HAULMONT

DTO

Живи быстро, гори ярко

Андрей Беляев

@belyaev_andrey

Что такое DTO

An object that carries data between processes in order to reduce the number of method calls. When you're working with a remote interface, such as Remote Facade, each call to it is expensive. As a result, you need to reduce the number of calls... The solution is to create a Data Transfer Object that can hold all the data for the call.

© Martin Fowler

Что такое DTO

- Носитель данных
- Описание данных, которые нам нужны

```yaml
NewUser:
  required:
    - email
    - password
    - username
  type: object
  properties:
    username:
      type: string
    email:
      type: string
    password:
      type: string
      format: password
```

```
type Owner {
  address: String
  city: String
  email: String
  firstName: String
  id: ID
  lastName: String
  telephone: String
}
```

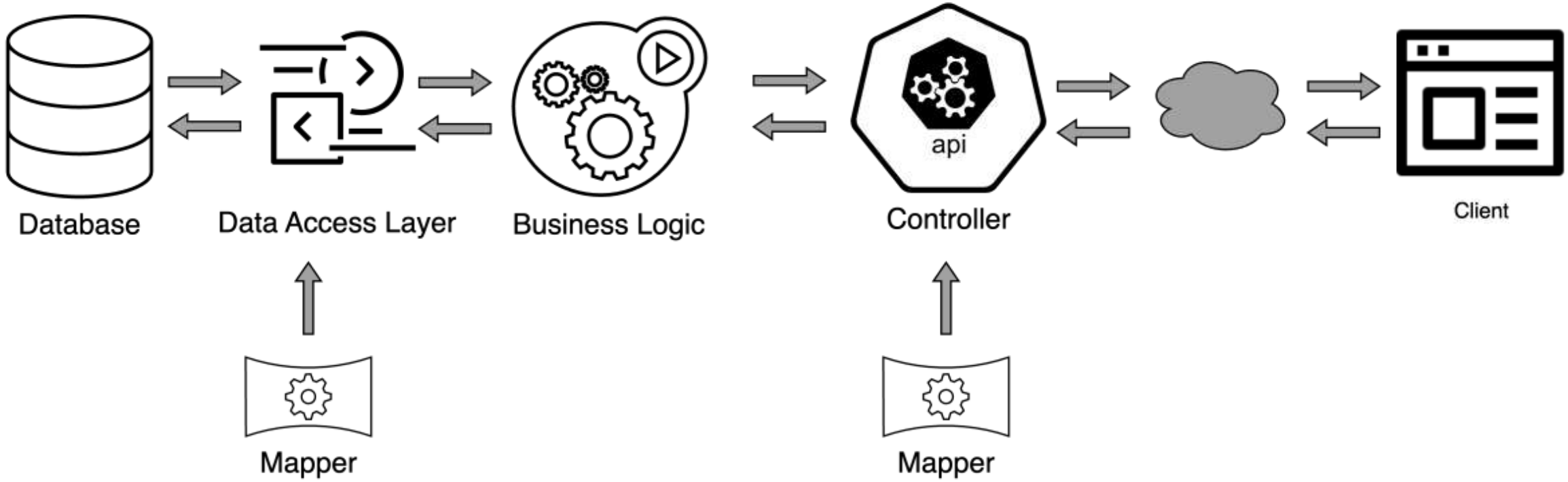
Этапы жизни DTO

- Создание
  - Наполнение данными
- Сериализация
- ...
- Десериализация

Как живут DTO

- Долго
- Это часть API
- Переживет приложение

Где используются DTO?



Adam Bein o DTO

1. Data Transfer Objects (DTOs) are introduced to decouple (JPA) entities from the UI
2. DTOs are also motivated by their typesafe nature
3. Lacking typesafety, JSON-P JsonObjects are not used as DTOs
4. Most DTOs are exposed as HTTP/JSON
5. Often 3rd party libraries are used to map a DTO into a JSON
6. Sometimes DTOs are called Value Objects
7. The vast majority of DTOs remains identical to entities over the entire lifecycle of a project
8. Copying data between DTOs and entities requires series of getter-setter invocations
9. DTOs are growing, the mapping logic is extracted into dedicated "mappers"
10. DTOs and mappers introduce a significant amount of untested code…
11. Getters and setters, constructor tests are written to increase the code coverage
12. Now: structural JPA entity code changes, affect DTOs, mappers and the corresponding unit test

https://www.adam-bien.com/roller/abien/entry/a_note_on_dtos

О чем поговорим

- **Виды DTO**
- **Этапы жизни DTO**
  - Создание
  - Сериализация
  - Десериализация
- **Потенциальные проблемы**

