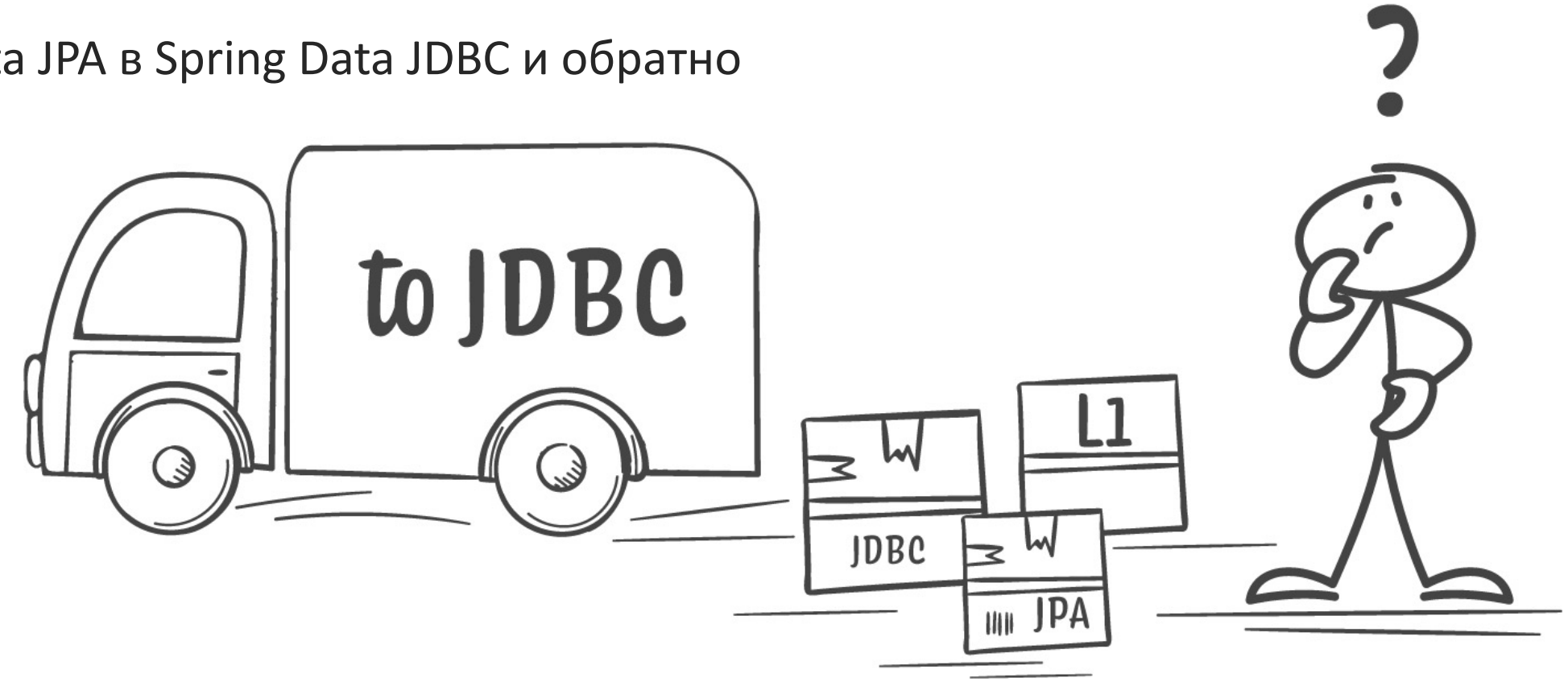
Один переезд равен двум пожарам

HAULMONT

Из Spring Data JPA в Spring Data JDBC и обратно



Немного стереотипов

- В Сибири всегда снег
- Все русские любят водку
- JPA тормозит

В Сибири всегда снег

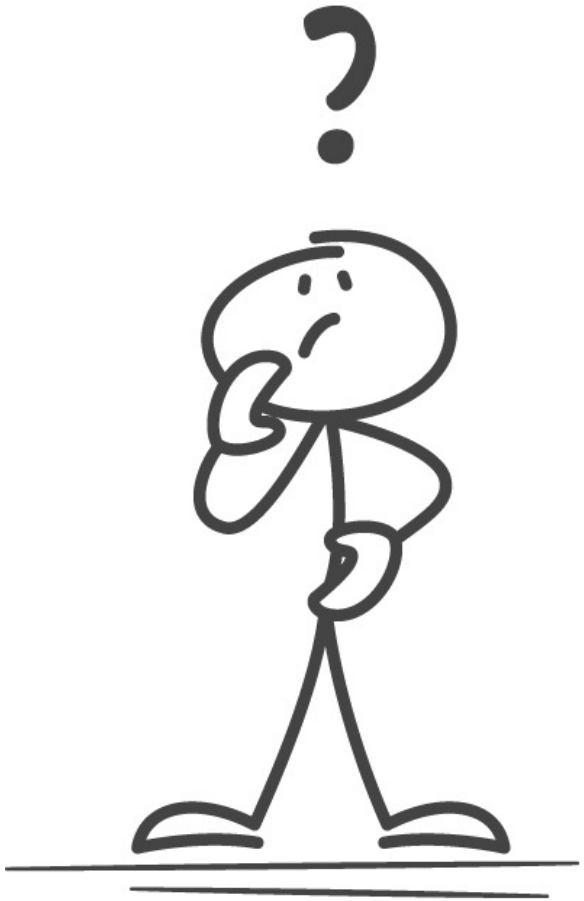
Все русские любят водку

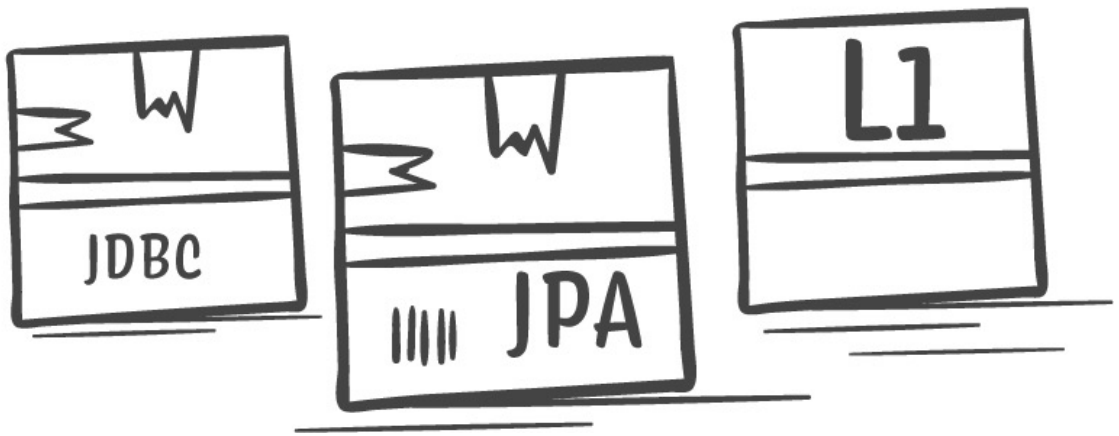
JPA...

