будет работать. Поскольку при исп. ALL сначала сохраняются все связанные с user сущности, а уже потом user (так себе обходной путь, просто особенность описана).



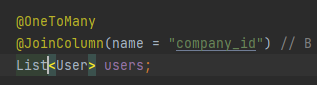
Но, по сути, cascade не используют при ManyToOne. Его используют часто при OneToMany.

**One to Many**

Типа OneToManyType нет, есть CollectionType и его наследники, ведь мы используем какие-то коллекции для такой связи. SetType, MapType и т.д.

Одна Company относится ко множеству User. Для этого у Compony коллекция из User. Для такой ассоциации использовать можно любую коллекцию.

Далее маппинг OneToMany можно реализовывать двумя различными способами. Используя @JoinColumn(name = company\_id) у Company нет полей для связи с users, так что связываем их через поле в users. Такой тип маппинг называется unidirectional. Для этого маппинга не обязательно, чтобы был маппинг ManyToOne.



Однако часто связь ManyToOne есть (т.е. User знает какой company принадлежит) и мы уже на эту связь добавляем связь OneToMany. Такая связь называется bidirectional. И в таком случае можно не указывать JoinColumn, а просто указать mappedBy, где просто указываем поле в User на Company.

Работа аннотации @OneToMany, аналогична работе аннотации @ManyToOne. Т.е. есть всё те же аргументы у аннотации fetch (по умолчанию Lazy, а не Eager), cascade и т.д.

поле users у Company будет иметь тип PersistenceBag. Это реализация LazyCollection, аналог Hibernate Proxy, но не для одной сущности, а для коллекции.

При bidirectional связи может возникнуть цикл. Например, при добавлении user в Set<User> внутри compony вызывается equals у user, который при сравнении двух user вызывает equals у Compony, который содержится внутри User и так по кругу. ->StackOverflow. Для того, чтобы этого избежать чаще всего у сущности, в которой есть связь @OneToMany или @ManyToMany мы исключаем из toString, equals hashCode поля с такими аннотациями. В общем, когда есть bidirectional связь, исключаем поля ассоциаций из этих 3 методов.

**Cascade types with collections**

При использовании OneToMany скорее всего захотите использовать CasacadeType = All, чтобы все сущности, связанные с этой сущностью, также сохранялись и удалялись вместе. На примере Company и User: чтобы при удалении компании удалились и все сотрудники, которые содержатся в этой компании. Можно поймать ошибку, если над OneToMany не будет CascadeType, а над ManyToOne будет. User удалится, удалит Company и останутся User’ы без company в БД, чего быть не может т.к. NOT NULL. Но если ALL указать над ManyToOne тоже, то работать будет опять же.

**Использовать Каскадное удаление лучше на уровне SQL**. Это быстрее по производительности и проще.

**OrphanRemoval**

Свойство @OneToMany и @OneToOne. Это свойство отвечает за удаление сущности из БД при удалении ссылки на него из содержащей сущности. Например, если мы поставим orphanRemoval = true, то при удалении пользователя из Set users внутри company, этот user удалится из БД. Стоит быть аккуратным с этим свойством, ведь на user могут ссылаться другие записи и при его удалении возникнет ошибка. Это как CascedeType.Remove. Только там user удалится из БД при удалении company, а тут при удалении ссылки на user из Company.

**Entity Equals and HashСode**

Уникальным идентификатором в SQL является id, но построить equals hashCode над этим полем не очень хорошая идея из-за коллекций, использующих эти методы. Например, у Company есть HashSet User, при создании новых User их id = null т.к. его проставляет БД при добавлении новой записи. И тогда все новые User будут эквивалентны друг другу т.е. их не сохранишь в такой коллекции.

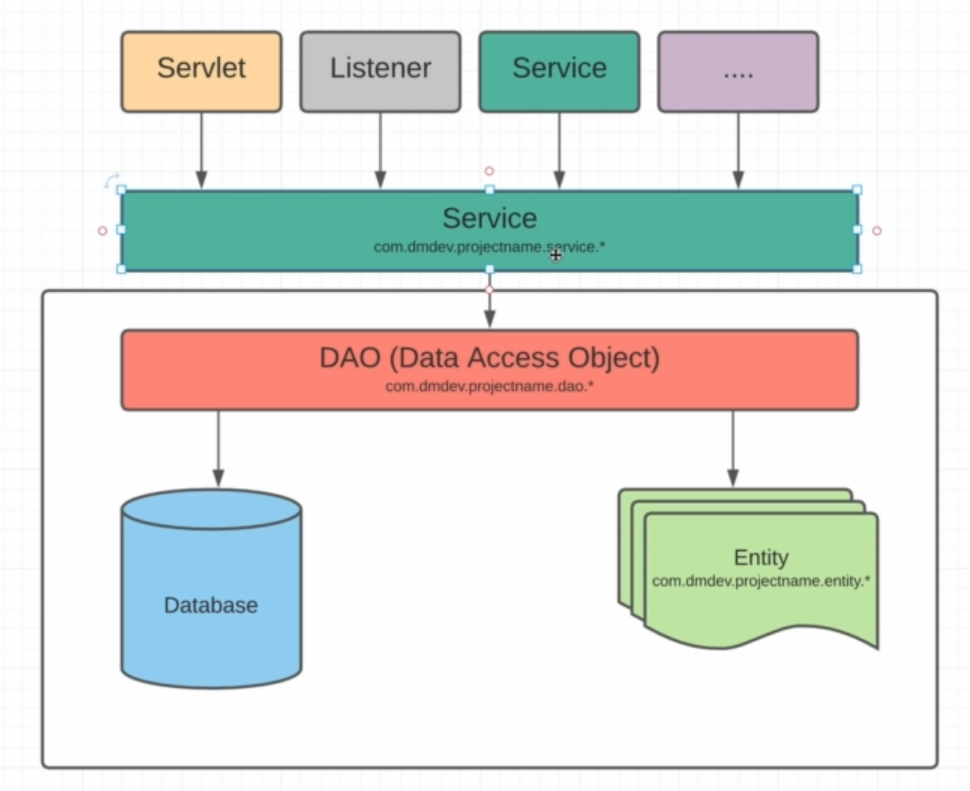
Решать эту проблему можно по-разному в зависимости от бизнес логики. Если есть какое-то NotNull поле, являющееся натуральным уникальным ключом, например username в User или name в Company, то можно использовать его для этих методов. Иначе в этих методах следует использовать все поля, но исключить поля с маппингом ассоциаций (ссылки на другие Entity) чтобы не было никакого зацикливания.





**LazyInitializationException**

Эта ошибка происходит, потому что и AbstarctPersistenceCollection и HibernateProxy содержит внутри себя сессию с помощью которой по ключу делаем запрос.



В реальных приложениях открывают и закрывают сессии на уровне сервисов. И пока мы делаем какую-то логику на уровне сервисов, мы не закрываем нашу сессию.

Но если нам необходимо поработать с какими-то данными на уровне Listener или Servlet, то мы используем паттерн DTO (data transfer object) т.е. мы переписываем данные из Entity в DTO, пока мы можем это сделать и уже потом из уровня сервисов возвращаем DTO. А с DTO можно спокойно работать т.к. сессия не нужна.

Есть метод session.getReference()который работает как get, но всегда возвращаем прокси к указанному объекту, не делая реального запроса к БД.

**OneToOne**

Тип данных OneToOneType в Hibernate.

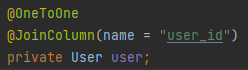
Есть 2 варианта как прописать такую зависимость. One to one это parent – child отношения.

**Вариант первый** – Primary Key.

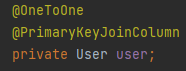
Первичный ключ у child (profile) не синтетический, а будет являться первичным ключом и таблицы chield и FOREIGN KEY на parent (users).



Чтобы сослаться на user можно использовать как обычно @JoinColumn(name = \*поле с Foreign Key\*).



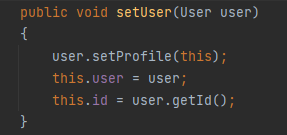
Но есть и другая аннотация, если наш Foreign Key является и Primary Key. Она говорит, что мы исп. первичный ключ таблицы нашей сущности для того, чтобы связаться с таблицей нашей сущности user. @PrimaryKeyJoinColumn



В свою очередь в User мы тоже можем сослаться на Profile через @OneToOne, и использовать просто параметр mappedBy = \*поле в классе profile\*

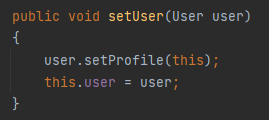


Этот вариант плох тем, что id profile должен полностью совпадать с id пользователя, и он не автогенерируемый. Также тем, что user уже должен быть сохранён в БД, для установки user в объект profile т.к. id user выдаётся средствами БД.



**Вариант второй** – Foreign Key.

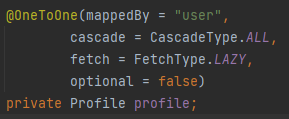
Первичный ключ у child синтетический и уникальный not null Foreign Key на parent. Соответственно потом вместо @PrimaryKeyJoinColumn обычный @JoinColumn(“user\_id”). В setUser мы больше не устанавливаем id для profile.



**Нюанс с Lazy initialization y OneToOne**

Даже если установить Lazy в profile у user, там всё равно не будет прокси объекта, а будет обычный класс profile. Не работает это потому что у User нет ключа на profile, чтобы потом Lazy его проинициализировать -> Hibernate не знает есть ли profile для такого user и можно ли будет его потом проинициализировать или туда нужно вписать null т.к. в таблице profile нет строки для такого user. Для этого он сделает select в profile, чтобы посмотреть, а можно ли будет его потом проинициализировать. А раз select уже всё равно делали, то тогда проинициализируем сразу.

Чтобы Lazy работал, нужно указать optional = false т.е. указываем, что сущность profile обязана быть в БД для нашего пользователя. Hibernate видит этот флаг и раздаёт proxy, не делая запроса к БД. Но это так будет работать только в первом варианте т.к. во втором у user всё ещё не поля хранящего profile\_id для lazy инициализации (а в первом варианте таким ключом является id самого пользователя).



Это так работает, потому что прокси утверждает, что объект есть, он просто ещё не проинициализирован.

Но в child Lazy будет работать как обычно т.к. у него есть ссылка на parent для lazy инициализации (user\_id).

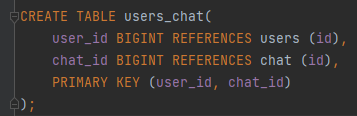
Вывод: лучше не использовать bidirectional(mappedBy) связь в OneToOne если нужна Lazy инициализация, либо отказаться от синтетического первичного ключа в profile.

Cascade часто используется в @OneToOne, но у parent(users). Поскольку если мы удаляем user мы хотим удалить и его profile.

**ManyToMany**

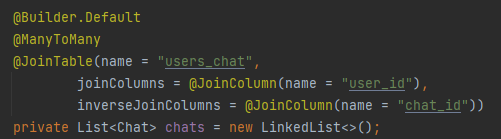
cascade не стоит использовать в Many to Many ведь пусть у нас есть Chat и User. При использовании Cascade мы будем либо удалять всех юзеров, которые в этом чате при удалении чата, либо удалять все чаты, которые есть у этого юзера и т.д.

Соответственно создали таблицу chat и таблицу users\_chat (но в этом варианте не будет сущности users\_chat):

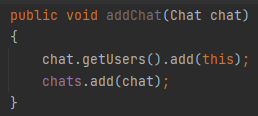


Встречается такой тип связи редко т.к. чаще всего в таблице по типу users\_chat будет ещё дополнительная информация вроде время добавления юзера и т.д. и свой синтетический id. -> уже @OneToMany у user’а.

Чтобы его прописать у более главной сущности (user) прописываем @JoinTable (как @JoinColumn, только таблицу)



**Важно.** Нужно добавлять и User’у в chats chat и чату в users user (как и раньше). Иначе не будет соответствия на уровне java модели, а такого быть не должно.



Для bidirectional связи указываем mappedBy



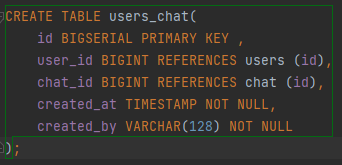
**Важный момент.** Hibernate добавит запись в таблицу users\_chat только при добавлении chat в объект User. Если добавить только user в объект Chat, то запись не появится в users\_chat т.к. у Chat используется mappedBy, а она read only.

Соответственно для того, чтобы удалить взаимосвязь чата и юзера (строку в users\_chat) нужно у user из поля chats удалить chat.

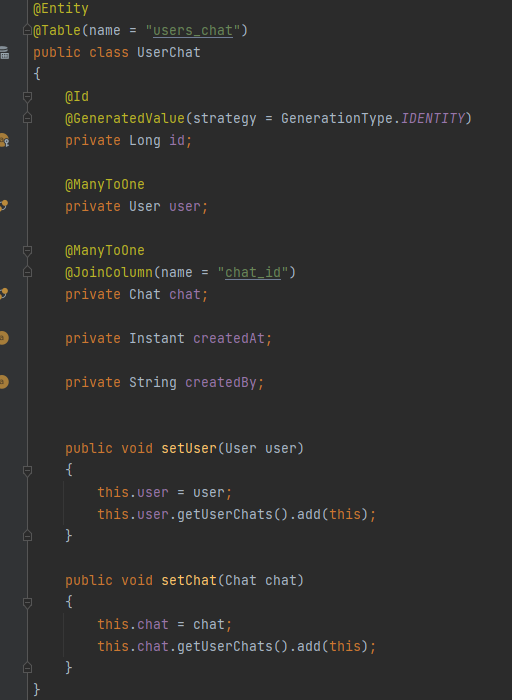
Не стоит оперировать Casacde type в этой связи. Лучше удалять chat и user через сессию.

**Как бы это выглядело скорее всего на практике:**

users\_chat была бы отдельной сущностью с синтетическим id. Например, с таким определением:



Создаём сущность соответственно для этой таблицы. Выглядеть она будет так:



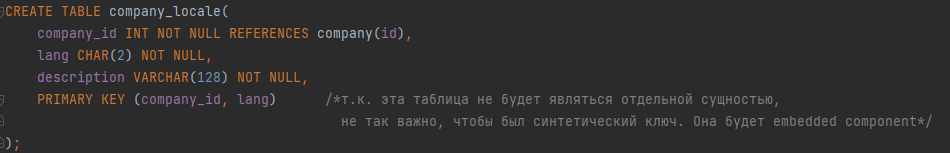
В этой сущности нет смысла указывать Cascade тоже. Однако в поле users\_chat внутри User или Chat в этом есть смысл (удалили user -> удаляем и записи какие у него есть чаты). Но опять же, Cascade лучше использовать внутри БД.

Именно такой вариант будет чаще всего встречаться в жизни.

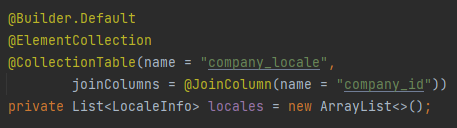
Просто к сведению: так будут делаться LEFT OUTHER JOIN (или INNER, если прописать optional = false) для user с users\_chat и chat с users\_chat, если мы будем назначать новый userchat этим двум сущностям, ведь в методах set есть getUserChats() -> инициализируем proxy.

**Element Collection**

@ElementCollection. Это о том, как использовать маппинг коллекций, чтобы элементы коллекций не были Entity. Создадим таблицу, скажем такую:



И создадим @Embeddable компонент с полями lang и description. А в Company просто вот такое поле:

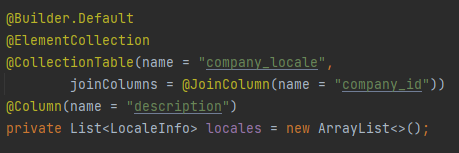


Аннотация @ElementCollection как раз и даёт знать Hibernate что тут происходит, чтобы он мог это обработать. С помощью @CollectionTable указываем из какой таблицы вынимать значения для данного компонента. С помощью joinColumns указываем какой столбец используется в качестве внешнего ключа для связи с Company. По умолчанию таблица будет \*названиекласса\_названиеполя\*, а имя столбца будет \*названиекласса\_id\*.

Можно также использовать @AttributeOverride, если имя поля в LocaleInfo отличается от имени колонки в company\_locale.

**Другой вариант использования**

Нам не нужен embedded component, а мы хотим, скажем, просто List<String> со всеми description для этой company из таблицы company\_locale. Тогда просто добавляем ещё одну аннотацию @Column(name = “description”), чтобы указать из какого столбца значения брать.



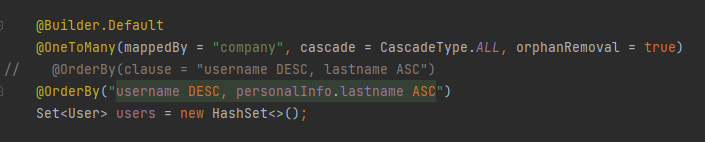
Но так мы сможем только считывать из таблицы company\_locale. Редко используется такой вариант.

**Collection ordering**

В Company есть users. Мы хотим не только этот users получить из БД, но и содержать их в нужном порядке на уровне приложения.

**Первый вариант. Через order by.**

Используем аннотацию @OrderBy. Если использовать @OrderBy из hibernate, то пишем SQL запрос. Если использовать @OrderBy из javax.persistent, то пишем HQL запрос.

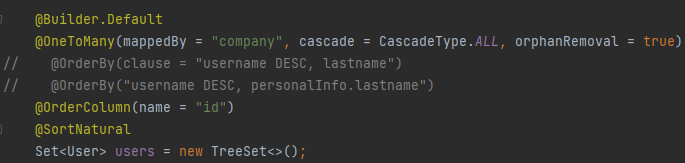


users объявлен как Set и дефолтная его реализация (до запроса к БД) это hashSet. Но когда мы получаем ответ из БД Hibernate подставляет свои коллекции. Это будет persistenceSet, но Hibernate видит, что данные нужны отсортированные и использует внутри LinkedHashSet. Так что порядок будет поддерживаться, даже если используется не List, а Set.

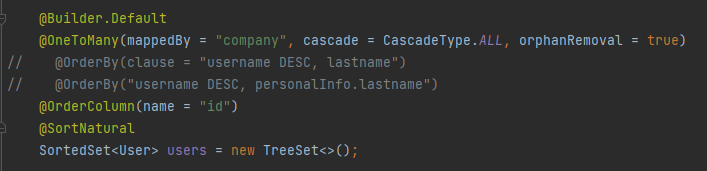
**Второй вариант. Сортировка in memory.**

@OrderColumn указывает имя столбца, по которому будет выполняться сортировка. Эту аннотацию самобытно использовать не будем, но она тоже выполняет сортировку. Она работает только с List, только с типом int и должен быть строгий порядок 1,2,3,4. Подойдёт для каких-то справочных таблиц.