Пример: Petclinic

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```java
@Entity
public class PetType {
    @Id
    private Long id;

    private String name;
}
```

```java
@Entity
public class Visit {
    @Id
    private Long id;

    private LocalDateTime visitDate;

    private String diagnosis;

    @ManyToOne
    private Pet pet;
}
```

Adam Bein o DTO

1. Data Transfer Objects (DTOs) are introduced to decouple (JPA) entities from the UI
2. DTOs are also motivated by their typesafe nature
3. Lacking typesafety, JSON-P JsonObjects are not used as DTOs
4. Most DTOs are exposed as HTTP/JSON
5. Often 3rd party libraries are used to map a DTO into a JSON
6. Sometimes DTOs are called Value Objects
7. The vast majority of DTOs remains identical to entities over the entire lifecycle of a project
8. Copying data between DTOs and entities requires series of getter-setter invocations
9. DTOs are growing, the mapping logic is extracted into dedicated "mappers"
10. DTOs and mappers introduce a significant amount of untested code…
11. Getters and setters, constructor tests are written to increase the code coverage
12. Now: structural JPA entity code changes, affect DTOs, mappers and the corresponding unit test

DTO как носитель данных

Разной структуры

- Java Class
- Records
- Maps

Желательные свойства

- Type safety
- Immutable
- Serializable
- Без циклических ссылок

Java Class - Entity

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

Не всегда работает

- Изменяющийся API
- Проблемы с версионированием API
- Проблемы с LAZY ассоциациями

Когда работает?

- Один клиент API

Java Class - POJO

```java
@Data
public class PetDto {

    private Long id;

    private String name;

    private Integer age;

    private PetTypeDto petType;

    private Set<VisitDto> visits;

}
```

- Просто
- Понятно
  - Наследование/полиморфизм
- Mutable/Immutable
- Boilerplate code
  - Lombok!

Records

```
public record PetDto (
  Long id,
  String name,
  Integer age,
  PetTypeDto petType,
  Set<VisitDto> visits)
implements Serializable {
}
```

- Immutable
- Меньше кода
- Не поддерживают наследование
- Java 14+

A *record class* is a shallowly immutable, transparent carrier for a fixed set of values, called the *record components*.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html

Maps

```
Map<String, Object>
```

- Работает везде
- Гибкие
- Не несут в себе описания нужных данных
- Нет типобезопасности

Виды DTO

	Java 8	Immutable	Type Safe
POJO	+	+/-	+
Entity	+	-	+
Record	-	+	+
Map	+	+/-	-

Этапы жизни DTO

- Создание
  - Наполнение данными (маппинг)
- Сериализация
- …
- Десериализация

Adam Bein o DTO

1. Data Transfer Objects (DTOs) are introduced to decouple (JPA) entities from the UI
2. DTOs are also motivated by their typesafe nature
3. Lacking typesafety, JSON-P JsonObjects are not used as DTOs
4. Most DTOs are exposed as HTTP/JSON
5. Often 3rd party libraries are used to map a DTO into a JSON
6. Sometimes DTOs are called Value Objects
7. The vast majority of DTOs remains identical to entities over the entire lifecycle of a project
8. Copying data between DTOs and entities requires series of getter-setter invocations
9. DTOs are growing, the mapping logic is extracted into dedicated "mappers"
10. DTOs and mappers introduce a significant amount of untested code…
11. Getters and setters, constructor tests are written to increase the code coverage
12. Now: structural JPA entity code changes, affect DTOs, mappers and the corresponding unit test

Пример: Petclinic

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```java
@Entity
public class PetType {
    @Id
    private Long id;

    private String name;
}
```

```java
@Entity
public class Visit {
    @Id
    private Long id;

    private LocalDateTime visitDate;

    private String diagnosis;

    @ManyToOne
    private Pet pet;
}
```

