УДК 004.415.2.043; 004.415.22; 004.415.23; 004.656
DOI: 10.25559/SITITO.16.202002.416-425

# Унифицированная модель данных и её применение в микросервисной архитектуре

**А. И. Балес**
ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Россия
119991, Россия, г. Москва, ГСП-1, Ленинские горы, д. 1
aleks_bales@mail.ru

## Аннотация

В настоящее время широкое распространение получила микросервисная архитектура, появляется множество приложений (микросервисов), которые так или иначе между собой взаимодействуют. Микросервисы, как правило, объединяются в целые платформы. Встает острый вопрос взаимодействия их друг с другом, хранения и обмена справочными данными. Эталонного решения данной проблемы не существует, на данный момент, но в данной работе описан симбиоз различных подходов, а также приведена конкретная реализация – унифицированная модель справочников в микросервисной платформе, которая позволяет обмениваться справочными данными десяткам микросервисов (на момент написания статьи количество сервисов в платформе составляет 57 штук). Была спроектирована метамодель, с помощью которой описываются сущности справочников: справочники, категории, связи между справочниками и между сущностями справочников. Архитектура такой модели микросервиса справочников позволяет абстрагироваться от конкретных реализаций СУБД, таким образом не происходит «сильной связности» с SQL или NoSQL базой данных. Данный микросервис имеет искусственно введеные ограничения запись данных от сторонних микросервисов. Миграция данных осуществляется вручную посредством Liquibase скриптов, что позволяет поддерживать справочники в одинаковом состоянии на всех стендах разработки, быстро и легко восстанавливать до любого состояния слепки справочников. Дальнейшее развитие микросервиса справочников может иметь следующий вид: разработка UI форм для ручного заполнения справочников и сохранение новых данных в модели или в скрипты Liquibase, что уже на данном этапе является очень актуальной проблемой для группы сопровождения и администраторов микросервисной платформы.

**Ключевые слова:** микросервисная архитектура, управление данными, унифицированная модель данных, справочники.

**Для цитирования:** Балес, А. И. Унифицированная модель данных и её применение в микросервисной архитектуре / А. И. Балес. – DOI 10.25559/SITITO.16.202002.416-425 // Современные информационные технологии и ИТ-образование. – 2020. – Т. 16, № 2. – С. 416-425.

# Unified Data Model and its Application in Microservice Architecture

**A. I. Bales**
Lomonosov Moscow State University, Moscow, Russia
1 Leninskie gory, Moscow 119991, Russia
aleks_bales@mail.ru

## Abstract

Today, the microservice architecture has become widespread and most popular in software engineering; many applications (microservices) are forced to communicate with each other. In the future, microservices are combined into entire application platforms. The acute issue of interaction of microservices with each other, storage and exchange of dictionary data have appeared in the minds of developers. A standard solution to this problem does not exist at the moment, but this article describes the symbiosis of various approaches, and also provides a specific implementation - a unified data model of dictionary in a microservice platform. It allows dozens of microservices to exchange dictionary data (now, the number of services in the platform is 57 pieces). This article describes the metamodel which help to design and create entities of dictionary, there are: category of dictionary, relations between categories and between directories. Such data model architecture of microservice allowed me to abstract from specific DBMS implementations, i.e. I am not hard tied on SQL or NoSQL database. This microservice has artificially introduced restrictions on data recording by other microservices. Data migration is carried out manually by Liquibase scripts, which allows maintaining dictionaries in the same state at all development stands, quickly and easily restore dictionaries to any state. Further evolution of the dictionary microservice could be as follows: the development of a user interface for filling dictionaries by administrator`s hands and saving new data in unify data models or auto generation Liquibase scripts. At this stage this problem is very urgent for the support group and administrators of the microservice platform.

**Keywords:** microservice architecture, data management patterns, unify data model, dictionary, canonical data model.

**For citation:** Bales A.I. Unified Data Model and its Application in Microservice Architecture. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2020; 16(2):416-425. DOI: https://doi.org/10.25559/SITITO.16.202002.416-425

## Введение

В настоящее время очень популярен подход к разработке промышленного ПО, основанный на выстраивании микросервисной архитектуры [9-10, 13-14]. Несмотря на огромную популярность, он имеет и свои трудности [12]. Одна из таких проблем – хранение и обмен справочными данными [10-11, 13, 18]. Зачастую многим микросервисам необходимо получать одну и ту же информацию, например, справочные данные, которые можно отнести к категории изменяемости как перманентные. Во избежание дублирования, в каждой схеме базы данных соответствующего микросервиса можно вынести перманентные или редко модифицируемые данные в отдельный сервис справочников. Для объединения моделей данных из различных микросервисов необходимо выстроить унифицированную модель-схему, которая позволит увеличить коэффициент переиспользования данных, расширять новые модели (в ООП такой принцип называется – наследование).

При попытке унифицировать взаимодействие с данными, стоит быть осторожным, дабы не перегрузить концептуальную схему базы данных и методы взаимодействия с ними. Для этого необходимо соблюдать границы микросервиса *(bounded context*[1]*)* [1], [15]. При построении модели микросервисов можно прибегнуть к современным практикам таким как *DDD (domain-driven design)*[2] [2], [15], она позволяет правильно с точки зрения логики выделить структуру, границы и функциональность микросервиса. Благодаря онтологии можно выстроить достаточно мощный механизм описания данных (метаданные), связей между данными, также появляется возможность абстрагироваться от способа хранения данных, конкретной реализации базы данных (SQL или NoSQL). Спроектированная модель будет достаточно легко проецироваться на реляционные таблицы (SQL), а также на NoSQL СУБД.

