**Как использовать Hibernate: основные проблемы и их решения**

В этом примере чувак создает какую-то сложную бизнес-логику с кучей всяческих мелочей и нюансов для формирования идентичности полей в таблицах, вместо использования обычного элементарного однозначно идентифицирующего поля uuid-формирования ключей. Нахрена???? Проблем наверное в жизни не хватает. Типа увеличивается скорость работы, так сейчас у всех на столах стоят компы, которые по мощности лет 15 назад считались бы серверами!

Меня зовут Андрей Слободяник, я уже более 10 лет работаю в Java enterprise проектах. В каждом из них были данные в базе, а доступ к ним осуществлялся с помощью JPA/Hibernate. При этом фреймворк использовался, как мне кажется, не совсем правильно: код мог быть компактнее, а производительность выше.

Эта статья — о наболевшем: основных проблемах, способах их исправления, и, главное, подходу, где уместен Hibernate.

*Слова «JPA» как стандарт и «Hibernate» как реализация используются как синонимы.*

Проверочный вопрос

@Column(name = "name", nullable = false, length = 32)

**private** String name;

* Для чего нужен атрибут nullable?
* Можно ли создать и сохранить entity, если в поле name будет значение null?
* Проверяется ли длина поля?
* В чем отличие этих атрибутов @Column и аннотаций @javax.validation.constraints.NotNull и @Size?

Если вы не уверены в ответах, добро пожаловать в статью.

Подход

Hibernate не нужен автоматически везде, где есть БД. Начинать следует не с неё. JPA по самому своему определению применяется, когда оказывается, что объекты Java нужно где-то хранить между выключениями приложения. Один из возможных вариантов — реляционная база данных.

Поэтому создайте **удобные** классы, описывающие вашу доменную модель так, как будто JPA у вас нет. Используйте всю выразительность Java: различные типы, композицию, наследование, коллекции, Maps, Enums. Только потом переведите её на JPA: добавьте ID, отношения и каскады. Проверьте созданные Hibernate таблицы, если необходимо, поправьте их.

Звучит слишком широко и абстрактно, поэтому давайте разберём на конкретном примере.

Техническое задание

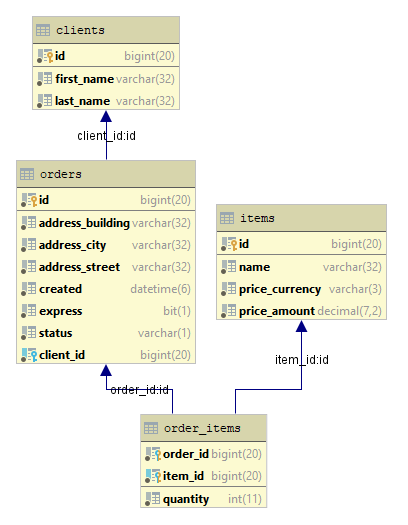
Предположим, мы разрабатываем магазин с заказами.

У заказа есть:

* дата создания;
* статус — новый, в обработке и т. д.;
* заказ может быть «срочный»;
* адрес доставки (каждый раз разный, поэтому смысла в нормализации пока нет);
* в заказ входят товары в каком-то количестве;
* у товара есть неизменная цена с валютой;
* заказ относится к определённому клиенту.

Обычный подход

Дальше почему-то происходит следующее. Вначале создаются таблицы, что-то вроде:



А потом для них — соответствующие entities. В зависимости от опыта разработчиков, в самом тяжелом случае получается такое.

@Entity

@Table(name = "orders")

public **class** **OrderEntity** {

@Id

**private** Long id;

**private** LocalDateTime created;

**private** String status; // see StatusConstants for available values

**private** Integer express;

**private** String addressCity;

**private** String addressStreet;

**private** String addressBuilding;

**private** Long clientId;

public static **class** **StatusConstants** {

public static **final** String NEW = "N";

public static **final** String PROCESSING = "P";

public static **final** String COMPLETED = "C";

public static **final** String DEFERRED = "D";

public static **final** List<String> ALL = Arrays.asList(NEW, PROCESSING, COMPLETED);

// oops, forgot to add "deferred" to the list

}

}

@Entity

@Table(name = "items")

public **class** **Item** {

@Id

**private** Long id;

**private** String name;

**private** BigDecimal priceAmount;

**private** String priceCurrency;

}

Это вполне рабочие entities, сделанные по принципу «поле в БД — такое же поле в классе», но далеко не лучшие. Из всех возможностей Hibernate мы используем только одну — маппинг между таблицами и классами. Фактически это работа в JDBC режиме.

Проблемы:

* адрес и цена «размазаны» по нескольким полям;
* целостность и операции с полем client делаются вручную;
* легко ошибиться со значениями поля status;
* нужно помнить, что флажок express представлен числом: 1 — true, 0 — false;

С некоторым опытом можно создавать более удобный маппинг, но сама идея — подгонять entities под таблицы — в случае JPA не верна.

К слову, подход строить приложение от базы данных тоже имеет место быть, но инструменты для него другие, например:

* база данных;
* сложные запросы, написанные вручную со всеми возможными оптимизациями;
* маппер в Java-классы типа [MyBatis](http://www.mybatis.org/mybatis-3/).

**JPA же предлагает другой принцип: вначале Java-классы, потом их сохранение в БД.**

Поэтому давайте отложим Hibernate в сторону и создадим объектную модель для исходной задачи.

Очевидно, что:

* для статуса заказа удобно использовать Enum;
* флажок «срочный» по самой своей сути — boolean;
* для адреса будет отдельный класс;
* для цены — тоже, причём уже есть готовый — Money из [JavaMoney](https://javamoney.github.io/);
* для товаров и их количества подойдёт Map;
* клиент заказа — это поле типа Client, а не числовой указатель на него.