Petclinic: DTO

```
public record PetDto(Long id, String name, Integer age
                     PetTypeDto petType, Set<VisitDto> visits) implements Serializable {
}
```

```
public record PetTypeDto(Long id, String name) implements Serializable {
}
```

```
public record VisitDto(Long id, String diagnosis
                       LocalDateTime visitDate) implements Serializable {
}
```

Маппинг DTO

- Одноименные свойства
- Разноименные свойства
- Вычисляемые свойства
- Мастер-детали
- Flattening

Как создаются DTO

- Вручную
- При помощи мапперов
- Встроенные средства фреймворков

Как рождаются DTO: вручную

- Понятно
- Явно
- Всегда есть занятие

Собственные фреймворки

- Эвристика
- Разметка аннотациями
- Файлы маппинга
- Нуждается в поддержке

Мапперы

Мапперы

MapStruct

- Annotation processor
- Маппер - интерфейс
- Кодогенерация
- Build-time
- Настройка аннотациями
  - И кодом

ModelMapper

- Рефлексия
- Маппер - бин
- Манипуляция с байткодом
- Run-time
- Настройка кодом

Конфигурация “из коробки” - MapStruct

```
@Mapper(unmappedTargetPolicy = ReportingPolicy.IGNORE,
              componentModel = MappingConstants.ComponentModel.SPRING)
public interface PetMapper {

    Pet toEntity(PetDto petDto);

    PetDto toDto(Pet pet);
    ... }
```

```
public record PetDto(Long id, String name,
                     PetTypeDto petType, Set<VisitDto> visits) implements Serializable { }
```

```
public record PetTypeDto(Long id, String name) implements Serializable { }
```

```
public record VisitDto(Long id,
                       LocalDateTime visitDate) implements Serializable { }
```

Конфигурация “из коробки” - ModelMapper

```
@Configuration
public class ModelMapperConfiguration {

    @Bean
    ModelMapper modelMapper() {

        ModelMapper mapper = new ModelMapper();

        return mapper;
    }
}
```

Что есть “из коробки”

	MapStruct	ModelMapper
Одноименные свойства	+	+
Разноименные свойства	-	-
Вычисляемые свойства	-	-
Flattening	-	+
Мастер-детали	+	+

Вычисляемые свойства

```
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name =
"pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```
@Entity
public class PetType {
    @Id
    private Long id;

    private String name;
}
```

```
public record PetFullAgeDto(Long id, String
nameAndAge)
       implements Serializable {
}
```

```
{
  "id": 1,
  "nameAndAge": "Tom cat 2"
}
```

Вычисляемые свойства - MapStruct

Встроенный язык для маппинга

```
@Mapping(target = "nameAndAge",
 expression = "java(\"%s %s %d\".formatted(pet.getName(), pet.getPetType().getName(), pet.getAge()))")
PetFullAgeDto toDto(Pet pet);
```

```
@Override
public PetFullAgeDto toDto(Pet pet) {

    Long id = pet.getId();

    String nameAndAge = "%s %s %d".formatted(pet.getName(), pet.getPetType().getName(), pet.getAge());

    PetFullAgeDto petFullAgeDto = new PetFullAgeDto( id, nameAndAge );

    return petFullAgeDto;
}
```

Вычисляемые свойства - MapStruct

@AfterMapping метод

```
@AfterMapping
default void setCalculatedField(Pet pet, @MappingTarget PetFullAgeDto petDto) {
    petDto.setNameAndAge("%s %s %d".formatted(pet.getName(), pet.getPetType().getName(), pet.getAge()));
}
```

```
@Override
public PetFullAgeDto toDto(Pet pet) {
    PetFullAgeDto petFullAgeDto = new PetFullAgeDto();

    petFullAgeDto.setId( pet.getId() );

    setCalculatedField( pet, petFullAgeDto );

    return petFullAgeDto;
}
```

Вычисляемые свойства - ModelMapper

Embedded Domain Specific Language (EDSL)

- Поддерживает только простые операции
- Для нашего случая не годится

Converter

```
Converter<Pet, String> petToInfoConverter = context -> {
    Pet pet = context.getSource();
    if (pet == null) return null;
    return "%s %s %d".formatted(pet.getName(), pet.getPetType().getName(), pet.getAge());
};

mapper.typeMap(Pet.class, PetFullAgeDto.class).addMappings(new PropertyMap<Pet, PetFullAgeDto>() {
    @Override
    protected void configure() {
        using(petToInfoConverter).map(source).setNameAndAge(null);
    }
});
```