## Микросервисы

В открытых источниках достаточно часто описываются современные подходы и практики к разработке программных систем (enterprise systems). Не каждую архитектуру системы можно отнести к микросервисной. Мартин Фаулер – один из основателей данного подхода, не дает точного описания что такое “микросервисная архитектура”, но сформулировал критерии, соответствующие микросервисной архитектуре[3] [3]. Для того чтобы описать все преимущества, необходимо понимать недостатки прошлых практик к разработке крупных коммерческих систем. Одна из самых распространенных практик, а вернее, архитектура имеет свое название – “монолитная”. В силу того, что все компоненты, модули имеют сильную связность между собой, и чтобы исправить или провести рефакторинг потребуется значительно количество ресурсов. Микросервисы – это не про маленькие монолитные системы, а про такой вид систем, которые можно переписать за две недели. Фаулер не считает микросервисную архитектуру панацеей, но приводит примеры, когда её использование ускоряло процесс разработки и доставки конечного продукта пользователям, что согласно практикам *Agile* является самым ценным в гибких методологиях разработки. Каждый из микросервисов имеет свой *Bounded Context* – бизнес модель *(domain model)*, которая является ядром сервиса, согласно одному из подходу к разработке микросервисной архитектуры – Domain Driven Design (*DDD*) [15]. Данное свойство сложно поддерживать в монолитных системах, так как между модулями установлена достаточно сильная связность, как физически, так и логически. Зачастую, при разбиении монолитного приложения на части, команды разработчиков сталкиваются с проблемой под названием закон Конвея[4] [4], когда структура команды выстроилась некорректно, и из-за этого структура/слои приложения сформировались также неправильно. Возникает множество проблем и ограничений при переходе от монолитной структуры к микросервисной [5], но появляются и свои плюсы:

- независимая разработка и установка (deploy);
- изолированность (данных, кода, ошибок);
- масштабируемость;
- логическое разбиение на компоненты/бизнес функционал позволяет легче воспринимать/читать исходный код, производить рефакторинг;
- достаточно соблюдать контракты (contracts)[5] [6] взаимодействия сервисов, а какой язык программирования или СУБД выбрать, решает команда разработчиков; получается некая гетерогенная система.

## Data Management Patterns

Согласно правилу изолированности, каждый микросервис должен скрывать свои данные и любое взаимодействие с ними напрямую к базе данных, т.е. другим сервисам запрещено напрямую осуществлять запись в таблицы базы данных других сервисов, но иногда делают послабление, предоставляя права на извлечение данных [7, С. 12].

Существует 6 основных подходов управления данными в микросервисах[6] [8], [16]:

- *Database-per-Service Pattern*
- *Shared Database Pattern*
- *Saga Pattern*
- *API Composition Pattern*
- *Event Sourcing Pattern*
- *CQRS Pattern*

### *A.* ***Database-per-Service***

*Database-per-Service* предполагает, что для каждого микросер-

[1] Fowler M. Bounded Context. 15 January 2014 [Электронный ресурс]. – URL: https://martinfowler.com/bliki/BoundedContext.html (дата обращения: 05.07.2020).

[2] Fowler M. Domain Driven Design. 22 April 2020 [Электронный ресурс]. – URL: https://martinfowler.com/bliki/DomainDrivenDesign.html (дата обращения: 05.07.2020).

[3] Lewis J. Microservices. 25 March 2014 [Электронный ресурс]. – URL: https://martinfowler.com/articles/microservices.html (дата обращения: 05.07.2020).

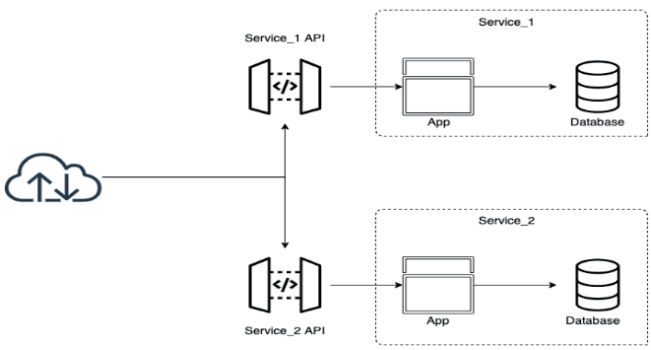
[4] Conway’s Law – A Theoretical Basis for the Microservice Architecture // Alibaba Cloud. – Dec 13, 2017. – [Электронный ресурс]. – URL: https://medium.com/@Alibaba_Cloud/conways-law-a-theoretical-basis-for-the-microservice-architecture-c666f7fcc66a (дата обращения: 05.07.2020).

[5] RESTful Contracts [Электронный ресурс]. – URL: https://www.signom.com/api/rest/docs (дата обращения: 05.07.2020).

[6] Microservices. Data managment [Электронный ресурс]. – URL: https://microservices.io/patterns/index.html#data-management (дата обращения: 05.07.2020).