В результате получается следующее:

public **class** **Order** {

**private** LocalDateTime created;

**private** Status status;

**private** boolean express;

**private** Address address;

**private** Map<Item, Integer> items;

**private** Client client;

}

public **class** **Address** {

**private** String city;

**private** String street;

**private** String building;

}

public **class** **Item** {

**private** String name;

**private** Money price;

}

public **class** **Client** {

**private** String firstName;

**private** String lastName;

}

Коллекции

Поскольку JPA умеет сам инициализировать коллекции, разработчики часто ограничиваются их объявлением, как в примере выше, а дальнейшие модификации осуществляют через геттер.

С учетом базового принципа «Класс должен уметь работать без JPA» лучше:

* инициализировать коллекцию сразу: в месте объявления либо в конструкторе;
* убрать сеттер;
* в геттере возвращать не модифицируемую копию;
* добавить модифицирующие методы.

Для нашего примера может быть так:

**private** Map<Item, Integer> items = **new** HashMap<>();

public Map<Item, Integer> getItems() {

**return** Collections.unmodifiableMap(items);

}

public void addItem(Item item) {

items.merge(item, 1, (v1, v2) -> v1 + v2);

}

public void removeItem(Item item) {

items.computeIfPresent(item, (k, v) -> v > 1 ? v - 1 : null);

}

Также уместно напомнить, что хотя Hibernate требует для своей работы пустой конструктор, для инициализации объектов можно и нужно использовать конструкторы с параметрами.

Подключаем JPA

Для этого требуются:

* Если хотим, чтобы класс хранился в отдельной таблице — аннотации @Entity и @Id, если в таблице другого класса — @Embeddable.
* Примитивы, числа, строки и даты Hibernate умеет сохранять сам.
* Для Enum-ов указываем @Enumerated и тип: хранить либо порядковый номер — EnumType.ORDINAL, либо строковое представление — EnumType.STRING.
* Ещё для Enum-ов и других объектов, которым достаточно одного поля в БД, удобно использовать AttributeConverter.
* Классы из нескольких полей и не соответствующие конвенциям Java Bean — Money в нашем случае — требуют @Type с описанием преобразования аналогично AttributeConverter-у. Для Money нужный класс [уже написан](http://jadira.sourceforge.net/).
* Для классов и коллекций указываются соответствующие отношения (ManyToOne, OneToMany, ElementCollection и т.д.).

После этого Hibernate вполне может создать необходимые таблицы.

Возможно, для кого-то эта информация будет новой, но @Table и различные @Column:

* не обязательны и содержат лишь уточняющие DDL атрибуты;
* не являются валидацией.

Ответ на вступительный вопрос.

@Column(name = "name", nullable = false, length = 32)

**private** String name;

... конвертируется в часть инструкции «create table»

name varchar(32) **not** null

В runtime Java никаких проверок на null и длину поля не происходит.

Поскольку при разработке приложения структура классов меняется, созданная JPA схема — это, скорее, заготовка для flyway/liquibase и/или in-memory БД.

Null

Null и в Java, и в базе данных следует использовать только тогда, когда нам действительно нужно значение «не определено». Во многих случаях такой необходимости нет. Ленясь инициализировать поля, мы либо подкладываем себе грабли в виде NPE, либо осыпаем код ненужными проверками на null.

Срочность заказа (поле express) на первый взгляд имеет три состояния — «да», «нет», «не указано». На практике, нам, скорее всего, будет достаточно двух — срочные и обычные заказы. Поэтому используйте примитивы (boolean) вместо классов (Boolean) там, где это возможно.

Важно помнить, что для поля id примитив использовать нельзя, поскольку для новых (transient) entities оно не определено. И, к сожалению, по техническим причинам (чтобы Hibernate мог создавать прокси) указывать модификатор **final** невозможно.

Именование полей

По умолчанию JPA использует такой naming convention для полей и классов fieldName (в java) -> field\_name (в БД).

Поэтому указывать в @Column(name = «another\_name») имеет смысл, если это не так. В нашем примере «Order» — служебное слово в SQL, поэтому я назвал таблицу «Order**s**», и остальные в множественном числе — для однообразия.

Ключи

Бывают естественные и [суррогатные](https://en.wikipedia.org/wiki/Surrogate_key). Суррогатные ключи обладают ощутимыми преимуществами — удобством и производительностью, поэтому будем использовать именно их.

Для всех entities (кроме Address) добавляем:

@Id

**private** Long id;

(Не)использование Id

При написании бизнес-логики объекты сравнивают между собой. Некоторые пишут так:

boolean **equals** = order.getId().**equals**(anotherOrder.getId());

... а принадлежность объекта к коллекции проверяют с помощью стрима

boolean contains = orders.stream().anyMatch(o -> o.getId().**equals**(someOrder.getId());

Это не компактно и не совсем верно. Лучше:

boolean **equals** = order.**equals**(anotherOrder);

boolean contains = orders.contains(someOrder);

Суррогатные ключи — IDs — не должны фигурировать ни в бизнес-логике, ни в запросах. Всегда работайте с объектами. Вместо параметра clientId

**select** o **from** **Order** o **where** o.client.id = :clientId

... должен быть объект

**select** o **from** **Order** o **where** o.client = :**client**

Позвольте JPA построить запрос самостоятельно.

Если в контексте нет объекта client и целиком он не нужен, достаточно использовать его reference. То есть вместо

Client client = em.find(Client.**class**, clientId);

... используем

Client client = em.getReference(Client.**class**, clientId);

Эквивалентность (методы equals/hashCode)

Чтобы сравнивать объекты, как было указано выше,

boolean **equals** = order.**equals**(anotherOrder);

... нужно определить методы equals и hashCode. Часто реализуют их через id:

@Override

**public** **boolean** **equals**(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

Order that = (Order) o;

**return** Objects.equals(id, that.id);

}

@Override

**public** **int** **hashCode**() {

**return** Objects.hash(id);

}

... но это не вполне корректно.

