

Data Access Layer

Матвей Попов

План

- Что такое DAL?
- Проблематика DAL
- Как с этими данными можно взаимодействовать?
- Какие инструменты и подходы существуют?
- Livecoding

Что такое Data Access Layer?

DAL - Data access layer

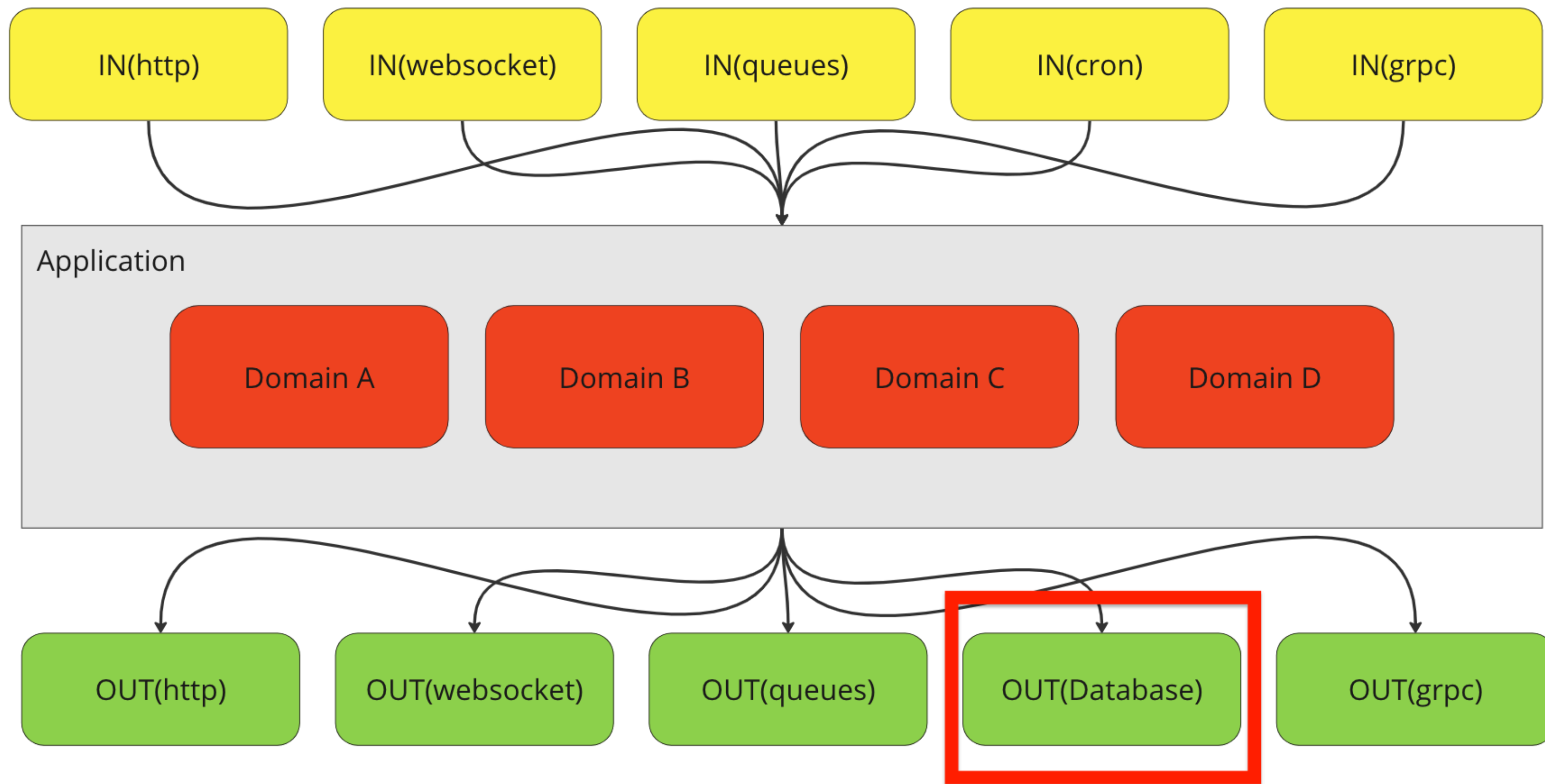
Это **слой** компьютерной программы, который предоставляет упрощенный **доступ** к данным, хранимым в постоянном **хранилище** какого-либо типа.