Flattening

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name =
"pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```java
@Entity
public class PetType {
    @Id
    private Long id;

    private String name;
}
```

```java
public record PetFlatDto(Long id, String name,
                                    Long petTypeId,
                                  String petTypeName)
              implements Serializable {
}
```

Flattening - MapStruct

```java
@Mapper(unmappedTargetPolicy = ReportingPolicy.IGNORE,
                componentModel = MappingConstants.ComponentModel.SPRING)
public interface PetFlatMapper {

    @Mapping(source = "petTypeName", target = "petType.name")
    @Mapping(source = "petTypeId", target = "petType.id")
    Pet toEntity(PetFlatDto petFlatDto);

    @InheritInverseConfiguration(name = "toEntity")
    PetFlatDto toDto(Pet pet);

}
```

Flattening

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name =
"pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```java
@Entity
public class PetType {
    @Id
    private Long id;

    private String name;
}
```

```java
public record PetFlatDto(Long id, String name,
                                    Long petTypeId,
                                  String petTypeName)
              implements Serializable {
}
```

POJO DTO + Lombok

Важно для MapStruct

```
<annotationProcessorPaths>
    <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.24</version>
    </path>
    <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok-mapstruct-binding
        </artifactId>
        <version>0.2.0</version>
    </path>
    <path>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-processor</artifactId>
        <version>1.5.3.Final</version>
    </path>
</annotationProcessorPaths>
```

Мапперы

	MapStruct	ModelMapper
Одноименные свойства	+	+
Разноименные свойства	+	+
Вычисляемые свойства	+	+
Flattening	+	+
Мастер-детали	+	+
Простота отладки	☺	⍰
Поддержка Records	+	-
Поддержка Lombok	⍰	☺
Производительность*	☺	😐

* https://www.baeldung.com/java-performance-mapping-frameworks

Проблема эффективности запросов

```java
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    private PetType petType;

    @OneToMany
    @JoinColumn(name = "pet_id")
    private Set<Visit> visits;

}
```

```java
public record PetBaseDto(Long id,
                         String name) {}
```

```java
public record PetDto(Long id, String name,
                     PetTypeDto petType,
                     Set<VisitDto> visits) {}
```

