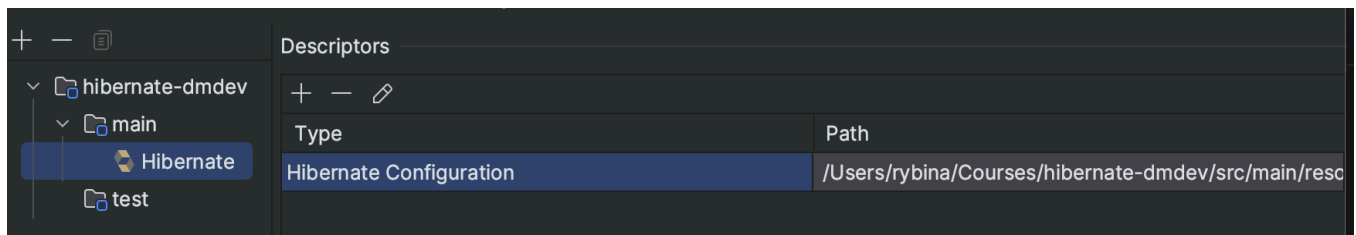


#Hibernate

0. Первое подключение

1. Для первого подключения нам необходимо подключить модуль **hibernate** в main (конфигурация структуры проекта) а также добавить дескриптор в виде xml конфигурации, который автоматически предложится нам при добавлении



2. Конфигурируем наш hibernate.csf.xml

```
<session-factory>
  <property name="connection.url">
    jdbc:postgresql://localhost:5432/hibernate
  </property>
  <property name="connection.username">
    postgres
  </property>
  <property name="connection.password">
    aqwsde322
  </property>
  <property name="connection.driver_class">
    org.postgresql.Driver
  </property>
  <property name="hibernate.dialect">
    org.hibernate.dialect.PostgreSQL10Dialect
  </property>
</session-factory>
```

POJO (*play old java object*) - класс, у которого все поля приватны и на них есть сеттеры и сеттеры

```
public class HibernateRunner {

    public static void main(String[] args) throws SQLException {
        Configuration configuration = new Configuration();
//        hibernate.cfg.xml by default
//        !!!
        configuration.addAnnotatedClass(User.class);
        configuration.configure();

//        sessionFactory == connectionPool
//        session == connection
        try (SessionFactory sessionFactory =
configuration.buildSessionFactory();
            Session session = sessionFactory.openSession();
        ) {
            session.beginTransaction();

            User user = User.builder()
                .lastName("lastN2")
                .age(20)
                .firstname("firstn")
                .username("test3")
                .birthday(LocalDate.of(2004, 2, 28))
                .build();
            session.save(user);
            System.out.println("OK");

            session.getTransaction().commit();
        }
    }
}
```

1. Преобразование типов в hibernate

Преобразование типов происходит с помощью имплементирования интерфейса **Type**, которое требует реализации многих методов. Такие классы имеют конструктор, где есть 2 класса-

дескриптора (1 - sql type, 2 - java type) пример:

```
public class LocalDateType
    extends AbstractSingleColumnStandardBasicType<LocalDate>
    implements LiteralType<LocalDate> {

    /**
     * Singleton access
     */
    public static final LocalDateType INSTANCE = new LocalDateType();

    public static final DateTimeFormatter FORMATTER =
        DateTimeFormatter.ofPattern( "yyyy-MM-dd", Locale.ENGLISH );

    public LocalDateType() {
        super( DateTypeDescriptor.INSTANCE, LocalDateJavaDescriptor.INSTANCE );
    }

    -----

    public abstract class AbstractSingleColumnStandardBasicType<T>
        extends AbstractStandardBasicType<T>
        implements SingleColumnType<T> {

        public AbstractSingleColumnStandardBasicType(SqlTypeDescriptor
            sqlTypeDescriptor, JavaTypeDescriptor<T> javaTypeDescriptor) {
            super( sqlTypeDescriptor, javaTypeDescriptor );
        }
    }
}
```

Sql дескриптор выполняет преобразование при конвертировании sql запроса и ответа, а Джава дескриптор выполняет механическое преобразование и используется внутри sql дескриптора

Также мы можем создать кастомный конвертор под какой-то наш тип (см. гитхаб с зависимостями с грейдл)

```
@Converter(autoApply = true)
public class BirthdayConvertor implements AttributeConverter<Birthday, Date> {

    @Override
```

```

public Date convertToDatabaseColumn(Birthday attribute) {
    return Optional.ofNullable(attribute)
        .map(Birthday::birthday)
        .map(Date::valueOf)
        .orElse(null);
}

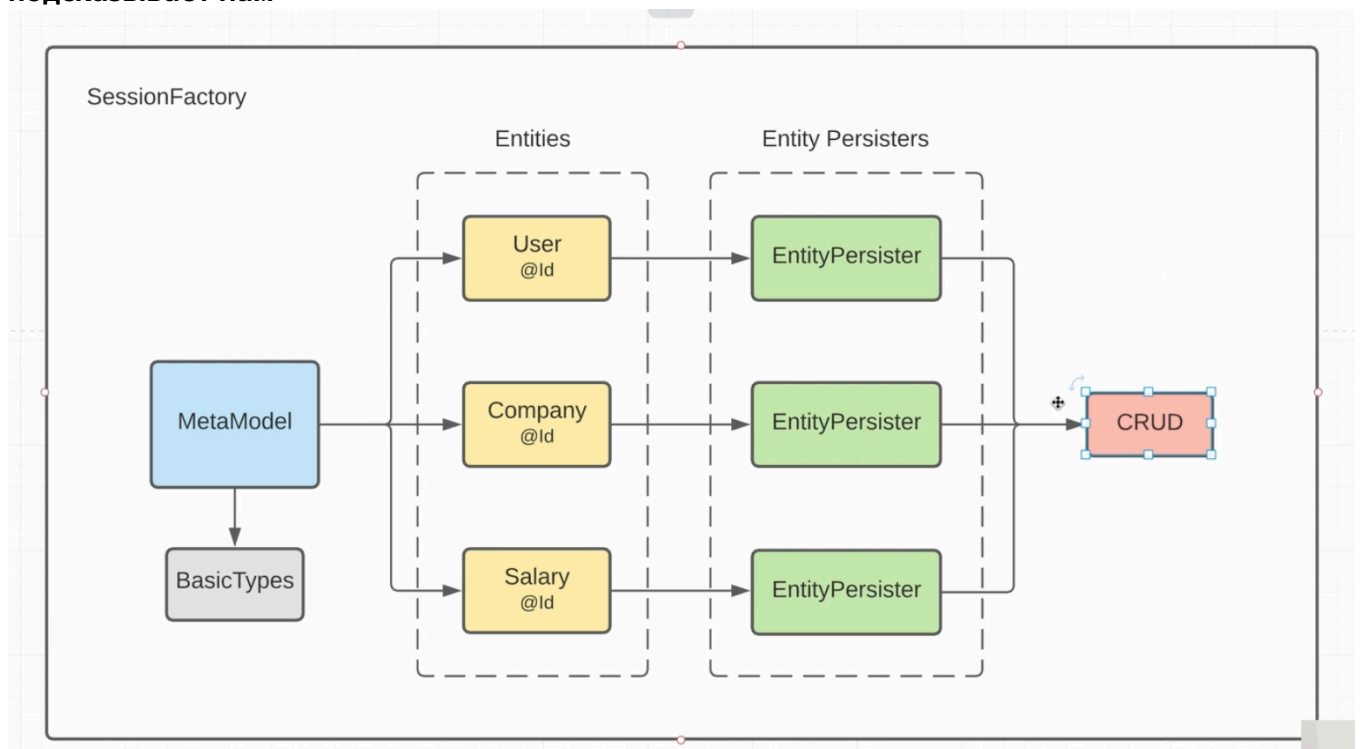
@Override
public Birthday convertToEntityAttribute(Date dbData) {
    return Optional.ofNullable(dbData)
        .map(Date::toLocalDate)
        .map(Birthday::new)
        .orElse(null);
}
}