виса выделяется отдельная база данных, в которой хранятся все модели[7] [16-17, 21]. Получение, обновление, добавление новых данных осуществляется посредством API (см. рис. 1). Основным преимуществом данного подхода является изолированность данных, т.е. только наш сервис может оперировать данными, относящимися к нему. Но в то же время, этот «плюс» является и жирным минусом, так как зачастую микросервисам необходимо выполнять объединение данных из других моделей (других микросервисов), и, чтобы это осуществить, приходится «вытаскивать» данные через API каждого сервиса, сохранять в буфере (оперативной памяти), а в API нашего сервиса производить объединение данных. Стоит также отметить, что не всегда под отдельный микросервис создается отдельная база, в целях экономии ресурсов можно использовать *Schema-per-Service*, когда в рамках одной базы данных выделяется схема пользователя, совпадающая, например, с именем микросервиса, которой раздаются отдельные права на чтение, запись, редактирование (grants).

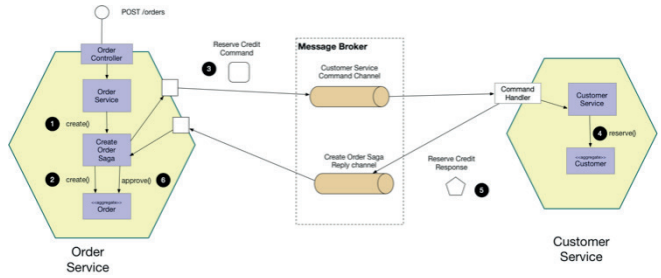
К одному из основных минусов данного подхода можно отнести трудности при объединении моделей из разных микросервисов.




F i g. 1. Database-per-Service pattern[8]

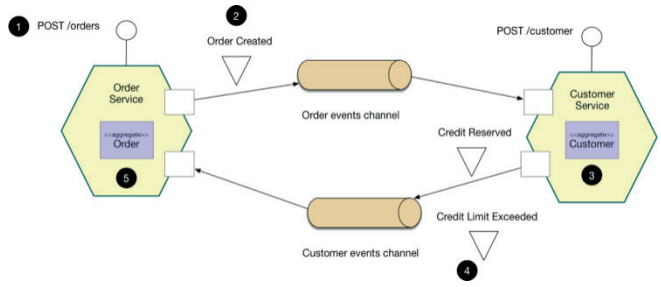
### *B.* ***Shared database***

*Shared* заказов (*Order Service*) и сервис покупатель (*Customer Service*). При создании заказа необходимо обновить информацию о заказе у покупателя. Create Order Saga в данной схеме является тем самым оркестратором, который создает событие о том, что заказ создан и кладет его в message broker, а Customer Service должен перехватить (handle) это сообщение и ответить аналогичным образом – сформировав сообщение для Order Service. После того как оркестратор получит обратное сообщение, он может считать транзакцию успешной и сделать commit локальной транзакции.




F i g. 2. Orchestrator-based Saga pattern [7, p. 122]

Принцип работы хореографа схож тем, что сервисы «подписываются» на нужные им события (сообщения). И исходя из тех или иных событий выполняют определенные действия. Аналогичный пример работы электронного магазина по схеме хореографа (рис. 5):




F i g. 3. Choreography-based Saga pattern [7, p. 118]

*Saga* позволяет рассмотреть альтернативу сложным (затратным по ресурсам и сложным в реализации) распределенным транзакциям (*2PC – two-phase commit*[9], *3PC* и т.д.). *Saga* делает послабление относительно мгновенности транзакции, т.е. предполагает что один из сервисов может быть не доступен в данный момент (минус синхронности), поэтому прибегает к плюсам асинхронности, не требуя мгновенного исполнения транзакции, а дожидается ответа от сервиса и только после этого принимает решение о дальнейшем продолжении. Основное правило в данном асинхронном подходе – необходимо иметь компенсационную транзакцию (событие/сообщение), чтобы можно было отменить (rollback) примененные изменения. Например, добавили в базу данных новый заказ, но получили ошибку при попытке обновить данные у покупателя, тогда необходимо отменить изменения по созданию заказа.

С появлением асинхронного взаимодействия и развития таких паттернов как *Saga,* требования теоремы *CAP*[10] переквалифицировались из *ACID* => *BASE*[11]. Что позволило сформировать новые правила таких систем.

[7] Pattern: Database per service [Электронный ресурс]. – URL: https://microservices.io/patterns/data/database-per-service.html (дата обращения: 05.07.2020).

[8] Там же.

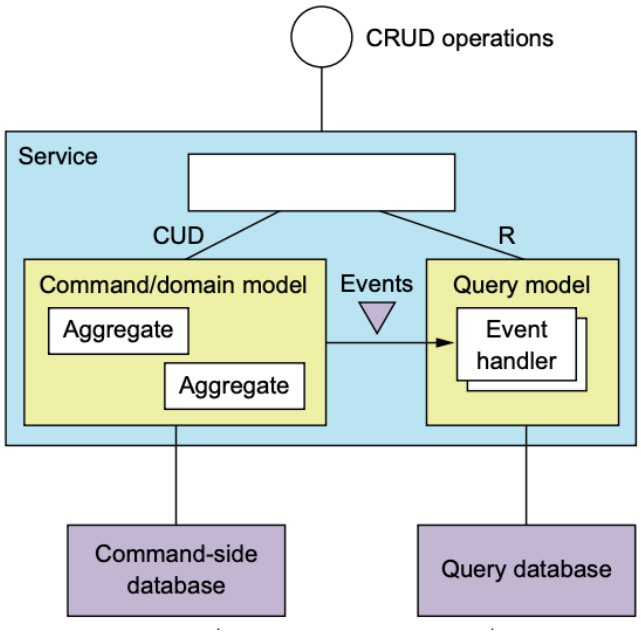
[9] Two-Phase Commit Mechanism // Oracle Database Online Documentation. 11g Release 1 (11.1) / Database Administration. – Oracle, Pp. 279. [Электронный ресурс]. – URL: https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222 (дата обращения: 05.07.2020).