JPA

- Нужны не все поля
- N+1 запрос
- Open in view

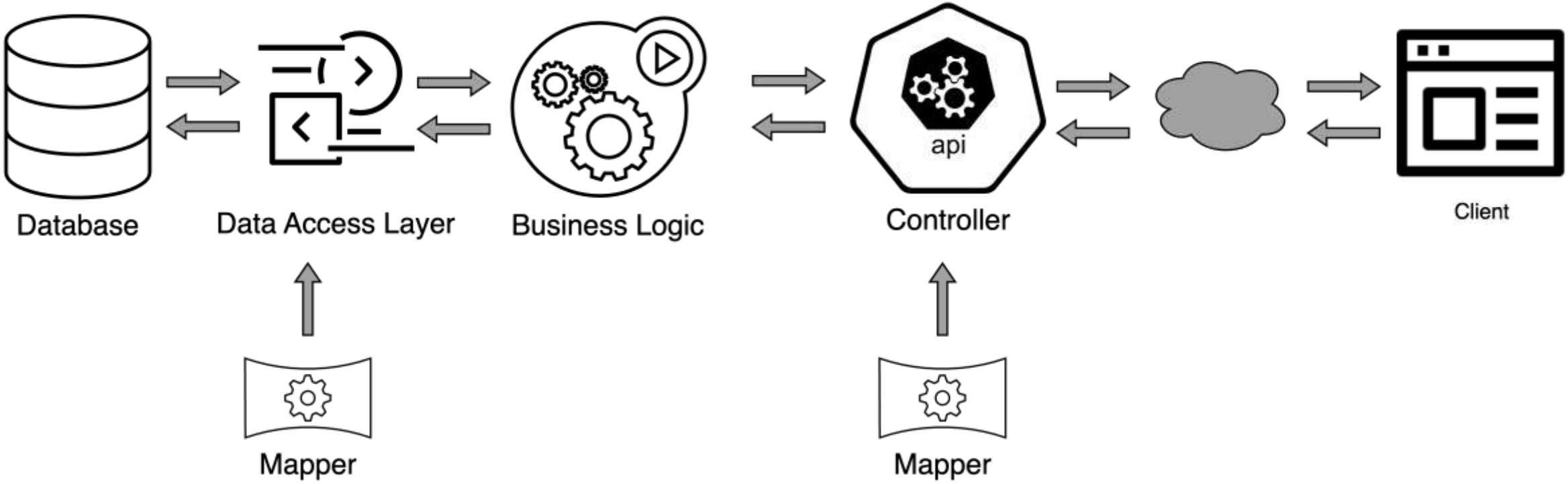
Не JPA

- Запрос на каждый DTO

Adam Bein o DTO

1. Data Transfer Objects (DTOs) are introduced to decouple (JPA) entities from the UI
2. DTOs are also motivated by their typesafe nature
3. Lacking typesafety, JSON-P JsonObjects are not used as DTOs
4. Most DTOs are exposed as HTTP/JSON
5. Often 3rd party libraries are used to map a DTO into a JSON
6. Sometimes DTOs are called Value Objects
7. The vast majority of DTOs remains identical to entities over the entire lifecycle of a project
8. Copying data between DTOs and entities requires series of getter-setter invocations
9. DTOs are growing, the mapping logic is extracted into dedicated "mappers"
10. DTOs and mappers introduce a significant amount of untested code…
11. Getters and setters, constructor tests are written to increase the code coverage
12. Now: structural JPA entity code changes, affect DTOs, mappers and the corresponding unit test

Где используются DTO?



Встроенные средства фреймворков

Напрямую из ORM

Hibernate

```
List<PetFlatDto> list =
session.createQuery("""
select new com.example.pet.dto.PetFlatDto
(p.id, p.name, p.petType.id, p.petType.name)
from Pet p
""", PetFlatDto.class).list();
```

- ResultTransformer (H5)
- TupleTransformer (H6)

```
Query<Map<String, ? extends Serializable>> mapQuery =
session.createQuery("""
select
p.id id, p.name name,
p.petType.id petTypeId, p.petType.name petTypeName
from Pet p
""", Object[].class)
        .setTupleTransformer(((tuple, aliases) ->
                Map.of(aliases[0], (long) tuple[0],
                       aliases[1], (String)tuple[1],
                       aliases[2], (long) tuple[2],
                       aliases[3], (String)tuple[3])
        ));
```

Напрямую из ORM

BlazeDS

```
@Entity
public class Pet {
    @Id
    private Long id;
    private String name;
    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    private PetType petType;

}
```

```
@EntityView(Pet.class)
public interface PetView {

  @IdMapping
  Long getId();

  @Mapping("CONCAT(name, ' ', age)")
  String getNameAndAge();
}
```

```
CriteriaBuilder<PetView> cb =
criteriaBuilderFactory.create(entityManager, PetView.class);

CriteriaBuilder<PetView> petViewBuilder =
evm.applySetting(EntityViewSetting.create(PetView.class), cb);

List<PetView> petViews = petViewBuilder.getResultList();
```

Spring Data JPA Репозитории

Проекции

```java
public interface PetInfo {
    Long getId();

    String getName();

    PetTypeInfo getPetType();

    Set<VisitInfo> getVisits();
}
```

```java
public interface VisitInfo {
    Long getId();

    LocalDateTime getVisitDate();
}
```

```java
public interface PetFullAgeInfo {
    Long getId();
    String getName();

@Value("#{target.getName + ' '
        + target.getPetType.getName + ' '
        +target.getAge}")
    String getNameAndAgeSpel();
}
```

```java
public interface PetFullAgeInfo {
    Long getId();
    String getName();
    Integer getAge();
    PetType getPetType();

    default String getNameAndAge() {
        return "%s %s %d".formatted(
          getName(), getPetType().getName(), getAge());
    }
}
```

HAULMONT

Spring Data JPA Репозитории

Проекции

```
public interface PetInfo {
    Long getId();

    String getName();

    PetTypeInfo getPetType();

    Set<VisitInfo> getVisits();
}
```

```
public interface VisitInfo {
    Long getId();

    LocalDateTime getVisitDate();
}
```