Disclaimer

Мы говорим о существующем и работающем приложении

JPA -> JDBC





О чем поговорим

- Зачем?
- О чем нужно думать при переезде
- Процесс
- Потери и находки

Зачем?

JPA генерирует «неправильные» запросы

- Что именно не так с запросами?
  - Структура запросов
  - Нет подсказок (хинтов) оптимизатора
- Маппинг
  - Не сохраняются коллекции

Зачем?

Производительность

- В чем конкретно проблема?
  - Расход памяти (L1 кэш)
  - Лишние запросы (N+1,…)

Решено, переезжаем!

Какие варианты?

- ~~JDBC~~
- ~~JDBC Template~~
- Spring Data JDBC

Идеальный сценарий

```
<!-- Spring and Spring Boot dependencies -->


<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

jdbc

Spring Data JDBC

9.2. Domain Driven Design and Relational Databases.

All Spring Data modules are inspired by the concepts of “repository”, “aggregate”, and “aggregate root” from Domain Driven Design. These are possibly even more important for Spring Data JDBC, because they are, to some extent, contrary to normal practice when working with relational databases.

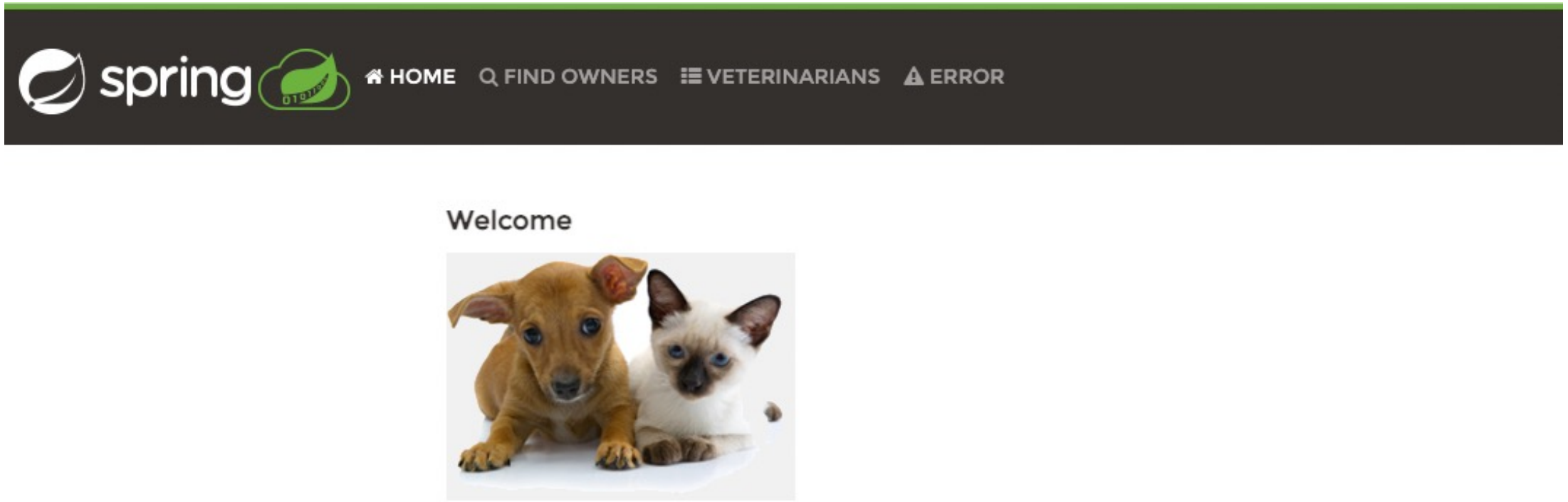
An aggregate is a group of entities that is guaranteed to be consistent between atomic changes to it. A classic example is an `Order` with `OrderItems`. A property on `Order` (for example, `numberOfItems` is consistent with the actual number of `OrderItems`) remains consistent as changes are made.

https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#jdbc.domain-driven-design

Spring Data JDBC

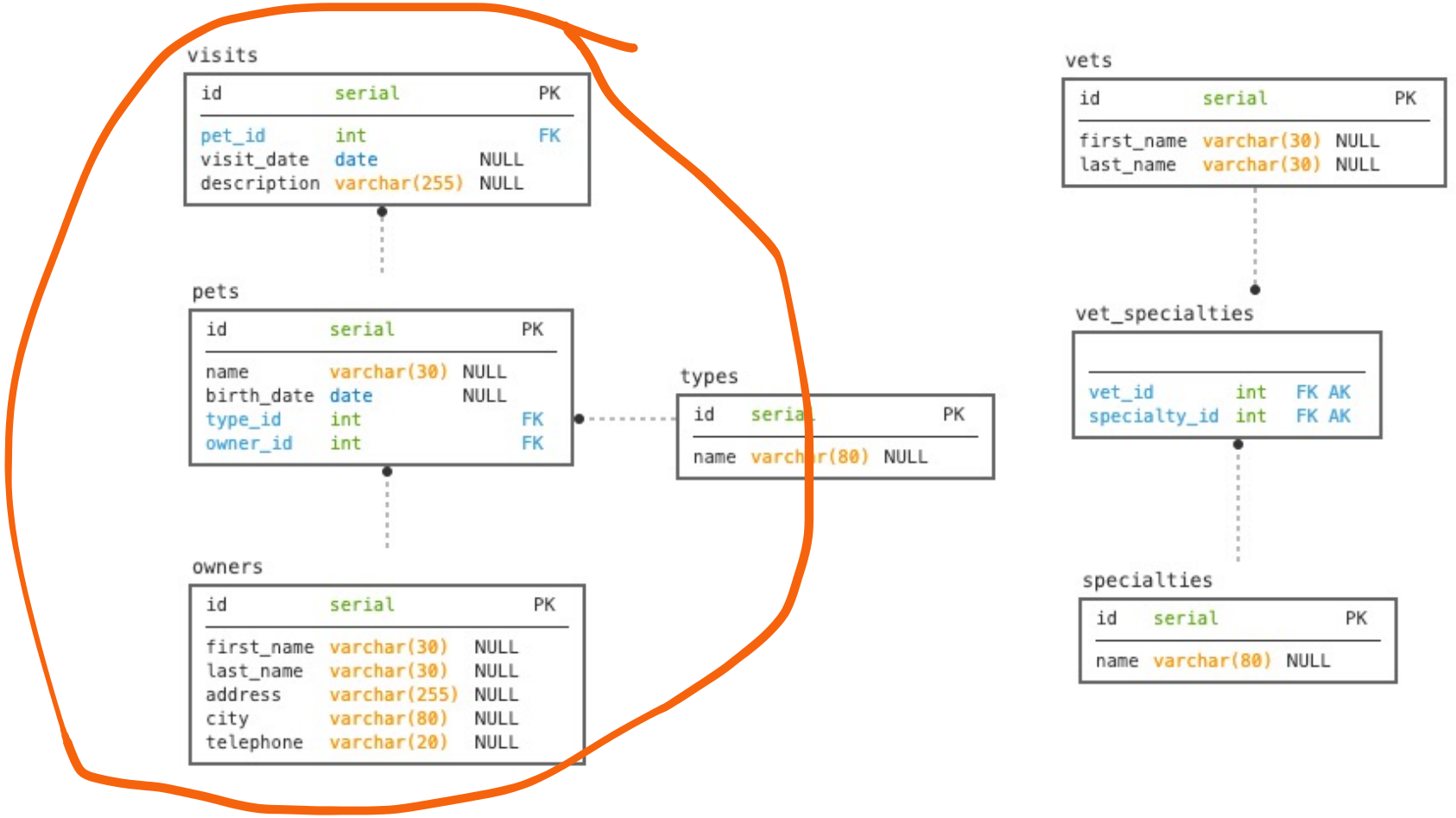
- Агрегаты
- Репозиторий для ~~каждой сущности~~ агрегата
- Нет контекста транзакции

Приложение для переезда



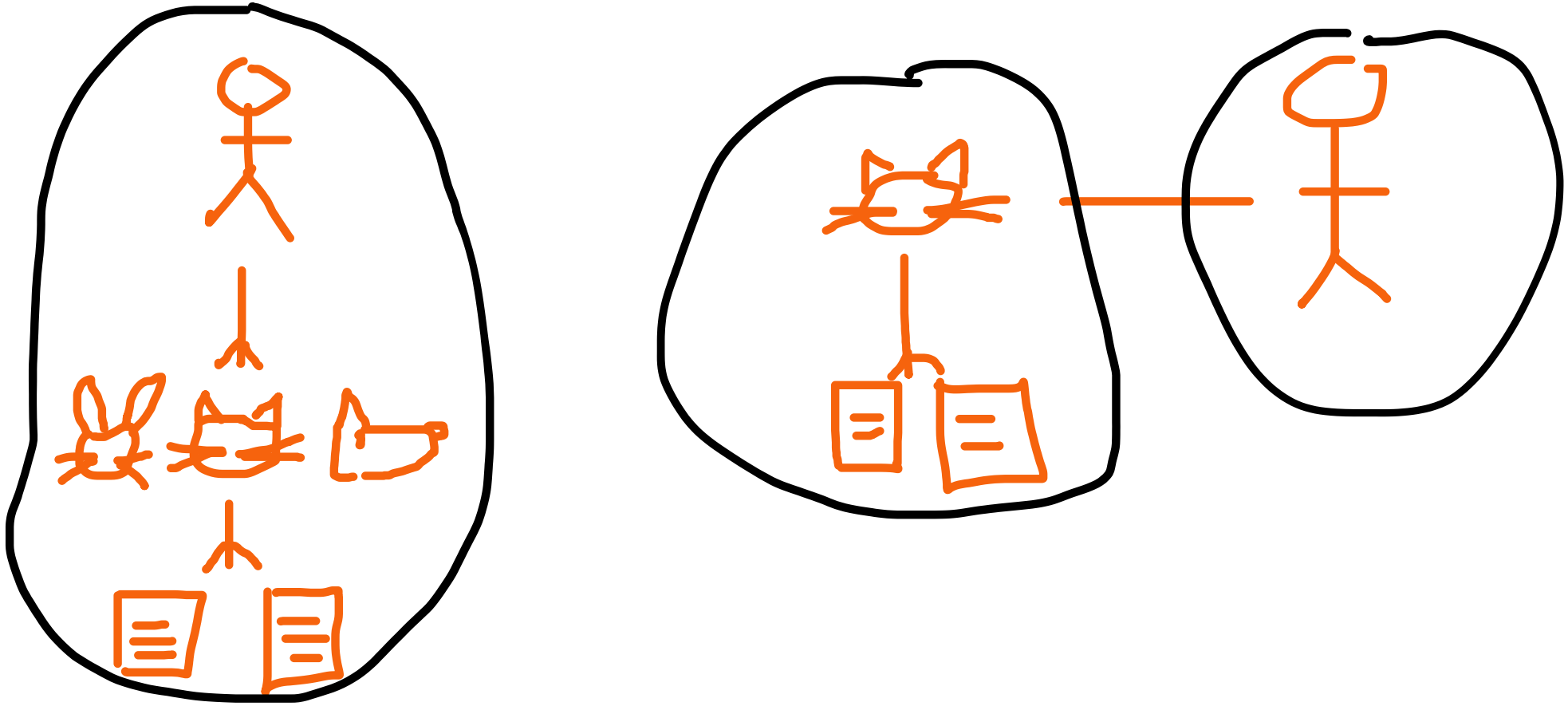
spring by Pivotal.

Pet Clinic – краткое содержание



Агрегат

Единица композиции, консистентная в рамках одной транзакции



Агрегат в JPA