[10] Nazrul S. S. CAP Theorem and Distributed Database Management Systems. Apr 24, 2018. [Электронный ресурс]. – URL: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e (дата обращения: 05.07.2020).

[11] Roe Ch. ACID vs. BASE: The Shifting pH of Database Transaction Processing. March 1, 2012. [Электронный ресурс]. – URL: https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/ (дата обращения: 05.07.2020).

### *C.* ***CQRS***

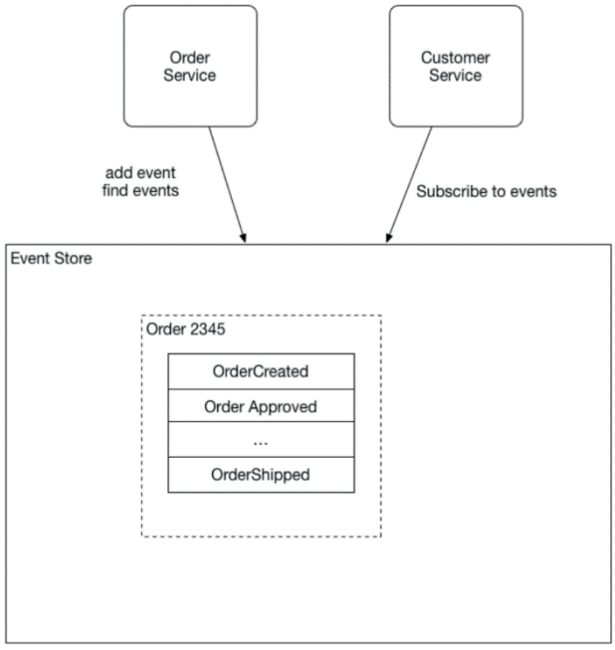
*CQRS* (*Command Query Request Segregation*) имеет схожие черты с *API Composition*, так как используется для извлечения данных из нескольких сервисов [22-25]. Но его главное отличие в том, что он сложнее в реализации. Он разделяет обращения к базе данных на операции (commands, CUD = Create, Update, Delete) и на запросы (query, read-only). Query database (рис. 6), как правило, подразумевает под собой материализованное представление (view) в терминах реляционных баз данных и агрегирует данные из нескольких микросервисов.




F i g. 4. CQRS pattern [7, p. 233]

### *D.* ***Event Sourcing***

Если вспомнить паттерн *Saga*, который сначала должен осуществить транзакцию записи данных в базу, а затем отправить событие подписавшимся сервисам через брокера сообщений, то можно найти ахиллесову пяту в данной схеме – у нас транзакция может не закоммититься (*committed*), а другие сервисы получат уведомление о появлении новой записи и тогда наша система станет не консистентной. И получается, что процесс добавления новой записи не является атомарным, так как представляет из себя две атомарные транзакции: вставка записи в базу и публикация сообщения в брокер. Поэтому для избегания таких проблем был придуман паттерн *Event Sourcing*, имеющий много общего с *Event-Driven Design*[12] [23-25]. Он предлагает вместо хранения модели данных в базе хранить историю событий (event) изменений модели, это позволит собрать в одном месте и историю событий изменения модели (*logs*), и сами модели (*persistence data model*). Чтобы восстановить текущее состояние модели, необходимо выполнить в порядке даты создания события все изменения над моделью (рис. 7).




F i g. 5. Event Sourcing pattern[13]

Все паттерны, связанные с обработкой событий, как правило, подразумевают обмен сообщениями, отождествляющими доменную модель.

## Унифицированная модель данных

Чтобы не проектировать одно и то же в каждом микросервисе, можно рассмотреть унифицированную справочную модель, именно справочную, так как во всем остальном сервисы отличаются доменными моделями[14]. Например, у нас есть два сервиса: *OrderService* и *NotifyService*, в каждом из них есть своя доменная модель, контекст, которая предусматривает взаимодействие со статусной моделью: у каждого заказа (объект *Order*) есть свой определенный статус, и у каждого оповещения (объект *Notify*) есть свой статус. Множество допустимых значений (домен) у каждого из них частично пересекается, но есть и свои уникальные значения. Согласно принципам *DDD*, для создания унифицированной модели необходимо выделить доменную модель, модель-справочник.

### *A. Объект* **Type**

Основополагающим в данной модели будет объект *Type*, который обладает общими свойствами для всех производных от него. Ему присущи следующие свойства:

- *Id* – идентификатор экземпляра *Type*
- *IdParent* – идентификатор “родителя”, используется для построения иерархических связей
- *IdCategory* – идентификатор экземпляра *TypeCategory*
- *Code* – уникальный системный идентификатор

[12] Fowler M. What do you mean by “Event-Driven”? 07 February 2017. [Электронный ресурс]. – URL: https://martinfowler.com/articles/201701-event-driven.html (дата обращения: 05.07.2020).