Другими словами, это:

- Более высокий уровень абстракции
- Соккрытие сложности реализации хранилища

**Разберемся глубже с
проблематикой**

Про что именно мы говорим?

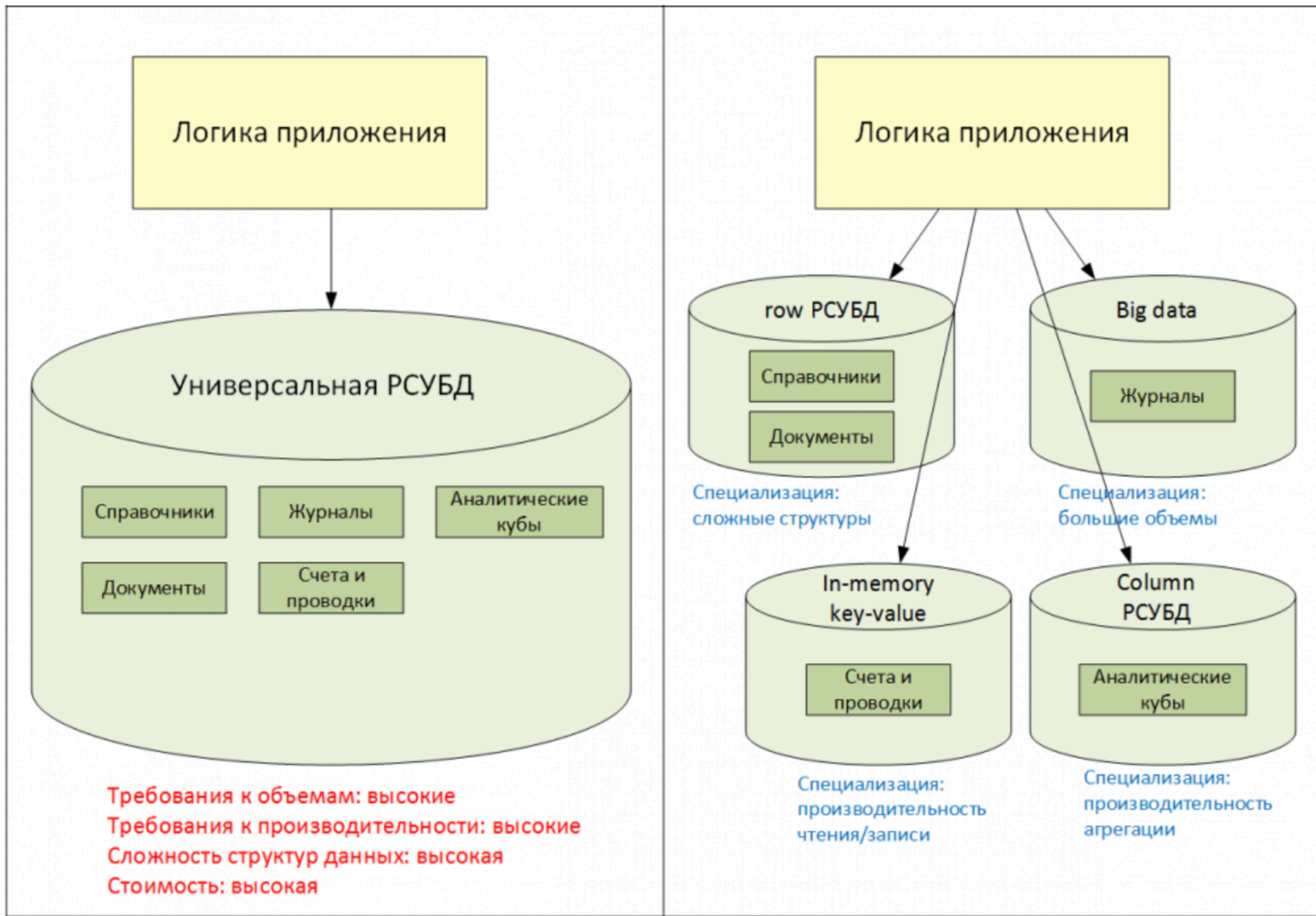


Для чего нам нужны данные?

**Для реализации бизнес
процессов**

А где эти данные могут быть?

- В одной специализированной БД
- В разных специализированных БД
- В разных БД с разной специализацией



**Как с данными можно
взаимодействовать?**

- Писать свой некоторый клиент к каждой базе данных
- Использовать как можно больше хранимых процедур
- Использовать специализированные SDK баз
- Использовать ORM
- Использовать Query DSL

**Как с данными нужно
взаимодействовать?**

**Необходимо максимально
абстрагироваться от конкретной БД**

Почему?