Проблемы

- Выбираются все поля
- N+1 запрос

Как решать

- Entity Graph

Spring Data Репозитории

Проблема

- Проекции с разным набором атибутов
- Один метод для извлечения данных

```
List<PetInfo> findByAgeGreaterThan(Integer age);

List<PetFullAgeInfo> findPetsByAgeGreaterThan(Integer age);
```

Решение

- Динамические методы

```
<T> List<T> findByAgeGreaterThan(Integer age, Class<T> type);
```

```
petRepository.findAllPetsByAgeGreaterThan(age, PetFullAgeInfo.class)
```

Как рождаются DTO

DIY фреймворки

- Простые/уникальные маппинги
- Нужно поддерживать

Отдельные мапперы

- Стабильные решения
- Поддержка все равно требуется
- Неэффективная выборка данных

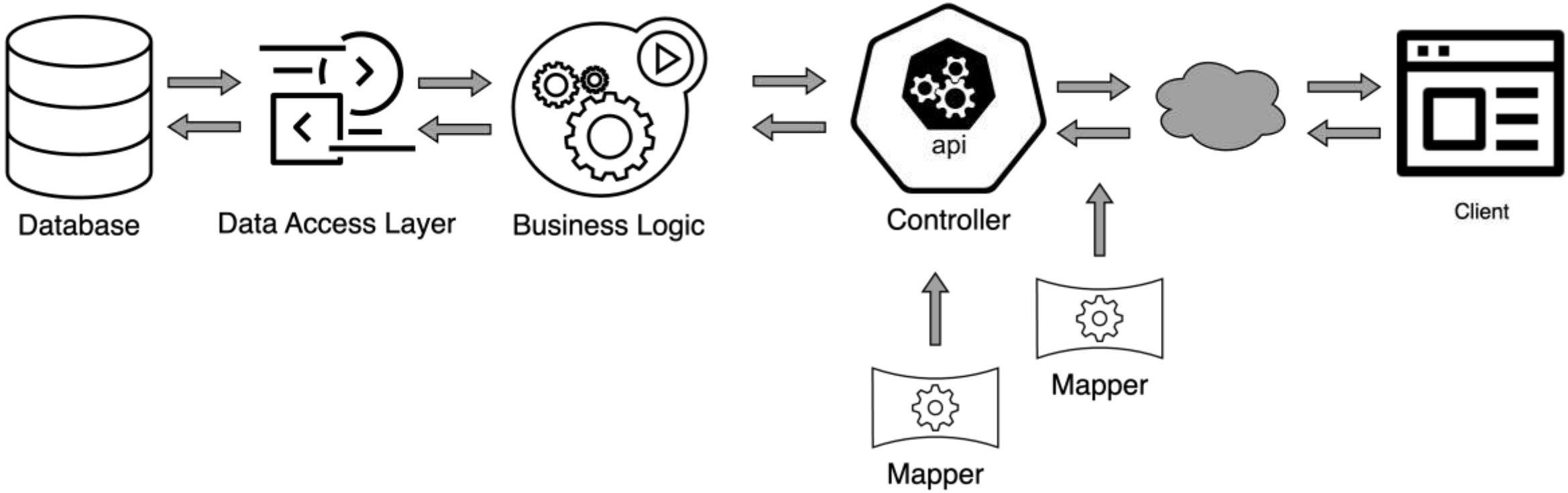
"Встроенные" мапперы

- Запросы эффективнее
- Код маппинга "размазан" по приложению
- Не всегда типобезопасно

Этапы жизни DTO

- Создание
  - Наполнение данными
- Сериализация
- ...
- Десериализация

Где используются DTO?



Во что сериализуем?

JSON

- Jackson
- Jersey
- Gson

XML

- Jackson
- JAXB

Сокрытие данных

Проблема:

- DTO
- Мапперы
- Это лишний код!