Из нашего принципа — не привязываться к JPA — следует, что суррогатный id не должен фигурировать в equals/hashCode. Важно помнить, что для новых (transient) entities поле id еще не инициализировано (равно null). Тем не менее, эквивалентность должна работать корректно независимо от состояния entity.

**Объекты должны сравниваться по бизнес-ключу, а не по id.**

Для класса Order в качестве бизнес-ключа напрашивается пара полей дата-клиент.

@Override

**public** **boolean** **equals**(Object o) {

**if** (**this** == o) **return** **true**;

**if** (!(o **instanceof** Order)) **return** **false**;

Order order = (Order) o;

**return** Objects.equals(created, order.created)

&& Objects.equals(client, order.client);

}

@Override

**public** **int** **hashCode**() {

**return** Objects.hash(created, client);

}

Обратите внимание, что Hibernate может создавать прокси-объекты и проверять класс следует не методом getClass(), а через instanceOf. К счастью, в Lombok этот момент учтен.

Только в исключительных случаях, если у класса нет ничего, что может быть использовано в качестве бизнес-ключа кроме поля id, equals() проверяем через тождество (==) и эквивалентность id, а hashCode() без полей вырождается в константу.

@Override

**public** **boolean** **equals**(Object o) {

**if** (**this** == o) **return** **true**;

**if** (!(o **instanceof** Foo)) **return** **false**;

Foo that = (Foo) o;

**return** Objects.equals(id, that.id);

}

@Override

**public** **int** **hashCode**() {

**return** 31;

}

Более детально тему раскрывает Влад Михальча:

* [«The best way to implement equals, hashCode, and toString with JPA and Hibernate»](https://vladmihalcea.com/the-best-way-to-implement-equals-hashcode-and-tostring-with-jpa-and-hibernate/);
* [«How to implement equals and hashCode using the JPA entity identifier (Primary Key)»](https://vladmihalcea.com/how-to-implement-equals-and-hashcode-using-the-jpa-entity-identifier/);
* [«How to implement Equals and HashCode for JPA entities»](https://vladmihalcea.com/hibernate-facts-equals-and-hashcode/).

Cascades

JPA призвано всячески упрощать рутинные операции. Для сохранения и удаления сложных объектов нет необходимости «пробегать» по структуре и повторять операции для вложенных объектов, достаточно указать каскады.

В нашем примере при создании нового заказа для нового пользователя не нужно сохранять их по отдельности, это сделает каскад.

Очевидный нюанс: прежде чем указывать CascadeType.ALL, подумайте, нужен ли включенный в него CascadeType.REMOVE. Опять же, для нашего примера — нет, при удалении заказа, клиент не удаляется, поэтому ALL не применяем.

Entity и DTO

Если мы разрабатываем Web-приложение, нам не обойтись без передачи entities на Front и обратно. Теория учит, что для передачи следует использовать отдельные DTO классы. Часто, в случае простых entities типа Client и Address в нашем примере, поля ClientDto и AddressDto будут точно такие же. Возникает соблазн не создавать отдельные классы, а использовать существующие. Это неверный подход.

Могут появиться поля, нужные только для DTO. Приходится маскировать их от сохранения в БД с помощью @Transient. Возможны изменения значений полей перед отправкой на UI. Чтобы эти модификации не отразились в БД, начинаются вызовы entityManager.detach().

Коллекции по умолчанию работают в режиме lazy loading и уходят на UI пустыми. Изменение режима на FetchType.EAGER закладывает серьёзную мину под производительность. Загрузка элементов коллекции теперь будет происходить во всех вопросах и создавать [N+1 проблему](https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping). Не делайте так. У Entity и DTO разная ответственность. Валидацию введенных пользователем данных — проверки @NotNull, @Size и т. д. — делает DTO.

Правильный подход — всегда создавать отдельные DTO классы. Чтобы сократить написание boilerplate кода по перекладыванию полей используйте [MapStruct](http://mapstruct.org/) или [аналоги](https://www.baeldung.com/java-performance-mapping-frameworks).

FetchType.EAGER

Исторически Hibernate по умолчанию использует режим EAGER загрузки в отношении ManyToOne и OneToOne, а во всех остальных случаях — LAZY. Рекомендуется использовать LAZY во всех случаях. Указать в запросе делать join вместо нескольких select-ов всегда возможно, а обратно — отключить EAGER для определённых случаев — нет.

Опять же, слово Владу — [«EAGER fetching is a code smell when using JPA and Hibernate»](https://vladmihalcea.com/eager-fetching-is-a-code-smell/).

EntityManager.flush() и clear()

Ещё один тревожный маркер — это многочисленные вызовы flush() и clear(). Почему-то вместо того, чтобы доверить управление entities фреймворку, разработчики начинают вмешиваться в этот процесс.

Навскидку приходят в голову только две исключительные ситуации, когда нужны эти методы:

* обработка очень большого количества данных (репорты), переполнения кеша 1-го уровня;
* вызов бизнес-логики из хранимых процедур в процессе транзакции, в таком случае нужен flush().

Во всех остальных случаях эти вызовы, скорее всего, лишние и только ухудшают производительность.

Project Lombok

Отличная штука, сокращает количество boilerplate кода. С JPA, однако, необходимо учитывать нюанс. @Data по умолчанию включает в себя @EqualAndHashCode и @ToString по всем полям, что в свою очередь может порождать каскад ненужных загрузок полей, игнорируя старательно указанный FetchType.LAZY, и зацикленные вызовы для bi-directional отношений.

Поэтому рекомендуется не использовать @Data, а  в  @EqualsAndHashCode и @ToString указывать только нужные поля.