- Ценовые или иные политики поставщиков хранилищ данных меняются
- С ростом сервиса существующие решения могут переставать удовлетворять требованиям
- Технологии хранения данных развиваются и предоставляют новые средства
- Появляются Open Source проекты, которые предоставляют бесплатные альтернативы

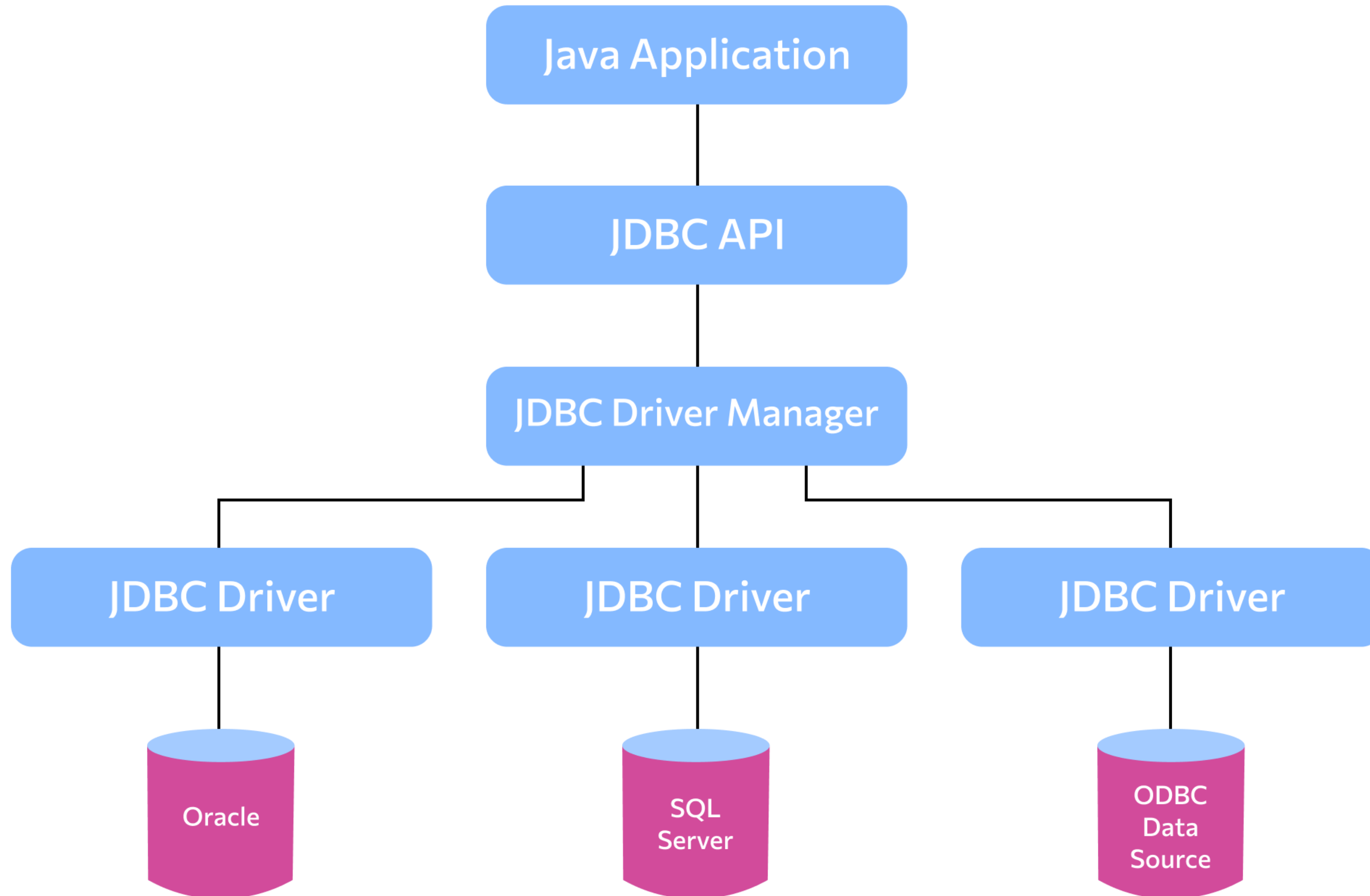
**Какие инструменты
существуют?**

**Будем идти от “старого” к
“новому”**

JDBC - Java Database Connectivity API

- Используют абсолютно все библиотеки
- Java реализация стандартного подхода к работе с базами данных
- Доступна в JDK
- Использует драйвера
- Существует с 1997 года