Проще на уровне приложения использовать TreeSet(реализуем Comparator<>). Но это если мы создаём новую Company и добавляем туда user. Если мы достаём из БД юзеров, то возвращается PersistenceSet, которая не соблюдает порядок. Чтобы это решить используем аннотацию @sortNatural, которая сортирует также, как и TreeSet (используя comparable)



Даже если указать SortedSet из java.util, а не Set, Hibernate всё равно не поймёт, что нужно подставить реализацию PersistenceSet с соблюдением порядка, без аннотации @SortNatural. Но при использовании SortedSet он выдаст ошибку, если не указать @SortNatural.

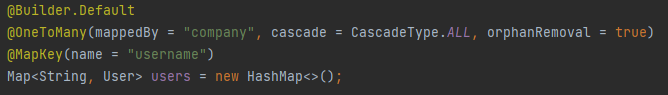


ещё есть аннотация @ SortComparator, если мы хотим, чтобы сущности были отсортированы не в порядке по умолчанию, а каким-то другим компаратором.

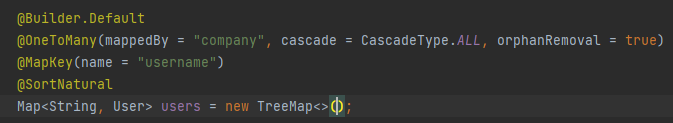
Лучше использовать первый вариант с сортировкой на сервере.

**Maps in mapping**

Всё остаётся, как и раньше, только нам нужно подсказать Hibernate какое поле, в сущности, использовать для ключа. Для этого есть аннотация @MapKey



Соответственно точно также можно получать отсортированные Map, используя @SortNatural:



Если захотим использовать какой-то класс, в качестве ключа, то используем аннотацию @MapKeyClass. Ещё есть @MapKeyEnumerated, для даты тоже отдельный есть, но всё это редко используется.

@MapKey используется только если значение является сущностью. Но если мы хотим в качестве значения использовать обычный String, то используем @ElementCollection и @MapKeyColumn.

**Databases in test**

Тесы в реальном приложении не запускают на реальной БД. т.к. необходимо предустанавливать специальное ПО для этого.

**Первый вариант решения**. In memory databases.

Все данные хранятся в памяти. У некоторых таких БД есть возможность сохранять состояние на диск. Такие БД максимально ускоряют процесс работы с данными. т.к. они в ОЗУ.

В test resources создаём такой же файл hibernate.cfg.xml, но описываем в нём всё для H2 БД, например (ещё зависимость для H2 нужна).

После тестов эта БД уничтожится. Как же накатить туда схему БД? Для этого используем Flyway или Liquidbase.

Или можно использовать генераторы. Для этого прописываем такую property. 

Работает это так:

Hibernate берёт маппинг сущностей, который мы задали с помощью аннотаций и т.д. и генерирует на основании его DDL. Соответственно он генерирует их не 1 в 1 т.к. мы написали в sql и их хорошо бы перепроверять.

Есть несколько стратегий генерации DDL:

update – смотрим разницу между готовой схемой и маппингом сущностей. И если разница есть, то мы изменяем и существующий DDL.

create – просто накатываем новую DDL на новую DDL.

create-drop – как только мы закрыли нашу SessionFactory происходит drop БД

validate – просто проверка существующей БД с маппингом сущностей.

Для более точной автогенерации DDL можно дополнительно давать информацию Hibernate. Например, @Column(unique = true, nullable = false). В Column есть даже columnDefinition = “” где можно написать SQL и он будет использоваться для определения указанной колонки.

Также есть ограничение, что в этой БД (H2) не будет тех же типов, что есть в нашей БД (postgresql например), например, JSON.

**Второй вариант решения**. Docker test containers.

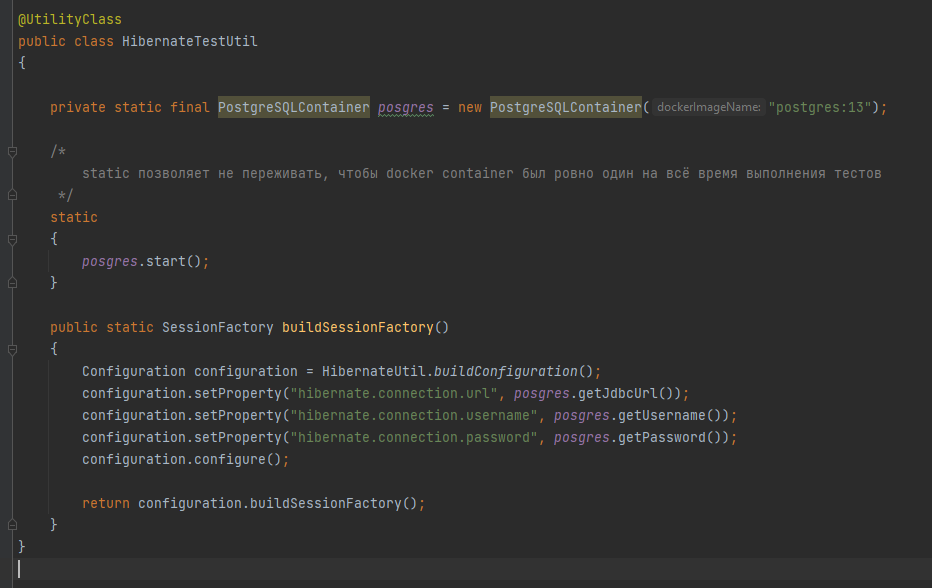
Этот вариант более предпочтителен в реальных приложениях. Тут нет проблемы совместимости БД, не нужна автогенерация DDL. и т.д. Также гарантируется идентичное состояние что во время тестов, что и в реальной БД.

Docker поднимает соответствующий контейнер во время старта тестов и тушит его как только тесты прошли.

Url, user и password предоставляется динамически так что их не нужно указывать в конфиге хибернейта.

Нужно написать немного кода, который поднимет docker образ.

Пишем что-то похожее на HibernateUtil, где мы конфигурировали и создавали SessionFactory для нашего приложения. Пользуемся конфигурацией нашего тестового приложения, но просим у контейнера и записываем при конфигурации динамически генерируемые настройки. А потом в тестах просим SessionFactory именно у этого класса, а не у HibernateUtil.



Нюанс:

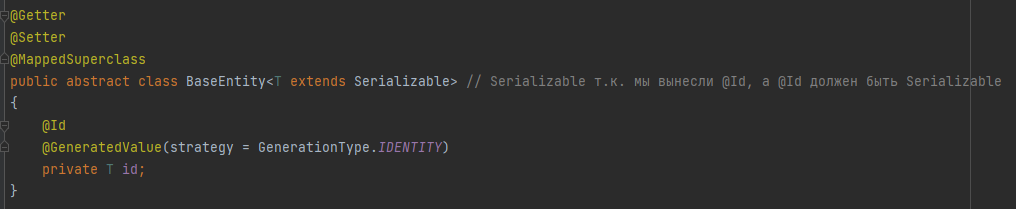
Докер поднимает 2 контейнера на самом деле. Один с постгрес, а второй управляет этим контейнером с постгрес. Он определяет, когда его нужно тушить. Если нет коннекшен пула к нему, то можно тушить. Ну там и др. стратегии есть.

**Mapped Superclass. Маппинг наследования.**

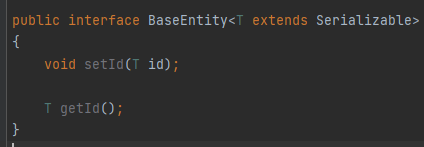
Наследование полей, **когда базовый класс не является отдельной сущностью** хибернейта.

Это нужно, когда, например, какая-то группа полей повторяется из сущности в сущность и мы хотим вынести их в одну сущность и унаследоваться от неё.

Всё что нужно – пометить суперкласс аннотацией @MappedSuperclass. Ещё дали этому классу геттеры и сеттеры т.к. мы захотим всё же их иметь для доступа к этому полю.

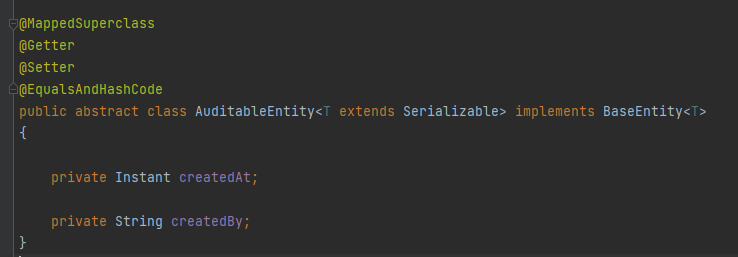


В продолжении примера, не все Сущности работают с Identity. Но все сущности обязаны иметь id, поэтому методы get/set подойдут для всех сущностей.



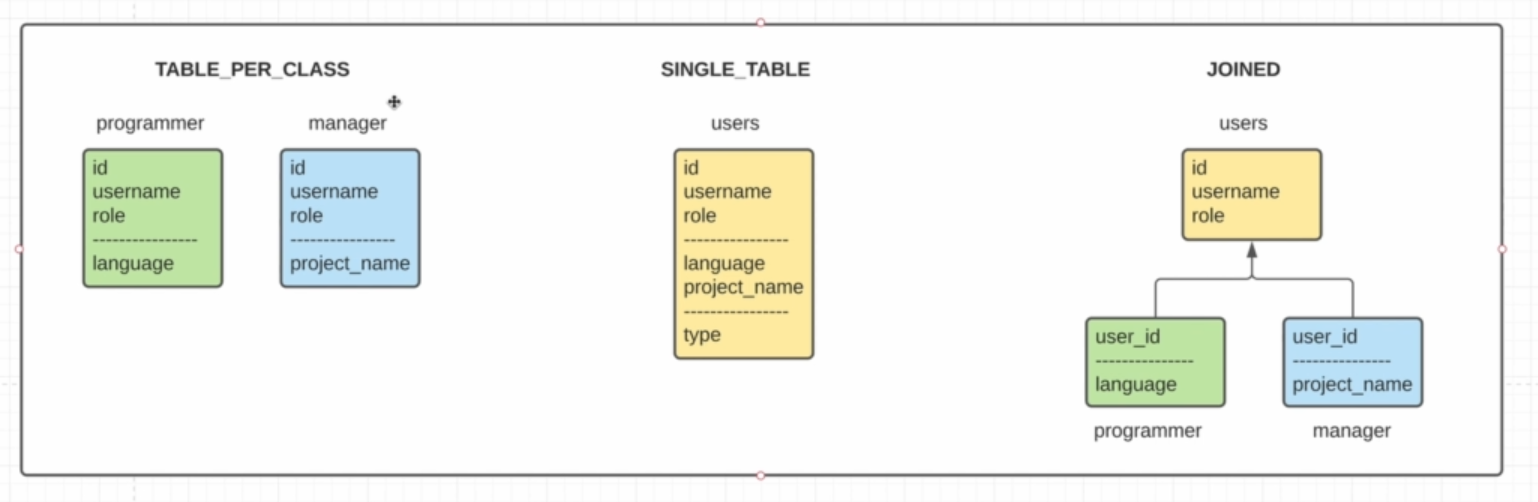
Какой-то подобный интерфейс можно использовать выше при работе с сущностями.

Могут всё же находится такие базовые поля, которые вы бы хотели вынести в общий класс и наследовать сущностями. В этом примере это поля для аудита (createdAt и createdBy). Но все наследники AuditableEntity будут Entity так что он тоже может реализовывать BaseEntity, чтобы не пришлось отдельно это прописывать в сущностях, наследующихся от AuditableEntity.



Наследование полей, **когда базовый класс является отдельной сущностью** хибернейта.

Есть 3 различных стратегии в таком случае маппинга наследования.



**Table per class**. (TABLE\_PER\_CLASS)

т.е. будет 2 наследника у таблицы users. Таблица programmer и таблица manager (все поля из users будут дублироваться в обоих таблицах).

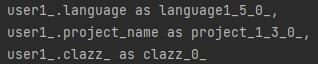
Создали классы, унаследовались, а User должны пометить специальной аннотацией @Inheritance. Теперь User не является самостоятельной сущностью, у нас для неё нет отдельной таблицы в БД. -> удаляем Builder и @Table тоже лучше удалять. Однако @Entity оставляем, чтобы можно было работать с этой сущностью в Hibernate.

Т.к. конфигурационных файла для Hibernate теперь 2, новые Entity нужно будет добавлять в оба. Чтобы не забыть это сделать, лучше добавлять новые entity в Configuration через java код т.к. он всего 1.

Т.к. мы можем использовать User.class в session.get для получение какого-то User по идентификатору, и каждая сущность должна быть уникальна, у неё есть первичный ключ. Но у Manager и Programmer у каждого свой Identity. А это значит, что у нас есть и Programmer и Manager с одним и тем же идентификатором -> для такой иерархии наследования следует использовать sequence 1 общий для двух таблиц. -> переделываем у User Identity на Sequence.

Если мы сделаем session.get(Programmer.class, 1L) то всё понятно, Hibernate пойдёт в таблицу programmer и оттуда достанет всё.

Но если мы сделаем session.get(User.class, 1L). То Hibernate не будет знать из какой таблицы доставать и сделает общий запрос, в котором пойдёт искать по обеим таблицам. В этом запросе будет поле и из Programmer (language) и из Manager (project\_name) + одно дополнительное поле user\_clazz, которое поможет Hibernate с определением объект какого класса возвращать.



“-“:

* Приходится дублировать общие поля и если мы добавляем какое-то общее поле, то приходится изменить обе таблицы
* Когда мы обращаемся к базовому классу приходится делать Union all запросов ко всем нашим наследникам
* Должны использовать общий sequence для всех таблиц – наследников, иначе не сможем оперировать с ними как с общей сущностью

“+”:

* Если нам нужен какой-то конкретный наследник, то мы обращаемся только к нужной таблице

**Single table** (SINGLE\_TABLE)

т.е. у наших наследников будет одна общая таблица и в ней будут как поля общие, так и совокупность всех полей у наследников + дополнительная колонка type по которой будет определяться какой класс нужно инстанцировать.

Для этого помечаем User аннотациями

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "type") (по умолчанию будет создаваться / искаться колонка “DTYPE”)

А классы наследники помечаем аннотацией @DiscriminatorValue("manager"). Это задаёт значение для колонки. По умолчанию это значение равно имени класса.

Также в данном варианте использования User уже не обязан быть абстрактным и может являться отдельной сущностью т.к. таблица для него уже есть.

При запросе конкретного класса hibernate указывает id и type в select. А при запросе базового класса просто не указывается type и запрос делается просто по id.

Итоги:

У нас одна таблица со всеми полями из всех наследников -> мы не можем давать какие-либо Constraint на их поля. Т.е. на language или project\_name. Например, NOT NULL и т.д.

Если запрос к базовому классу нет никакого UNION ALL мы обращаемся всего к 1 таблице, но и при поиске конкретной реализации мы обращаемся к этой же таблице, а она большая.

Можем использовать IDENTITY, а не SEQUANCE.

Простота в реализации, но портим нормализацию.

**Joined** (JOINED).

Есть отдельные таблицы для конкретных реализаций и дополнительная таблица с общими полями. По сути, такая же иерархия, как и в Java.

Прописываем у базового класса:

@Inheritance(strategy = InheritanceType.JOINED)

А у наследников

@PrimaryKeyJoinColumn(name = "id") т.к. внешний ключ в конкретном классе является и внутренним ключом для этой таблицы (user\_id на картинке).

Для вставки Programmer требуется вставить в 2 таблицы. В user и в programmer. Остальные операции тоже усложняются т.к. нужно работать с двумя таблицами. А при запросе базового класса мы делаем соединение со всеми таблицами наследниками и ищем конкретную реализацию (а если не находим, то создаём user).

Из плюсов – данные нормализованы, но ухудшается производительность. На практике используется single table чаще всего. Из-за производительности. Если же важна нормализация, то используют Joined.

**Hibernate querying (HQL)**

Чтобы делать SELECT, UPDATE, INSERT, DELETE по более сложным механизмам, чем по 1 сущности, просто через id есть 2 основных механизма:

* HQL (чаще используется)
* Criteria API

В HQL мы оперируем сущностями и полями, а не таблицами и колонками, как в SQL.

session.createQuery("select u from User u where u.personalInfo.firstname = 'Ivan'");

Аналогично:

select u.\* from users u where u.firstname = Ivan

session.createQuery() возвращает Query. Query аналог PrepareStatement. Основные методы Query:

list()

uniqueResult() но он выдаст exception, если запрос ничего не вернёт или вернёт не 1 результат.

uniqueResultOptional()

кучу setParametr()

setMaxResult() – limit

setFirstResult() – offset

setHint() – дополнительные параметры, который мы хотим передать вместе с нашим запросом. Например, время выполнения нашего запроса (timeout) и т.д. Основные есть в классе QueryHints.

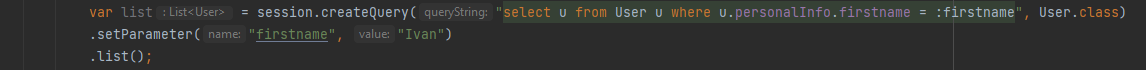
Есть flushMode при вызове query. Настраивая этот параметр можно указать должна ли сессия сделать flush, перед выполнением query. Этот параметр необходим т.к. query идёт в обход persistence context, ведь это обычный запрос в БД, а нам возможно потребуется синхронизировать данные в БД и приложении перед запросом.

Но эта настройка даже вынесена в метод setFlushMode(). По умолчанию там auto (т.е. будет flush перед запросом). Другой вариант – COMMIT (означает что flush делаться будет только с commit транзакции или явном вызове flush). Не рекомендуют менять это.

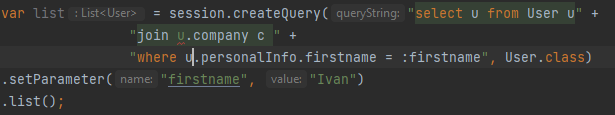
Но list() будет возвращать не параметризованную коллекцию. Чтобы вернулась параметризованная, можно задать второй параметр с классом типа данных, который мы ожидаем получить.

Но с HQL нам только idea помогает с проверкой синтаксиса. Для компилятора это строка.

Как и в PrepareStatement можно задавать параметры в запросе:

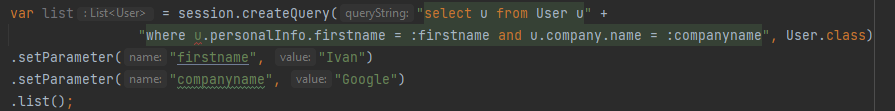


Нужно прописывать маппинг в сущностях, потому что в дальнейшем это поможет в написании запросов. Ведь в HQL тоже можно делать join.



При join указываем поле у сущности для связи с другой сущностью.

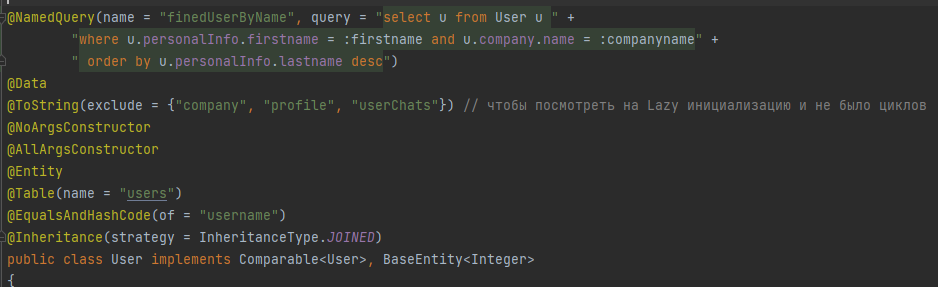
Вверху явный join. HQL поддерживает и неявный join. Мы можем обратиться к компании сразу через user. И это будет аналогично первому запросу.