Правильные entities

Применяя всё вышеизложенное к нашим классам, получаем:

@Getter

@EqualsAndHashCode(of = {"firstName", "lastName"})

@ToString(of = {"firstName", "lastName"})

@NoArgsConstructor

@Entity

@Table(name = "**clients**")

public **class** **Client** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

@Column(length = 32, nullable = false)

**private** String firstName;

@Column(length = 32, nullable = false)

**private** String lastName;

public Client(String firstName, String lastName) {

**this**.firstName = firstName;

**this**.lastName = lastName;

}

}

@Getter

@EqualsAndHashCode(of = "name")

@ToString(of = {"name", "price"})

@NoArgsConstructor

@Entity

@Table(name = "**items**")

public **class** **Item** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

@Column(nullable = false, length = 32)

**private** String name;

@Columns(

columns = {

@Column(name = "price\_currency", length = 3, nullable = false),

@Column(name = "price\_amount", precision = 7, scale = 2, nullable = false)

}

)

@Type(**type** = "org.jadira.usertype.moneyandcurrency.moneta.PersistentMoneyAmountAndCurrency")

**private** Money price;

public Item(String name, Money price) {

**this**.name = name;

**this**.price = price;

}

}

@Getter

@EqualsAndHashCode(of = {"city", "street", "building"})

@ToString(of = {"city", "street", "building"})

@NoArgsConstructor

@AllArgsConstructor

@Embeddable

public **class** **Address** {

**private** String city;

**private** String street;

**private** String building;

}

@AllArgsConstructor

public enum **Status** {

NEW("N"),

PROCESSING("P"),

COMPLETED("C"),

DEFERRED("D");

@Getter

**private** **final** String code;

}

@Converter(autoApply = true)

public **class** **StatusConverter** **implements** **AttributeConverter<Status**, **String>** {

@Override

public String convertToDatabaseColumn(Status status) {

**return** status.getCode();

}

@Override

public Status convertToEntityAttribute(String code) {

**for** (Status status : Status.values()) {

**if** (status.getCode().equals(code)) {

**return** status;

}

}

**throw** **new** IllegalArgumentException("Unknown code " + code);

}

}

@Getter

@EqualsAndHashCode(of = {"created", "client"})

@ToString(of = {"created", "address", "express", "status"})

@NoArgsConstructor

@Entity

@Table(name = "**orders**")

public **class** **Order** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

@Column(nullable = false)

**private** LocalDateTime created = LocalDateTime.now();

@AttributeOverrides({

@AttributeOverride(name = "city", column = @Column(name = "address\_city", nullable = false, length = 32)),

@AttributeOverride(name = "street", column = @Column(name = "address\_street", nullable = false, length = 32)),

@AttributeOverride(name = "building", column = @Column(name = "address\_building", nullable = false, length = 32))

})

**private** Address address;

@Setter

**private** boolean express;

@Column(length = 1, nullable = false)

@Setter

**private** Status status = Status.NEW;

@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST, optional = false)

**private** Client client;

@ElementCollection

@Column(name = "quantity", nullable = false)

@MapKeyJoinColumn(name = "item\_id")

**private** Map<Item, Integer> items = **new** HashMap<>();

public Order(Address address, Client client) {

**this**.address = address;

**this**.client = client;

}

public Map<Item, Integer> getItems() {

**return** Collections.unmodifiableMap(items);

}

public void addItem(Item item) {

items.merge(item, 1, (v1, v2) -> v1 + v2);

}

public void removeItem(Item item) {

items.computeIfPresent(item, (k, v) -> v > 1 ? v - 1 : null);

}

}

Многочисленные аннотации @Table и @Column присутствуют только для того, чтобы сгенерировать точно такую же схему, как на диаграмме. Пример учебный, на практике для генерации id лучше использовать sequence.

Внимательный читатель должен заметить, что использование поля client c lazy-загрузкой в Order.equals() противоречит рекомендациям раздела Lombok. Если бизнес-логика позволяет, лучше реализовать эквивалентность без него.

Заключение

JPA достаточно обширная тема, а Hibernate, к сожалению, содержит большое количество «gotchas», чтобы рассмотреть все нюансы в одной статье.

Много полезного и интересного [в блоге Влада Михальча](https://vladmihalcea.com/tutorials/hibernate/).

За кадром остались:

* наследование;
* uni- и bi-directional отношения;
* стратегии работы с коллекциями;
* criteriaBuilder;
* batching;
* QueryDSL;
* и многие другие темы.

При встрече с ними я бы рекомендовал придерживаться основного посыла, который всячески старался проиллюстрировать — **вначале полностью рабочая объектная модель**, потом вопросы, как сохранить её в БД, не наоборот.

Резюмируя, составим check list потенциальных проблем, которые освещены в статье:

* entities состоят из большого количества примитивных полей, а не из классов;
* вместо enum-ов используются строковые константы;
* методы equals/hashCode не определены;
* в бизнес-логике и запросах фигурируют id;
* происходит смешение Entity и Dto в одном классе;
* использование FetchType.EAGER, в том числе по умолчанию в @ManyToOne;
* ненужные вызовы flush() и clear();
* неаккуратное использование Lombok.

[**[https://s.dou.ua/img/avatars/40x40_148411.jpg](https://dou.ua/users/dmitry-bugay/)Dmitry Bugay**](https://dou.ua/users/dmitry-bugay/)[10.05.2019 13:21](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580909)

Статья неплоха, но по сути, является компиляцией бессчетного количества других статей, например того же Влада.

в бизнес-логике и запросах фигурируют id;

В реальных приложениях идентификаторы всегда фигурируют. Давайте это признаем, вместо того чтобы реплицировать теоретический идеализм. В тех же внешних запросах вида clients/123/orders/456/items/789/price.  
Ту да же относится и порицаемое автором еквалс и хешкод по ид. В теории да, все красиво очень.  
На практике же, есть у вас миллион ордеров с полем name="shirt", color="BLUE«. И все еквивалентны, поздравляю :) Но все принадлежат разным лицам, и являются различными записями, а у вас они все будут равны. Какой практический смысл вы получите с такого равенства?  
Мы давно перешли на еквалс по айдишнику (если быть точным, то по ууидам), и это упростило все. Потому что на практике важна идентичность, а не равенство по бизнес-параметрам. С вероятностью 99,99% бизнес-идентичные данные, принадлежащие разным графам, никогда не пересекутся в одной логической операции и/или транзакции и вам нечего будет сравнивать. В остальных случаях бизнес-равенство является более сложным понятием, оно определяется конкретной потребностью и для него пишется отдельный компаратор или несколько.  
Помимо этого, как вы смотрите на то, что бизнес-логика очень часто должна *позволять хранить бизнес-дубликаты в коллекциях*? Вам нужны в коллекциях *разные* объекты которые еквалс друг другу? Точно?

entities состоят из большого количества примитивных полей, а не из классов;

В самом факте того, что сущность состоит из большого количества примитивных полей, нет ничего страшного, в ней просто может быть много плоских данных.