JDBC Architecture



Преимущества и недостатки

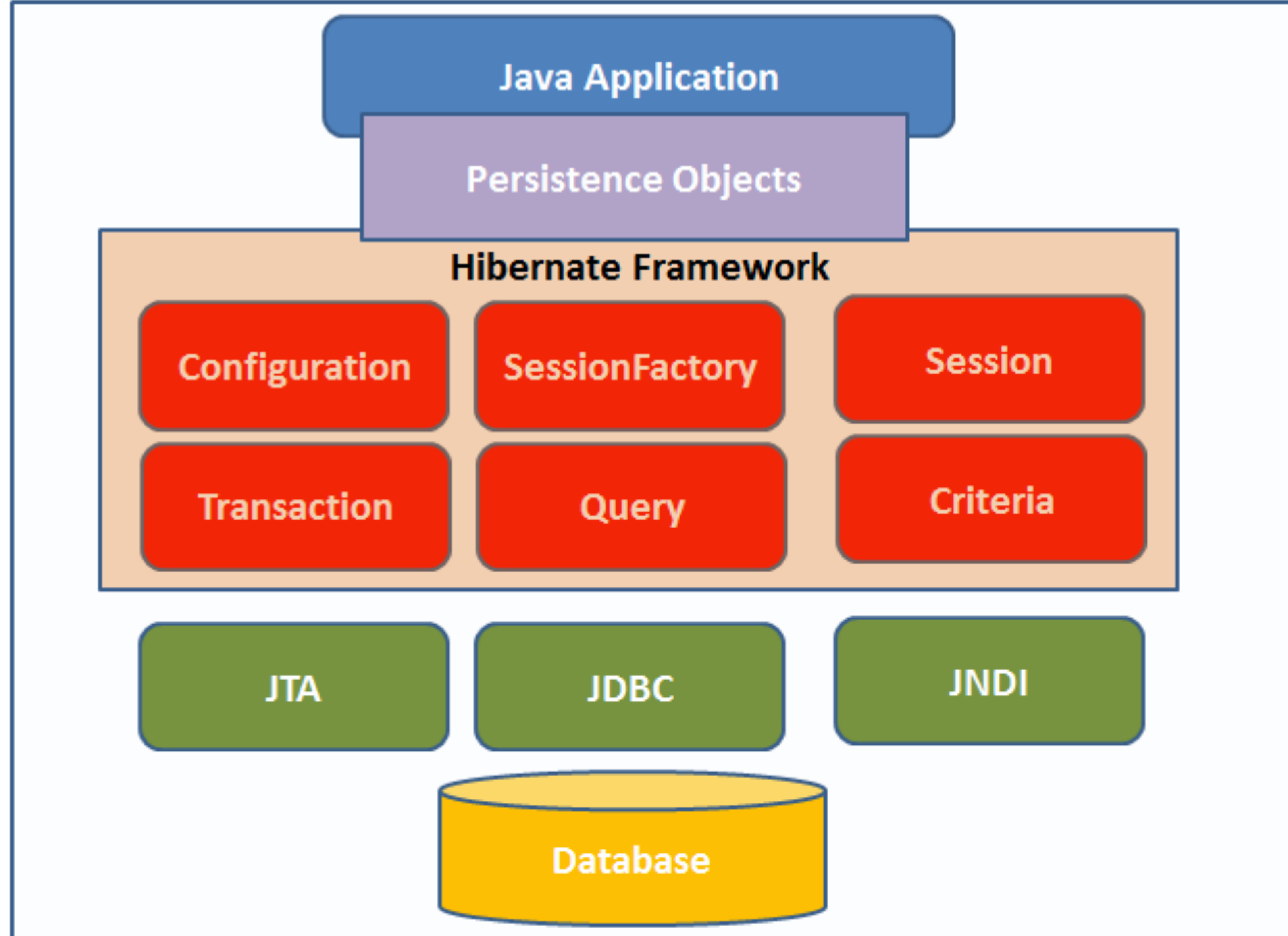
- Гибкий и проверенный временем подход
- Не зависим от реляционной базы данных
- Однако зависим от диалекта SQL
- Не очень удобно соотносить объекты и схемы в таблицах
- Необходимо писать избыточно много кода