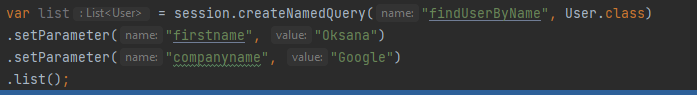
Он рекомендует использовать явные join, чтобы был более SQL подобный синтаксис. + можно руководить какой именно join хотим использовать.

Также можно использовать order by и т.д. Всё тут есть.

Мы также можем делать именованные запросы (на практике не так часто делается). Если какой-то большой запрос часто используется мы можем сохранить его, без использования дополнительного класса, а просто сохранив его в сущности, используя @NamedQuery.



А используем так:



Но так они разрастаются и появляется много загромождающего кода.

Интерфейс Query позволяет делать не только select запросы. Тогда будет возвращаться не сущность, а количество удалённых / изменённых / вставленных строк. И тогда используем не list(), а, например, executeUpdate().

Но некоторые СУБД могут возвращать не количество строк, а сами строки. Для этих и других особенностей есть метод session.createNativeQuery(), где мы пишем уже sql запрос. Рекомендуется использовать если нужна какая-то оптимизация или необходимый функционал нельзя получить, используя HQL (пример пор строки).

**Все сущности, полученные с помощью любых Query будут положены в PersistenceContext.**

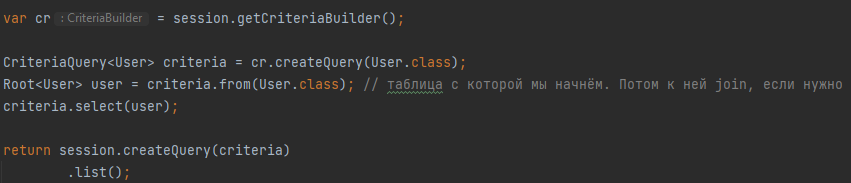
Рекомендуется использовать синтаксис более похожий на SQL.

т.е. “select u form User u”, а не “from User”.

**Criteria API**

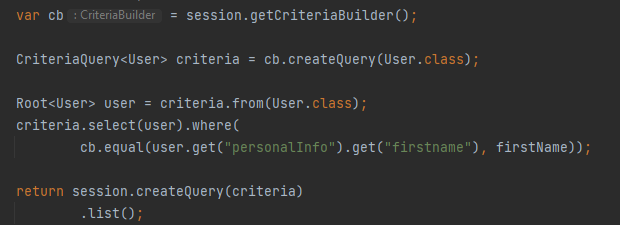
Позволяет строить динамические запросы. В большинстве случаев, это такие запросы, где where условия варьируются. Если писать такой запрос просто через строку, то это было бы очень сложно т.к. добавлять where или and, а где-то or и т.д. и так очень много ошибок можно допустить, ведь это обычная строка. Для создания таких запросов можно использовать criteria api. Criteria громоздкая и непонятная. Есть отдельные фреймворки для построения динамических запросов с человеческим api.

Простой пример использования:

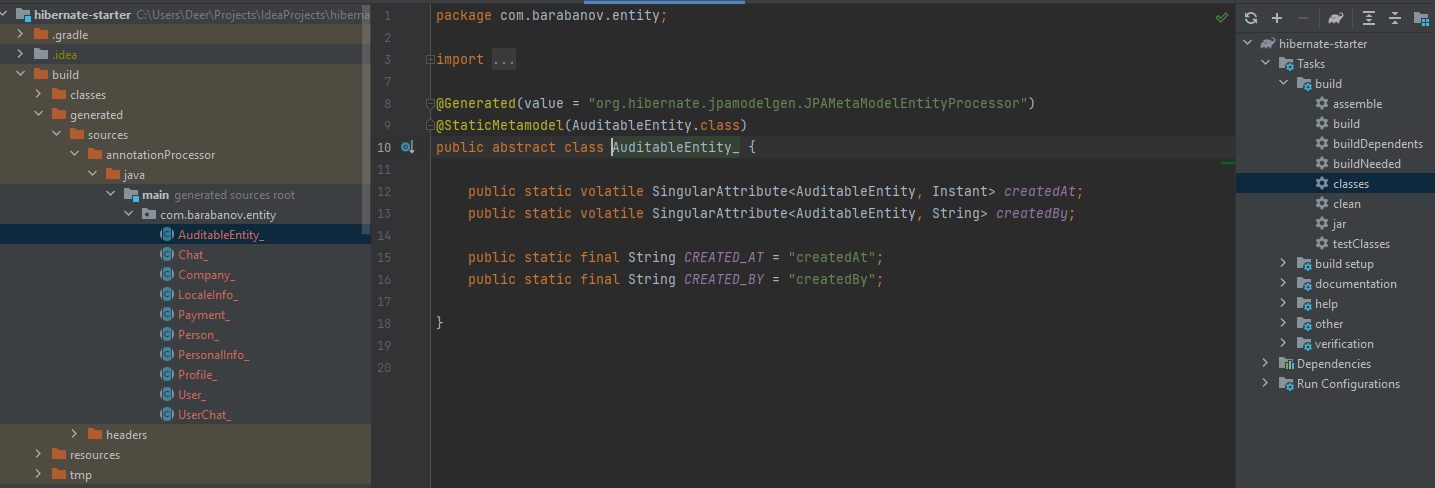


Мы создаём criteria. И ему всё назначаем. Но делается всё через CriteriaBuilder. В нём все методы equal, like, or и т.д.

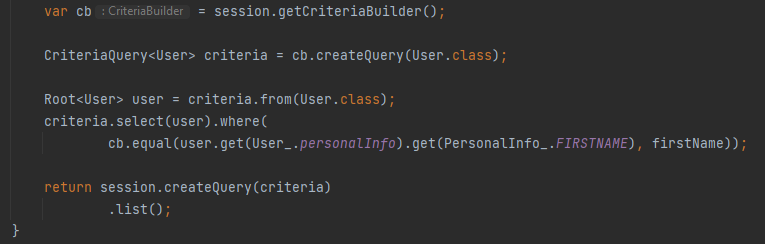
И придётся обращаться к полям, используя строки, как на примере:



Чтобы это избежать есть jpamodelgen (версия должна быть такая же), который сгенерирует просто константы для каждой нашей сущности, которые мы можем использовать вместо прописывания через строки.



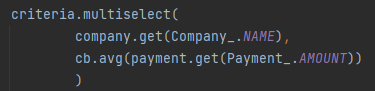
И предыдущий запрос выглядит так:



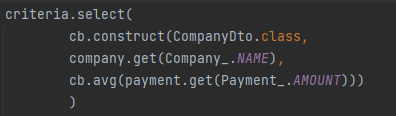
В реальных приложениях приходит DTO с множеством полей. Мы как-то так составляем набор условия для where и передаём их criteria.



Если мы хотим вернуть List<Object[]>, то мы можем использовать multiselect

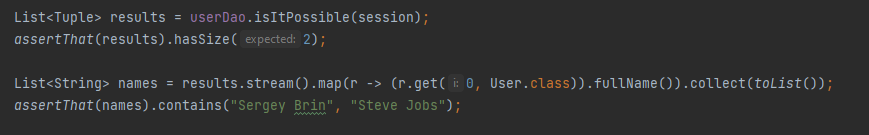


Но есть и функционал для использования DTO. Мы просто указываем какой DTO нужно использовать и передаём аргументы для его конструктора. Hibernate сам достанет эти поля, вызовет конструктор со всеми параметрами и создаст Dto, который и вернёт. Т.е. вот так (это называется Projection):



**Tuple**

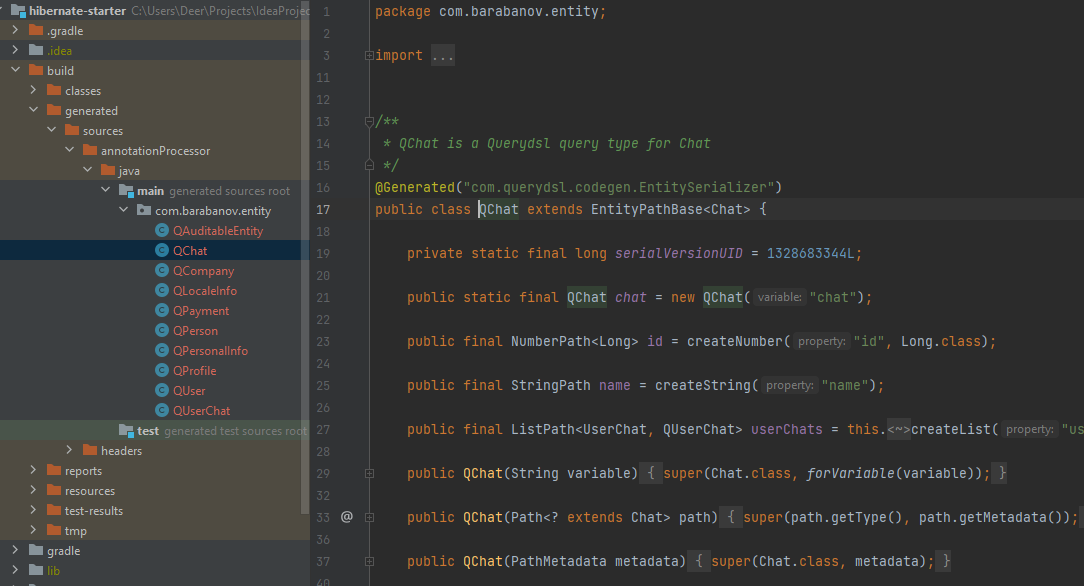
По сути, просто List с Object, которому в get можно вторым параметром предать тип, который должен вернуть этот get. Пример использования:



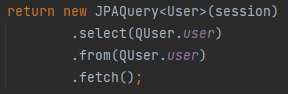
Делать через cb.tuple().

**Querydsl**

Т.к. Criteria API довольно неудобная, но динамические запросы строить всё равно нужно, появлялись библиотеки для построения таких запросов, но не через Criteria API. Querydsl одна из таких. (он генерирует классы не с “\_” в конце, как jpamodelgen, а с “Q” вначале).

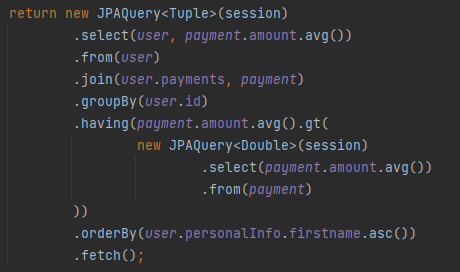


Тут есть один основной класс JPAQuery, используя его строим запрос. Пример простого запроса findAll:



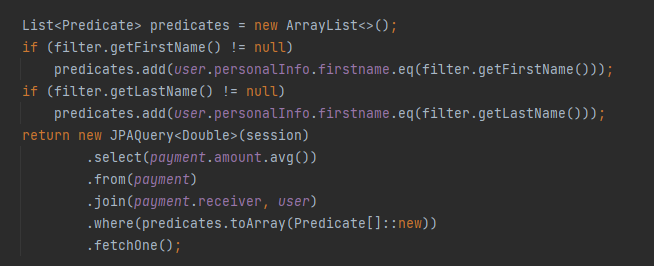
Querydsl не умеет работать с DTO, но у неё есть есть класс Tuple. На Querydsl писать даже быстрее, чем на HQL и его рекомендуется использовать на реальных приложениях.

Пример запроса, возвращающего список: сотрудник (объект User), средний размер выплат, но только для тех сотрудников, чей средний размер выплат больше среднего размера выплат всех сотрудников. Упорядочить по имени сотрудника.



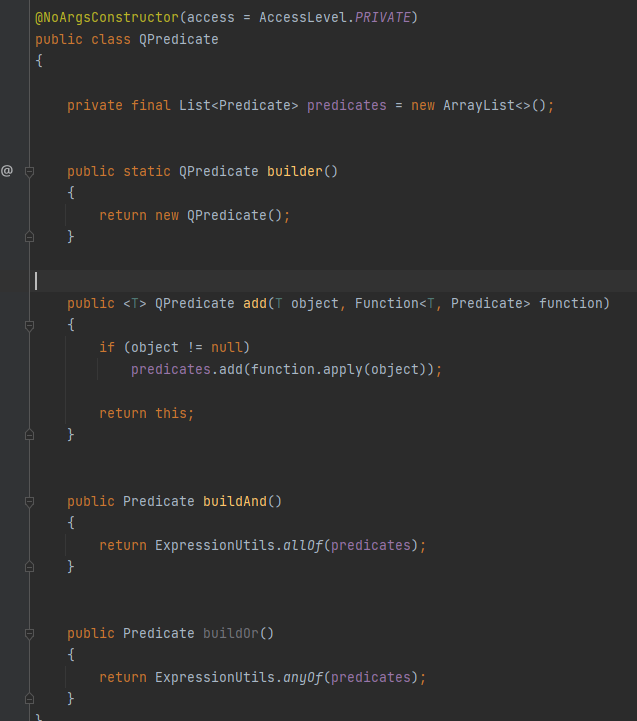
**Filter**

Мы хотим иметь какой-то такой функционал динамического создания запросов (в filter просто поля с значениями для фильтрации):



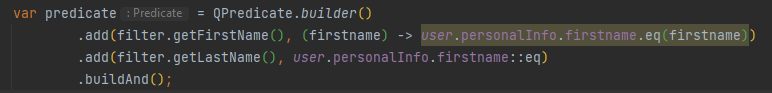
Однако, чтобы избежать множественных if нам необходимо будет написать свой builder предикатов таких.

Создаём вот такой класс (обычно такой класс создаётся на уровне сервисов, а не внутри DAO. DAO принимает просто predicate уже):



И так получается всё красиво:)

Первая add для наглядности. Это другие Predicate. Это из querydsl Predicate. В нём сути записано само выражение. Тут нет метода test().



**N + 1 Select problem**

Не стоит использовать OneToOne bidirectional связь тогда, когда у chield сущности первичный ключ – синтетический (т.к. не получится сделать Lazy инициализацию см. OneToOne).

Стоит предпочитать Lazy инициализацию, т.к. под бизнес логику не подгадаешь что и где потребуется или нет, а могут выполняться лишние запросы к БД.

Поскольку у нас делаются LEFT OUTHER JOIN при OneToMany, если сделать несколько таких OneToMany и всем поставить EAGER у нас будет большой sql запрос, где будет декартово множество всех этих таблиц т.к. у нас LEFT OUTHER JOIN каждый раз будет делаться.

MultipleBagFetchException говорит о том, что нельзя получать несколько ассоциаций одним запросом, если это Bag (List). Например, мы хотим запрашивать все userChats и payments у User как EAGER. Будет эта ошибка т.к. Hibernate не будет понимать какие строки относить к userChats, а какие к payments. А PersistenceSet не зависит от порядка, так что если использовать не List, а Set то работать то будет. Но декартово произведение идея вообще не лучшая.

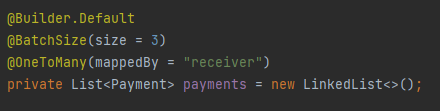
Также все EAGER не работают для HQL запросов, criteria API и т.д. И даже если будет EAGER у OneToMany всё равно не будет LEFT OUTHER JOIN, а будут отдельные запросы для инициализации связанных сущностей. Потому что не работали бы limit, offset и др. функции из-за декартово множества.

Но если бы мы хотели 5 User у нас сделалось бы 5 запросов для User, потом 5 запросов для payments, потом 5 запросов для userChats и т.д. Т.е. вместо 1 запроса для одного User мы сделаем N + 1 для каждого User. Короче везде используем LAZY, т.к. EAGER (именно одним запросом с JOIN) не работает при использовании HQL и др. запросов.

**BatchSize**

Итак, при запросе к нашему proxy объекту всё равно будет делаться отдельный запрос, чтобы достать все сущности. Но можно не делать отдельный запрос, чтобы достать все payments для user, а можно перечислить в запросе несколько user\_id и достать payments сразу для нескольких user. Это не рекомендуется использовать на практике.

@BatchSize ставим прям над полем. 3 значит количество user\_id в запросе к payment



Но при OneToOne это не работает + полностью не решает проблему N + 1. Также просто пометить поле compony в User ничего не даст. Но можно пометить класс Compony @BatchSize(size = 3) и тогда эта аннотация будет играть роль во всех маппингах, где используется Compony.

BatchSize хоть и улучшает нашу производительность, но не решает проблему т.к. N может стремиться к бесконечности, а мы просто уменьшаем N таким образом в несколько раз.

**Fetch**

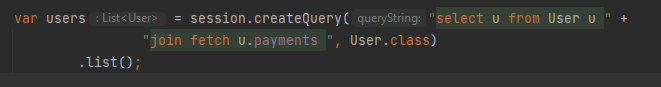
@Fetch ставим вместо @BatchSize. Для коллекций походит только FetchMode.SUBSELECT. SELECT и JOIN нельзя использовать с коллекциями т.к. будет опять декартово произведение. А SUBSELECT делает подзапрос для того, чтобы понять для каких User доставать payments. Подзапрос – это просто запрос, который мы использовали для получения User’ов. Тут проблема N + 1 решается т.к. у нас возвращаются Payment’ы для всех нужных нам User всего за 1 запрос.

Но Fetch не работает, для отдельных сущностей, без коллекций т.е. для ManyToOne, OneToOne, ManyToMany, при запросах через HQL и т.д. При запросе этих сущностей по id также всё хорошо и будет подзапрос. А через HQL будет всё та же N + 1. Например, если пометить compony в User как Fetch c JOIN при запросе через HQL всё равно будет отдельный запрос для Compony. Кроме того, с Fetch мы делаем получение наших payment’ов в любом случае, даже когда этого не хотим. Но в каких-то случаях нам может потребоваться только один User с Payment, а всем остальным, полученным этим запросом User’ам payments не нужны. С Fetch всё равно все User’ы получат payments т.к. они будут в подзапросе.

Тоже рекомендуется не использовать этот вариант решения N + 1.

**Query Fetch**

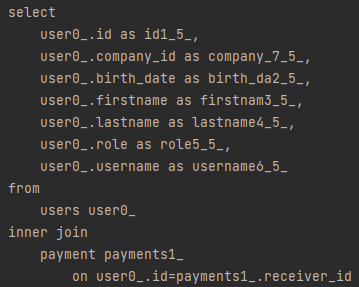
Этот вариант используется уже на практике. Использование Fetch на уровне HQL запроса.



Так у нас будет именно 1 запрос, где вернуться сразу все необходимые поля.