```java
@Entity
@Table(name = "owners")
public class Owner extends Person {

    3 usages
    @Column(name = "address")
    @NotEmpty
    private String address;

    3 usages
    @Column(name = "city")
    @NotEmpty
    private String city;

    3 usages
    @Column(name = "telephone")
    @NotEmpty
    @Digits(fraction = 0, integer = 10)
    private String telephone;

    1 usage
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "owner_id")
    @OrderBy("name")
    private List<Pet> pets = new ArrayList<>();
```

```java
@Entity
@Table(name = "pets")
public class Pet extends NamedEntity {

    2 usages
    @Column(name = "birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    1 usage
    @ManyToOne
    @JoinColumn(name = "type_id")
    private PetType type;

    1 usage
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "pet_id")
    @OrderBy("visit_date ASC")
    private Set<Visit> visits = new LinkedHashSet<>();
```

HAULMONT

Агрегат в Spring Data JDBC

```java
public class Owner {

    3 usages
    @NotEmpty
    private String address;

    3 usages
    @NotEmpty
    private String city;

    3 usages
    @NotEmpty
    @Digits(fraction = 0, integer = 10)
    private String telephone;

    2 usages
    @MappedCollection
    private Set<Pet> pets;
```

```java
public class Pet {

    4 usages
    @Id
    private Integer id;

    3 usages
    private String name;

    3 usages
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    3 usages
    @Column("type_id")
    private Integer type;