и многие другие темы.

...например вот о чем. Развесили вы по всем бест-практисам лейзи лоадинга, все вроде красиво. Но тут у вас есть запрос который чтото достает, и отдает в жсон. А это чтото — глубокой вложенности, и все на лейзи. Привет, Hibernate.initialize(graphRootEntity); В JPA нет такой функции, привет платформ-депендент апи.  
А потом оказывается, что отдельные компоненты графа нужны, отдельные нет. А у вас уже хибернейт.инишлайз стоит. В результате появляется сущность, описывающая все нужные данные с расставленным везде игером.

А еще есть такой вариант, когда бизнес-идентификатор хранится отдельным полем как ентити, и нужен чуть более чем вообще всегда. Лейзи лоадинг такого объекта вообще будет ошибкой. При этом сделать его плоским нельзя, потому что таблицу идентификации используют многие разные сущности.  
Хороший пример — хранилище ШКИ в логистике.

Я не порицаю лейзи лоадинг, мне просто не нравится тиражируемое заблуждение «лейзи = силвербуллет, игер = сжечьеретика», которая сквозит в большинстве статей про хиб/жпа.

Изменение режима на FetchType.EAGER закладывает серьёзную мину под производительность.

Это не совсем так. Зависит, внезапно, бадум-тссс от того, какие у вас сценарии работы с сущностями. Неожиданно, да? :)  
Рассмотрим крайности.  
Сценарий 1, примитивный.  
У нас круд. GET cats/123 и так далее. 100% запрошенной из базы инфы конвертируется в жсон и отдается в веб. Так вот, в таком сценарии на фечтайп абсолютно пофиг, и лейзи будет даже вреден. Просто потому что вам все равно нужно отдать всю информацию, которая есть в базе. Игером или лейз, но надо ехать. Поэтому можно не парится и ставить игер везде, пускай хиб до итс бест и нафигачит джойнов, все равно вам все это вытягивать.  
Сценарий 2, сложный.  
У вас приложение, обратное круду. Т.е. по запросу вызывается каскад сложнейшей логики, которая будет доставать/апдейтить/создавать кучу данных через кучу джойнов и форейн кеев, при этом для большинства операций будет использован лишь факт связи и ограниченное количество данных, а обратно будет отдано очень ограниченное количество динамических сгенерированных данных. В таком случае, таки действительно необходим лейзи, поскольку хиб будет играть роль не столько собирателя данных сколько формирователя необходимого графа, где не все данные действительно нужны, а чать работы будет происходить вообще с одними лишь идентификаторами.

Загрузка элементов коллекции теперь будет происходить во всех вопросах и создавать N+1 проблему.

Н+1 не является такой уж серьезной проблемой. Более того, на практике он неизбежен.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[10.05.2019 17:12](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581038)

Спасибо за развёрнутый коментарий.

В остальных случаях бизнес-равенство является более сложным понятием, оно определяется конкретной потребностью и для него пишется отдельный компаратор или несколько.

С этим, конечно, сложно спорить.

Помимо этого, как вы смотрите на то, что бизнес-логика очень часто должна позволять хранить бизнес-дубликаты в коллекциях? Вам нужны в коллекциях разные объекты которые еквалс друг другу? Точно?

А из-за этого, мне кажется, массово используется List вместо Set. Что в свою очередь приводит к невозможности отличить элементы в Cartesian product.

Развесили вы по всем бест-практисам лейзи лоадинга, все вроде красиво. Но тут у вас есть запрос который чтото достает, и отдает в жсон.

Json аннотации прямо на entity? Или есть Dto?

Я не порицаю лейзи лоадинг, мне просто не нравится тиражируемое заблуждение «лейзи = силвербуллет, игер = сжечьеретика», которая сквозит в большинстве статей про хиб/жпа.  
Это не совсем так. Зависит, внезапно, бадум-тссс от того, какие у вас сценарии работы с сущностями. Неожиданно, да? :)

Я не хотел бы, чтобы меня приняли за фанатика lazy, но как быть, если eager устраивал до поры до времени, а потом внезапно появляется сценарий где эти данные не нужны, а их выборка тормозит запрос?

Н+1 не является такой уж серьезной проблемой. Более того, на практике он неизбежен.

Без конкрентного сценария и замеров невозможно сказать, насколько серьёзна проблема.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_ed2hf4uzfzi_0ql0fvM.jpg](https://dou.ua/users/yurij-gagarin-1/)Dyvak Yurii**](https://dou.ua/users/yurij-gagarin-1/) Software engineer[13.09.2019 16:00](https://dou.ua/lenta/articles/how-to-use-hibernate/#1673479)

Спасибо большое за доклад на Logeek Night. Во время доклада мы представили что DB нет, а есть только java, и у нас получились идеальные условия для разработки! А теперь представьте что DB всё-таки есть, и её охраняют 3 опасных DBA.  
Как везти разработку в таких условиях?  
Было бы очень интересно услышать ваше виденье inheritance, и тот случай когда нужно строить сложные структуры с разворотом таблицы?

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[13.09.2019 16:12](https://dou.ua/lenta/articles/how-to-use-hibernate/#1673498)

Спасибо за комментарий.

А теперь представьте что DB всё-таки есть, и её охраняют 3 опасных DBA.  
Как везти разработку в таких условиях?