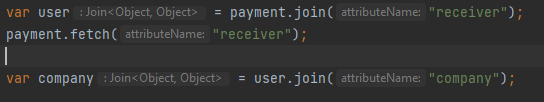
А просто с join не будет полей для инициализации payment в select



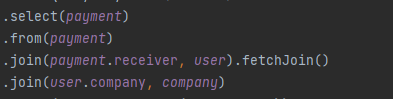
И payments будут потом инициализироваться отдельными подзапросами для каждого user.

Этот способ работает и для отдельных сущностей, а не только для коллекций, однако стоит быть осторожнее т.к. продублируются все поля для всех строк. Но тем не менее для одной единственной связи это не так плохо, а если связь OneToOne то вообще вполне хорошо и будет гораздо выше по производительности.

Для использования fetch в Criteria API есть ключевое слово fetch



В querydsl:

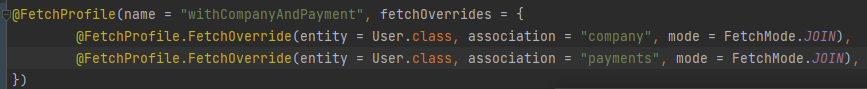


Fetch отличный вариант для построения запросов HQL, querydsl и т.д.

Но всё что он делает – добавляет колонки в запросе, чтобы можно было проинициализировать сущности сразу, не делая дополнительного подзапроса. Но это не работает при работе с одной единственной сущностью (если не делать HQL запрос). При использовании getById мы будем делать дополнительные запросы всё равно. Для решения этого есть другие инструменты.

**Fetch profile**

Первый инструмент для решения проблемы N+1 при работе с одной единственной сущностью (при работе с HQL используется query fetch).

Создаём стратегию того, как выполнять запрос. Для этого используем аннотации @FetchProfile и FetchOverride. Т.к. FetchOverride может ставиться и над пакетом, приходиться записывать и параметр для какого класса эта аннотация. Теперь над классом User такая аннотация (и запрос на select user будет содержать сразу все поля для инициализации company и payments за 1 запрос с left outer join’ами).  Также в сессии нам нужно указать какие профайлы для запросов использовать.



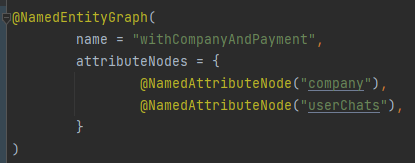
Нельзя использовать limit и offset т.к. будет декартово произведение user на payment. Но т.к. мы это всё пишем для getById нам limit и offset не нужны.

Такой вариант не работает для HQL запросов. Это работает только для работы с одной сущностью. Этот вариант решения проблемы не предпочтителен.

**Entity Graphs**

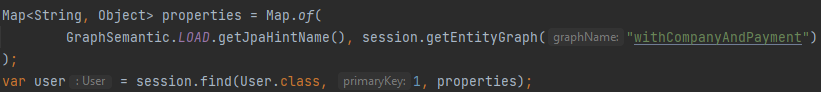
Это инструмент для решения N + 1проблемы, как при работе с HQL запросами, так и при работе с одной сущностью. Т.е. вместо fetch query + fetchProfile можно использовать Entity Graphs.

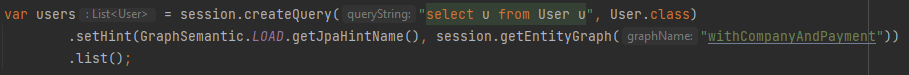
Использование аналогично fetchProfile. Помечаем класс аннотацией @NamedEntityGraph



Аналогично помечаем участвующие поля аннотацией @NamedAttributeNode. Однако этим полям мы можем задавать такой же сабграф, чтобы запросить сразу уже какие-то поля внутри этих полей. Например, chat внутри userChats.

Далее нужно передавать хинт с названием этого графа при HQL запросах и обычной работе с одной сущностью. Однако в методе get нет параметра для задание хинтов, так что используем более гибкий метод find().





GraphSemantic.FETCH говорит о том, что атрибуты, помеченные AttributeNodes будут EAGER, а все остальные LAZY.

GraphSemantic.LOAD говорит о том, что атрибуты, помеченные AttributeNodes будут EAGER. Остальные атрибуты будут загружаться также, как они помечены в метаданных (LAZY или EAGER).

Вот так мы указали что хотим, чтобы у User инициализировался userChats, а внутри него инициализировались объекты Chat, используя subgraph и всё это сделалось через 1 запрос к БД.



Однако такие аннотации получаются довольно громоздкими и Hibernate предоставляет программный механизм для создания этих графов. Выглядит это так:



Создание этих Entity graph можно выделить в отдельный утилитный граф

**Best Practices в N + 1 проблеме**

1. Избегать @OneToOne bidirectional связи с синтетическим ключом у child.
2. Используем LAZY везде. EAGER не срабатывает (в том смысле что одним запросом не запрашивается при HQL, а создаются отдельные запросы)
3. Не следует предпочитать @BatchSize и @Fetch
4. Использовать query fetch (HQL, Criteria API, QuerySQL)
5. Предпочитать EntityGraph API, а не @FetchProfile. Его можно вынести в отдельный класс, а не просто аннотацией и он универсален.

Но стоит сначала писать бизнес логику, а уже потом оптимизацию, при необходимости. Оптимизация до окончания разработки делает разработку и поддержание кода сложнее, а оптимизация вообще эвристическое занятие и что и где лучше оптимизировать заранее сказать нельзя.

**Transactions and Locks**

Транзакция – единица работы в рамках соединения с БД. Она выполняется полностью (commit) или полностью откатывается (rollback).

ACID – набор свойств, которыми должна обладать транзакция в БД.

* Atomicity – Атомарность гарантирует, что никакая транзакция не будет зафиксирована частично. Будут выполнены либо все её подоперации, либо не будет выполнено ни одной.
* Consistency – Согласованность. Каждая успешная транзакция фиксирует только допустимые результаты. (это автоматически достигается в реляц. БД)
* Isolation – Изолированность. Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.
* Durability – Устойчивость. Независимо от проблем на нижних уровнях изменения, сделанные успешно завершенной транзакцией, должны остаться сохранёнными после возвращения системы в работу.

Проблемы транзакций:

* Lost Update (нет в современных СУБД)
* Dirty Read
* Non Repeatable Read
* Phantom Read

Для решения этих проблем существуют разные уровни изолированности транзакций:

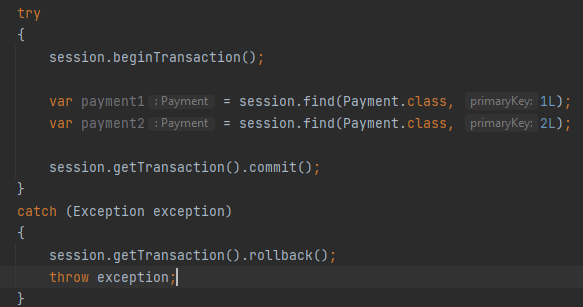


Чем выше уровень изоляции, тем хуже по производительности наше приложение. Так, по сути, для SERIALIZABLE наши транзакции выполняются последовательно т.к. там блокируется вся таблица для транзакции (на самом деле там делается что-то изолированных копий, но по производительности всё равно гораздо хуже).

В PosgreSQL по умолчанию READ COMMITED.

**JPA Transactions**

По-хорошему приходилось бы делать так: открывать транзакцию, делать работу, закрывать транзакцию. Проверять не получилось ли ошибки. Если получилась откатывать транзакцию и прокидывать ошибку дальше. Т.е. было бы что-то такое каждый раз при работе с сессией:



Кстати, объектом Session после неудачной операции нельзя пользоваться. Он портится.

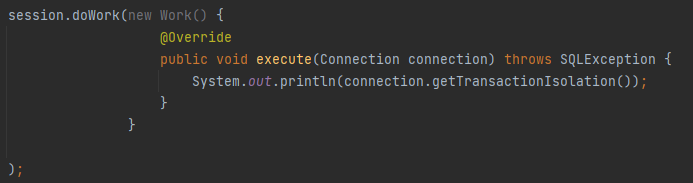
Есть механизм удобнее. Аннотация @Transactional. Суть в том, что какой-то отдельный класс открывает нашу сессию, открывает транзакцию. Вызывает основной функционал нашего кода и сам делает коммит или роллбэк. В Hibernate из коробки не реализован такой механизм. Но в спринге есть и там это вручную делать не нужно, только если захотим обработать какой-то специфичный exception самим.

Методов для установки уровня изолированности транзакций нет. Это делают через properties.



Значения: 1,2,4,8

Чтобы посмотреть уровень изолированности можно использовать следующий подход:



У connection есть и методы для установки такого уровня, но это не следует использовать. Его следует устанавливать через property. doWork() это по сути метод, которые даёт нам возможность работать напрямую с connection у session.

Но вообще лучше оставлять уровень дефолтным для вашей СУБД и устанавливать нужные вам блокировки только во время выполнения определённых запросов.

**Locks Optimistic**

Это первый вариант решения проблемы параллельного выполнения транзакций. Это решение с помощью java кода. Используя это нам не нужно прибегать к увеличению уровня изолированности транзакций, что хорошо сказывается на нашей производительности.

При запросе find() 3м параметром можно передавать LockModeType:



При решении через приложение:

* OPTIMISTIC
* OPTIMISTIC\_FORCE\_INCREMENT

При решении через БД:

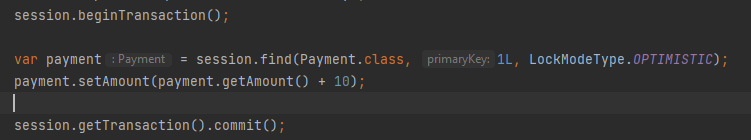
* PESSIMISTIC\_READ
* PESSIMISTIC\_WRITE
* PESSIMISTIC\_FORCE\_INCREMENT

На самом деле указывать OPTIMISTIC необязательно т.к. блокировки решаются на уровне самого приложения. Но нам нужно зайти в сущность и прописать аннотацию @OptimisticLocking(). В ней есть параметр type, по умолчанию он VERSION.

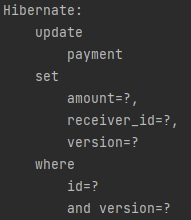
При использовании VERSION необходимо указать поле, в котором будет храниться версия строчки. т.е. будет просто ещё 1 колонка, в которой будет написана версия строки. Версия будет меняться при изменении строки.

Можно использовать вместо int Timestamp, но часы могут быть рассинхронизированы и это не рекомендуется использовать.

Теперь мы делаем update payment.



Hibernate будет делать следующий запрос

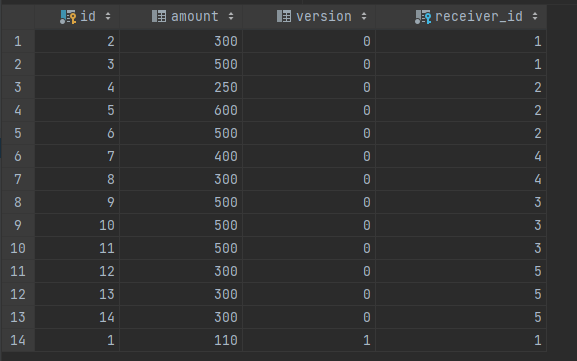


т.е. он проверяет в where не только id, но и версию строки, которую он хочет обновить. И если версии не совпадают -> не будет найдено строк -> update вернёт “0” т.е. изменил 0 строк -> он видит 0 и кидает OptimisticLockException.

Также он изменяет версию строки сам (в set есть поле version).

Т.е. так у нас решается проблема last commit wins и становится first commit wins.

Соответственно таблица payment выглядит вот так:



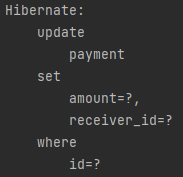
OPTIMISTIC\_FORCE\_INCREMENT обновит version, даже если мы не делали никаких изменений.

Для OptimisticLockException соответственно делаем уже try catch нашей сессии. И можем выдавать сообщение пользователю что, например, номер уже забронирован.

**Другие типы optimictickLock**

All – работает как и VERSION только нет отдельного поля и мы в where сравниваем не version, а все поля нашей сущности с строкой в БД и если они не совпадают, значит кто-то другой изменил нашу сущность -> ошибка.

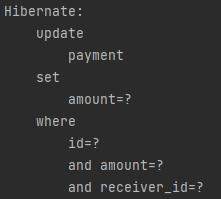
Но операция update для нашей сущности хранится закешированной (как и другие crud), чтобы каждый раз её динамически не составлять (там просто каждый раз обновляются все поля):



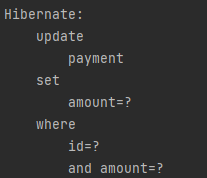
Хоть мы и обновили только поле amount, receiver\_id тоже перезапишется, но на то же значение, что и было.

Однако, у нас Update не будет стандартный. Он будет на основании всех наших полей. И для этого указываем аннотацию @DynamicUpdate. Также есть @DynamicInsert.

Изменили amount в payment



А Dirty смотри лишь dirty context и в where добавляет только те поля, которые изменились. Т.е. если изменилось amount в where мы добавим только amount.

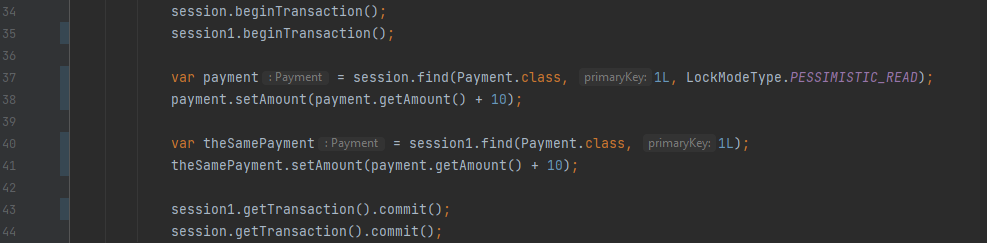


Короче на практике используют VERSION. Можно использовать ALL, а в DIRTY много подводных камней.

**Locks pessimistic**

Решение проблемы параллельного выполнения транзакций на уровне БД, но не поднимая уровня изолированности транзакций.

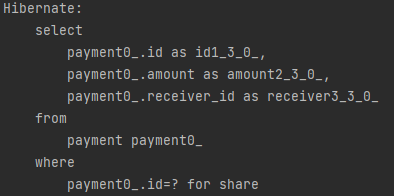
Такой код подвиснет до таймаута:



Произойдёт это, потому что session1 попробует сделать update, но эта строка будет заблокирована session, так что мы просто будем ждать на 43 строчке (а блокировку session отдать не может т.к. она отдаётся на 44 строке)

Однако select в 40 к этой строке сделать получится т.к. такой тип блокировки (for share в postgresql) это позволяет сделать. Что какие блокировки позволяют делать стоит смотреть для конкретной СУБД.

На строке 37 выполнился такой запрос:

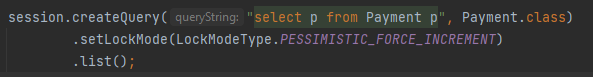


Если заменить PESSIMISTIC\_READ на PESSIMISTIC\_WRITE, то будет не for share в запросе, а for update на этом же месте.

И подвиснем мы точно также. Он ещё более строгий, чем for share, но обычный select он также не блокирует в posgresql.

PESSIMISTIC\_FORCE\_INCREMENT использует поле с @Version дополнительно его изменяя. Это нужно, если мы хотим поддерживать некую версионность нашего payment. В любом случае не будет недопустимых операций, но так ещё и версии будут записываться, если это нам нужно.

LockMode устанавливается в HQL так:



Устанавливается блокировка на все строки, которые мы вернули запросом, поэтому лучше её использовать в HQL только если не так уж много строк возвращается. Тем более такую т.к. тут нам придётся ещё и обновлять каждую строку, чтобы изменить версию (если мы не только на read их вытащили).

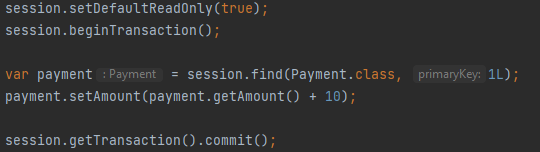
Чтобы избежать таких подвисаний, нужно установить таймаут. в HQL это setHint (\*как долго он будет держать этот lock\*) или setTimeout().

Итак, общее правило: мы используем какие-то optimistic или pessimistic блокировки только запросов, а не повышаем уровень изолированности в целом.

**Read Only Mode**

Если мы знаем, что наши сущности, запрашиваемые из БД, не будут изменяться и нужны нам только для чтения, то мы можем выполнять разные оптимизации на уровне приложения и на уровне БД.

Чтобы обезопасить себя от лишних изменений наших сущностей мы можем установить session.setDefaultReadOnly(true); Но это для всех наших сущностей. Либо можем установить это даже для какой-то конкретной сущности. Hibernate видит этот флаг и не делает dirty check для этой сущности. В этой проверке сравниваются состояние сущности как только она попала в persistenceContext с состоянием, когда делается flush. Она трудоёмкая.

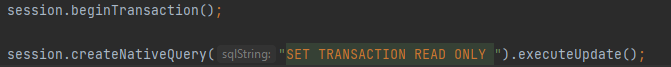


В таком случае всё будет работать, но payment не обновится в БД.

Можно устанавливать не только через session, но и в HQL. Так readOnly установится всем сущностям, которые вернуться от этого запроса.

через setReadOnly или setHint.

Но это оптимизация на уровне приложения. А можно пометить readOnly в БД. И тогда даже если мы пошлём изменение в БД из нашего приложения, то пробросится exception.



Кроме того, если мы дадим БД знать о read only, то она может выполнит какие-то оптимизации для вашей транзакции. Это стоит смотреть для конкретной СУБД.

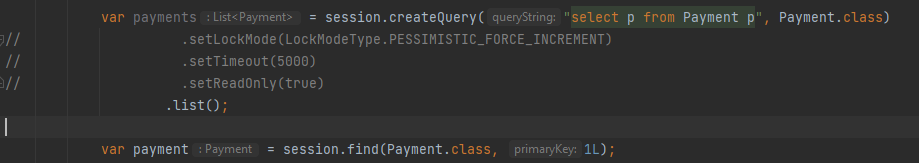
Рекомендуется использовать setDefaulrReadOnly для только read транзакций. И + если хотим обезопасить себя от изменений, чтобы выбрасывалась ошибка и получить какие-то оптимизации выполняем set transaction read only для БД, после начала транзакции.

**Nontransactional Data Access**

Hibernate предоставляет нам функционал для выполнения sql запросов без явного открытия транзакций. Это автокоммит мод (как в mysql. Открывается автоматически транзакция, выполняется код и она закрывается).

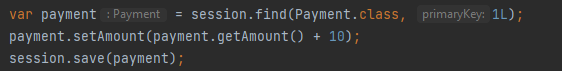
Но у Hibernate поведение необычное в этом моде. Можно делать read.

Это будет работать и без транзакций

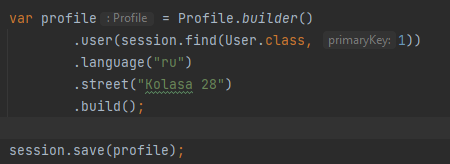


Можно изменять объекты без транзакций. Тогда просто они не изменятся в БД.