```

Аннотация `@Converter(autoApply = true)` заменяет строку кода
`configuration.addAttributeConverter(new BirthdayConvertor(), true);`

3. Работа с Entity

Энтити лучше инициализировать в hibernate xml конфете, иначе оно автоматически не подсказывает нам



Session factory имеет метамодель, в которой хранятся все наши таблицы и типы данных, а также класс для каждого типа данных, который маппит его. Таблицы, в свою очередь имеют энити перистор, **каждая**,
ентити персистор - класс, который производит манипуляции с таблицей (crud)

4. Persistence context

Каждая сессия имеет свой **persistence context — First-level cache**, он хранит в себе мапу {id, entity}, то есть кеширует все наши get

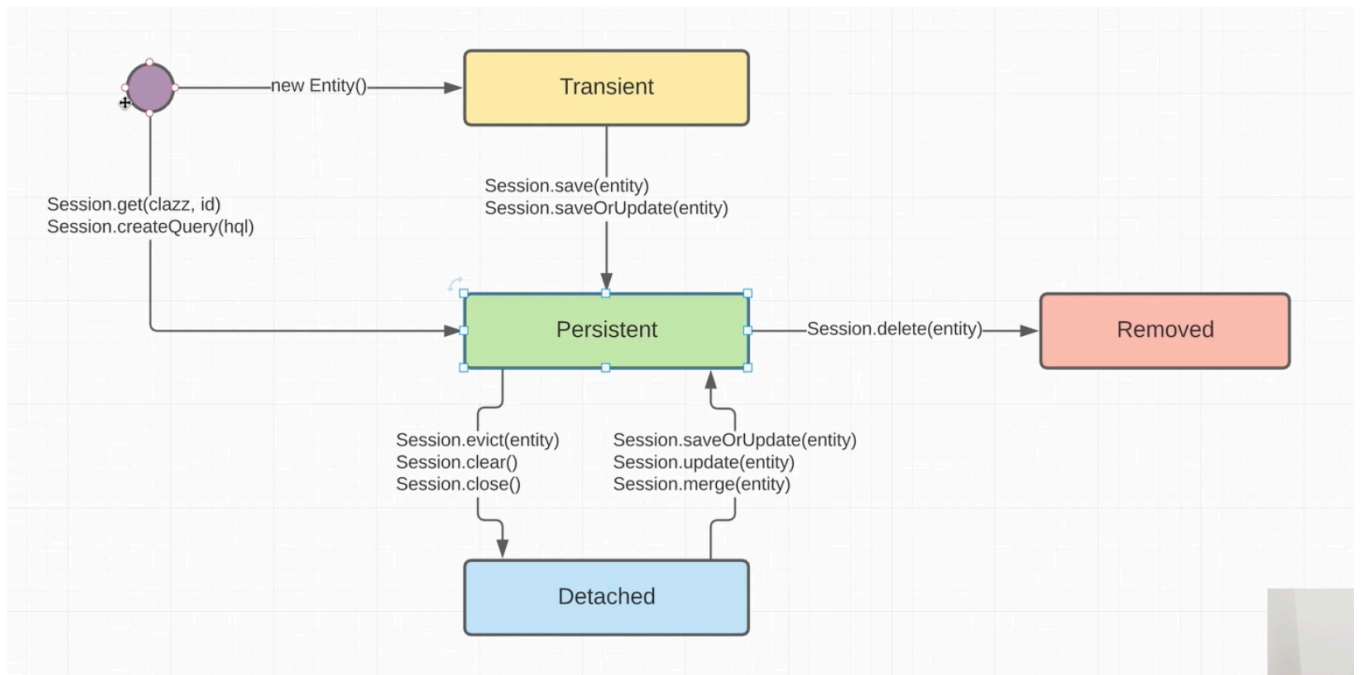
First-level cache в Hibernate, также известный как сессионный кеш, это встроенный кеш, который Hibernate использует для хранения объектов сущности на уровне сессии. Он ассоциирован с конкретной сессией Hibernate и по умолчанию всегда включен. Цель этого кеша - уменьшить количество запросов к базе данных за счет хранения объектов, уже загруженных в текущей сессии.

Все сущности, которые лежат в persistence контексте при изменении конечно же меняются и в бд

Как удалить сущность из кеша

```
session.evict(entity)
session.close()
session.clear()
```