    2 usages
    @MappedCollection
    private Set<Visit> visits;
```

HAULMONT

О чем мы (обычно) не думаем?

- Связи между сущностями
- Генерация ID
- Cascade операции
- Batch операции
- L1 cache
  - Отслеживание состояния
- Методы Spring Data репозиториев
  - findBy…
- Boot-time валидация

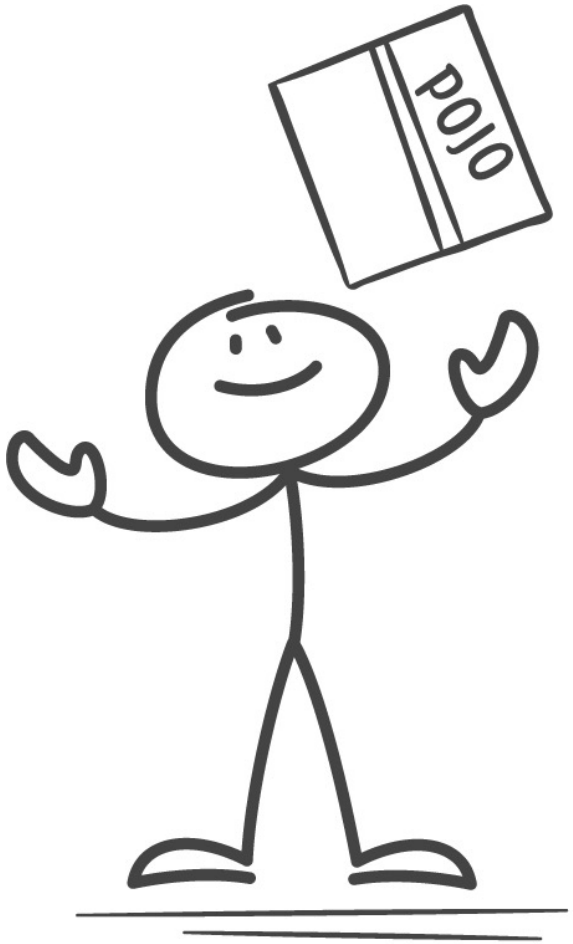
Через что нам надо пройти

- Не компилируется
- Не запускается
- Не работает

```java
@Entity
@Table(name = "vets")
public class Vet extends Person {

    4 usages
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "vet_specialties", joinColumns = @JoinColumn(name = "vet_id"),
            inverseJoinColumns = @JoinColumn(name = "specialty_id"))
    private Set<Specialty> specialties;
```

Не компилируется

Переделываем сущности



```java
@Entity
@Table(name = "pets")
public class Pet {

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

    @Column(name = "name")
    private String name;

    @Column(name = "birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    @ManyToOne
    @JoinColumn(name = "type_id")
    private PetType type;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private Owner owner;

    @OneToMany(mappedBy = "pet")
    private List<Visit> visits;
```

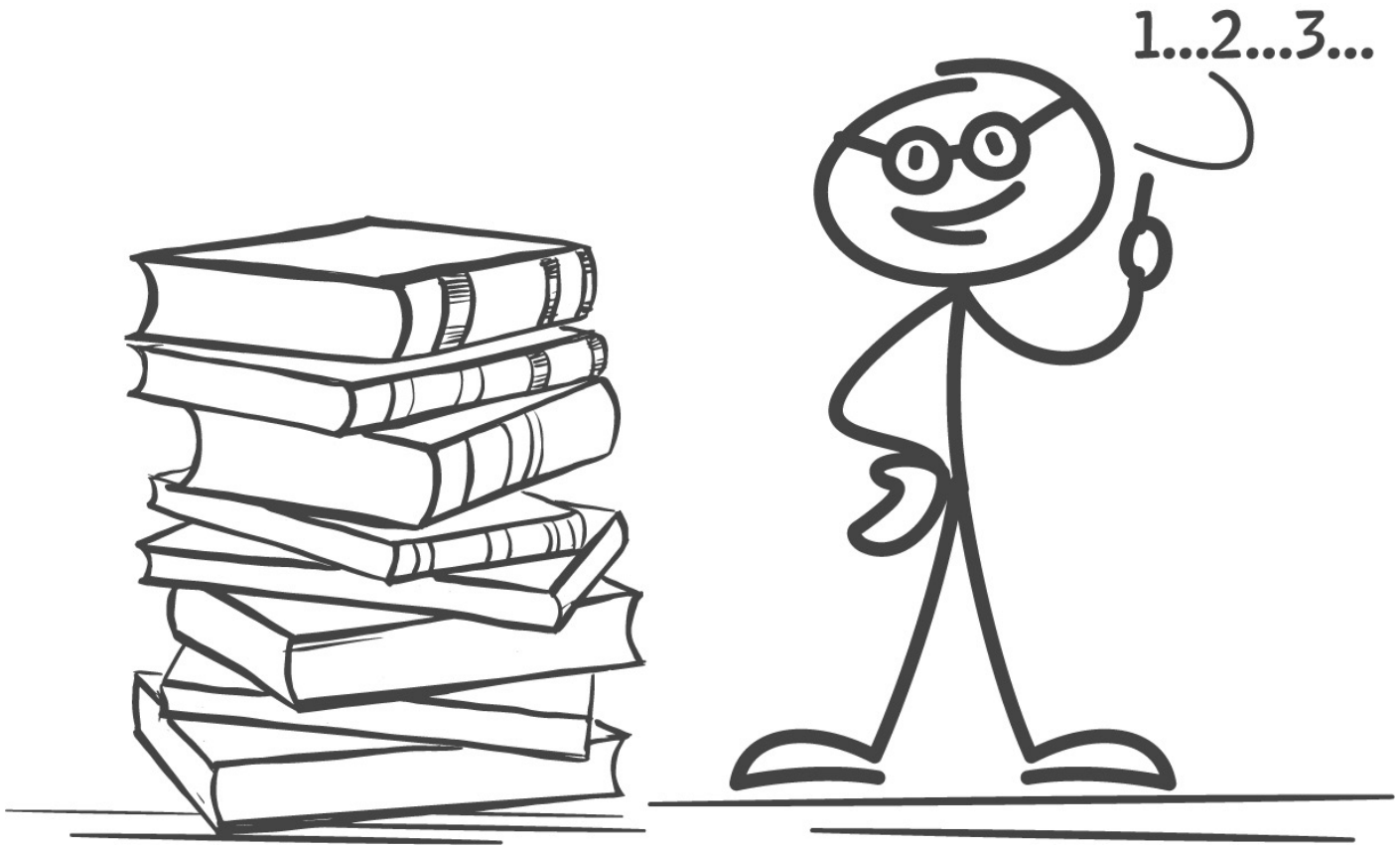
```java
@Column("type_id")
private Integer type;

@Column("owner_id")
private Integer owner;