Это тоже будет работать. Но в БД update не будет



А если сделать руками flush() то будет ошибка.

И даже это сработает, но только если id - IDENTITY

Короче там много тонкостей, так сложнее и вообще рекомендуется не использовать такой подход или использовать только на чтение.

**Listeners & Interceptors**

**Entity callbacks**

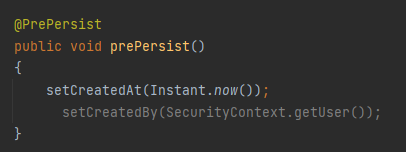
Есть интерфейс Callback. Они нам нужны для того, чтобы перехватывать какие-то события из нашего жизненного цикла сущностей и что-то выполнять в этот момент.

бывают вот такие callback:

* @PreUpdate
* @PostUpdate
* @PrePersist
* @PostPersist
* @PreRemove
* @PostRemove
* @PreLoad

И мы можем внедриться в каждую из этих частей жизни сущности и что-то сделать. И как видно, это аннотации. Так что мы можем реализовать это либо в наших сущностях, либо в отдельных Listner. Callback реализуют EntityCallback и ListenerCallback.

Например, мы хотим перед тем, как сохранить нашу сущность Payment установить ей поля createdAt и createdBy (содержащиеся в AuditableEntity). Всё что нужно для этого сделать:



Метод в Payment должен быть помечен этой аннотацией, иметь любое имя и быть void.

Остальные аннотации использовать аналогично.

Но если мы хотим организовать такой аудит для всех наших AuditableEntity. Мы должны перенести все эти аннотации из Payment в AuditableEntity и это также будет работать.



Однако в сущностях должны быть только поля, а prePersist, preUpdate какой-то функционал -> лучше их выносить в отдельные классы (Listner’ы).

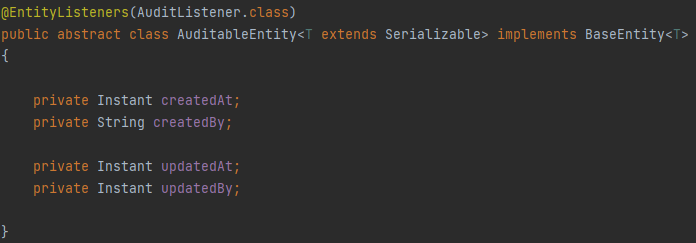
**Listener callbacks**

Просто вынесем логику из сущности. Для этого создадим такой класс:



В нём нам уже придётся принимать сущность т.к. раньше мы понимали к какой сущности этот callback через this. Интерфейс CallBack как раз принимает параметр Object entity для этого. И вообще-то должен быть Object, но т.к. мы знаем, над каким типом мы повесим этот AuditableListener -> мы знаем какого типа будет объект в runtime мы можем написать вместо Object тип AuditableEntity<?>.

Hibernate не знает про этот Listner и чтобы он про него узнал есть специальная аннотация @ EntityListeners() чтобы оповестить Hibernate.



Есть только 1 ограничение. Если мы хотим сделать несколько одинаковых аннотаций, их придётся сделать в разных Listener т.е. 2 @PrePersist написать в одном Listener нельзя.

Другой пример:

Можно часто встретить различные денормализации БД и Listner’ы отлично подходят для решения проблем такого типа. Например, мы хотим отображать в чате количество наших пользователей в нём. Для этого нам необходимо достать userChats и вызвать size(). Что не очень круто. В реальной практике часто прибегают к денормализации.

В данном случае сделаем count в Chat. И теперь нам нужно следить за этим полем и обновлять его. Создаём Listner для UserChat, которое будет обновлять поле count в chat.

Вот такой Listener:

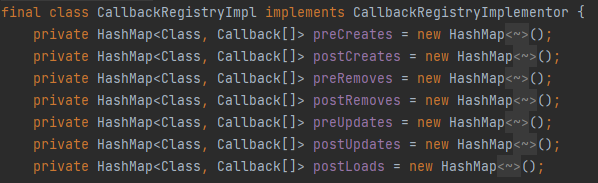


И размещаем его над UserChat.

Но теперь нам нужно выполнять операции, связанные с UserChat только через Hibernate. Если мы это будем делать вручную, через SQL запросы, то мы нарушим наш count в chat. В этом и есть сложность денормализации, зато мы заметно выигрываем в производительности.

Ну и можно отправлять всякие сообщения в этих listener. Например, отправить сообщению пользователю, если появился новый и т.д.

Хранится это так: по классу получаются все его callback определённого типа.

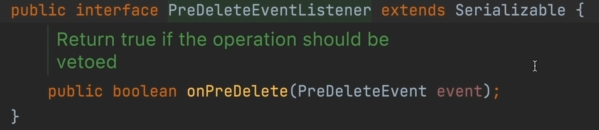


**Event Listeners**

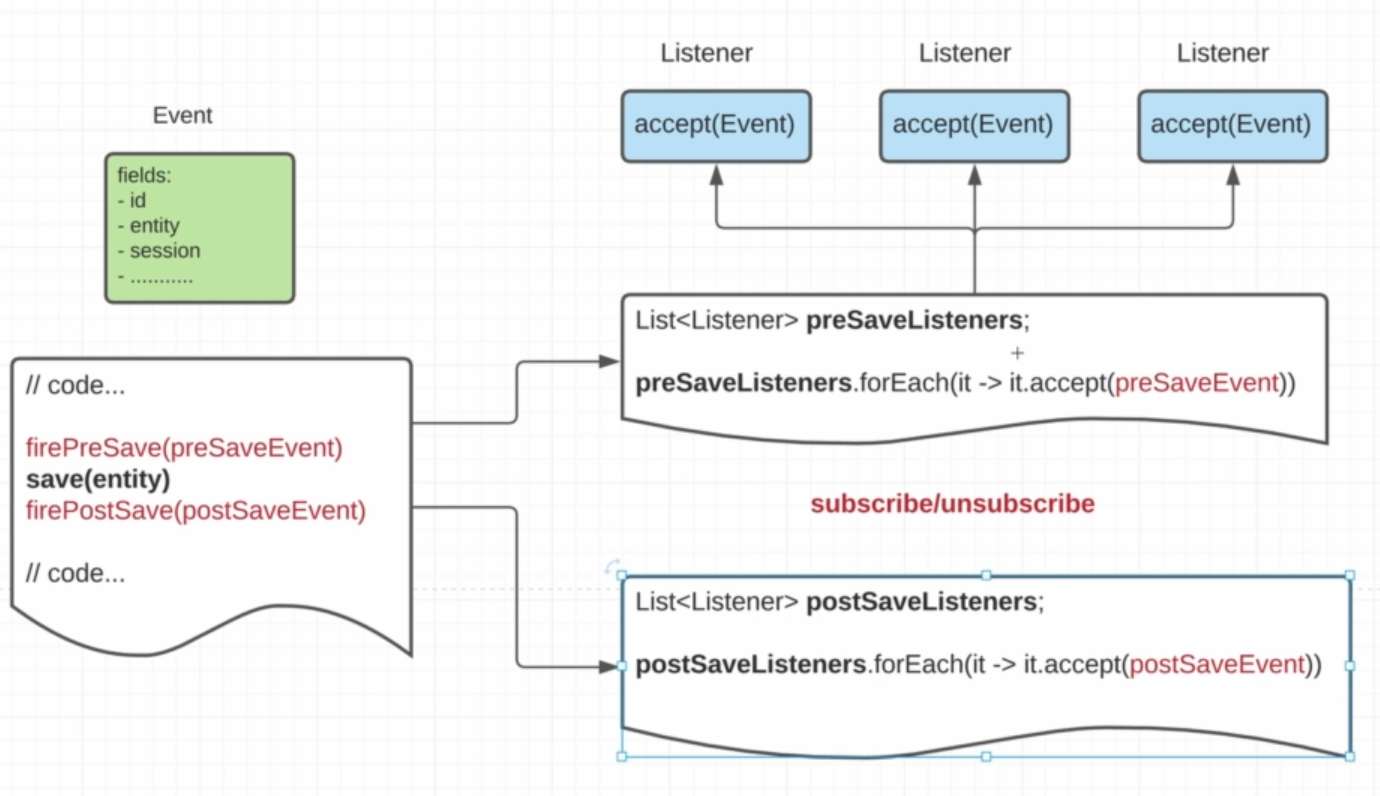
Это ещё одно средство для работы с Listener, помимо CallBack, которое принято в Hibernate.

Даже save сделан через event и listener.

Сами объекты Listener это обычные функциональные интерфейсы. В них всего 1 метод, который нужно переопределить. Он занимается прослушиванием какого-то типа (PreDeleteEvent в примере)



Получается такая схема:

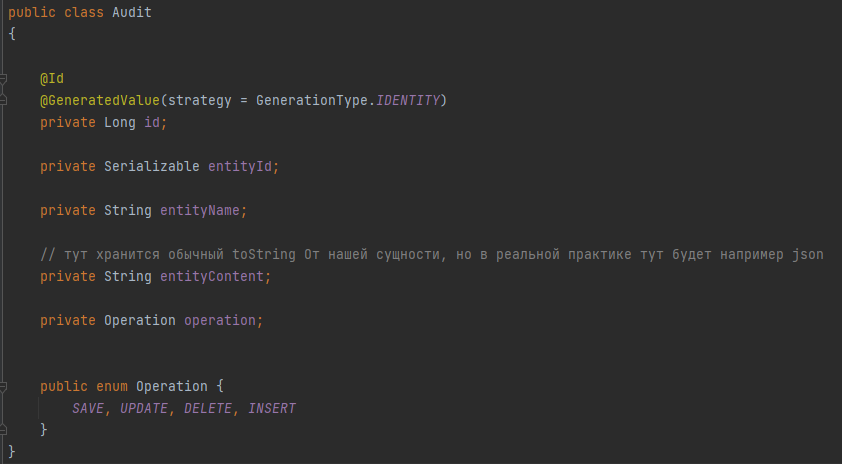


Добавление в списки Listener либо удаление называется subscribe/unsubscribe.

Event это по сути DTO просто со множеством полей.

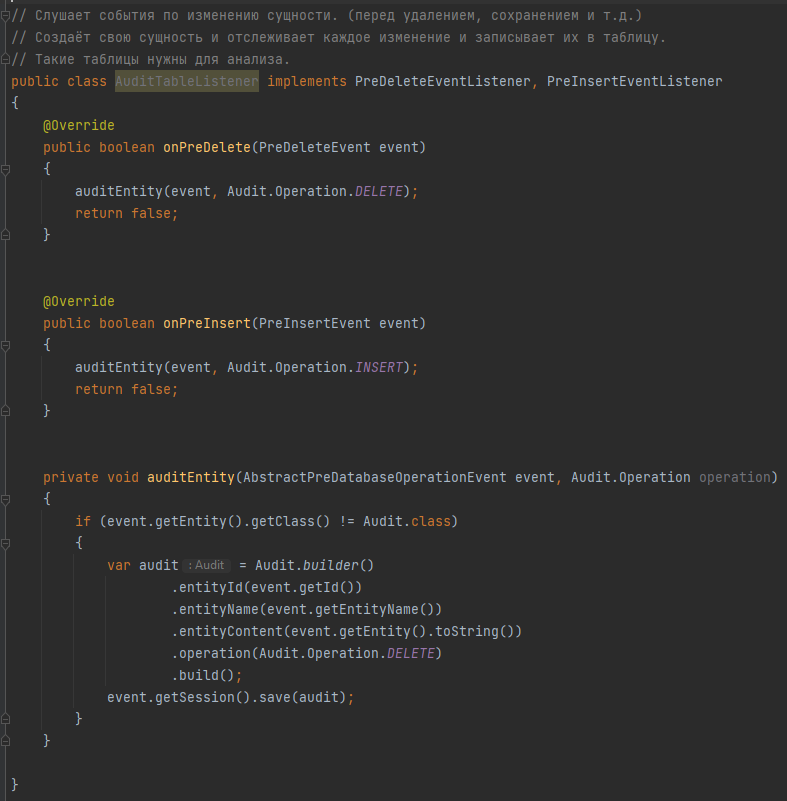
Но в Hibernate даже save сделан через Listener.

Создадим свой класс Audit которой будет обычным Entity:



Такая таблица Audit потом используется для анализа.

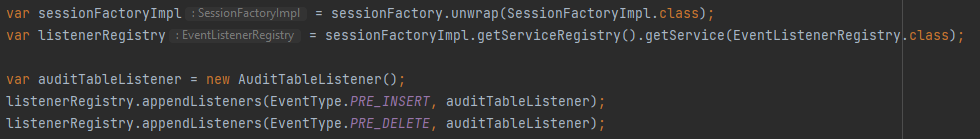
Создадим свой Listener:



if внутри auditEntity() нам нужен т.к. при сохранении в БД audit у нас опят вызовутся onPreInsert listener’ы и чтобы не было цикла и мы не делали аудит на наш удит мы написали if.

Теперь нам нужно добавить этот Listener в соответствующую группу (т.е. в preDelete группу и PreInsert).

Делается это через SessionFactory:



Метод unwrap просто делает каст до конкретной реализации SessionFactoryImpl.

Итог: мы подключились в жизненном цикле сущностей при создании и удалении. И точно также можем подключиться в любой другой момент жизненного цикла из EventType.

**Interceptors**

Ещё 1 механизм. Редко используется на практике. Есть интерфейс Interceptor. В нём куча методов для всех возможных точек внедрения. Типо onSave, onFlushDirty, onDelete и т.д. таких куча. Реализовывать это сложно.

Есть класс EmptyInterceptor. В нём прописаны дефолтные реализации для этих методов или просто пустые тела, если метод void.

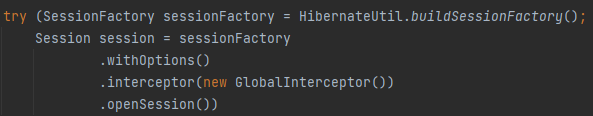
Мы наследуемся от EntityEnterceptor и переопределяем нужные нам методы. Можно создать всего 1 Enterceptor, в отличии от Listner’ов которых мы можем создать сколько угодно.

Т.к. он всего 1, подключать его через configuration:



Но на самом деле мы можем задать Interceptor для какой-то конкретной сессии. Если мы хотим, чтобы в этой сессии использовался не тот Interceptor, который достался от SessionFactory, а какой-то свой, то можно написать это:

Но опять же он будет только 1, просто другой.

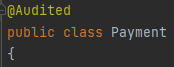


Итог: рекомендуется использовать event Listener (но он общий, для внедрения в жизненный цикл всех сущностей, если if не писать, как я понял). А если мы хотим внедриться в жизненный цикл только какой-то 1 сущности, то используем callback (вроде так, лично мой вывод).

**Hibernate Envers**

Мы создали AuditTableListener для аудита нашей системы, но есть hibernate envers. Это встроенный механизм аудита для нашей системы. Точно также реализованный через Listener’ы.

Для того, чтобы таблица аудировалась всё что нам нужно сделать – поставить над ней соответствующую аннотацию. Например, я хочу чтобы для таблицы Payment сохранялись и логировались все её изменения. Для этого я помечаю класс Payment @Audited.

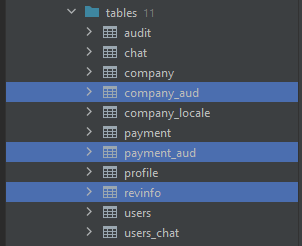


Нюанс:

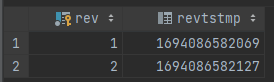
Если в нашей сущности (Payment) есть маппинг на другие сущности, то мы также должны аудировать и эти сущности (поставить @Audited над User). Если же мы не хотим аудировать эти сущности, то тогда ставим над полем @NotAudited.

Всё. Расставили аннотации (над Payment и Company), запустили. Появились

3 таблицы:

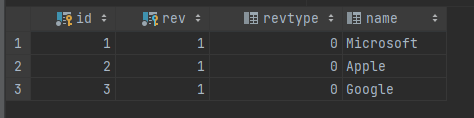


В таблице revInfo сохраняется ревизия. Эта ревизия соответствует транзакции (revtstmp – время, когда произошла транзакция). Как только мы открыли транзакцию создаётся запись в revInfo.



Далее все изменения записываются в соответствующие таблицы company\_aud и payment\_aud в которых есть ссылка на ревизию (транзакцию) в рамках которой они были сделаны.

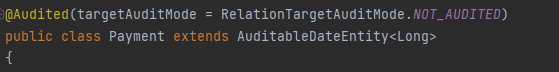
Пример таблицы company\_aud:



* id – это id сущности
* rev – номер ревизии (транзакции)
* revtype – тип операции (0 – create, 1- update, 2 - delete)
* все поля нашей сущности Company (кроме тех, что @NotAudited) т.е. в данном случае просто name

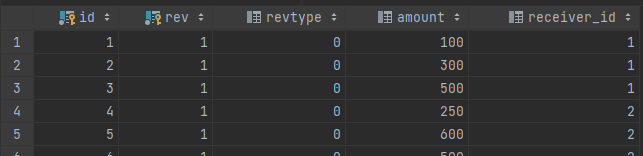
Но что, если мы можем поменять поле User в Payment? @NotAudited нас уже не устроит, ведь мы хотим видеть изменение этого поля, но аудировать User мы тоже не хотим. Для этого в @Audit есть несколько полей.

targetAuditMode если прописать NOT\_AUDITED т.е. не аудировать сущности, которые являются полями в нашем Payment.



Но если наше поле является не сущностью, а коллекцией, мы всё равно должны ставить над ней @NotAudited (логично, ведь коллекция не является полем в нашей таблице).

Теперь таблица будет содержать receiver\_id поле тоже

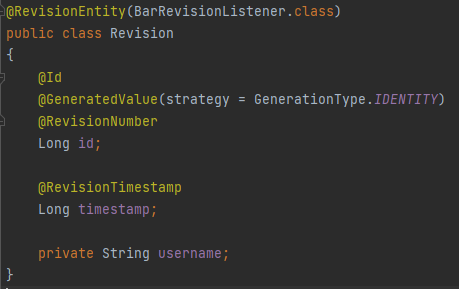


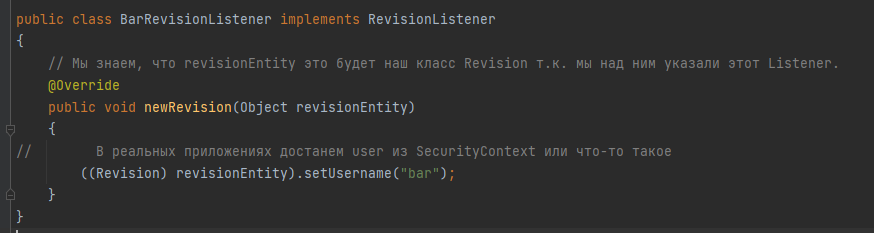
В коде всё это реализовано через event Listner’ы Hibernate. Например, insert. После него вызываются PostInsertEventListner’ы. Есть listener Envers PostInsertEventListnerImpl который проверяет помечена ли сущность аннотацией @Audited. И если да, то производит аудит её. Всё.