5. Entity lifecycle



6. Refresh, Merge

refresh - подтягивает данные из базы данных и накладывает их на нашу сущность

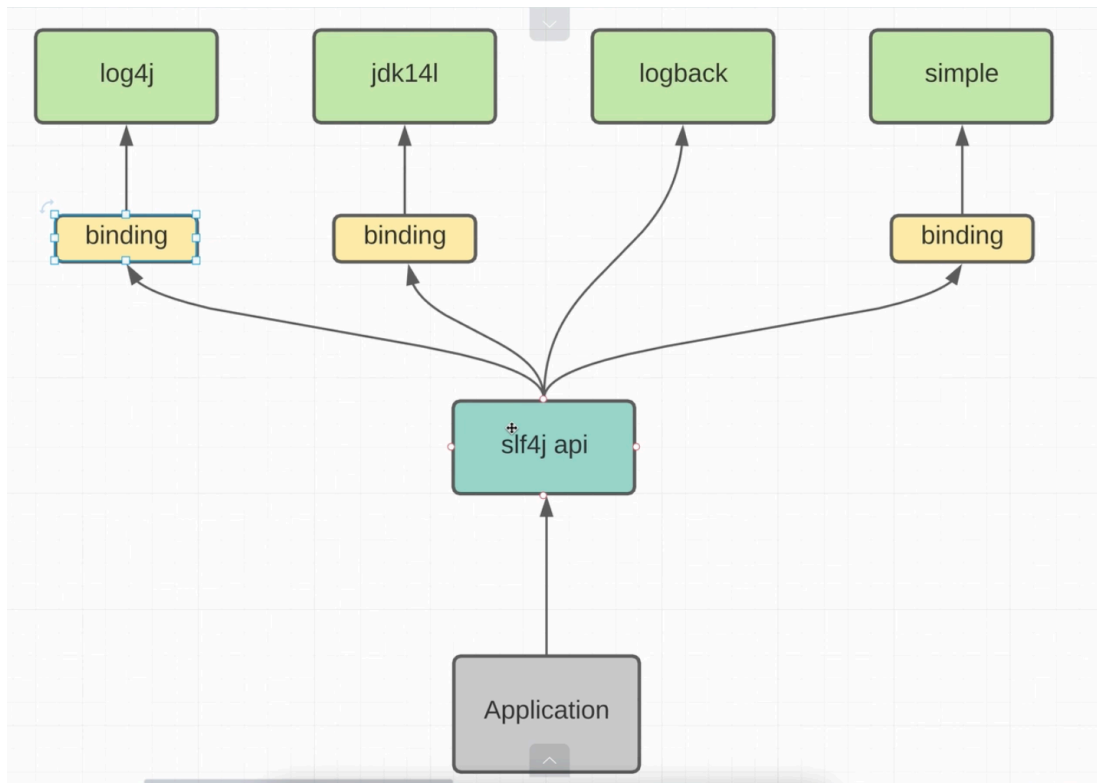
merge - наоборот изменяет данные в базе данных на наши локальные и возвращает **новую сущность** нам

7. Java Persistence JPA

Jpa - набор интерфейсов и аннотаций, которые дают возможность маппить сущности из базы данных в джава классы

Hibernate - конкретная реализация JPA

8. Логирование



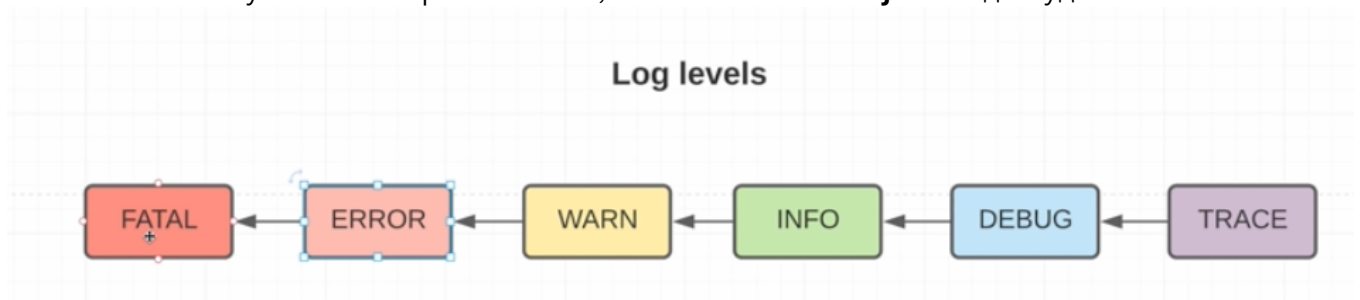
По умолчанию если мы имеем зависимость

```
implementation 'org.slf4j:slf4j-api:2.0.11'
```

Мы можем работать только с logBack. Для того, чтобы работать с иными реализациями **slf4j**, нам надо подключить соответствующий *binding*

Log4j (best practice): `implementation 'org.slf4j:slf4j-log4j12:2.0.7'`

Т.к зависимость тут является транзитивной, то зависимость **slf4j** мы можем удалить



Базовая конфигурация:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>

  <appender name="console" class="org.apache.log4j.ConsoleAppender">
```

```

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
        </layout>
    </appender>

    <root>
        <level value="info" />
        <appender-ref ref="console" />
    </root>

</log4j:configuration>

```

9. Cascade(опасная штука)

Каскады как оказалось можно добавлять как на дочернюю таблицу, так и на родительскую. Но! **cascade = cascadeType.PERSIST** будет срабатывать только на родительских сущностях, т.к есть требование *parent-first*, сначала должна быть по логике вещей создана родительская сущность. Однако данное поведение можно обойти используя **cascade = cascadeType.ALL**, так делать нежелательно. Обычно лучшей практикой считается вешать каскадирование на родительскую сущность, а остальное каскадное поведение настраивать непосредственно в sql

Hash коллекции

Если у нас каскадирование настроено на какую-то хеш коллекцию дочерних сущностей (внутри родительской как и должно быть), то мы вполне можем столкнуться с проблемой, что у нас не получится нормально работать с этой коллекцией, т.к наше хеширование зачастую идет по айдишникам, а при создании айдишник естественно не указан и добавление в коллекцию нормально не срабатывает. **Решение:** ставить хеширование по какому-то другому уникальному not null полю

10. OrphanRemoval в one-to-many

— Что делать с таблицей родителем если мы из него удаляем какой-то дочерний элемент?

Два варианта:

- OrphanRemoval = true, удаляем дочерний элемент из базы данных при комете транзакции
- OrphanRemoval = false - ни-че-го (дефолт)

11. @ElementCollection

ElementCollection - @Embeddable прикрепленный класс, экземпляры которого мы можем иметь не одну к родительскому классу,

Аналог @Embedded только в множественном числе :)

```
@Embeddable
public class LocaleInfo {

    private String lang;
    protected String description;
}
```

```
@ElementCollection
@CollectionTable(name = "company_locale", joinColumns = @JoinColumn(name =
"company_id"))
// @AttributeOverride(name = "название внутри прикрепленного класса", column
= @Column(name = "название колонки в бд"))
private List<LocaleInfo> localeInfos = new ArrayList<>();
```

12. Как работают прокси

Каждый прокси в гибокнейте имплементирует интерфейсы `HibernateProxy`, `ProxyConfiguration`, а также хранит `byteBuddyInterceptor` - `lazyInitializer` (наследуется от `BasicLazyInitializer`). В `byteBuddyInterceptor` есть метод **intercept**, который принимает в себя **прокси объект, метод (через рефлекссию) и аргументы**, в этом методе решается вызывать ли настоящий объект, айдишник которого лежит внутри прокси, или заглушку

13. Работа с мапой

```
@Builder.Default // написано для того, чтобы при нашем билдере ставилось
дефолтное значение
@OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval =
false)
@MapKey(name = "username")
```

```
@SortNatural
```

```
private Map<String, User> users = new TreeMap<>();
```

- Если мы хотим сортировать по ключу, то мы обязаны поставить интерфейс, реализующий **sortedMap**, например **Treemap** и аннотацию `@MapKey(name = "username")`, если мы используем

@MapKeyColumn (y @ElementCollection)

В JPA аннотация `@ElementCollection` используется для определения коллекции простых или встраиваемых типов. Когда вы используете `@ElementCollection` с `Map`, JPA предполагает, что ключ (`@MapKeyColumn`) и значение карты хранятся в колонках таблицы, связанной с коллекцией.