[13] Pattern: Database per service [Электронный ресурс]. – URL: https://microservices.io/patterns/data/database-per-service.html (дата обращения: 05.07.2020).

[14] Evan E. Domain-Driven Design: Tackling Complexity in the Heart of Software. – 1st Ed. – Addison-Wesley Professional, 2003.

- *BCode* – бизнес идентификатор
- *Name* – наименование
- *Note* – описание
- *Virtual* – признак абстрактности
- *SortOrder* – порядковый номер экземпляра из множества экземпляров одного отношения в разрезе *IdCategory*
- *Deleted* – признак актуальности/жизни экземпляра

Атрибут *Code* является уникальным текстовым идентификатором объекта *Type* наравне с *Id*. Также мы накладываем дополнительные ограничения на свойство *BCode,* он также является уникальным текстовым значением, но уже в рамках определенной *IdCategory.* Пример использования свойства *Virtual*: у нас есть иерархия объектов, и нас интересуют только листовые узлы данной иерархии, тогда мы помечаем все промежуточные от корня до листа узлы как *Virtual = 1*.

Рассмотрим варианты проекции объекта статус (*Status*) на объект *Type:*

```
{ // Type
   {                                  {
      id: 1,                             id: 2,
      idParent: null,                    idParent: null,
      idCategory: 1,                     idCategory: 2,
      code: '',                          code: '',
      bCode: '',                         bCode: '',
      name: 'Новый',                     name: 'Новый',
      note: 'Новый',                     note: 'Новый',
      virtual: 0,                        virtual: 0,
      sortorder: 0,                      sortorder: 0,
      deleted: 0                         deleted: 0
   },                                  },

   {                                  {
      id: 3,                             id: 4,
      idParent: null,                    idParent: null,
      idCategory: 1,                     idCategory: 2,
      code: '',                          code: '',
      bCode: '',                         bCode: '',
      name:'Выполняется',                name: 'Прочитано,
      note: 'Выполняется',               note: 'Прочитано',
      virtual: 0,                        virtual: 0,
      sortorder: 1,                      sortorder: 1,
      deleted: 0                         deleted: 0
   },                                  }

}
```

*B.* *Объект* **TypeCategory**

Для описания принадлежности объектов к разным категориям, необходимо описать объект *TypeCategory,* который позволит осуществлять логическое разделение данных в модели. Свойства *TypeCategory* отлично проецируются на объект *Type,* который является базовым для любого в домене микросервиса справочников. *TypeCategory* можно описать следующим образом:

- *Id* – ссылка на экземпляр *Type*
- *TableName* – название отношения, которое расширяет перечень свойств отношения *Type*
- *ViewName* – аналогично предыдущему, вместо отношения используется представление
- *ClassName* – полное имя класса, используется для корректной сериализации/десериализации

Рассмотрим наши сервисы: *OrderService* и *NotifyService.* Им соответствуют два экземпляра объекта *TypeCategory:*

```
{ // TypeCategory
   {                                  {
      id: 4,                             id: 5,
      tableName: '',                     tableName: '',
      viewName: '',                      viewName: '',
      className: ''                      className: ''
   },                                  }
}
```

*С помощью TypeCategory можно выделять новые справочники, которые будут дополнять атрибутивный состав более «узкого» объекта.*

*Поле id* в *TypeCategory ссылается на объект Type.*

```
{ // Type
   {                                  {
      id: 4,                             id: 5,
      idParent: null,                    idParent: null,
      idCategory: null,                  idCategory: null,
      code:                              code:
'Category_OrderStatus',            'Category_NotifyStatus',
      bCode: 'OrderStatus',              bCode: 'NotifyStatus',
      name: 'Статусы                     name: 'Статусы
заказов',                          уведомлений',
      note: 'Категория                   note: 'Категория
статусов заказов',                 статусов уведомлений',
      virtual: 0,                        virtual: 0,
      sortorder: 0,                      sortorder: 0,
      deleted: 0                         deleted: 0
   },                                  }
}
```

*Получается, зная идентификатор категории статусов, относящихся к OrderService*, мы сможем получить множество всех статусов для сервиса.

*XI.* *Объект* **RLType**

Чтобы связывать экземпляры двух объектов необходимо описать объект *RLType.* Он будет иметь следующий атрибутивный состав:

- *Id* – идентификатор экземпляра *Type*
- *IdType1* – идентификатор экземпляра объекта №1
- *IdType2* – идентификатор экземпляра объекта №2
- *IdRLTypeCategory* – идентификатор экземпляра *RLTypeCategory*

Попробуем описать связь объекта *User* и *Role,* оба объекта наследуются от *Type*, но с помощью *TypeCategory* мы можем добавить расширяющие поля каждому из объектов.

```
{ // Type
  {
    id: 9,
    idParent: null,
    idCategory: null,
    code: 'Category_Users',
    bCode: 'Users',
    name:'Пользователи',
    note: 'Пользователи',
    virtual: 0,
    sortorder: 0,
    deleted: 0
  },
  {
    id: 10,
    idParent: null,
    idCategory: 9,
    code:
'Users_JohnSmith13',
    bCode:
'JohnSmith13',
    name: 'John Smith',
    note: 'Пользователь
John Smith',
    virtual: 0,
    sortorder: 5,
    deleted: 0
  }
}
```

```
{ // TypeCategory
  {
    id: 9,
    tableName: 'User',
    viewName: ''.
    className: ''
  }
}
```

```
{ // User
  {
    id: 10,
    firstName: 'John',
    lastName: 'Smith',
    email: 'temp@temp.com'
  }
}
```

Для расширения объекта *Type* до объекта *User*, нам потребовалось указать атрибут tableName = 'User', причем идентификаторы у расширяемого экземпляра *Type* совпадают с *User*.