Таблица revInfo сгенерирована автоматически, но мы можем сделать свою вместо неё с какими-то дополнительными полями. Для этого создадим сущность Revision. В неё должно быть как минимум 2 поля. Id (должен быть числовым) и timestamp (дата или лонг) название может быть любое.

Чтобы Hibernate envers понял, что именно эту сущность нужно использовать, а не создавать свою, мы помечаем класс аннотацией @RevisionEntity (может быть только одна на весть проект). И пометить поля аннотациями @RevisionNumber и @RevisionTimestamp.

Дальше, например, мы хотим добавить поле, указывающее на того, кто выполнил эту транзакцию. Добавим поле String user для этого. Hibernate envers сам устанавливает значения в timestamp, а в id это делает БД. Но чтобы устанавливать значение в другие наши поля, нам нужно создать RevisionListner и указать его в параметрах.



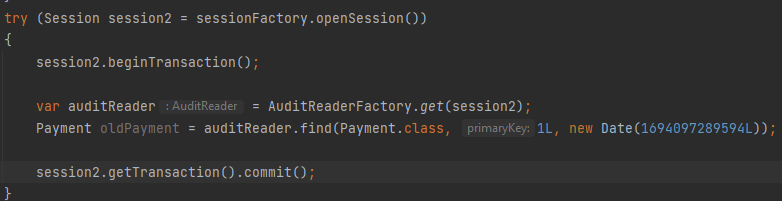


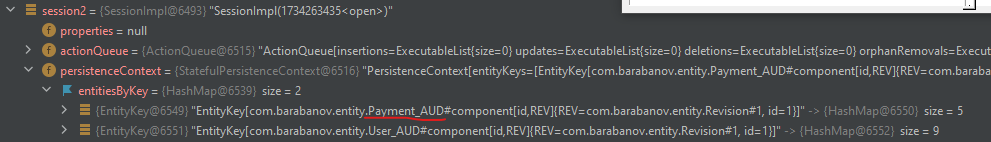
**Преимущества разбиения аудита на отдельный таблицы для каждой сущности**

Company\_aud и другие таблицы для аудита – не являются сущностями, а просто какие-то таблицы и мы не можем обращаться к ним через session.

Для этого у нас есть AuditReaderFactory.get(\*session\*) он возвращает AuditReader, работа с которым очень похожа на работу с Session.

find() нам возвращает Payment, но в persistenceContext будет не объект Payment, а Payment\_AUD. т.е. динамически создался класс Payment\_AUD.



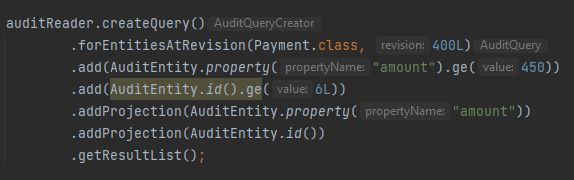


Если мы сделаем такой же запрос через session (без указания ревизии) мы получим тоже payment, но в persistenceContext добавится обычный Payment.

Если у нас всего 3 ревизии объекта, а в revision мы укажем больше 3х, то получим всё равно 3ю ревизию объекта.

У auditReader есть createQuery(). Он позволяет делать запросы наподобие Criteria API. addProjection позволяют выбирать какие-то конкретные поля у таблицы.

Пример:



Можно не только перемещаться во времени (как в Git), но и сохранять текущее состояние определённой ревизии в нашу БД.

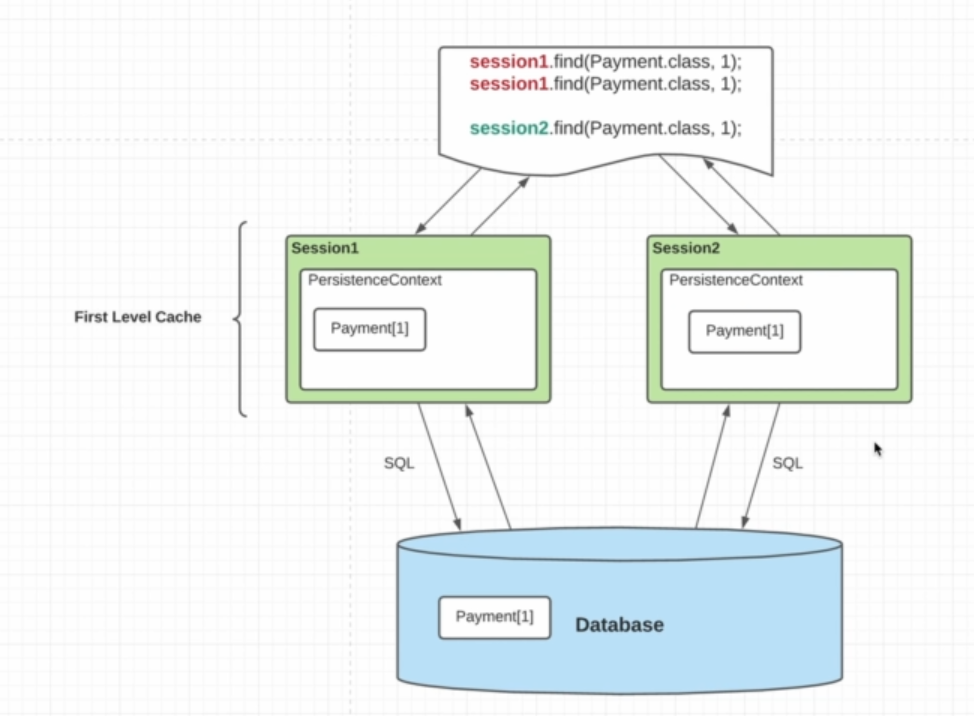
т.е., например мы хотим накатить наш oldPayment на текущее состояние в БД.



И это очень мощный функционал, который мы не можем предоставить с помощью единственной таблицы audit.

**Second Level Cache**

Вспомним про First level cache:



Код будет выполняться так:

Первый find() посмотрит в PersistenceContext для Session1, увидит что там нет объекта по такому ключу и сделает запрос к БД.

Второй find() найдёт объект в PersistenceContext.

Третий find() посмотрит в PersistenceContext для Session2, увидит что там нет объекта по такому ключу и сделает запрос к БД.

т.е. 2 запроса к БД, 2 разных объекта Payment, а строка одна и та же в БД.

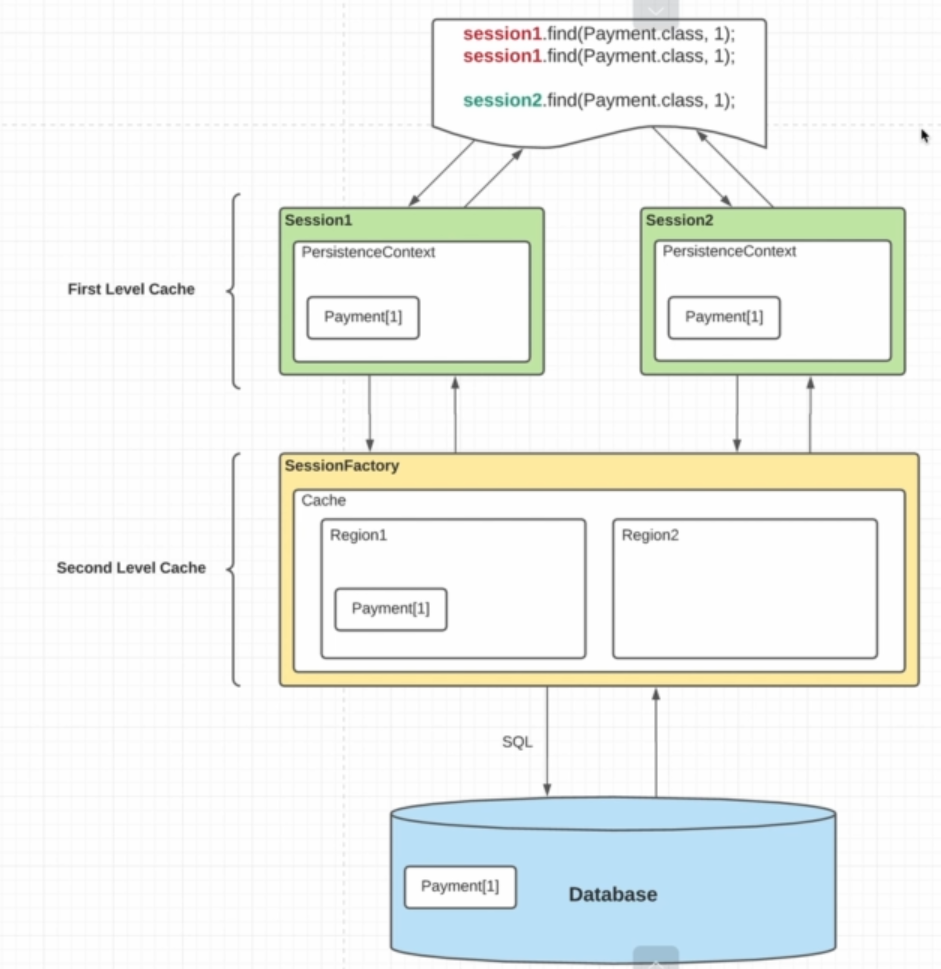
First level cache включен всегда. Первый уровень в Session. Второй уровень на уровне SessionFactory. Second level cache нужно настраивать и включать вручную.

Сам кеш разбит на регионы. Мы сами решаем, что за регионы там будут. Регион – аналог persistenceContext. Там будет Map<key – Serialized entity>. Но тут Entity будет не в виде объекта, а в виде значения.

Разбивать на регионы нужно чтобы их можно было тоньше настраивать по объёму памяти, который будет занимать этот region, как долго будет жить сущность в этом region.

Каждой сущности приписывается определённый region. Обычно, каждой сущности приписывается свой отдельный region.

Как он работает:



Для первой строки обращаемся в persistenceContext для Session1, но теперь вместо обращения к БД мы идём в SessionFactory и смотрим есть ли в его кэше, в том region, который мы установили для нашего Payment закешированное значение -> нет -> делаем запрос в БД. Помещаем Payment в region1 и в persistenceContext session1.

Для второй строки идём просто в persistenceContext session1.

Для третьей строки идём в persistenceContext для Session2, там нет объекта. Идём в region для Payment, там есть объект. Возвращаем его в Session2 в persistenceContext.

Cache Miss – значит обратились к кешу, но не нашли там какое-то значение (первый find)

Cache Put – когда не нашли какой-то сущности, обратились к БД и положили значение в кеш(первый find)

Cache Hit – когда обращаемся к кешу и пытаемся достать сущность, которая в нём есть. Т.е. мы попали в кеш удачно (третий find)

SecondLevelCache нам пригодится когда мы делаем запросы на получение каких-то справочных значений. Справочник стран, категорий в интернет-магазине, меню еды. В общем, данные, которые почти никогда не меняются.

Плохо использовать для сущностей, которые часто меняются.

Чтобы подключить этот кеш есть 2 интерфейса.

1. RegionFactoryTemplate – занимается тем, что создаёт region’ы для наших сущностей.
2. DomainDataStoreAccess – должны быть методы для получения из кеша значения, положить в него значение, удалить из кеша значение и т.д.

Это интерфейсы. Нам нужно либо их реализовывать самим, либо подключать библиотеки с реализацией этих интерфейсов.

Библиотека – jcache (это просто API). А конкретная реализация ehcache:ehcache. Модуль hibernate-ehcache не следует использовать, он deprecated.

Теперь у нас есть реализации JCacheRegionFactory и JCacheAccessImpl.

Но функционал простой и вообще это можно и самому даже сделать.

Теперь скажем Hibernate, что мы хотим использовать second level cache:

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

И укажем провайдера кеша (класс реализации):

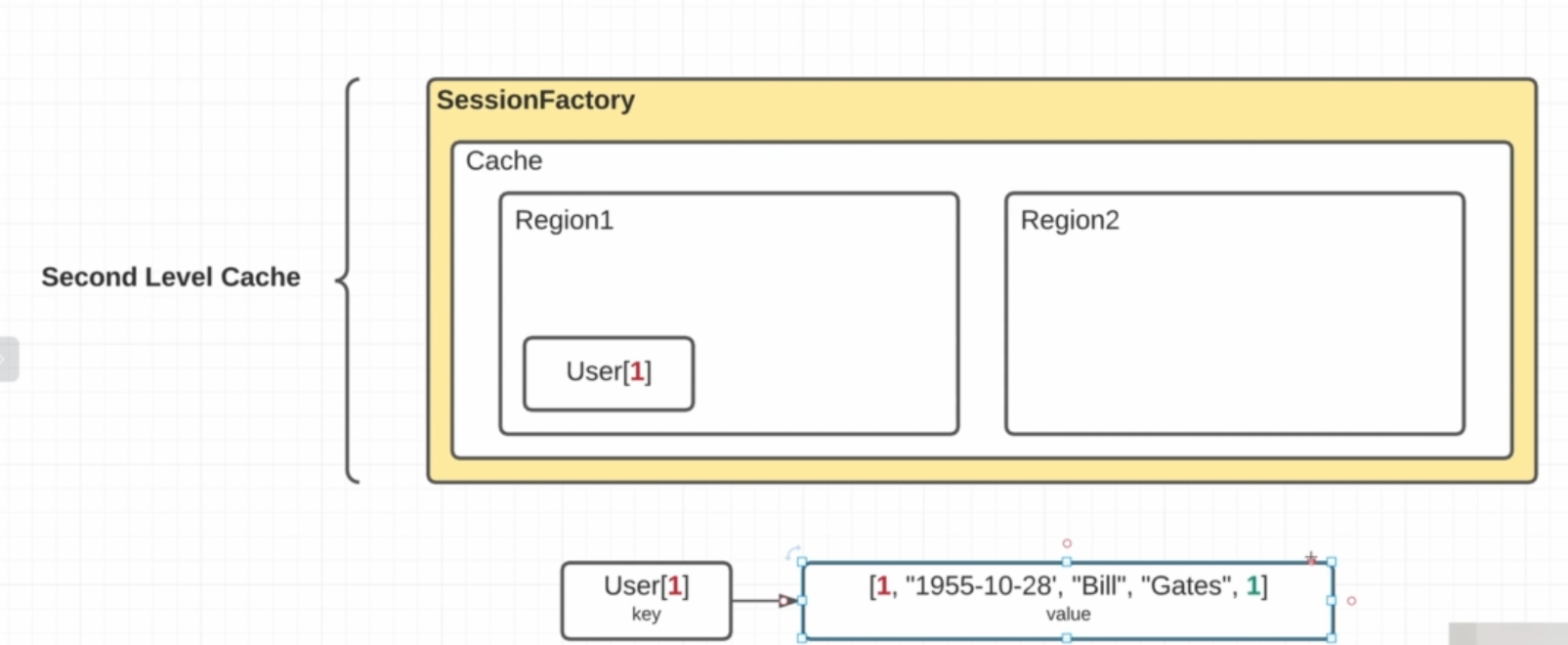


**@Cache**

Кеш второго уровня должен быть явно указан для той сущности, которую мы хотим закешировать, в отличии от кеша первого уровня.

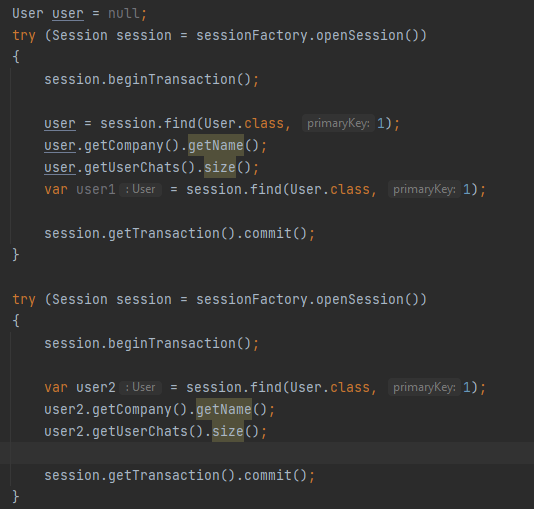
В нашем примере помечаем User аннотацией @Cache

При использовании кеша второго уровня всё равно создаётся новый объект (при использовании кеша первого возвращается один и тот же). Потому что сохраняется в secondLevelCache не сущность, а его сериализованная форма.



Но поле Company в User (ManyToOne) не кешируется (т.к. мы не ставили аннотацию Cache над ней). Она Lazy стоит так что при запросе к company у User из разных сессий будет 2 запроса к БД для получения company для этого user. Пометим и Company @Cache. Тогда company тоже будет класться в кеш при первом обращении к ней.

Но что, если мы хотим в нашем User кешировать не только Company (ManyToOne), но и коллекцию userChats. Если мы просто поставим аннотацию Cache над UserChat, то всё равно будет выполняться повторный запрос, чтобы вытащить userChat для user, который уже есть в кеше.

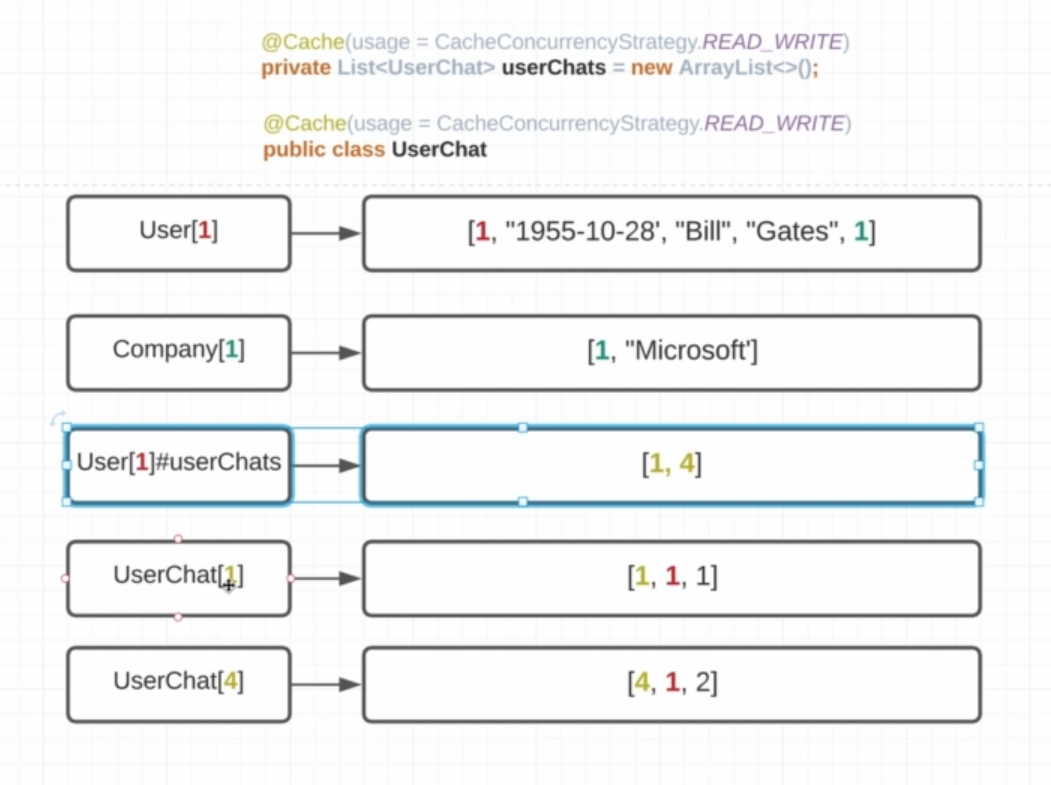


Это будет так работать ведь у User нет поля, где хранились бы id UserChat, которые ему принадлежат. -> нужно искать какие UserChat ему принадлежат в БД.