Решение – разметка классов

- @XmlTransient
- @JsonIgnoreProperties
- @JsonIgnore
- @JsonView

JsonView

```java
@Entity
@Table(name = "pet")
public class Pet {
    @Id
    @Column(name = "id", nullable = false)
    @JsonView(Views.Public.class)
    private Long id;

    @Column(name = "name")
    @JsonView(Views.Public.class)
    private String name;

    @Column(name = "age")
    @JsonView(Views.Admin.class)
    private Integer age;

    @ManyToOne
    @JoinColumn(name = "pet_type_id")
    @JsonView(Views.Public.class)
    private PetType petType;

    @OneToMany(orphanRemoval = true)
    @JoinColumn(name = "pet_id")
    @JsonView(Views.Public.class)
    private Set<Visit> visits = new LinkedHashSet<>(); }
```

```java
public class Views {
public static class Public {}
public static class Admin extends Public
{}
}
```

```java
@GetMapping("/jsonview")
@JsonView(Views.Public.class)
public List<Pet> findAllJsonView() {
    return petService.findAll();
}
```

JsonView – проблемы

JPA

- LazyInit
- N+1 запрос

Для всех

- Вытаскиваем все данные
- Версионирование API
- Комбинаторный взрыв

GraphQL

- DTO не нужны?
- Схема описывает возвращаемые типы
- Мы сами собираем себе DTO

```
type Owner {
  address: String
  city: String
  email: String
  firstName: String
  id: ID
  lastName: String
  telephone: String
}
```

```
{
  Owner {
    id,
    firstName,
    lastName,
    email
  }
}
```

GraphQL - недоработки

```
type Owner {
  address: String
  city: String
  email: String
  firstName: String
  id: ID
  lastName: String
  telephone: String
}
```

Постраничная выборка

- Отдельный тип данных

Эффективность запросов

- Выбираются все поля
- N+1 запрос

Безопасность

- Сложнее скрывать поля

JSON Relational Duality View

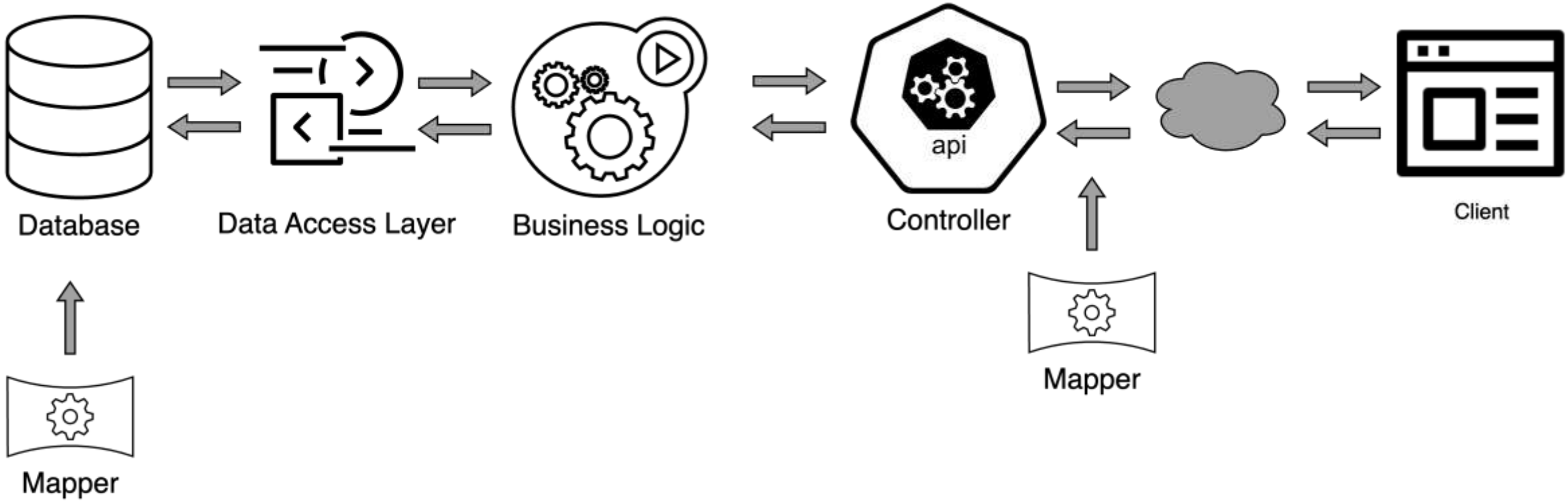
Lukas Eder
@lukaseder

Well folks, I have only been playing with Oracle 23c JSON DUALITY views for a few minutes, but holy shit! 🤯

oracle-base.com/articles/23c/j...

That indeed is an ORM killer

Где используются DTO?