@MappedCollection
private List<Visit> visits;
```

Diff

```
@Entity
@Table(name = "pets")
public class Pet {

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

    @Column(name = "name")
    private String name;

    @Column(name = "birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    @ManyToOne
    @JoinColumn(name = "type_id")
    private PetType type;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private Owner owner;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "pet_id")
    private List<Visit> visits;
```

```
@Table("pets")
public class Pet {

    @Id
    private Integer id;

    @Column("name")
    private String name;

    @Column("birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    @Column("type_id")
    private Integer type;


    @Column("owner_id")
    private Integer owner;


    @MappedCollection
    private List<Visit> visits;
```

Меняем связи и считаем запросы



Связи между сущностями

- Поддерживаются one-to-many, one-to-one
- Для many-to-* – reference ID
  - Или AggregateReference<T, ID>
  - Для many-to-many - + дополнительная сущность
- Все связи – eager
  - N+1 запрос

```java
@Entity
@Table(name = "pets")
public class Pet {

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

    @Column(name = "name")
    private String name;

    @Column(name = "birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    @ManyToOne
    @JoinColumn(name = "type_id")
    private PetType type;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private Owner owner;

    @OneToMany(mappedBy = "pet")
    private List<Visit> visits;
```

SELECT

JOIN

JOIN

+1 SELECT (optional)

```java
@Table("pets")
public class Pet {

    @Id
    private Integer id;

    @Column("name")
    private String name;

    @Column("birth_date")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;

    @Column("type_id")
    private Integer type;

    @Column("owner_id")
    private Integer owner;

    @MappedCollection
    private List<Visit> visits;
```

SELECT

+1 SELECT (optional)

+1 SELECT (optional)

+1 SELECT

Создаем ID

```java
@Entity
@Table(name = "pets")
public class Pet {

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

```java
@Table("pets")
public class Pet {


    @Id
    private Integer id;
```

```java
@Bean
public ApplicationListener<?> idSetting() {

    return (ApplicationListener<BeforeConvertEvent>) event -> {

        if (event.getEntity() instanceof Pet) {
            setId((Pet) event.getEntity());
        }
    };
}
```

Идем в базу и выбираем ID из последовательности

Генерация ID

- Пока только IDENTITY поля
- Все остальное – вручную
  - SEQUENCE
  - UUID

```
@Id
private UUID id = UUID.randomUUID();
```

Генерация ID

```java
@Override
public <T> T save(T instance) {

   Assert.notNull(instance, "Aggregate instance must not be null!");

   RelationalPersistentEntity<?> persistentEntity = context.getRequiredPersistentEntity(instance.getClass());

   Function<T, MutableAggregateChange<T>> changeCreator =
                           persistentEntity.isNew(instance) ?
                           this::createInsertChange :
                           this::createUpdateChange;

   return store(instance, changeCreator, persistentEntity);
}
```

Сразу присвоить ID -> UPDATE

- Либо явно пишем Insert
- Либо пишем callback

```java
@Bean
public ApplicationListener<?> idSetting() {

   return (ApplicationListener<BeforeConvertEvent>) event -> {

      if (event.getEntity() instanceof Pet) {
         setId((Pet) event.getEntity());
      }
   };
}
```

Конвертеры данных

- JSON
- JSONB
- BOOLEAN
- …

9.11. Custom Conversions

Spring Data JDBC allows registration of custom converters to influence how values are mapped in the database. Currently, converters are only applied on property-level.

- Нужно будет переписать и зарегистрировать

Конвертеры данных

```
@Column("visit_doc")
private VisitDoc visitDoc;
```

```
@WritingConverter
public static class VisitDocToString implements Converter<VisitDoc, String> {
   @Override
   public String convert(VisitDoc source) {
      return source.toString();
   }
}
```

```
@ReadingConverter
public static class StringToVisitDoc implements  Converter<String, VisitDoc> {
   @Override
   public VisitDoc convert(String source) {
      return VisitDoc.fromString(source);
   }
}
```

Конвертеры данных

```
@Column("visit_doc")
private VisitDoc visitDoc;
```

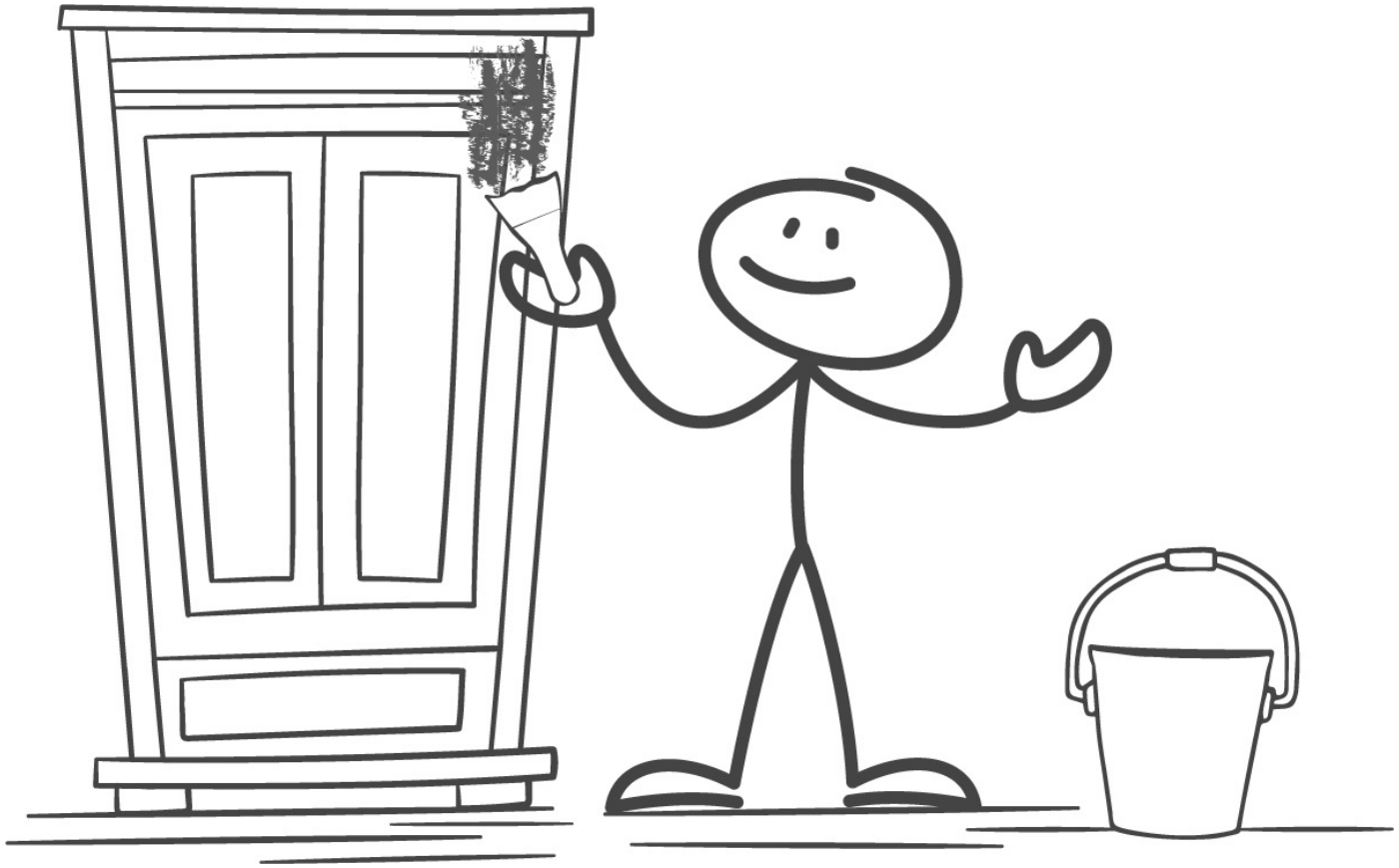
```
@Configuration
public class ConvertersConfiguration extends AbstractJdbcConfiguration {

   @Override
   protected List<?> userConverters() {
      return List.of(new VisitDocToString(), new StringToVisitDoc());
   }

}
```

```
Caused by: java.lang.IllegalArgumentException Create breakpoint : Cannot query by nested property: pets.name
    at org.springframework.data.jdbc.repository.query.JdbcQueryCreator.validateProperty(JdbcQueryCreator.java:137) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
    at org.springframework.data.jdbc.repository.query.JdbcQueryCreator.validate(JdbcQueryCreator.java:127) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
    at org.springframework.data.jdbc.repository.query.PartTreeJdbcQuery.<init>(PartTreeJdbcQuery.java:108) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
    at org.springframework.data.jdbc.repository.support.JdbcQueryLookupStrategy.resolveQuery(JdbcQueryLookupStrategy.java:122) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
```