Чтобы не выполнялся повторный запрос для того, чтобы узнать UserChat у кешированного user, пометим поле userChats у User тоже аннотацией Cache.



Тогда закешируются все id UserChat (появится отдельный кеш для userChat’в пользователя). User[]#userChats на картинке.



По этим id’шникам мы опять обращаемся в кеш и поискать userChat (а над ними тоже уже стоит Cache они тоже уже есть в кеше второго уровня, после первого запроса).

Если забыть поставить над классом UserChat аннотацию, то просто при поиске userChat’ов по их id их не будет в кеше и произойдут запросы к БД (а вот запросов на userChat для такого-то user id не будет т.к. мы знаем какие userChat нам нужны).

Так что для кеширования коллекций нужно ставить 2 аннотации.

**Region**

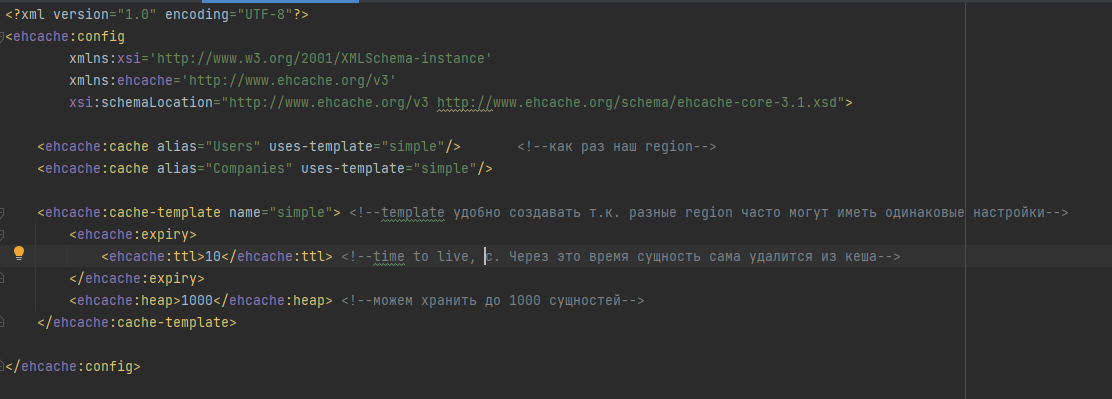
Region – всего лишь область памяти в нашем кеше, которая используется для хранения сущностей (их сериализованного значения).

Когда мы сохраняем User мы указываем region. Если мы его не указываем, создаётся region по умолчанию и он кладётся туда. По умолчанию для каждой сущности (не только сущности, а везде, где мы указываем аннотацию @Cache) создаётся отдельный region.

Чтобы задать region явно, мы можем в аннотации @Cache прописать region явно.

Чтобы настраивать region нужны специальные настройки, в зависимости от выбранного кеш провайдера. В случае ehcache мы должны настраивать region через xml маппинг.

Пример:



Укажем в Hibernate конфиге путь до конфига для конфига кеша:



**ConcurrencyStrategy**

Это то, как работают транзакции с нашем кешем. Т.е. локи и одновременная работа с нашим кешем.

READ\_ONLY – мы не можем сущность изменить, только удалить или добавить (обычно справочники какие-то. Справочник стран и т.д.)

READ\_WRITE – добавлять, удалять и изменять сущности. Происходит lock сущности, пока какая-то транзакция пытается изменить её. Когда транзакции видят, что стоит lock (т.е. кто-то обновляет её), они делают запрос в БД, а не в кеш. А после того, как изменяющая сущность транзакция делает commit, сущность вновь кладётся в кеш.

NONSTRICT\_READ\_WRITE – почти как обычный read\_write, но чуть менее жесткий. Прочитать при необходимости.

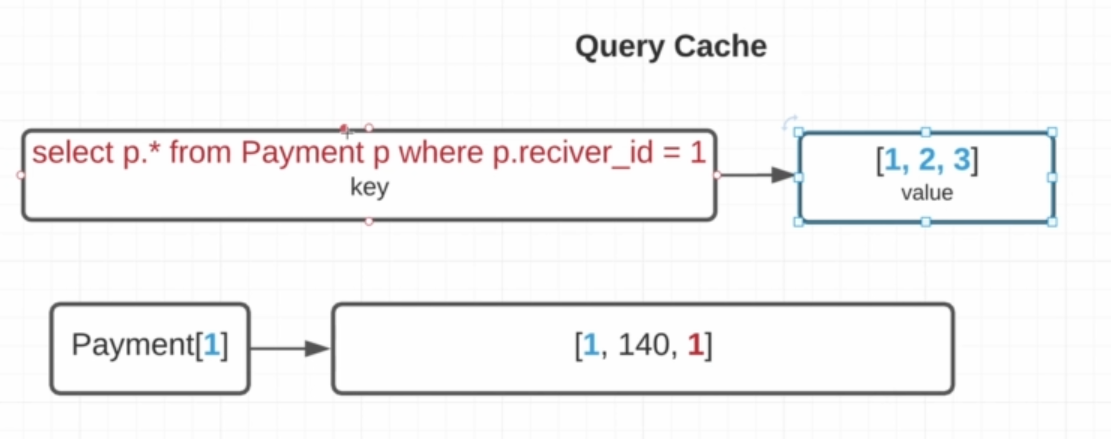
TRANSACTIONAL – распределённые транзакции. Подробнее читать, если что.

**Query Cache**

Second level cache позволяет сохранять не только сущности, но и SQL запросы.

С сущностями используется класс + ключ для получения значения.

С запросами мы используем всю строку нашего запроса т.е. String – ключ нашего Query cache. А в карте значений будут аналогично только id.

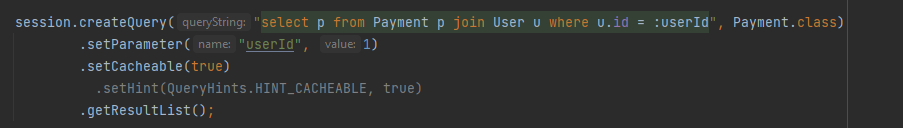


Но в этом примере нам, как и раньше нужно поставить @Cache над Payment иначе мы сохраняем только id -> возникнет проблема как с коллекциями. Сколько id мы сохранили, столько запросов и произойдёт. А с @Cache над Payment сущности с такими id вытащатся тоже и кеша.

Указываем, что мы будем использовать кеш запросов:



setCacheble даёт знать, чтобы мы поместили этот запрос в кеш запросов.



Без указания setCacheble не будет искаться в кеше запросов.

Можно указать <property name="hibernate.generate\_statistics">true</property>

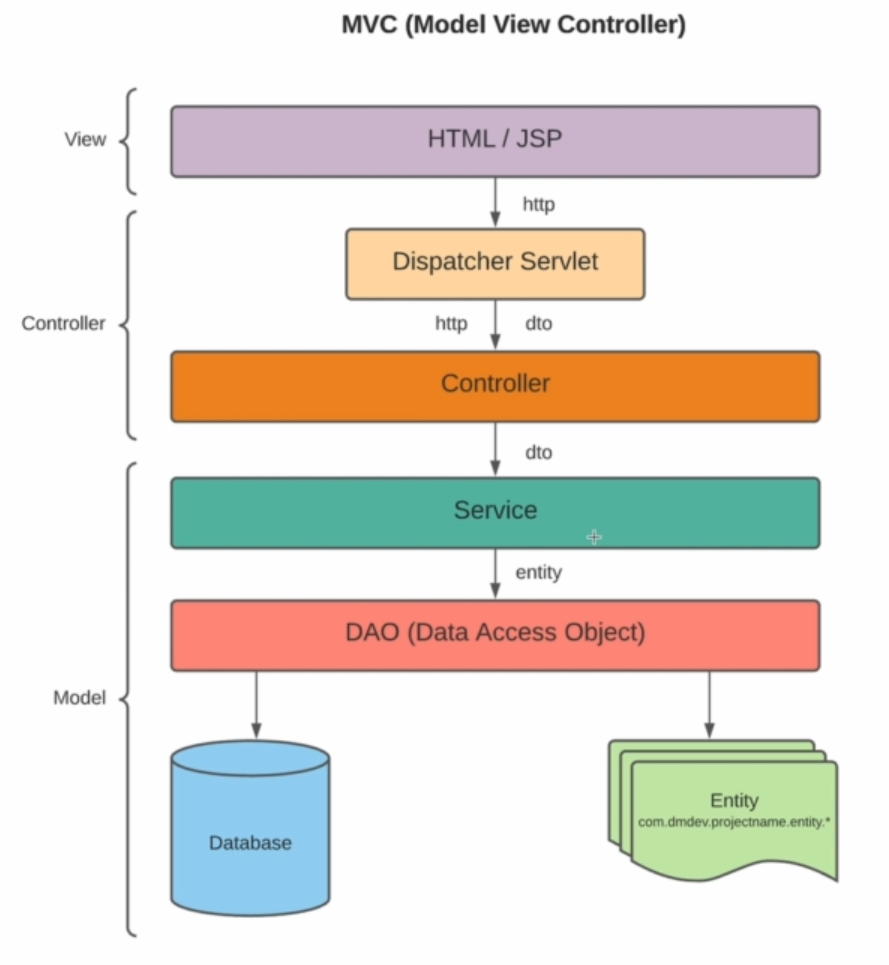
Для генерации статистики по использованию кеша. Там тьма различных данных. Например, так можно смотреть сколько раз был put, miss, hit для конкретного региона, сколько JDBC stmnt было сделано и куча всего всего всего.

Итог: кеш стоит использовать, он используется везде. Но делать это стоит аккуратно и продумано, иначе можно даже ухудшить производительность приложения.

**DAO & Repository**

**CRUD**

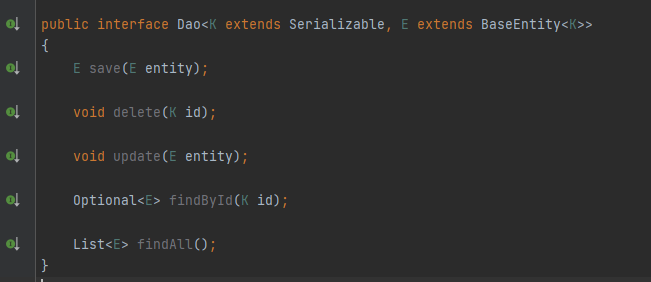
MVC (Model View Controller)



Hibernate будет использоваться в 2х слоях. DAO и Service.

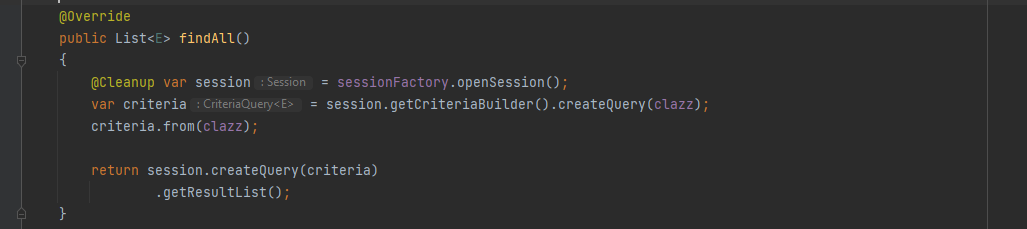
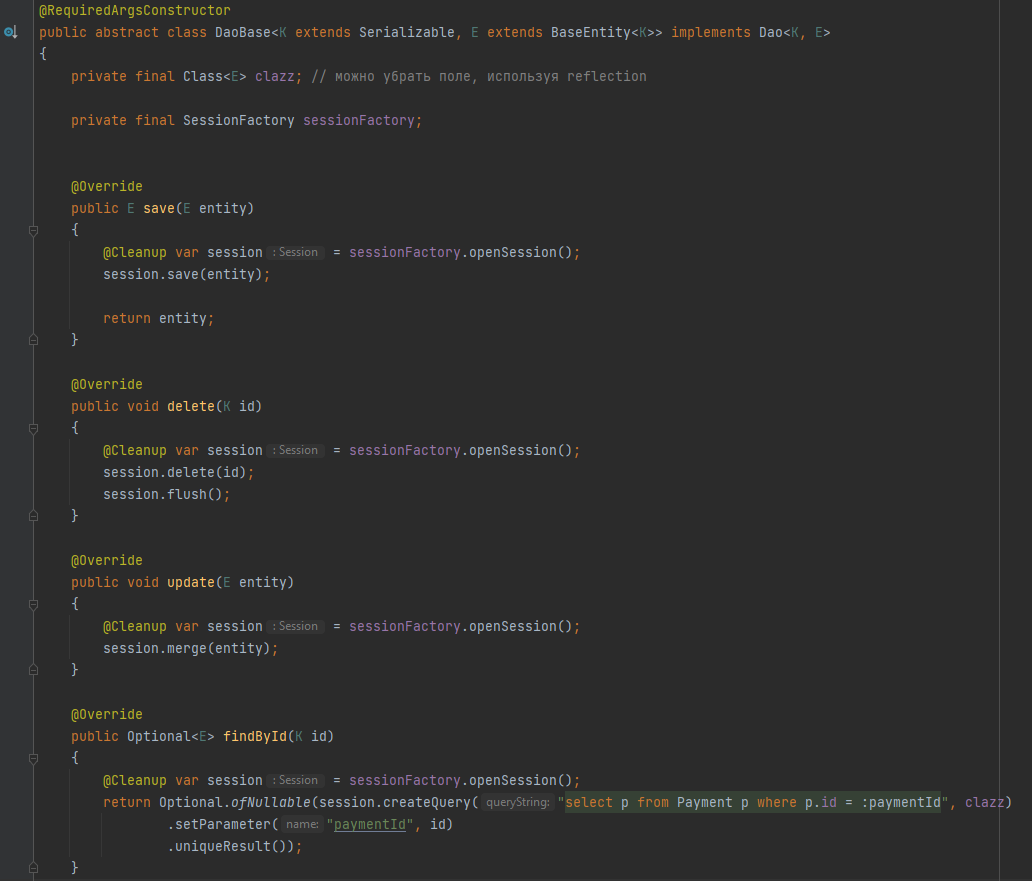
Реализуем для примера.

Создадим интерфейс DAO:

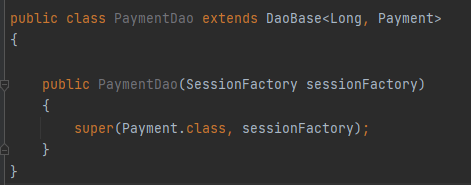


Поскольку эти методы будут реализовывать все Entity, чтобы каждый раз их не переписывать, сделаем абстрактный класс с их реализацией.

Получим вот такой Базовый класс:



Тогда конкретные Dao будут выглядеть уже так:



Теперь у каждой сущности есть свой DAO и нам нужно всего лишь сделать extends. Остальные операции уже добавляем по мере необходимости в конкретные DAO.

Но вообще-то на слое DAO мы занимаемся маппингом из реляционной, в сущности, и обратно. А тут мы этого не делаем, а взаимодействуем лишь с сессией. Т.е. мы работаем с каким-то слоем, который стоит между Service DAO. Такому слою дали имя Repository. Он работает DAO, но не работает с конкретными маппингами сущностей.

Поэтому правильнее именовать не Dao и DaoBase, а Rerository и RepositoryBase.

!!!(Однако именно при таком подходе у нас постоянно будут Lazy…Exception т.к. мы закрываем нашу сессию, с помощью которой мы получили сущность, а нам может потребоваться эта сессия для получения каких-то полей сущности.)

**CurrentSessionContext**

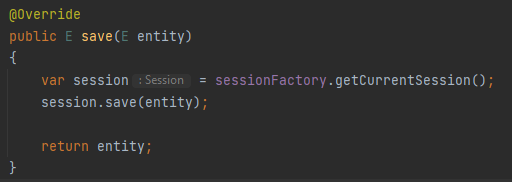
Проблема в том, что мы закрываем нашу сессию после вызова каждого метода, а это плохо.

Что делать? Мы работаем с этими методами на уровне сервисов.

В каждый метод передавать Session, но тогда на уровне сервисов работать с Session и обязаны в каждый метод передавать session, что очень неудобно. Мы видим параметры, о которых не обязаны знать.

Но и @Cleanup мы убрать не можем т.к. тогда сессия не будет закрываться и пул быстро израсходуется и будет утечка.

Решение: не открывать новую сессию, а получать текущую, sessionFactory.getCurrentSession()

****

Но как Hibernate будет понимать откуда вынимать эту текущую сессию?

Для этого есть специальный класс, отвечающий за стратегию того, как мы получаем эту сессию. Интерфейс – CurrenSessionContext. Нам нужен класс ThreadLocalSessionContext. Он автоматически при открытии транзакции создаёт сессию и когда мы делаем commit или rollback транзакции он удаляет её из TreadLocal переменной (и закрывает её).

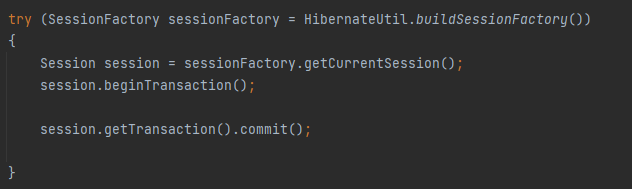
Класс ManagedSessionContext – работает так же, только автоматически не закрывает / открывает сессии.

Скажем Hibernate какой SesssionContext использовать:



Теперь в HibernateRunner мы делаем не openSession, а тоже getCurrentSession().

Теперь нам даже не нужно её закрывать. Главное начинать и завершать транзакции. А без транзакции будет ошибка.



На уровне сервисов всегда идёт работа с транзакциями. Здесь происходит открытие транзакции, вызов всех необходимых методов, согласно нашей бизнес логике и закрытие транзакции.

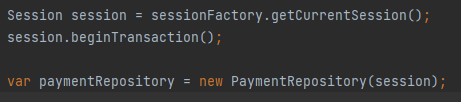
Дальше.

В RepositoryBase у нас зависимость на SessionFactory. А по-хорошему она должна быть на EntityManager (ну или Session. Session это реализация EntityManager).