**Необходимо сильнее
унифицировать подход**

Hibernate и JPA

- Hibernate - реализация ORM модели
- Унифицированный подход
- Использует аннотации для настройки
- Очень много функционала

Hibernate Architecture



```

1  package com.ferum_bot.quotesapi.models.entity
2
3  import ...
4
5
6
7
8  Ferum-bot
9  @Entity
10
11  data class QuoteEntity(
12
13      @Id
14      @GeneratedValue(strategy = GenerationType.AUTO)
15      @Column(name = "id")
16      var id: Long? = null,
17
18      @Column(name = "text", length = 5000)
19      var text: String,
20
21      @ManyToOne
22      @JoinColumn(name = "author_entity_id", referencedColumnName = "id")
23      var author: AuthorEntity? = null,
24
25      @ManyToOne
26      @JoinColumn(name = "tag_entity_id", referencedColumnName = "id")
27      var tag: TagEntity? = null,
28
29      @JsonFormat(pattern = "dd.MM.yyyy HH:mm:ss")
30      var createdAt: LocalDateTime? = null,
31
32      @JsonFormat(pattern = "dd.MM.yyyy HH:mm:ss")
33      var updatedAt: LocalDateTime? = null,
34  ) {
35
36  Ferum-bot
37  override fun equals(other: Any?): Boolean {
38      if (this === other) return true
39      if (other == null || Hibernate.getClass(proxy: this) != Hibernate.getClass(other)) {
40          return false
41      }
42      other as QuoteEntity
43
44      return id != null && id == other.id
45  }
46
47  Ferum-bot
48  override fun hashCode(): Int = 916666166
49
50

```

А что же такое JPA?

JPA - Java Persistence API

- Hibernate был настолько крут, что его решили стандартизировать
- JPA - это спецификация
- Hibernate - имплементация JPA

Плюсы и минусы

- JPA очень продуман и очень мощный
- Предоставляет гигантский API
- Сам генерирует SQL запросы
- Очень просто организовывать DataLayer в приложении
- Очень тяжело контролировать генерируемые запросы
- Объектная модель не всегда соответствует реляционной

**Может есть какой-то
компромиссный подход?**

QueryBuilder

- Библиотека все еще валидирует запросы
- Пользователь сам контролирует получаемые SQL
- Все еще унифицировано
- Нужно писать заметно больше кода

- Хорошо абстрагируемся от конкретной технологии

JOOQ

```
31  override fun updateProfileEntityAccess(access: ProfileEntityAccess): ProfileEntityAccess {
32      val authorEntityAccessValue = accessTransformer.transformToDatabaseValue(access.data.access.accessForAuthor)
33      val responsibleEntityAccessValue = accessTransformer.transformToDatabaseValue(access.data.access.accessForResponsible)
34      val followerEntityAccessValue = accessTransformer.transformToDatabaseValue(access.data.access.accessForFollower)
35      val noRoleEntityAccessValue = accessTransformer.transformToDatabaseValue(access.data.access.accessForNoRole)
36
37      jooqConfiguration.dsl().transaction { cfg ->
38          cfg.dsl() DSLContext
39              .insertInto(
40                  PROFILE_ENTITY_ACCESS,
41                  PROFILE_ENTITY_ACCESS.ID,
42                  PROFILE_ENTITY_ACCESS.ENTITY_META_ID,
43                  PROFILE_ENTITY_ACCESS.SECURITY_PROFILE_ID,
44                  PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_AUTHOR,
45                  PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_RESPONSIBLE,
46                  PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_FOLLOWER,
47                  PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_NO_ROLE
48              ) InsertValuesStep7<ProfileEntityAccessRecord!, UUID?, UUID?, UUID?, Byte?, Byte?, Byte?, Byte?>
49              .values(
50                  access.id,
51                  access.data.entityMetaId,
52                  access.data.securityProfileId,
53                  authorEntityAccessValue,
54                  responsibleEntityAccessValue,
55                  followerEntityAccessValue,
56                  noRoleEntityAccessValue
57              )
58              .onDuplicateKeyUpdate() InsertOnDuplicateSetStep<ProfileEntityAccessRecord!>
59              .set(PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_AUTHOR, authorEntityAccessValue) InsertOnDuplicateSetMoreStep<ProfileEntityAccessRecord!>
60              .set(PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_RESPONSIBLE, responsibleEntityAccessValue)
61              .set(PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_FOLLOWER, followerEntityAccessValue)
62              .set(PROFILE_ENTITY_ACCESS.ACCESS_VALUE_FOR_NO_ROLE, noRoleEntityAccessValue)
63              .execute()
64          }
65
66      return access
67  }
68
```