```
@Builder.Default
@ElementCollection
@CollectionTable(name = "company_locale", joinColumns = @JoinColumn(name =
"company_id"))
@MapKeyColumn(name = "lang")
@Column(автоматически оставшаяся колонка, которая и не внеш ключ и не key)
private Map<String, String> localeInfos = new HashMap<>();
```

В вашем случае, `@CollectionTable(name = "company_locale", joinColumns = @JoinColumn(name = "company_id"))` указывает на то, что для хранения элементов карты `localeInfos` используется таблица `company_locale`, а `company_id` служит внешним ключом, связывающим каждую запись с конкретной компанией. `@MapKeyColumn(name = "lang")` указывает, что колонка `lang` в этой таблице будет использоваться для хранения ключей карты.

Для значений карты JPA будет использовать другую колонку этой таблицы, не указанную явно. По умолчанию JPA создает колонку для значений, используя некоторое стандартное именование (например, `value`), если вы не укажете имя колонки для значений с помощью аннотации `@Column` внутри `@ElementCollection`.

Пример таблицы `company_locale` для вашей карты `localeInfos` может выглядеть так:

company_id	lang	value
1	en	English
1	fr	Français
2	en	English

Здесь `company_id` связывает записи с конкретной компанией, `lang` используется как ключ в вашей карте `localeInfos`, а `value` - это значение, связанное с каждым ключом.

Если вы хотите явно указать имя колонки для значений, вы можете использовать аннотацию `@Column` внутри `@ElementCollection`, например:

```
@ElementCollection
@CollectionTable(name = "company_locale", joinColumns = @JoinColumn(name =
"company_id"))
@MapKeyColumn(name = "lang")
@Column(name = "locale_value") // Явно указываем имя колонки для значений
private Map<String, String> localeInfos = new HashMap<>();
```

Это заставит JPA использовать колонку `locale_value` для хранения значений карты `localeInfos`.

Если колонок уже больше, чем 3, то имеет смысл вынести в отдельную сущность с аннотацией `@Embeddable` нашу табличку и привязывать ее так, как указано в 11 пункте

14. Наследованные

14.1 Table per class

Используется, если у 2 допустим таблицы с одинаковыми полями, (в бд у нас наследованная нет !)

Важный момент

Так как мы выносим общие данные в одну таблицу, НО айдишники генерируются независимо друг от друга, то сущностей общего типа (родителя) будет 2 с одинаковым айдишником и будет вылетать ошибка

Для того, чтобы избежать данного поведения, стоит изменить тип генерации первичных ключей

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
private Integer id;
```

14.2 Single table

В данной стратегии нет проблем с айдишником, все хранится в одной единой таблице и в джава коде мы определяем с каким наследником работаем только с помощью

`@DiscriminatorColumn(name = "type")`, в каждом наследнике соответственно нужно проставить `@DiscriminatorValue("m")`

Минусы:

- все в одной таблице (денормализированная)
- нельзя навесить ограничения по типу not null

14.3 Joined

Традиционный вид наследования

Id у наследников является и первичным ключом и внешним

```
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class User {

    -----

    @PrimaryKeyJoinColumn(name = "id")
    public class Manager extends User {
```

15. N + 1

Основные правила:

1. Avoid **@OneToOne bidirectional**
2. Use fetch type **LAZY** everywhere
3. Don't prefer **@BatchSize**, **@Fetch**
4. Use query **fetch** (HQL, Criteria API, Querydsl)
5. Prefer **EntityGraph API** than **@FetchProfile**

Но стоит помнить, что оптимизация без повода ухудшает performance приложения

15.1 @BatchSize(кол-во)

BatchSize в ArrayList работает так, что то количество, которое мы указали - это количество тех прокси, что должны распаковаться И когда мы каким-то образом захотим распаковать прокси у которого стоит эта аннотация, то распакуется сразу еще то количество, что мы указали, это будут прокси, которые лежат уже в перзистенс контекста (например если мы распакуем детей у 1 сущности, то они распакуются и у n количество ее соседей)

```
select
    payments0_.receiver_id as receiver3_4_1_,
    payments0_.id as id1_4_1_,
    payments0_.id as id1_4_0_,
    payments0_.amount as amount2_4_0_,
    payments0_.receiver_id as receiver3_4_0_
from
    Payment payments0_
where
    payments0_.receiver_id in (
        ?, ?
    )
```

При связи **@OneToMany**, аннотацию надо ставить надо маппингом, а при аннотации **@ManyToOne** над сущностью! Иначе работать не будет

15.2 @Fetch

Подходит только для коллекций и надо использовать SUBSELECT

```
@Builder.Default
@Fetch(FetchMode.SUBSELECT)
@OneToMany(mappedBy = "receiver", fetch = FetchType.LAZY)
private List<Payment> payments = new ArrayList<>();
```

```
users = session.createQuery("from User where id > 3", User.class).list();

users.forEach(u -> u.getPayments().forEach(payment -> payment.getId()));
```

Распаковывает так

```
select
    payments0_.receiver_id as receiver3_4_1_,
    payments0_.id as id1_4_1_,
    payments0_.id as id1_4_0_,
    payments0_.amount as amount2_4_0_,
    payments0_.receiver_id as receiver3_4_0_
from
    Payment payments0_
where
    payments0_.receiver_id in (
        select
            user0_.id
        from
            public.users user0_
        where
            user0_.id>3
    )
```

15.3 @FetchProfile

Работает только в случае, когда мы получаем сущность по айдишнику, т.е одну, что конечно же портит всю его гибкость

```
@FetchProfile(name = "withCompanyAndPayment", fetchOverrides = {
    @FetchProfile.FetchOverride(
        entity = User.class, association = "company", mode =
FetchMode.JOIN
    ),
```

```

        @FetchProfile.FetchOverride(
            entity = User.class, association = "payments", mode =
FetchMode.JOIN
        )
    })
    public class User {