JSON Relational Duality View

```
create json relational duality view pets_dv as
select json {'petId'      : p.id,
             'petName'    : p.name,
             'petAge'     : p.age,
             'visits' :
               [ select json {'visitId'      : v.id,
                              'visitDate'    : v.visitDate,
                              'diagnosis'    : v.diagnosis}
                   FROM visits v with insert update delete
                   WHERE v.petId = p.id ]}
from pets p with insert update delete;
```

```
select json_serialize(p.data pretty) from pets_dv p;
```


```
{
    "_metadata" :
        { "etag" :
"E546E2220E8F9620E36C2A7F8858D6F7",
        "asof" : "00000000001FA9FA" },
    ”petId": 1,
    ”petName": "Tom",
    ”petAge": 2,
    "visits": [
      {
        ”visitId": 1,
        "visitDate": "2023-04-
09T23:00:29",
        "diagnosis": "Illness"
      }
    ]
  }
```

JSON Relational Duality View

```
create json relational duality view pets_dv as
select json {'petId'     : p.id,
             'petName'   : p.name,
             'petAge'    : p.age,
             'visits' :
               [ select json {'visitId'     : v.id,
                              'visitDate'   : v.visitDa
                              'diagnosis'   : v.diagnos
                  FROM visits v with insert update del
                  WHERE v.petId = p.id ]}
from pets p with insert update delete;
```

```
insert into pets_dv p (data)
values ('
{
  "petId" : 12,
  "petName" : "Tommy",
  "petAge" : "4",
  "visits" : [
    {
      "visitId" : 15,
      "visitDate" : "2023-04-12T09:00:00",
      "diagnosis" : "Rabies"
    }
  ]
}');
```

JSON Relational Duality View

- Нужны ли теперь DTO?
- Нужен ли ORM?
- Пока непонятно
  - Привязка к структуре данных
  - Версионирование
  - Скорость выборки
  - Скорость вставки данных

Этапы жизни DTO

- Создание
  - Наполнение данными
- Сериализация
- ...
- Десериализация

Валидация

```
@Data
@NoArgsConstructor
public final class PetDto {
    private Long id;
    @Size(min = 4, max = 16)
    private String name;
    private PetTypeDto petType;
    private Set<VisitDto> visits;
}
```

- Response DTO
- Request DTO
- Синхронизация валидаторов
  - Первая версия API
  - Вторая?

Преобразование в сущности

Частичное обновление

```
@BeanMapping(nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.IGNORE)
Pet partialUpdate(PetDto petDto, @MappingTarget Pet pet);
```

```
@Override
public Pet partialUpdate(PetDto petDto, Pet pet) {
    if ( petDto == null ) {
        return pet;
    }

    if ( petDto.id() != null ) {
        pet.setId( petDto.id() );
    }
    if ( petDto.name() != null ) {
        pet.setName( petDto.name() );
    }
...
```

Преобразование в сущности

Обратные ссылки

```
@Entity
public class Pet {
    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "pet", orphanRemoval = true)
    private Set<Visit> visits = new LinkedHashSet<>();}
```

```
@Entity
public class Visit {
    @Id
    private Long id;

    private LocalDateTime visitDate;

    @ManyToOne
    @JoinColumn(name = "pet_id")
    private Pet pet;}
```

```
@Mapper
public interface PetMapper {
    Pet toEntity(PetDto petDto);

    PetDto toDto(Pet pet);

    @AfterMapping
    default void linkVisits(@MappingTarget Pet pet) {
        pet.getVisits().forEach(visit -> visit.setPet(pet));}}
```

```
@Override
public Pet toEntity(PetDto petDto) {

    Pet pet = new Pet();
    pet.setId( petDto.getId() );
    pet.setVisits(
      visitDtoSetToVisitSet(petDto.getVisits()));
    linkVisits( pet );

    return pet;}
```

Итого

DTO – часть API

- Может пережить приложение
- Может быть нескольких версий

Задача DTO

- Быть сериализованным
- Decoupling

Маппинг – почти неизбежен

Проблема эффективности запросов

- N+1
- Проекции
- Отдельные запросы
- Планируйте заранее архитектуру доступа к данным


ЖЫЗНЬ с ДТО


ЖЫЗНЬ БЕЗ ДТО