Прочитал вопрос как «Что делать, если схема БД уже создана?» Два варианта: возможно Hibernate не нужен вообще, см. MyBatis. Либо JPA и «дизассемблировать» таблицы, большое количество полей «сгруппировать» с помощью @Embeddable, использовать enum и т.д.

ваше виденье inheritance

Я не сторонник использования наследования, это специфичный приём и должен применять аккуратно, вот моя старая, несколько наивная статья [dou.ua/...​n-vs-inheritance-in-java](https://dou.ua/lenta/articles/composition-vs-inheritance-in-java/)

Если наследование всё таки нужно, и проектировать от Java, то Hibernate прекрасно его поддерживает и предоставляет три стратегии для этого.

Если «дизассемблировать» таблицы, то @MappedSuperClass для повторяющихся полей (например, в каждой таблице есть аудит createdBy, createdAt, modifiedBy, modifiedAt) мне не кажется хорошей идеей. Я предолжил бы использовать композицию и @Embeddable.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_85453.jpg](https://dou.ua/users/igor-lytvynenko/)Igor Lytvynenko**](https://dou.ua/users/igor-lytvynenko/) Java Developer[18.05.2019 10:50](https://dou.ua/lenta/articles/how-to-use-hibernate/#1585503)

Є декілька різних позицій по цьому питанню.

1. Proxy задає життєвий цикл, здавалося б data об’єкта.

У вашій логіці є одне протиріччя. З однієї сторони, ви будуєте класи OOP-way — ціль достойна. Але з іншої, використовуєте для цього досить сильну магію.

В магії, як такій, нічого поганого немає, але в цьому випадку, кінець-кінцем, доведеться працювати з proxy об’єктами, а не з чистими data класами. І це в багато чому множить на нуль всі старання (так як ви вже не можете з цим об’єктом працювати як з data, він прив’язаний до storage/session, управляється ним та без нього не існує).

2. Ви натягуєте data об’єкт на базу.

Ви згадали про DTO. І вірно зазначили, що DTO часто відрізняються. Більше того, є сенс також розрізняти DTO (структури в запитах/відповідях api), DAO(структури для роботи з storage) та data-класи (структури для роботи в пам’яті). В більшості випадків, як мінімум з початку розробки, всі ці три набори класів сутностей тупо ідентичні. І в усіх цих випадках є сенс НЕ плодити сутності і взяти, наприклад, DAO класи і використовувати їх всюди. Як тільки десь щось змінюється — це все рефакториться та сутність розділяється.

Так ось, на вашому прикладі, перша модель — це для DAO, а друга — це для data. І мапити їх через ORM — це костиль, який спрацює лише в такому випадку як ви описали. Додайте сюди, скажем searchString в DAO (який генериться з інших полів не очевидним чином, щоб потім з нього пошук LIKE-ом робити). Та додайте в data-класі, замість Map items якусь специфічно оптимізовану колекцію (уявімо, для прикладу, що Item не товар, а щось, що має якісь координати, і колекція — не Map, а якийсь geo індекс). Також в data можна додати пару atomic примітивів та lock-ів для роботи в synchrinized. А тепер зліпіть ці дві сутності в одну і зробіть марінг в базу.

Суть не в тому, щоб довести вашу ідею до абсурду, а в тому, щоб показати, що ваш спосіб «пофіксити» «звичайний підхід» ламається, якщо рухатися в тому ж напрямку «покращення» стуктури, в якому ви кажете, що він круто працює.

3. Базу даних потрібно «поважати» ... вона все-таки зберігає для вас ваші дані :)

БД — не проста штука. І коли даних багато і складних, треба вирішити як з ними швидко працювати. І тут логіка починає тягнутися: в sql створюються денормалізовані таблиці та хитрі індекси, щось виноситься в nosql, щось в elastic search, а шось взагалі в щось схоже на annoy індекс чи in-memory структури.

Hibernate-like ORM ж кажуть: «та забийте, ми все зробимо ... ви тільки гляньте що ми можем і розберіться», але в результаті, по факту, там закрито дуже мало тих методів, що можна використати для оптимізації роботи з БД і зроблено це так, що ще можна пальці повідстрілювати.

4. В коментарях згадувався аудит...

Я не працював з аудитом в Hibernate, але з того, що доводилось робити, хуки на базу — було б дуже грубим рішенням.

В тих проектах, де ми робили аудит-логи, ці логи були на рівні чуть вище бази. Наприклад лог міг виглядати як: «[req:12355][user:45][ip:127.0.0.1][agent:android/chrome] Авторизація Петро Петро Петрович»,  
а не " Петро Петро Петрович lastLogin changed from XXX to YYY«.

5. Неочевдний життєвий цикл

Можливо, це все Hibernate і вміє, але є сенс звертати увагу на наступне.

Сутності бувають різні. Крім класичних строго транзакційних (користувач змінив name чи створив замовлення), ще є словники (які вигідно записати в map в пам’ять і оновлювати кожні 10хв або по зовнішньому trigger-у), логи (append-only, можна писати bulk-ом), статистика (можна накопичувати і писати агреговані зміни раз в хвилину)

Кожен з цих випадків працює по різному і, в цілому, ломає роботу з одним «графом» об’єктів.

Окремо є сенс згадати транзакційність...  
Я не знаю як в кого, але в нас з нею є вже традиційні особливості. І поки не видно яка може бути альтернатива retry-циклу з ідемпотентним try/catch-ем з явним контролем того які запити і в якому пооядку робляться.

А то буває тут kill −9 прилітає (чи щось типу пропало з’єднання до бази, якийсь lock wait exception чи disk is full, або, взагалі dead lock) а потім шукай що там де не так.

------  
В цілому, для ясності, я не є великий противник ORM. ORM — це часто зручно. Але від ORM я чекаю простий і швидкий mapper на data клас, простий crud api, зручний api для raw sql, optional query builder та утиліти для крайніх випадків (mass select/insert/update, управління транзакціями, тощо).