Не стартует

```
Caused by: java.lang.UnsupportedOperationException Create breakpoint : Page queries are not supported using string-based queries. Offending method: public abstract org.springframework.data.domain.Page org
.springframework.samples.petclinic.owner.OwnerRepository.findAll(org.springframework.data.domain.Pageable)
    at org.springframework.data.jdbc.repository.query.StringBasedJdbcQuery.<init>(StringBasedJdbcQuery.java:106) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
    at org.springframework.data.jdbc.repository.support.JdbcQueryLookupStrategy.resolveQuery(JdbcQueryLookupStrategy.java:118) ~[spring-data-jdbc-2.3.3.jar:2.3.3]
    ... 58 common frames omitted
```

Обновляем репозитории



Методы Spring Data репозиториев

```
List<Owner> findByPets_NameLike (String name);
```

Частично поддерживаются

- Для корневой сущности
- Для всего остального - @Query

```
if (!path.getParentPath().isEmbedded() && path.getLength() > 1) {

   throw new IllegalArgumentException(
         String.format("Cannot query by nested property: %s",
                                 path.getRequiredPersistentPropertyPath().toDotPath()));

}
```

```
List<Owner> findByPets_NameLike (String name);
```

```
@Query("select * from owners o join pets p on o. id = p.owner_id and p.name like :name")
List<Owner> findByPets_NameLike (@Param("name") String name);
```

Убираем JPQL

```java
@Query("SELECT DISTINCT owner FROM Owner owner left join  owner.pets WHERE owner.lastName LIKE :lastName% ")
@Transactional(readOnly = true)
Page<Owner> findByLastName(@Param("lastName") String lastName, Pageable pageable);
```

```java
@Query("SELECT * FROM owner WHERE last_name LIKE concat(:lastName,'%')")
@Transactional(readOnly = true)
Collection<Owner> findByLastName(@Param("lastName") String lastName);
```

```java
if (queryMethod.isPageQuery()) {
   throw new UnsupportedOperationException(
         "Page queries are not supported using string-based queries. Offending method: " + queryMethod);
}
```

JPQL vs SQL

```
@Query("SELECT DISTINCT owner FROM Owner owners left join owner.pets WHERE owner.lastName LIKE :lastName% ")
@Transactional(readOnly = true)
Page<Owner> findByLastName(@Param("lastName") String lastName, Pageable pageable);
```

```
Application run failed

Reason: Validation failed for query for method
public abstract Page OwnerRepository.findByLastName (String, Pageable)!
```