```

Вывод sql

Hibernate:

```

select
    user0_.id as id1_0_0_,
    user0_.company_id as company_7_0_0_,
    user0_.info as info2_0_0_,
    user0_.birthday as birthday3_0_0_,
    user0_.firstname as firstnam4_0_0_,
    user0_.lastName as lastname5_0_0_,
    user0_.username as username6_0_0_,
    company1_.id as id1_2_1_,
    company1_.name as name2_2_1_,
    payments2_.receiver_id as receiver3_4_2_,
    payments2_.id as id1_4_2_,
    payments2_.id as id1_4_3_,
    payments2_.amount as amount2_4_3_,
    payments2_.receiver_id as receiver3_4_3_,
    profile3_.id as id1_5_4_,
    profile3_.language as language2_5_4_,
    profile3_.street as street3_5_4_,
    profile3_.user_id as user_id4_5_4_
from
    public.users user0_
left outer join
    Company company1_
        on user0_.company_id=company1_.id
left outer join
    Payment payments2_
        on user0_.id=payments2_.receiver_id
left outer join
    Profile profile3_
        on user0_.id=profile3_.user_id
where
    user0_.id=?

```

Вывод sql

id1_0_0_	company_7_0_0_	info2_0_0_	birthday3_0_0_	firstnam4_0_0_	lastnam
1	1	null	1955-10-28	Bill	Gates
1	1	null	1955-10-28	Bill	Gates
1	1	null	1955-10-28	Bill	Gates
2	2	null	1955-02-24	Steve	Jobs
2	2	null	1955-02-24	Steve	Jobs
2	2	null	1955-02-24	Steve	Jobs
3	3	null	1973-08-21	Sergey	Brin
3	3	null	1973-08-21	Sergey	Brin
3	3	null	1973-08-21	Sergey	Brin

15.4 @NamedEntityGraphs

15.4.1 С помощью аннотаций

attributeNodes - те поля, которые будут подгружаться сразу

```
@NamedEntityGraph(  
    name = "WithCompanyAndChat",  
    attributeNodes = {  
        @NamedAttributeNode("company"),  
        @NamedAttributeNode(value = "userChats", subgraph =  
"chatskillallmen")  
    },  
    subgraphs = {  
        @NamedSubgraph(name ="chatskillallmen", attributeNodes =  
@NamedAttributeNode("chat"))  
    }  
)  
public class User {
```



```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;

@Embedded
private PersonalInfo personalInfo;

@Column
private String username;

@Column(columnDefinition = "jsonb")
@Type(type = "json")
private String info;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "company_id")
// @NotAudited
private Company company;

@OneToOne(mappedBy = "user", cascade = CascadeType.ALL)
// @NotAudited
private Profile profile;

@Builder.Default
// @BatchSize(size = 5)
@NotAudited
@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
private List<UserChat> userChats = new ArrayList<>();

@Builder.Default
@NotAudited
// @Fetch(FetchMode.SUBSELECT)
@OneToMany(mappedBy = "receiver", fetch = FetchType.LAZY)
private List<Payment> payments = new ArrayList<>();

public String fullName() {
    return getPersonalInfo().getFirstname() + " " +
getPersonalInfo().getLastName();
}
}

```

Использование:

```
RootGraph<?> graph = session.getEntityGraph("WithCompanyAndChat");

Map<String, Object> properties = Map.of(QueryHints.HINT_FETCHGRAPH, graph)
```

15.4.1 Ручная настройка

```
RootGraph<User> userGraph = session.createEntityGraph(User.class);
userGraph.addAttributeNodes("company", "userChats");
SubGraph<UserChat> userChatSubGraph = userGraph.addSubgraph("userChats",
UserChat.class);
userChatSubGraph.addAttributeNodes("chat");
```

Использование:

```
Map<String, Object> properties = Map.of(QueryHints.HINT_LOADGRAPH, userGraph);

var user = session.find(User.class, 1, properties);
```

Вывод sql

Все равно нельзя вписать к слову в attributeNodes больше одной коллекции

Hibernate:

```
select
    user0_.id as id1_0_0_,
    user0_.company_id as company_7_0_0_,
    user0_.info as info2_0_0_,
    user0_.birthday as birthday3_0_0_,
    user0_.firstname as firstnam4_0_0_,
    user0_.lastName as lastname5_0_0_,
    user0_.username as username6_0_0_,
    company1_.id as id1_2_1_,
    company1_.name as name2_2_1_,
    userchats2_.user_id as user_id5_6_2_,
    userchats2_.id as id1_6_2_,
    userchats2_.id as id1_6_3_,
    userchats2_.chat_id as chat_id4_6_3_,
    userchats2_.createdAt as createda2_6_3_,
    userchats2_.createdBy as createdb3_6_3_,
```

```

        userchats2_.user_id as user_id5_6_3_,
        chat3_.id as id1_1_4_,
        chat3_.name as name2_1_4_,
        profile4_.id as id1_5_5_,
        profile4_.language as language2_5_5_,
        profile4_.street as street3_5_5_,
        profile4_.user_id as user_id4_5_5_
from
    public.users user0_
left outer join
    Company company1_
        on user0_.company_id=company1_.id
left outer join
    users_chat userchats2_
        on user0_.id=userchats2_.user_id
left outer join
    Chat chat3_
        on userchats2_.chat_id=chat3_.id
left outer join
    Profile profile4_
        on user0_.id=profile4_.user_id
where
    user0_.id=?

```

16. Транзакции

16.0 ACID

Атомарность гарантирует, что никакая транзакция не может быть выполнена частично

Consistency (*согласованность*) - каждая успешная транзакция фиксирует только допустимые значения

Изолированность - транзакции не должны оказывать влияние на результат на друг друга

Durability (*устойчивость*) - независимо от проблем на низших уровнях, все закомиченные, а то есть успешные транзакции должны оставаться сохраненными

Transaction isolation issues

- **Lost Update** - потерянное обновление
- **Dirty Read** - "грязное" чтение
- **Non Repeatable Read** - неповторяющееся чтение
- **Phantom Read** - фантомное чтение

DM
dev

Уровни изолированности транзакций

Уровень изоляции	Фантомное чтение	Неповторяющееся чтение	«Грязное» чтение	Потерянное обновление
SERIALIZABLE	+	+	+	+
REPEATABLE READ	-	+	+	+
READ COMMITTED	-	-	+	+
READ UNCOMMITTED	-	-	-	+

16.1 Lock Optimistic

Всего есть три типа оптимистического лока:

OptimisticLockType.VERSION

OptimisticLockType.VERSION сравнивает сущности по их версии и если мы пытаемся обновить сущность имея неактуальную версию, то выпадает ошибка

```
@OptimisticLocking(type = OptimisticLockType.VERSION)
```

```
public class Payment {
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;

@Version
private Long version;
```