Получается так:

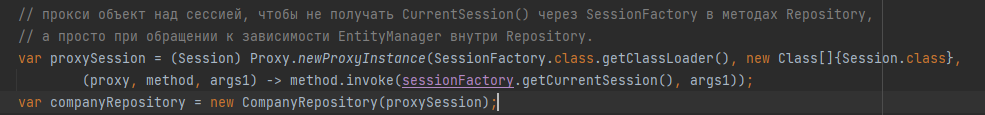


Но до этого в многопотоке мы бы для разных потоков получали бы разные session. А сейчас, если мы сделаем просто что-то типо такого:



то работать ничего не будет. Session будет открываться, когда начинается транзакция и закрываться. Как только произойдёт вызов метода PaymentRepository в рамках другого потока, то ничего работать не будет.

Создадим свой динамический прокси (т.к. работаем через интерфейсы. Если бы был какой-то базовый класс, то можно было бы использовать прокси через наследование). Так это будет работать и при многопоточности.



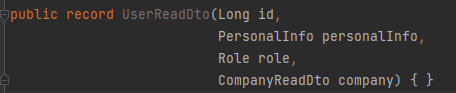
И теперь нам потребуется создать только один экземпляр PaymentRepository (как это и должно быть).

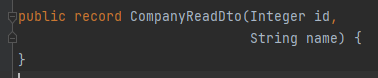
**Service**

Как работать на уровне сервисов с Dao и Repository.

Сервис может иметь зависимости на те Repository, которые ему нужны.

Метод find возвращает какой-то DTO. Сервис работает с другими слоями приложения, либо с другими сервисами и мы захотим работать с DTO, чтобы не словить LazyInitializationException или другие. Так что создали такие DTO:

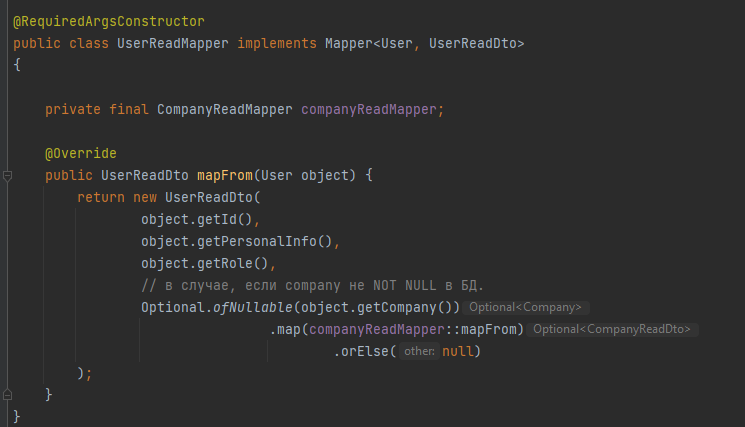


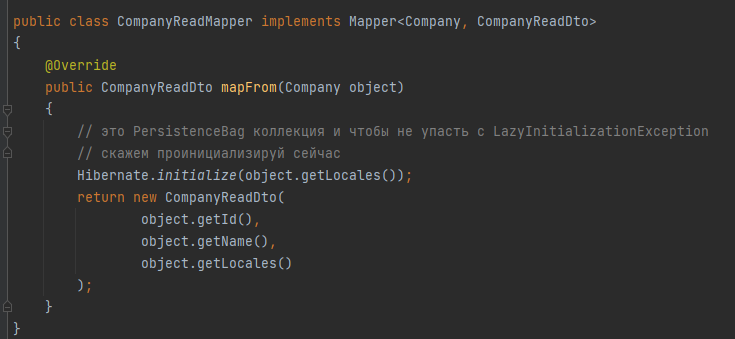


Теперь нам нужен меппер, который конвертирует User в UserReadDto. Создали интерфейс Mapper с методом [T mapFrom(F object);]

Создали такой меппер. Мы это делаем вручную, но есть различные фреймворки для автоматизации этого процесса. Лучший – **MapStruct**

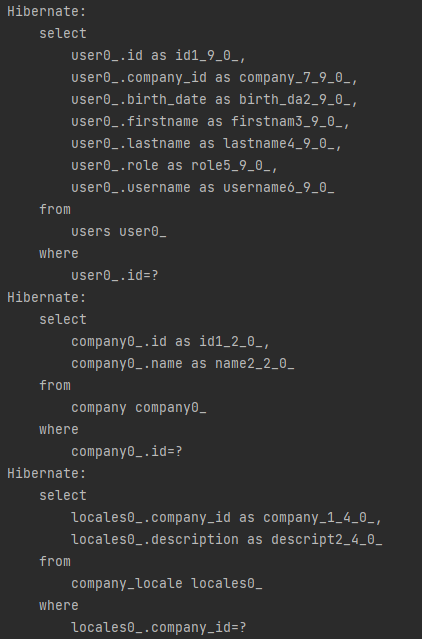
Вот такие написали:





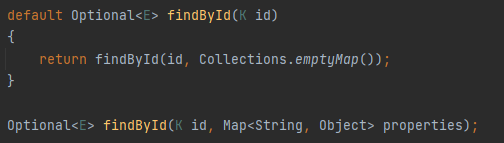
Они тоже инстацируются только раз. Соответственно в UserService зависимость от UserReadMapper.

С такой реализацией у нас опять будет несколько запросов для получения нашего DTO, вместо всего одного запроса. -> опять N + 1



В таком случае можно использовать entity граф. Тогда в findById мы должны передавать наши properties (с помощью них мы можем добавить наш entity граф, с помощью которого мы сразу запросим все поля через join), а сейчас этого нет в нашем интерфейсе.

Сделаем такие методы для этого:



Все EntityGraph, мы можем вынести в константы и добавить их в UserRepository, чем харкодить их в сервисах вот так:

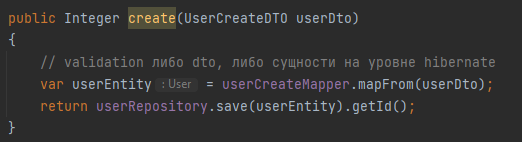


Не всегда стоит гнаться за тем, чтобы доставить всё одним запросом. Некоторые сущности или поля лучше доставать отдельными запросами т.к. данные будут дублироваться в ResultSet и стоит об этом думать, а лучше тестировать.

Кроме того, если мы хотим считать наш User в какой-то другой DTO, мы можем сделать ещё и метод findById, принимающий mapper.

Перейдём к методу create. Он принимает UserCreateDto. Первое что мы делаем – валидация пришедших данных. Теперь нам опять нужно делать map dto в сущность. И потом просто делаем save и возвращаем id.

Но при создании User из UserCreateDto у нас есть companyId, но нет Company. Для того, чтобы получить Company в UserCreateMapper мы можем сделать зависимость от CompanyRepository.

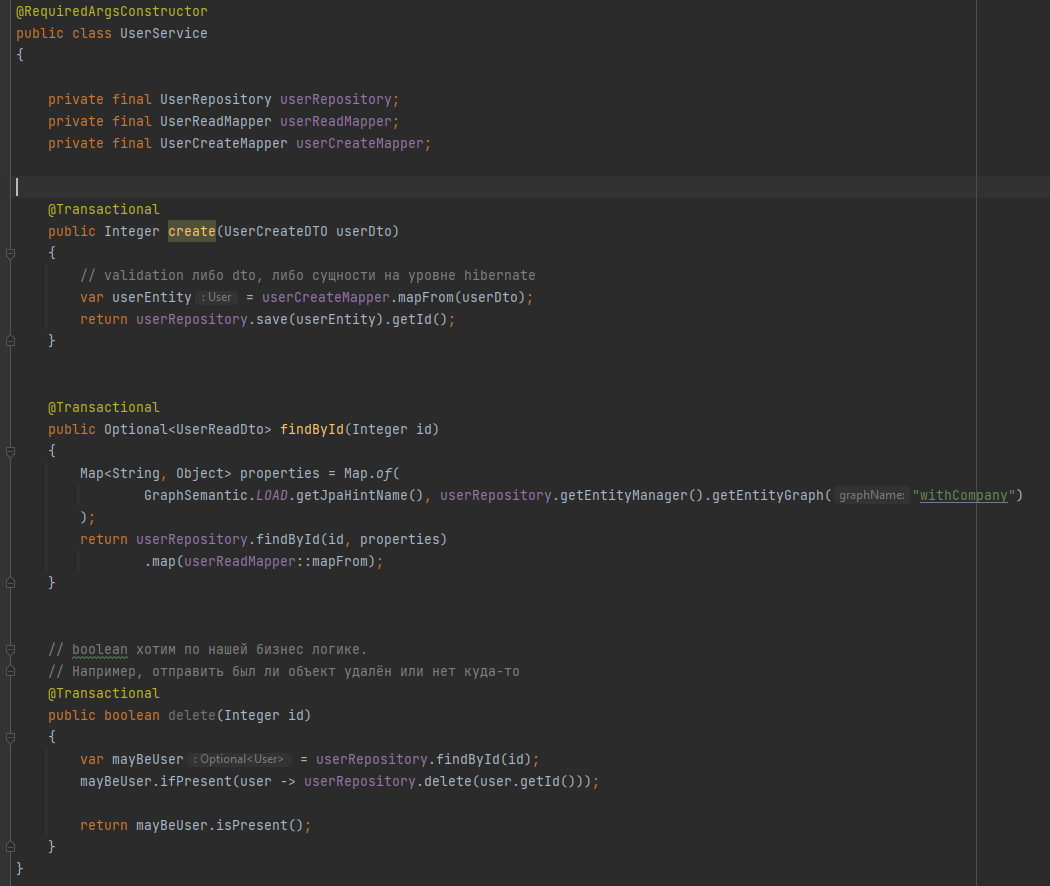


Получилось это.

Идём дальше. У нас до сих пор осталось ручное управление транзакциями. А такого ручного управления не должно быть. На уровне сервисов должно происходить открытие / закрытие или роллбэк транзакции. Это мы можем сделать с помощью @Transactional.

Т.е. те методы, в которых мы хотим открывать транзакцию, мы помечаем как @Transactional

Соответственно выглядит это так:

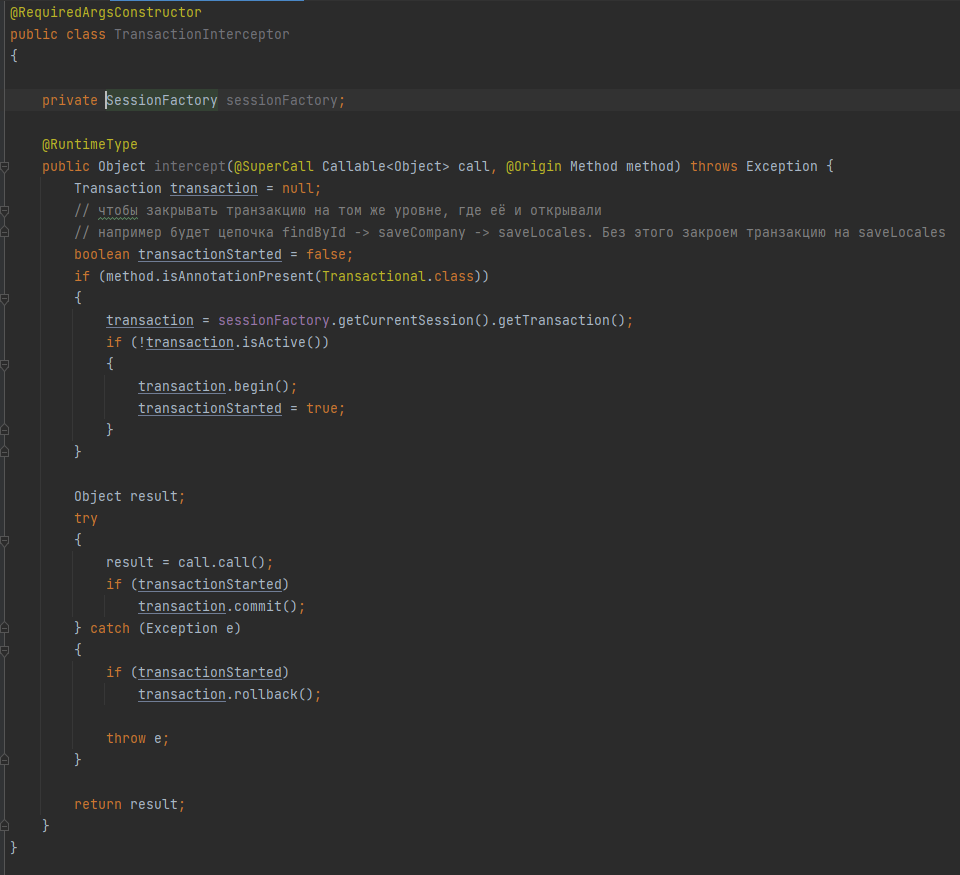


Но тут нам каждый метод придётся помечать @Transactional. Можно пометить класс аннотацией @Transactional и тогда все методы класса будут с ней, но это уже в спринге. В Hibernate так нельзя.

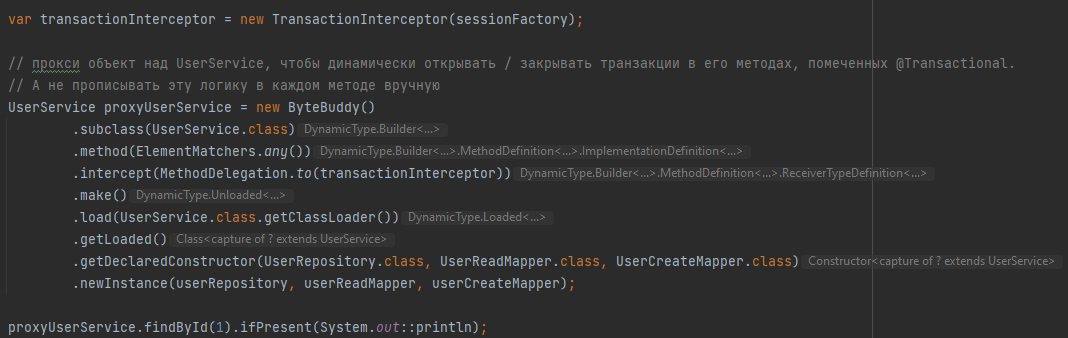
Создадим proxy на наш UserService для автоматического управления транзакциями.

В данном случае через наследование т.к. нет интерфейса UserService (создавать по 1 интерфейсу на 1 реализацию неудобно. Это больше захламляет.)

Для этого создадим TransactionInterceptor. Этот класс будет открывать и закрывать транзакции.



Теперь создадим сам proxy на наш UserService, исопользуя API ByteBuddy (её использует Hibernate для создания прокси) для создания прокси через наследование, а не делая это вручную.

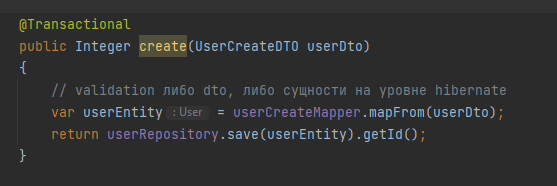


Теперь нам не нужно коммитить транзакции. Всё будет происходить автоматически.

Мы реализовали свой функционал для того, чтобы обрабатывать транзакции динамически, во время вызова userService. Для этого нам пришлось создать Proxy.

**Валидация (Validation)**

У нас осталась недоделанной валидация при create сущности в UserService.



Вместо того, чтобы писать свой валидатор мы можем взять готовый механизм валидации. JSR 303. (есть новее стандарт в Jakarta 2.0, но её Hibernate ещё не поддерживал на момент выхода курса)

Суть JSR 303 в том, что мы используем готовые аннотации и интерфейсы для того, чтобы валидировать наши объекты. Подключим её (идёт отдельной зависимостью)

implementation 'org.hibernate:hibernate-validator:6.0.22.Final' Это последняя версия 1.0 спецификации. Дальше уже 2.0.

Для работы этой зависимости потребуются ещё 2:

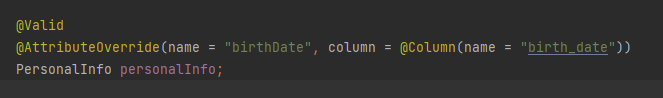
implementation 'javax.el:javax.el-api:3.0.0'

implementation 'org.glassfish:javax.el:3.0.0'

Дальше.

Например, пометим birthDate в PersonalInfo у User @NotNull из javax.validation.constraints. И таких аннотаций куча.

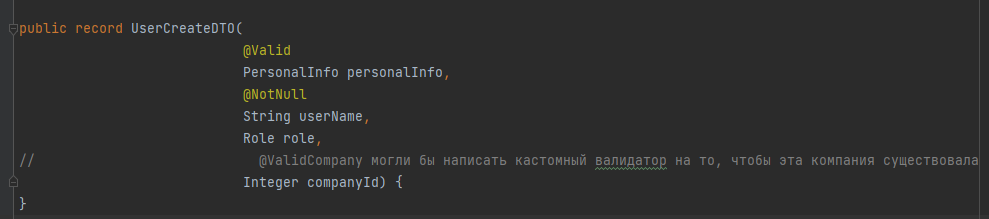
Но т.к. PersonalInfo это уже вложенный объект, нам нужна аннотация @Valid.



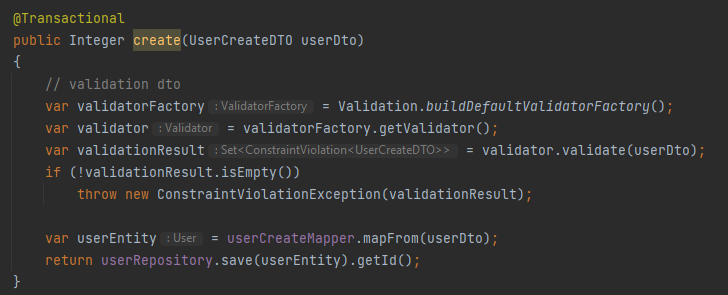
Чтобы мы зашли внутри этого объекта и уже там проверили.

Расставили аннотации над User. Больше ничего не нужно. Просто запускаем. Hibernate увидит аннотации и проверит всё. В случае, если условие будет нарушено выкинется InvocationTargetException. Эта валидация получается не на уровне сервисов, а ниже.

Однако, валидацию лучше всего использовать на уровне сервисов, а не на уровне Hibernate, когда он делает запрос в БД (поэтому помечаем аннотациями не сущность User, а UserCreateDto)



Теперь нам в create нужен валидатор. Hibernate делает это из коробки, но т.к. мы производим валидацию на уровень выше, то пропишем это вручную. На самом деле объект validator thread safe достаточно всего 1 объекта на приложение. Но это опять dependency injection, чего пока нет. Так что пусть пока так.



Мы валидируем DTO и если что ловим этот exception и говорим пользователю что пошло не так и какие поля нужно поправить т.е. не доходим до уровня Hibernate, что мы и хотели.

В метод validate можно передавать группу. Это просто маркер. По умолчанию валидируются аннотации без группы. А аннотации с какими-то группами участвуют в валидации только если передать эту группу в validate. Группа – это интерфейс-маркер.

Например, пометили роль такой группой



(создали интерфейс Update check. Это интрефейс маркер) И эта аннотация будет учитываться только если мы передадим в validate этот маркер.

Совет: для валидации использовать эту спецификацию. Эту же спецификацию использует и спринг.