Робота ж з DDL, каскадний select/update/delete, детектор змін в об’єктах, кеш об’єктів з БД, автоматичні join-и, автоматична підтримка купи БД одночасно, та своя мова запитів — це все тільки ускладнює роботу.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_4692.png](https://dou.ua/users/mgu/)Mykola Gurov**](https://dou.ua/users/mgu/) Жирный тролль[12.05.2019 11:55](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581619)

Вопрос не троллинга ради, а чтобы понять для себя: зачем вообще использовать Хибернейт?

Пока вижу две ситуации:  
1) демо/пилотный проект. Работа с БД не важна, в простейшем случае не нужно об этом думать — все запустит Spring/JPA/Hibernate.  
2) коробочный продукт реально поддерживающий работу с разными базами данных.

В остальных случаях — какие проблемы решает Хибернейт не создавая больших, когда есть вменяемый jOOQ, Spring JDBC templates, etc?

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_148411.jpg](https://dou.ua/users/dmitry-bugay/)Dmitry Bugay**](https://dou.ua/users/dmitry-bugay/)[12.05.2019 12:18](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581621)

какие проблемы решает Хибернейт

Проблемы данных с большими графами связей. Если вам нужно вытащить сущность, которая является корнем графа на 20-30 узлов, и вы решите делать это плейн олд ждбц, вы это сделаете (возможно), но это замедлит разработку вашего проекта в десятки раз.

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_4692.png https://s.dou.ua/img/avatars/25x25_photo2019-08-2122-56-58_nnkXdHs.jpg https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg [Еще 31 комментарий](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[12.05.2019 14:26](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581633)

Классический ответ таков — JPA отражает доменную область на БД без написания sql вручную.  
Т.е. можно свободно оперировать объектами:

Order order = **new** Order(

**new** Address("Kiev", "Shevchenko av.", "1"),

**new** Client("Mykola", "Gurov")

);

Item item1 = **new** Item("aaa", Money.of(100, "USD"));

order.addItem(item1);

em.persist(order);

Потом делать

Item item2 = **new** Item("bbb", Money.of(50, "EUR"));

order.addItem(item2);

order.addItem(item1);

... и Hibernate выполнит все нужные insert-ы и update-ы и расставит id-шники.

\* совсем маленький нюанс [stackoverflow.com/...​ate-using-jpa-annotations](https://stackoverflow.com/questions/12391498/mapping-a-java-util-mapentity-entity-in-hibernate-using-jpa-annotations)

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_4692.png https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 32 комментария](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/25x25_6233.png](https://dou.ua/users/silverwolf/)Bogdan Shyiak**](https://dou.ua/users/silverwolf/) SE or Dev[12.05.2019 19:14](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581698)

1) демо/пилотный проект. Работа с БД не важна, в простейшем случае не нужно об этом думать — все запустит Spring/JPA/Hibernate.

Все-таки не «демо/пилотный», а проект где нужно часто менять требования/структуру БД. То есть большенство проектов в первые 0,5-1 год своей жизни. Плюс всякие админки и тд КРУД приложения. В теории с появлением [spring.io/...​projects/spring-data-jdbc](https://spring.io/projects/spring-data-jdbc) хибер может уйти на ... второй план.

какие проблемы решает Хибернейт не создавая больших, когда есть вменяемый jOOQ, Spring JDBC templates, etc?

Spring JDBC templates? Рылли? Даже в инфраструктуре спринга репозитории на ДжДБС теплейте занимают больше времени на разработку чем с хибером (это по опыту последних проектов). Сюда же количество багов, особенно когда люди не включат БД в воркфлоу тестов.  
Сюда же надо не забывать о том что существуют джуниоры, которые на ДжДБС теплейте такого трешака могут натворить (чтобы код был типобезопасным и переиспользуемым :) )

jOOQ как бы платный. Это шоустопер для энтерпрайзов.

№ 3: оказывается все еще есть много проектов где не используют спринг :) А по «альтернативным JPA» СО ответы сложнее найти.

Но в общем Хибернейт — гуано. По статье тоже есть вопросы, так сказать, но обсуждать мне их не хочетсо.

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_6233.png https://s.dou.ua/img/avatars/25x25_4692.png https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg [Еще 8 комментариев](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_101567.jpg](https://dou.ua/users/sarieflive-/)Андрей Плотников**](https://dou.ua/users/sarieflive-/) Java Software Engineer[11.05.2019 09:18](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581305)

Такое чувство что автор не знает о чем пишет

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[11.05.2019 09:33](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581312)

«Любая дискуссия в интернете постепенно скатывается на личности, поэтому чтобы сэкономить время, начну сразу с них»?

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_101567.jpg https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 19 комментариев](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_61931_9Cfmuxu.jpg](https://dou.ua/users/roman-gorodishcher/)Roman Gorodishcher**](https://dou.ua/users/roman-gorodishcher/) SDE[10.05.2019 14:47](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580967)

ORM не нужны.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_av2_fxavJgF.jpg](https://dou.ua/users/vlad/)Włodzimierz Rożkow**](https://dou.ua/users/vlad/) інфлюенсер в t.me/full\_of\_hatred[10.05.2019 15:57](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580990)

Фреймворки не нужны.

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_56064.jpg https://s.dou.ua/img/avatars/25x25_228820.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 4 комментария](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_av2_fxavJgF.jpg](https://dou.ua/users/vlad/)Włodzimierz Rożkow**](https://dou.ua/users/vlad/) інфлюенсер в t.me/full\_of\_hatred[10.05.2019 14:12](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580951)

Найголовніше шо має мати будь-яка ORM — це логування всіх реальних запитів у базу з таймінгами виконання і змінними які передаються. Бо як би нам не хотілося, абстрагуватися від бази, але все одно там чи сям протікає і треба дивитися чому твій запит робить N+1 або чому довго виконується і так далі.