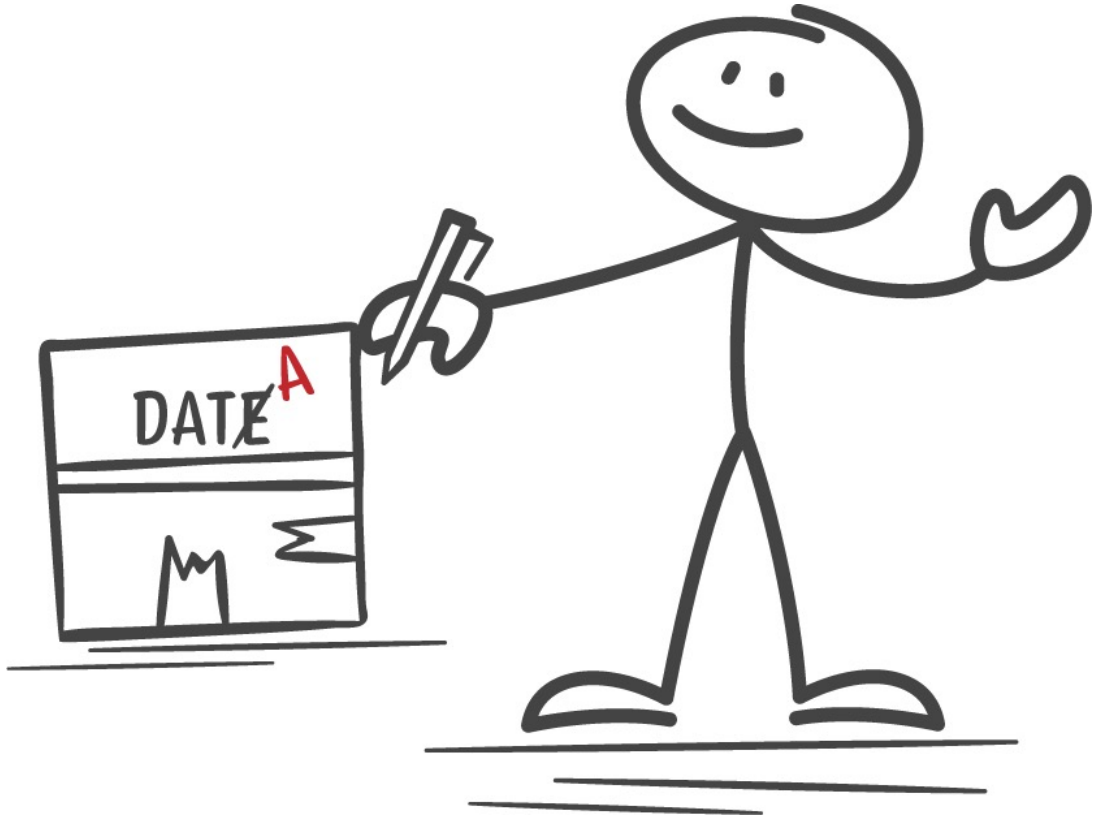
JPQL vs SQL

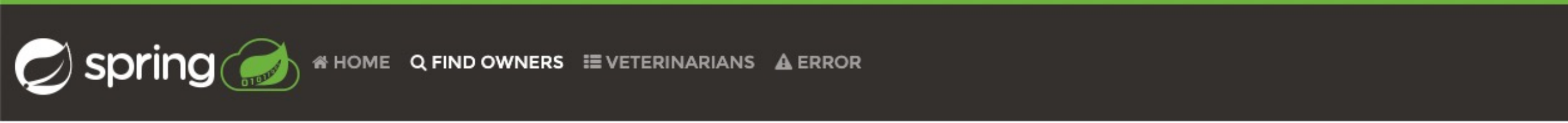
```
@Query("SELECT * FROM owners WHERE last_name LIKE concat(:lastName,'%')")
@Transactional(readOnly = true)
Collection<Owner> findByLastName(@Param("lastName") String lastName);
```

```
Failed to complete request: org.springframework.jdbc.BadSqlGrammarException:
PreparedStatementCallback;

bad SQL grammar [SELECT * FROM owners WHERE last_name LIKE concat(?,'%')];

nested exception is java.sql.SQLSyntaxErrorException: (conn=55) Table
'petclinic.owners' doesn't exist
```

Не работает

Обновляем данные





Pet

Owner

Jean Coleman

Name

Max

Birth Date

04.09.1995

Type

cat

Update Pet

JPA

```java
@PostMapping("/pets/{petId}/edit")
public String processUpdateForm(@Valid Pet pet, BindingResult result, Owner owner, ModelMap model) {
      owner.addPet(pet);
      this.petRepository.save(pet);
      return "redirect:/owners/{ownerId}";
}
```

```
Hibernate: select * from pets pet0_ where pet0_.id=?

Hibernate: * from owners owner0_ left outer join pets pets1_ on owner0_.id=pets1_.owner_id where owner0_.id=?

Hibernate: select * from types pettype0_ where pettype0_.id=?

Hibernate: select * from types pettype0_ where pettype0_.id=?

Hibernate: update pets set name=?, birth_date=?, owner_id=?, type_id=? where id=?
```

JDBC

```java
@PostMapping("/pets/{petId}/edit")
public String processUpdateForm(@Valid Pet pet, BindingResult result, Owner owner, ModelMap model) {
      pet.setOwner(owner);
      this.petRepository.save(pet);
      return "redirect:/owners/{ownerId}";
}
```

```
JdbcTemplate: [SELECT * FROM `owner` WHERE `owner`.`id` = ?]

JdbcTemplate: [select * from pet_type order by name]

JdbcTemplate: [UPDATE `pet` SET `name` = ?, `birth_date` = ?, `type_id` = ?, `owner_id` = ? WHERE `pet`.`id` = ?]

JdbcTemplate: [DELETE FROM `visit` WHERE `visit`.`pet_id` = ?]
```

Каскадные операции

- Применяются для агрегатов
- Есть особенности
  - Вставка – как обычно
  - Обновление – пересоздание дочерних записей
  - Удаление - каскадное

Batch операции

- Пока не поддерживаются
- Реализуется в JdbcTemplate

Batch операции - реализация

```
public interface OwnerRepository extends Repository<Owner, Integer>, WithOwnerBatch<Owner>


public class WithOwnerBatchImpl implements WithOwnerBatch<Owner> {

    private final String saveSql = """
        insert into owner (id, first_name, last_name, address, city, telephone)
        values (?, ?, ?, ?, ?, ?)
        """;

    @Override
    public <S extends Owner> Iterable<S> saveAll(final Iterable<S> entities) {

        int[] keys = this.jdbcTemplate.batchUpdate(saveSql, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException {
                Owner entity = list.get(i);
                ps.setInt(1, //assign ID);
                ps.setString(2, entity.getFirstName());
                ps.setString(3, entity.getLastName());
             //etc
         }
   });
```

Избавляемся от ненавистного L1 cache

L1 cache

- **Нет контекста транзакции**
- **Все detached**
  - Нет отслеживания состояния сущности
  - Сохранение изменений – явное
  - Insert/Update
    - @Id
    - @Version
    - implements Persistable
    - EntityInformation

Переносим визиты со скидкой

```java
@Service
public class VetService {

   @Transactional
   public List<Visit> postponeVisitsForVet(Vet vet, LocalDate fromDate, LocalDate toDate) {
      List<Visit> visits = visitRepository.findByDateAndVet(fromDate, vet);
      visits.forEach(visit -> {
         visit.setDate(toDate);
         visitService.applyDiscount(visit.getId(), BigDecimal.TEN);
      });
      return visits;
   }

}
```

SAVE

```java
@Service
public class VisitService {

   @Transactional
   public void applyDiscount(Integer visitId, BigDecimal discount) {
      Visit v = visitRepository.findById(visitId);
      //Applying discount
   }
```