- *OPTIMISTIC* - обновляет версию сущности если мы в ней что-то меняем
- *OPTIMISTIC_FORCE_INCREMENT* обновляет версию сущности в любом случае

```
Payment payment = session.find(Payment.class, 1, LockModeType.OPTIMISTIC --
можно и не писать, оно стоит по умолчанию);
```

так работает это в бд, логично, что если версия уже обновилась, то сущность по прошлой версии мы не найдем

```
update
  Payment
set
  amount=?,
  receiver_id=?,
  version=?
where
  id=?
  and version=?
```

OptimisticLockType.ALL и OptimisticLockType.DIRTY

OptimisticLockType.ALL проверяет версию сущности не по специальному полю, а по всем полям, если сейчас нет в бд сущности с такими полями, значит она была изменена, значит мы не можем сделать такое грязное обновление. OptimisticLockType.DIRTY проверяет только по грязному полю, если другие поля менялись, нам не важно

@OptimisticLocking(type = OptimisticLockType.ALL

```
@OptimisticLocking(type = OptimisticLockType.ALL)
```

```
@DynamicUpdate
public class Payment {
```


```
Hibernate:
    update
        Payment
    set
        amount=?
    where
        id=?
        and amount=?
        and receiver_id=?
```

@OptimisticLocking(type = OptimisticLockType.DIRTY

```
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
public class Payment {
```

```
Hibernate:
    update
        Payment
    set
        amount=?
    where
        id=?
        and amount=?
```

16.2 Lock Pessimistic

 Lock Pessimistic делает блокировку уже на уровне запроса в базу данных

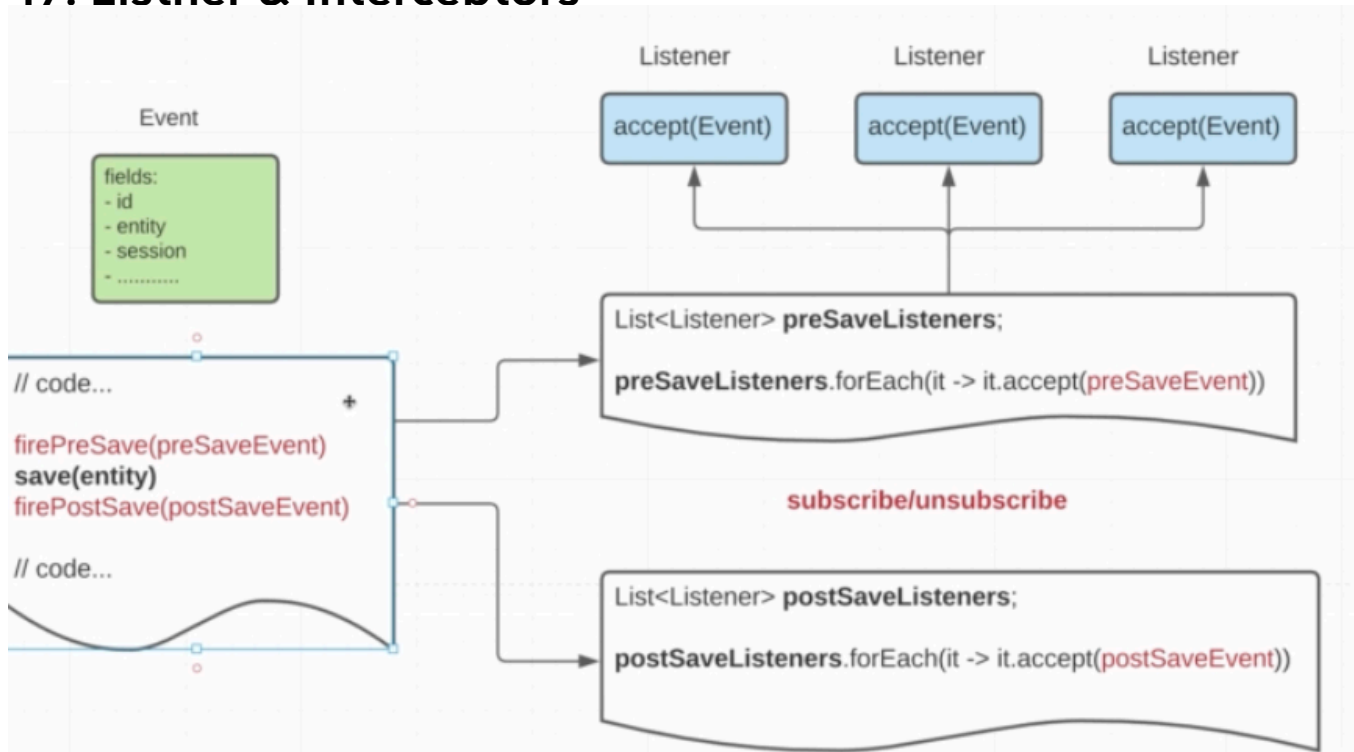
Поставить все сущности в readOnlyMode

```
session.setDefaultReadOnly(true);
```

```
session.createQuery("from Payment ", Payment.class)
    .setLockMode(LockModeType.PESSIMISTIC_READ)
    .setHint("javax.persistence.lock.timeout", 5000)
    .list();
```

```
Payment payment = session.find(Payment.class, 1, LockModeType.PESSIMISTIC_READ);
```

17. Listener & Interceptors



17.1 Введение, коллбеки. Callbacks using entities

Данный способ представляет их себя внедрение в разные этапы жизненного цикла сущностей

```
public abstract class AuditableEntity<T extends Serializable> extends
BaseEntity<T> {

    private Instant createdAt;

    private Instant updatedAt;

    @PrePersist
    public void prePersist() {
        setCreatedAt(Instant.*now*());
    }

    @PreUpdate
    public void preUpdate() {
        setUpdatedAt(Instant.*now*());
    }
}
```

```
}  
}
```

17.2 Listeners callbacks

Этот способ отличается тем, что мы выносим нашу событийную логику в иной класс

```
public class AuditListener {  
  
    @PrePersist  
    public void prePersist(AuditableEntity<?> entity) {  
  
        entity.setCreatedAt(Instant.now());  
  
//        setCreatedAt(Instant.now());  
    }  
  
    @PreUpdate  
    public void preUpdate(AuditableEntity<?> entity) {  
  
        entity.setUpdatedAt(Instant.now());  
  
//        setUpdatedAt(Instant.now());  
  
    }  
}  
  
-----  
  
public class UserChatListener {  
  
    @PostPersist  
    public void postPersist(UserChat userChat) {  
  
        Chat chat = userChat.getChat();  
  
        chat.setCount(chat.getCount() + 1);  
    }  
  
    @PostRemove  
    public void postRemove(UserChat userChat) {  
  
        Chat chat = userChat.getChat();
```



```

        chat.setCount(chat.getCount() - 1);
    }
}