С помощью такой онтологии мы смогли описать категорию "Пользователи" и одного из пользователей с именем 'John Smith'. Опишем две роли, например, администратор и наблюдатель, будем считать, что расширять атрибутивный состав для объекта роль не требуется:

```
{ // Type
  {
    id: 11,
    idParent: null,
    idCategory: null,
    code: 'Category_Roles',
    bCode: 'Roles',
    name: 'Роли',
    note:'Системные роли',
    virtual: 0,
    sortorder: 0,
    deleted: 0
  },
  {
    id: 13,
    idParent: null,
    idCategory: 11,
    code: 'Roles_Observer',
    bCode: 'Observer',
    name: 'Наблюдатель',
    note: 'Системная роль
Наблюдатель',
    virtual: 0,
    sortorder: 1,
    deleted: 0
  }
  {
    id: 12,
    idParent: null,
    idCategory: 11,
    code: 'Roles_Admin',
    bCode: 'Admin',
    name: 'Администратор',
    note: 'Системная роль
Администратор',
    virtual: 0,
    sortorder: 0,
    deleted: 0
  },
}
```

```
{ // TypeCategory
  {
    id: 11,
    tableName: '',
    viewName: ''.
    className: ''
  }
}
```

Чтобы назначить пользователю какую-либо роль, необходимо отразить этой в экземпляре объекта *RLType*:

```
{ // Type
  {
    id: 14,
    idParent: null,
    idCategory: null,
    code:
'CategoryRelation_User2Role',
    bCode: 'User2Role',
    name:'Связь Пользователь –
Роль',
    note:'Связь Пользователь –
Роль',
    virtual: 0,
    sortorder: 1,
    deleted: 0
  }
}
```

```
{ // RlTypeCategory
  {
    id: 14,
    inheritenceType1: 0,
    inheritenceType2: 0
  }
}
```

```
{ // RlType
  {
    id: 15,
    idType1: 10,
    idType2: 12,
    idRLTypeCategory: 14
  }
}
```

Таким образом мы смогли задать связь Пользователь – Роль, в частности, присвоили пользователю 'John Smith' роль 'Администратор'.

Вернемся к примеру про статусы в *OrderService* и *NotifyService*, предположим, что домен статусов сервиса заказов полностью "покрывает" статусы из сервиса оповещений (на языке множеств это звучит так: A ⊂ B, где A – домен статусов *NotifyService*, а B – домен статусов *OrderService*). Чтобы не дублировать одни и те же данные для каждой категории (*Category* для *OrderStatus* и *Category* для *NotifyStatus*), мы можем воспользоваться объектом *RLType* и "задать" в нем связи принадлежности статуса к сервису по аналогии как мы «связали» Роль и Пользователя.

*D.* *Объект* **RLTypeCategory**

- *Id* – идентификатор экземпляра *Type*
- *InheritenceType1* – признак наследования для Type1
- *InheritenceType2* – признак наследования для Type2

Для более наглядного описания свойств объекта *RLTypeCategory*, рассмотрим пример: нам необходимо реализовать ролевую модель в микросервисе, чтобы к определенным компонентам имели доступ люди с конкретными ролями. У нас есть модели *User* и *Role*.

Представим, что одному пользователю может соответствовать несколько ролей, например, тестировщик и администратор,

или же есть руководитель, который совмещает в себе роли как администратора, так и наблюдателя. Для этого необходимо у экземпляров объекта *Type* с *id* равным 13 и 14 указать ссылку на «родителя» – роль Руководитель.

```
{ // Type
  {
    id: 13,
    idParent: 20,
    idCategory: 11,
    code: 'Roles_Observer',
    bCode: 'Observer',
    name: 'Наблюдатель',
    note: 'Системная роль Наблюдатель',
    virtual: 0,
    sortorder: 1,
    deleted: 0
  },
  {
    id: 12,
    idParent: 20,
    idCategory: 11,
    code: 'Roles_Admin',
    bCode: 'Admin',
    name: 'Администратор',
    note: 'Системная роль Администратор',
    virtual: 0,
    sortorder: 0,
    deleted: 0
  }
}
```

Теперь необходимо добавить роль Руководитель:

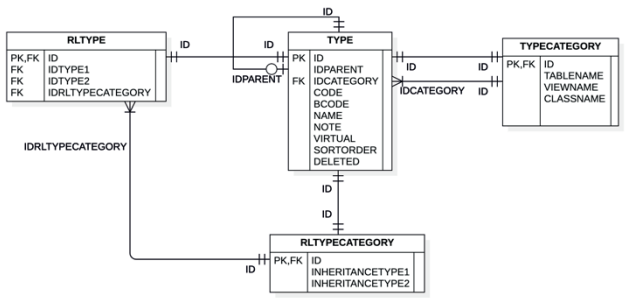
```
{ // Type
  {
    id: 20,
    idParent: null,
    idCategory: 11,
    code: 'Roles_Chief',
    bCode: 'Chief',
    name: 'Руководитель,
    note: 'Системная роль Руководитель',
    virtual: 0,
    sortorder: 2,
    deleted: 0
  }
}
```