JPA

```
67  fx  Ferum-bot
68      override fun createNewQuote(quote: CreateQuoteModel): QuoteEntity {
69          val tagEntity = tagsDataSource.findById(quote.tagId).get()
70          val authorEntity = authorsDataSource.findById(quote.authorId).get()
71          val quoteEntity = QuoteEntity(
72              text = quote.text,
73              author = authorEntity,
74              tag = tagEntity,
75          )
76          val resultEntity = quotesDataSource.saveAndFlush(quoteEntity)
77          updateTag(tagEntity, resultEntity)
78          updateAuthor(authorEntity, resultEntity)
79
80          return resultEntity
81      }
82
```

А что будем использовать мы?

EXPOSED



- Написан на Kotlin для Kotlin приложений
- Представляет облегченную ORM
- Можно использовать в виде DSL в SQL
- Поддержка Coroutine из коробки
- Использует JDBC под капотом
- Многомодульная архитектура

Примеры

Описание таблицы

```
8
  Ferum-bot
9  object PostsTable: LongIdTable( name: "posts") {
10
11    val name = text( name: "name", eagerLoading = true)
12
13    val text = text( name: "text", eagerLoading = true)
14
15    val isVisible = bool( name: "is_visible").default( defaultValue: true)
16
17    val isDeleted = bool( name: "is_deleted").default( defaultValue: false)
18
19    val isEdited = bool( name: "is_edited").default( defaultValue: false)
20
21    val author = reference(
22      name = "author_user_id", foreign = UserProfilesTable,
23      onDelete = NO_ACTION, onUpdate = NO_ACTION
24    )
25
26    // val settings = reference(
27    //   name = "post_settings_id", foreign = PostsSettingsTable,
28    //   onDelete = NO_ACTION, onUpdate = NO_ACTION
29    // )
30
31    val createdAt = datetime( name: "created_date").clientDefault { localDateTimeNow() }
32
33    val updatedAt = datetime( name: "updated_date").clientDefault { localDateTimeNow() }
34  }
```

Описание сущности

```
14
    Ferum-bot
15 class Post(id: EntityID<Long>): LongEntity(id) {
    Ferum-bot
16     companion object: CallbackLongEntityClass<Post>(PostsTable)
17
18     var name by PostsTable.name
19
20     var text by PostsTable.text
21
22     var author by PostsTable.author
23
24     val comments by PostComment referrersOn PostCommentsTable.post
25
26     val reactions by PostReaction referrersOn PostReactionsTable.post
27
28     var tags by PostTag via PostToTagTable
29
30     var isVisible by PostsTable.isVisible
31
32     var isDeleted by PostsTable.isDeleted
33
34     var isEdited by PostsTable.isEdited
35
36     var createdDate by PostsTable.createdDate
37
38     var updatedAt by PostsTable.updatedDate
39
```

Пример запроса

```
42  fx  Ferum-bot
43      override fun getPostsWithAuthorIds(
44          authorIds: Collection<EntityID<Long>>, pageNumber: Int, pageSize: Int
45      ): List<Post> {
46          val offset = (pageNumber - 1).toLong() * pageSize
47          val query = PostsTable
48              .select {
49                  PostsTable.author inList authorIds
50              }
51          .orderBy(PostsTable.createdDate, SortOrder.DESC_NULLS_LAST)
52          .limit(pageSize, offset)
53          val result = Post.wrapRows(query)
54          return result.toList()
55      }
```

Ferum-bot

Плюсы и Минусы

- Лаконичен и легковесен
- Использует весь функционал Kotlin
- Можно использовать как ORM так и QueryBuilder
- Поддерживает очень мало баз данных
- Очень молодая библиотека
- Небольшое сообщество
- Очень активно развивается

А теперь подключим Exposed
в наш проект

Livcoding: Добавляем хранение данных

Выводы

Kotlin - это все еще очень круто!

Домашнее задание

**Нужно добавить хранение
данных в ваше приложение**

Материалы

Будут в README

Спасибо за внимание!