```

И подключаем эти слушатели на присущие классы с помощью аннотации

@EntityListeners(<Нужный слушатель>.class)

```

@EntityListeners(AuditListener.class)
public abstract class AuditableEntity<T extends Serializable> extends
BaseEntity<T> {

```

17.3 Event listeners (best practices)

Суть в том, что мы тоже реализуем классы, в которые уже внедрена гибернейтовская логику, то есть мы наследуемая уже от каких-то Listenerов и переопределяем их методы И по итогу принимает на вход event

```

public class AuditTableListener implements PreDeleteEventListener,
PreInsertEventListener {

    @Serial
    private static final long serialVersionUID = 6031724244654156176L;

    @Override
    public boolean onPreDelete(PreDeleteEvent event) {
        if ((event.getEntity() instanceof Audit)) {
            return false;
        }

        var audit = Audit.builder()
            .entityId(event.getId())
            .entityName(event.getEntityName())
//            .entityContent(event.getEntity().toString())
            .operation(Audit.Operation.DELETE)
            .build();

```

```

        event.getSession().save(audit);

        return false;
    }

    @Override
    public boolean onPreInsert(PreInsertEvent event) {
        if ((event.getEntity() instanceof Audit)) {
            return false;
        }

        var audit = Audit.builder()
            .entityId(event.getId())
            .entityName(event.getEntityName())
            // .entityContent(event.getEntity().toString())
            .operation(Audit.Operation.INSERT)
            .build();

        event.getSession().save(audit);

        return false;
    }
}

```

```

private static void registerListeners(SessionFactory sessionFactory) {
    SessionFactoryImpl sessionFactoryImpl =
        sessionFactory.unwrap(SessionFactoryImpl.class);

    EventListenerRegistry eventListenerRegistry =
        sessionFactoryImpl.getServiceRegistry().getService(EventListenerRegistry.class);

    AuditTableListener auditTableListener = new AuditTableListener();

    eventListenerRegistry.appendListeners(EventType.PRE_INSERT,
        auditTableListener);
    eventListenerRegistry.appendListeners(EventType.PRE_DELETE,
        auditTableListener);
}

```

17.4 Interceptor

 Interceptor - глобальный слушатель с большим функционалом чем просто abstractListener

```
configuration.setInterceptor(new GlobalInterceptor());
```

```
public class GlobalInterceptor extends EmptyInterceptor {

    @Serial
    private static final long serialVersionUID = 8995472881574080246L;

    @Override
    public int[] findDirty(Object entity, Serializable id, Object[]
currentState, Object[] previousState, String[] propertyNames, Type[] types) {
        return super.findDirty(entity, id, currentState, previousState,
propertyNames, types);
    }
}
```

18. Hibernate envers

Это библиотека, которая самая создает аудиты, для использования необходима зависимость

```
implementation 'org.hibernate:hibernate-envers:5.6.15.Final'
```

Пример использования:

Для того, чтобы прослушивать сущность, надо пометить ее @Audited,

Также вложенные маппинги, надо также пометить либо @Audited либо @NotAudited ИНАЧЕ будет ошибка

```
@Audited
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Embedded
    private PersonalInfo personalInfo;

    @Column
```

```

private String username;

@Column(columnDefinition = "jsonb")
@Type(type = "json")
private String info;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "company_id")
@NotAudited
private Company company;

@OneToOne(mappedBy = "user", cascade = CascadeType.ALL)
@NotAudited
private Profile profile;

@Builder.Default
// @BatchSize(size = 5)
@NotAudited
@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
private List<UserChat> userChats = new ArrayList<>();

@Builder.Default
@NotAudited
// @Fetch(FetchMode.SUBSELECT)
@OneToMany(mappedBy = "receiver", fetch = FetchType.LAZY)
private List<Payment> payments = new ArrayList<>();

public String fullName() {
    return getPersonalInfo().getFirstname() + " " +
getPersonalInfo().getLastName();
}
}

```

@Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)

Дает возможность не писать над маппингом ??toOne @NotAudit, но к сожалению, придется все равно писать над коллекциями.

```

@Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)
public class User {

    @Id

```

```

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Embedded
    private PersonalInfo personalInfo;

    @Column
    private String username;

    @Column(columnDefinition = "jsonb")
    @Type(type = "json")
    private String info;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "company_id")
    //    @NotAudited
    private Company company;

    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL)
    //    @NotAudited
    private Profile profile;

    @Builder.Default
    //    @BatchSize(size = 5)
    @NotAudited
    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<UserChat> userChats = new ArrayList<>();

    @Builder.Default
    @NotAudited
    //    @Fetch(FetchMode.SUBSELECT)
    @OneToMany(mappedBy = "receiver", fetch = FetchType.LAZY)
    private List<Payment> payments = new ArrayList<>();

    public String fullName() {
        return getPersonalInfo().getFirstname() + " " +
getPersonalInfo().getLastName();
    }
}

```

Custom revinfo

Intro

В envers всем занимаются реализации интерфейса `EnversListener` с одним методом, на каждый вид события своя реализация,

Они уже наследуются от других классов, но вот пример

```
public class EnversPostInsertEventListenerImpl extends BaseEnversEventListener
implements PostInsertEventListener {
    public EnversPostInsertEventListenerImpl(EnversService enversService) {
        super( enversService );
    }

    @Override
    public void onPostInsert(PostInsertEvent event) {
        final String entityName = event.getPersister().getEntityName();

        if
        ( getEnversService().getEntitiesConfigurations().isVersioned( entityName ) ) {
            checkIfTransactionInProgress( event.getSession() );

            final AuditProcess auditProcess =
            getEnversService().getAuditProcessManager().get( event.getSession() );

            final AuditWorkUnit workUnit = new AddWorkUnit(
                event.getSession(),
                event.getPersister().getEntityName(),
                getEnversService(),
                event.getId(),
                event.getPersister(),
                event.getState()
            );
            auditProcess.addWorkUnit( workUnit );

            if ( workUnit.containsWork() ) {
                generateBidirectionalCollectionChangeWorkUnits(
                    auditProcess,
                    event.getPersister(),
                    entityName,
                    event.getState(),
                    null,
                    event.getSession()
                );
            }
        }
    }
}
```

```

    );
}
}
}

@Override
public boolean requiresPostCommitHandling(EntityPersister persister) {
    return
getEnversService().getEntitiesConfigurations().isVersioned( persister.getEntityName() );
}
}

```

WorkUnit - строка одной нашей ревизии (в revinfo)