Остается назначить пользователю 'John_Smith' роль Руководитель (*idType2*: 20) и указать в поле *inheritenceType2*: 1.

```
{ // RlTypeCategory
  {
    id: 14,
    inheritenceType1: 0,
    inheritenceType2: 1
  }
}
```

```
{ // RlType
  {
    id: 15,
    idType1: 10,
    idType2: 20,
    idRLTypeCategory: 14
  }
}
```

Указав *inheritenceType2*: 1 мы сообщаем системе, что все дочерние элементы у *idType2* также обладают связью *idRLTypeCategory*: 14 (связь Пользователь – Роль), т. е. 'John Smith' также получит роль и Администратор, и Наблюдатель. В итоге, мы имеем следующую ER диаграмму сущностей (рис. 8).

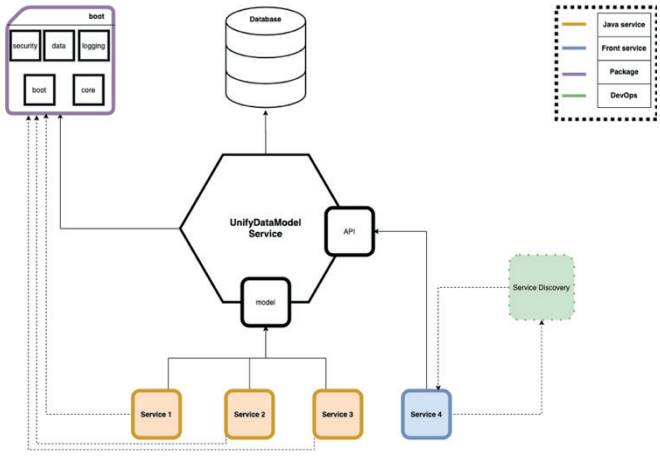



F i g. 6. ER диаграмма

## Применение

Унифицированная модель данных позволяет не только описывать сложные связи между сущностями (выстраивать онтологию), но и упрощает процесс взаимодействия и интерпретации данных от сервиса к сервису (упрощение интеграции между сервисами) [18]. Унифицированная модель выступает в роли *CDM* (*Canonical Data Model*) [25]. *CDM* основывается на межсервисном взаимодействии посредством сообщений, структура которых представляются в виде общей модели *(canonical data model)*[15]. Несмотря на то, что данный паттерн происходит из *SOA*, он достаточно актуален и имеет место быть в микросервисной архитектуре.

В силу ограничений на микросервис – хранение справочных (перманентных) данных, можно использовать следующую схему взаимодействия между сервисами (рис. 9).




F i g. 7. Microservice communication

Каждый сервис имеет право использовать данные, модели микросервиса справочника, но только в режиме чтения. Добавление и редактирование данных осуществляется посредством

[15] du Preez T. Canonical Data Models & Microservices: Clarifying the conflicting views. –Deloitte, 2016. [Электронный ресурс]. – URL: https://www2.deloitte.com/content/dam/Deloitte/za/Documents/strategy/ZA_Deloitte_Digita_Canonical_Schemas.pdf (дата обращения: 05.07.2020).

Liquibase скриптов. Использование API микросервиса справочников может быть реализовано как в библиотечной форме, так и в форме *RESTful* запросов к серверу. Второй вариант более универсальный, ему отдается предпочтение при использовании в гетерогенных микросервисных системах/платформах, где каждый сервис реализован на своем стеке технологий. Я скрестил несколько паттернов: *schema-per-service*, *shared database* (readonly режим), saga для добавления пользователей. Данные в справочнике физически никогда не удаляются, они могут быть помечены флагом *deleted*: 1 (логическое удаление).

## Заключение

Микросервис справочников применяется в промышленной платформе мониторинга корпоративных клиентов банка, данная платформа насчитывает порядка 20 сервисов как бизнес, так и технических, которые извлекают и оперируют моделями из микросервиса справочников. Данный сервис предоставляет следующие преимущества:

- Онтология
- Ролевая модель
- Привилегии
- Пользователи
- Категории справочников
- Интеграция между сервисами
- Расширение моделей (пример с моделью *User*)
- Экономия ресурсов, оптимизация избыточности моделей (все справочники в одном месте, не надо дублировать одно и то же в каждом сервисе)

Микросервис справочников имеет смысл применять, когда количество микросервисов достаточно велико и они образуют целую платформу (более 5 микросервисов).