І з цим у хібера серйозні проблеми. Дефолтне логування просто нікуди не годиться, воно зроблено на диво погано для такого популярного проекту.

Для порівняння подивіться на логи ActiveRecord, та це просто небо і земля.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_148411.jpg](https://dou.ua/users/dmitry-bugay/)Dmitry Bugay**](https://dou.ua/users/dmitry-bugay/)[10.05.2019 14:48](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580968)

Дефолтне логування просто нікуди не годиться, воно зроблено на диво погано для такого популярного проекту.

Вот тут согласен полностью. Логи хиба отвратные. С форматом логов еще можно было бы мириться, если бы он умел вставлять в лог параметры вместо ?, ?, ?, ?.

на логи ActiveRecord

АР это концепция а не имплементация. С имплементациями в джаве не очень. А причина одна —  Мартин теоретик. АР как идея сама по себе в реализации превращается в костыль. Даже теоретически АР означает что внутри объекта должна содержаться ссылка на провайдер какойнибудь персистентности, что убивает все влажные мечтания о поджо, объектности, слабой связности и теде и тепе.

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_av2_fxavJgF.jpg https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 21 комментарий](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[10.05.2019 16:31](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581011)

Нехорошо, да. DataSource Proxy типа [github.com/ttddyy/datasource-proxy](https://github.com/ttddyy/datasource-proxy) в помощь

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_av2_fxavJgF.jpg](https://dou.ua/users/vlad/)Włodzimierz Rożkow**](https://dou.ua/users/vlad/) інфлюенсер в t.me/full\_of\_hatred[10.05.2019 14:09](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580945)

Изменение режима на FetchType.EAGER закладывает серьёзную мину под производительность. Загрузка элементов коллекции теперь будет происходить во всех вопросах и создавать N+1 проблему.

В мене від цієї штуки спершу серйозно бомбануло.  
Чому там відразу не зроблено по-нормальному, без N+1?

Треба думати про це два рази — перший раз коли пишеш EAGER і другий раз коли пишеш join fetch. Бісить.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[10.05.2019 16:37](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581016)

Навіщо писати EAGER?  
По-нормальному зробити не вийде, одночасно можуть бути сценарії, коли треба діставати вкладені колекції і коли ці дані не потрібні

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_av2_fxavJgF.jpg https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_121029.jpg [Еще 9 комментариев](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_148411.jpg](https://dou.ua/users/dmitry-bugay/)Dmitry Bugay**](https://dou.ua/users/dmitry-bugay/)[10.05.2019 13:40](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580924)

делете

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 51 комментарий](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_98272_D2Fj9MB.jpg](https://dou.ua/users/victor-mikhaylov/)Victor Mikhaylov**](https://dou.ua/users/victor-mikhaylov/) junior manual tester[10.05.2019 12:54](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580894)

хорошо написал. к сожалению на практике приходится иметь с легаси проектами, где структура базы уже спроектирована (и часто не лучшим образом)

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[10.05.2019 16:51](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581025)

Спасибо.  
Да, обычно структура БД уже есть и в своей практике не припомню с ней каких-то особых проблем. А вот как её потом в entities «дизассемблировать» — тут самое веселье и начинается.  
Например: в каких-то таблицах есть createdBy, createdOn, modifiedBy, modifiedOn. Почему их выносят в супер-класс? Почему не композиция с @Embeddable?

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg https://s.dou.ua/img/avatars/25x25_148411.jpg [Еще 7 комментариев](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/40x40_87907.jpg](https://dou.ua/users/ievgen-safronenko/)Ievgen Safronenko**](https://dou.ua/users/ievgen-safronenko/) Senior Software Engineer[10.05.2019 12:26](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580872)

Спасибо за статью!

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[10.05.2019 17:18](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581042)

Спасибо за отклик!

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_me21_moUEx5F.jpg](https://dou.ua/users/sergey-morenets/)Sergey Morenets**](https://dou.ua/users/sergey-morenets/) Java evangelist[10.05.2019 12:19](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580864)

Спасибо за статью. За кадром также остались достаточно важные темы: потенциальные проблемы и bottlenecks Hibernate, почему Hibernate, а не EclipseLink и т.д.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_148411.jpg](https://dou.ua/users/dmitry-bugay/)Dmitry Bugay**](https://dou.ua/users/dmitry-bugay/)[10.05.2019 13:33](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580921)

Hibernate, а не EclipseLink

А у вас есть конкретные примеры преимущества EclipseLink над Hibernate? Если да — опишите их в статье, думаю всем будет интересно почитать.  
Если нет — то в чем смысл вопроса?

bottlenecks Hibernate

На моей практике «ботлнек хибернейта» при инвестигейте оказывался временной проблемой железа в стойке или самой бд :)

[Ответить](javascript:;)

[Поддержать](javascript:;)

https://s.dou.ua/img/avatars/25x25_me21_moUEx5F.jpg https://s.dou.ua/img/avatars/25x25_av2_fxavJgF.jpg https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg [Еще 4 комментария](https://dou.ua/lenta/articles/how-to-use-hibernate/)

[**[https://s.dou.ua/img/avatars/25x25_av2_fxavJgF.jpg](https://dou.ua/users/vlad/)Włodzimierz Rożkow**](https://dou.ua/users/vlad/) інфлюенсер в t.me/full\_of\_hatred[10.05.2019 14:35](https://dou.ua/lenta/articles/how-to-use-hibernate/#1580962)

проблемы и bottlenecks Hibernate

Які там можуть бути ботлнеки якщо всі проблеми з ормами від повільних запитів у базу?

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_32_o8McEnA.jpg](https://dou.ua/users/andriy-slobodyanyk/)Andriy Slobodyanyk**](https://dou.ua/users/andriy-slobodyanyk/) Java Developer[11.05.2019 05:44](https://dou.ua/lenta/articles/how-to-use-hibernate/#1581273)

Задачка на внимательность — что в моём примере является Hibernate specific?