Наверное, заработало

Spring Data JPA -> Spring Data JDBC

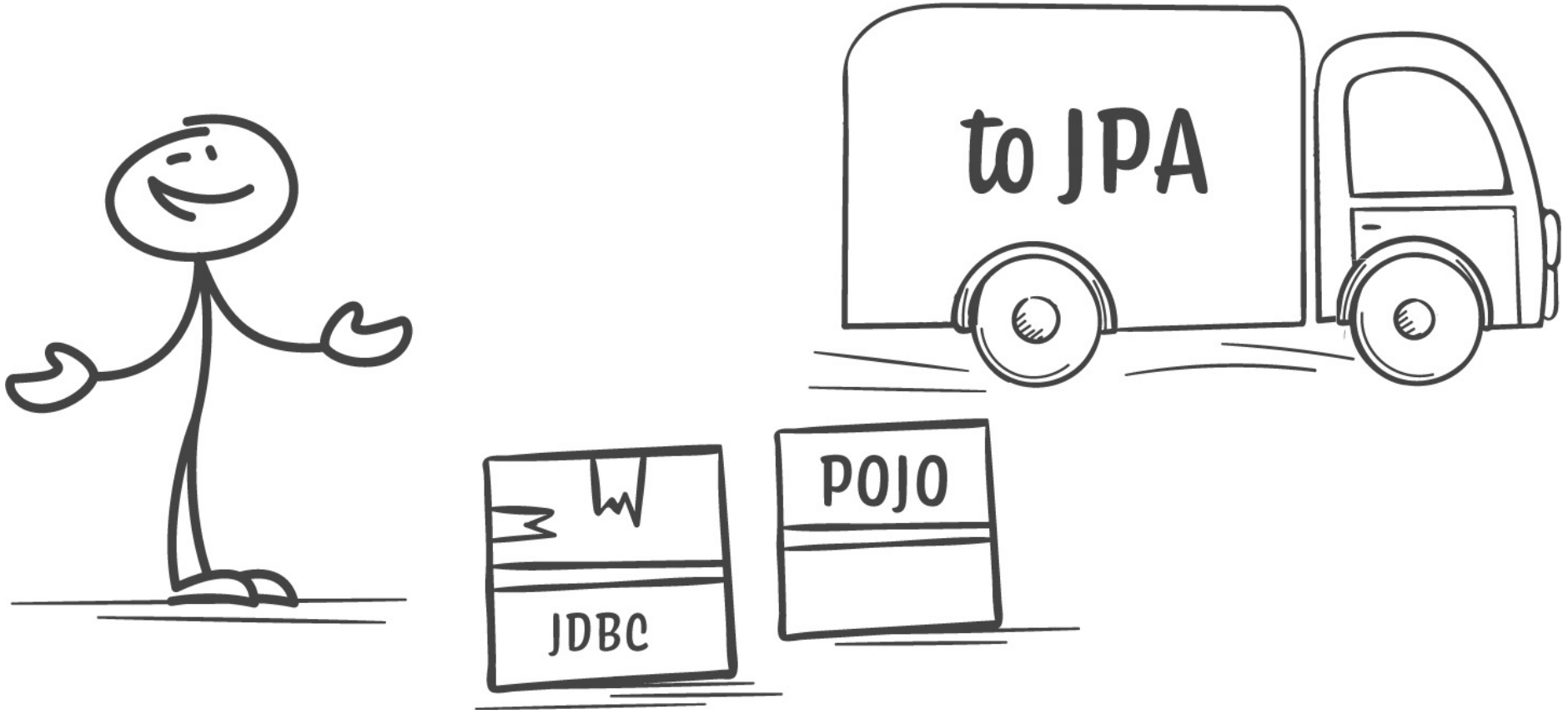
- Разберитесь в причинах
  - Неправильный SQL
  - Неправильный маппинг
  - Большое количество запросов
- Попробуйте обойтись существующим решением
  - JPQL
  - Native SQL
  - Entity Graph
- Не обязательно переводить все целиком
  - Можно переписать только часть модулей
  - JPA для CRUD, неJPA для сложной выборки

Spring Data JPA -> Spring Data JDBC

Если решили переезжать, обратите внимание на:

- Связи
- Lazy коллекции
- Генерация ID
- Batch операции
- Явное сохранение сущностей
- Пересмотреть Derived запросы
- JPQL перевести в SQL

JDBC -> JPA



Зачем?

Начинаем изобретать JPA

- Нужно работать со сложными графами сущностей
- Придумываем стратегии генерации ID
- Отслеживание состояния сущностей
- Реализуем ленивую загрузку
- Придумываем кэш запросов

Что нас ждет?

- Переделка модели данных
- Смена образа мышления
  - Объекты - первичны
  - Не писать простые запросы
- Работа с Transaction Context
  - Следить за состоянием сущностей
  - Думать про LAZY

Какие ещё альтернативы?

MyBatis

- Ближе к JDBC Template
- Контроль над запросами и маппингом
- Но переписывать много

MyBatis - Entities

```
public class Owner {

  private Integer id;

  private String firstName;

  private String lastName;

  private String address;

  private String city;

  @Digits(fraction = 0, integer = 10)
  private String telephone;

  private Set<Pet> pets;
}
```

MyBatis - Mappers

```java
@Mapper
public interface OwnerMapper {

  List<Owner> findOwners();

  Owner findById(Integer id);

}
```

```java
@Mapper
public interface OwnerMapper {

  @Select("select * from owners")
  List<Owner> findOwners();

  @Select("select * from owners where id = #{id}")
  Owner findById(Integer id);

}
```

```xml
<sql id="owner_column_list">
    id, first_name, last_name, address, city, telephone
</sql>

<select id="findById" resultMap="ownerMap">
    SELECT
    <include refid="owner_column_list"/>
    FROM owners WHERE id = #{id}
</select>
```

Какие ещё альтернативы?

jOOQ

- «Типизированный» SQL - DSL
- Кодогенерация
- Можно комбинировать с Hibernate

jOOQ - Entities

```
public class Owner {

  private Integer id;

  private String firstName;

  private String lastName;

  private String address;

  private String city;

  @Digits(fraction = 0, integer = 10)
  private String telephone;

  private Set<Pet> pets;
}
```

jOOQ - Repositories

```
@Repository
@Transactional(readOnly = true)
public class OwnerRepository {

    private final DSLContext dsl;

    public OwnerRepository(DSLContext dsl) {
        this.dsl = dsl;
    }

}
```

```
public Owner findById(Integer id) {
        return this.dsl.select().from(OWNER)
                .leftJoin(PET)
                .on(OWNER.ID.eq(PET.OWNER_ID))
                .leftJoin(PET_TYPE)
                .on(PET.TYPE_ID.eq(PET_TYPE.ID))
                .where(OWNER.ID.eq(id))
                .fetch()
                .intoGroups(OWNER)
                .values()
                .stream()
                .map(records -> {
                    Record6<Integer, String, String, String, String, String> record6s = records
                            .into(OWNER.ID, OWNER.FIRST_NAME, OWNER.LAST_NAME, OWNER.ADDRESS, OWNER.CITY, OWNER.TELEPHONE)
                            .get(0);
                    List<Pet> pets = records.sortAsc(OWNER.ID)
                            .stream()
                            .filter(record -> record.get(PET_TYPE.ID) != null)
                            .map(record -> new Pet(record.get(PET.ID), record.get(PET.NAME), record.get(PET.BIRTH_DATE),
                                    record.into(PetType.class), null, null))
                            .collect(toList());
                    return new Owner(record6s.value1(), record6s.value2(),
                            record6s.value3(), record6s.value4(), record6s.value5(),
                            record6s.value6(),
                       new HashSet<>(pets));
                }).findFirst()
                .orElseThrow(() -> new IllegalArgumentException("Cannot find Owner!"));
```

Если решили переезжать

Обратите внимание на:

- Связи
- Lazy коллекции
- Генерацию ID
- Batch операции
- Явное сохранение сущностей

Спасибо за внимание