AuditProcess - это и есть revinfo в нашей бд

Создаем кастомные revinfo

Айдишник и таймстемп генерируются автоматом

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@RevisionEntity(value = MyOwnRevisionListener.class)
public class Revision {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @RevisionNumber
    private Long id;

    @RevisionTimestamp
    private Long timestamp;

    private String userInfo;
}

```

```
public class MyOwnRevisionListener implements RevisionListener {

    @Override
    public void newRevision(Object revisionEntity) {
        ((Revision) revisionEntity).setUserInfo("yaaa");
    }
}
```

MyOwnRevisionListener реализовывает запись в поля, которые не генерируют автоматически

Зачем? Использование - перемещения по версиям

```
User oldUser = auditReader.find(User.class, 1, 1L);
```

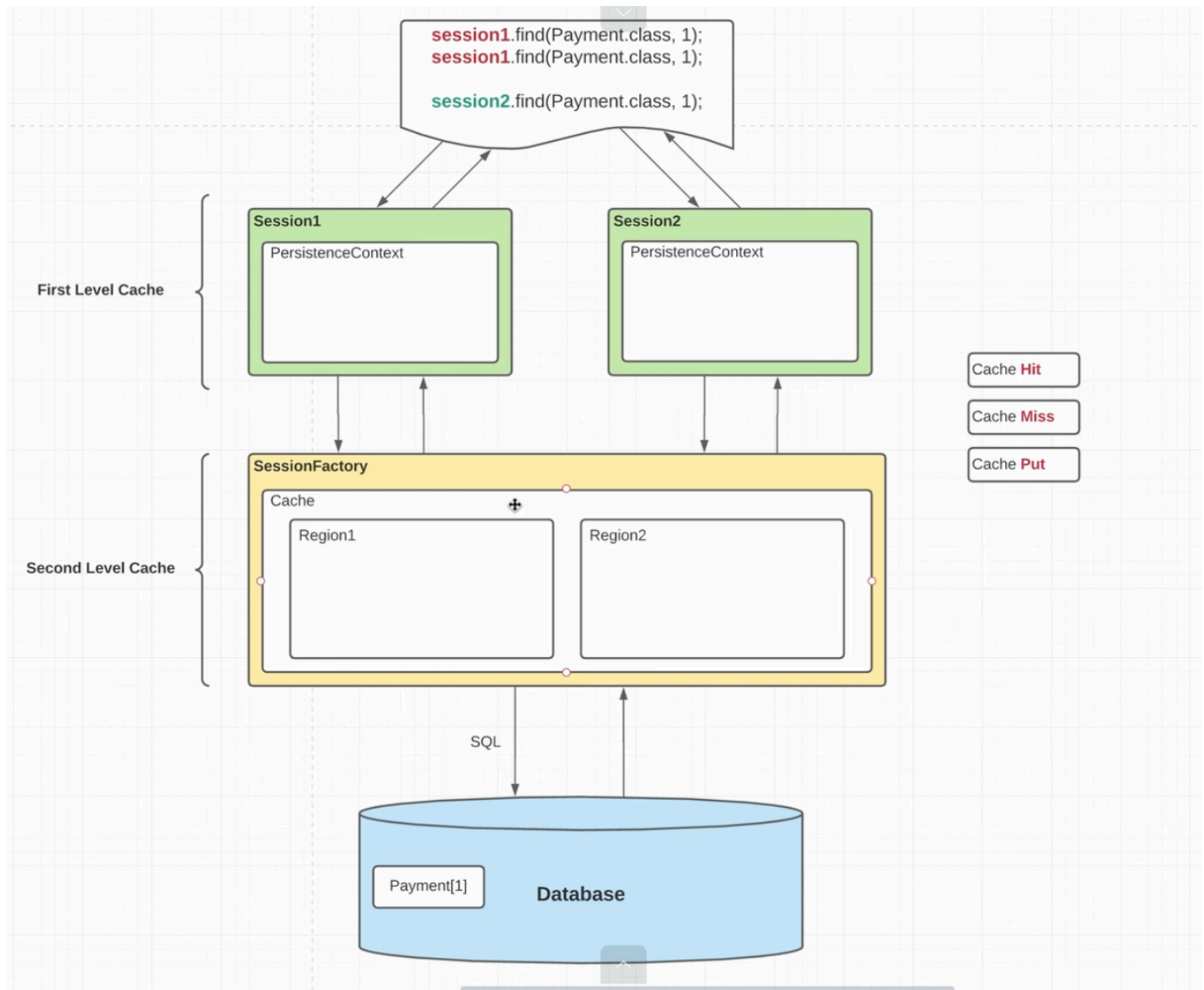
session2.replicate(oldUser, ReplicationMode.OVERWRITE); сущность перепишется и в бд!

```
auditReader.createQuery() AuditQueryCreator
    .forEntitiesAtRevision(Payment.class, revision: 400L) AuditQuery
    .add(AuditEntity.property( propertyName: "amount").ge( value: 450))
    .add(AuditEntity.property( propertyName: "id").ge( value: 6L))
    .addProjection(AuditEntity.property( propertyName: "amount"))
    .addProjection(AuditEntity.id())
    .getResultList();
```

19. Second level cache

Second level cache - обычно выключен. Хранится в sessionFactory, делится на регионы.

Регионы по умолчанию создаются для каждой сущности и именуются по полному пути его класса



- cache miss - сущность в кеше не найдена
- cache put - сущность закеширована
- cache hit - сущность найдена

Сущности хранятся во втором уровне кеша сериализованными, поэтому каждый раз, когда мы вытаскиваем сущность мы, мы будем вытаскивать новый объект класса, т.е десериализованные данные

Как подключать second level cache

1. Подключить необходимую библиотеку

```
implementation 'org.hibernate:hibernate-jcache:5.6.15.Final'
```

```
implementation 'org.ehcache:ehcache:3.9.7'
```

2. Внести правки в конфигурацию

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.jcache.internal.
JCacheRegionFactory</property>
```

3. Помечаем необходимые сущности

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class User {
```

Если мы хотим кешировать коллекции внутри сущностей (маппинги), то нам надо ! Ставить @Cache И над полем в сущности И над самой сущностью.

19.2 SecondQueryCache

Кеширование запросов сохраняет строку (с уже вставленными в нее параметрами вместо “?”). Важно при использовании также кешировать сущность, которую мы получаем, а иначе оно будет лишь сохранять строки и делать повторно запросы в бд (собственно такая же проблема как и маппинг коллекций сущностей)

```
<property name="hibernate.cache.use_query_cache">true</property>
```

```
        List<Payment> payments = session.createQuery("from Payment
where receiver.id = :userId", Payment.class)
                .setParameter("userId", 2)
                .setCacheable(true)
//                .setCacheRegion()
                .getResultList();
```