## References


[1] Yussupov V., Breitenbücher U., Krieger C., Leymann F., Soldani J., Wurster M. Pattern-based Modelling, Integration, and Deployment of Microservice Architectures. In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC).* Eindhoven, Netherlands; 2020. p. 40-50. (In Eng.) DOI: https://doi.org/10.1109/EDOC49727.2020.00015

[2] Arcelli D., Cortellessa V., Di Pompeo D., Eramo R., Tucci M. Exploiting Architecture/Runtime Model-Driven Traceability for Performance Improvement. In: *2019 IEEE International Conference on Software Architecture (ICSA).* Hamburg, Germany; 2019. p. 81-90. (In Eng.) DOI: https://doi.org/10.1109/ICSA.2019.00017

[3] Carvalho L., Garcia A., Assunção W.K.G., de Mello R., de Lima M.J. Analysis of the criteria adopted in industry to extract microservices. In: *Proceedings of the Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice (CESSER-IP '19).* IEEE Press; 2019. p. 22-29. (In Eng.) DOI: https://doi.org/10.1109/CESSER-IP.2019.00012

[4] Pautasso C., Zimmermann O., Amundsen M., Lewis J., Josuttis N. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*. 2017; 34(1):91-98. (In Eng.) DOI: https://doi.org/10.1109/MS.2017.24

[5] Neri D., Soldani J., Zimmermann O., Brogi A. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*. 2020; 35(1-2):3-15. (In Eng.) DOI: https://doi.org/10.1007/s00450-019-00407-8

[6] Jamshidi P., Pahl C., Mendonça N.C., Lewis J., Tilkov S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*. 2018; 35(3):24-35. (In Eng.) DOI: https://doi.org/10.1109/MS.2018.2141039

[7] Richardson C. Microservices Patterns: With examples in Java. 1st Ed. Manning Publications; 2018. (In Eng.)

[8] Knoche H., Hasselbring W. Using Microservices for Legacy Software Modernization. *IEEE Software*. 2018; 35(3):44-49. (In Eng.) DOI: https://doi.org/10.1109/MS.2018.2141035

[9] Taibi D., Lenarduzzi V., Pahl C., Janes A. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: *Proceedings of the XP2017 Scientific Workshops (XP '17).* Association for Computing Machinery, New York, NY, USA; 2017. Article 23. p. 1-5. (In Eng.) DOI: https://doi.org/10.1145/3120459.3120483

[10] Messina A., Rizzo R., Storniolo P., Urso A. A Simplified Database Pattern for the Microservice Architecture. In: *Proceedings of the DBKDA 2016: The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications*. Lisbon, Portugal; 2016. p. 35-40. (In Eng.) DOI: https://doi.org/10.13140/RG.2.1.3529.3681

[11] Villaça L.H.N., Azevedo L.G., Siqueira S.W.M. Microservice Architecture for Multistore Database Using Canonical Data Model. In: *XVI Brazilian Symposium on Information Systems (SBSI'20)*. Association for Computing Machinery, New York, NY, USA; 2020. Article 20. p. 1-8. (In Eng.) DOI: https://doi.org/10.1145/3411564.3411629

[12] Al-Debagy O., Martinek P. A Comparative Review of Microservices and Monolithic Architectures. In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI).* Budapest, Hungary; 2018. p. 000149-000154. (In Eng.) DOI: https://doi.org/10.1109/CINTI.2018.8928192

[13] Smid A., Wang R., Cerny T. Case study on data communication in microservice architecture. In: *Proceedings of the Conference on Research in Adaptive and Convergent Systems (RACS '19).* Association for Computing Machinery, New York, NY, USA; 2019. p. 261-267. (In Eng.) DOI: https://doi.org/10.1145/3338840.3355659

[14] Hasselbring W., Steinacker G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* Gothenburg; 2017. p. 243-246. (In Eng.) DOI: https://doi.org/10.1109/ICSAW.2017.11

[15] Merson P., Yoder J. Modeling Microservices with DDD. In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C).* Salvador, Brazil; 2020. p. 7-8. (In Eng.) DOI: https://doi.org/10.1109/ICSA-C50368.2020.00010

[16] Dobaj J., Iber J., Krisper M., Kreiner C. A Microservice Architecture for the Industrial Internet-Of-Things. In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18).* Association for Computing Machinery, New York, NY, USA; 2018. Article 11. p. 1-15. (In Eng.) DOI: https://doi.org/10.1145/3282308.3282320

[17] Kumar S.S., Shastry P.M.M. Database-per-service for e-learn-

ing system with micro-service architecture. In: *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon).* Bangalore; 2017. p. 705-708. (In Eng.) DOI: https://doi.org/10.1109/SmartTechCon.2017.8358462

[18] Kaneshima E., Braga R.T.V. Patterns for enterprise application integration. In: *Proceedings of the 9th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP '12).* Association for Computing Machinery, New York, NY, USA; 2012. Article 2. p. 1-16. (In Eng.) DOI: https://doi.org/10.1145/2591028.2600811

[19] Akbulut A., Perros H.G. Software Versioning with Microservices through the API Gateway Design Pattern. In: *2019 9th International Conference on Advanced Computer Information Technologies (ACIT).* Ceske Budejovice, Czech Republic; 2019. p. 289-292. (In Eng.) DOI: https://doi.org/10.1109/ACITT.2019.8779952

[20] Schäffer E., Mayr A., Fuchs J., Sjarov M., Vorndran J., Franke J. Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions. *Procedia CIRP.* 2019; 86:86-91. (In Eng.) DOI: https://doi.org/10.1016/j.procir.2020.01.017

[21] Amaral M., Polo J., Carrera D., Mohomed I., Unuvar M., Steinder M. Performance Evaluation of Microservices Architectures Using Containers. In: *2015 IEEE 14th International Symposium on Network Computing and Applications.* Cambridge, MA; 2015. p. 27-34. (In Eng.) DOI: https://doi.org/10.1109/NCA.2015.49

[22] Munonye K., Martinek P. Evaluation of Data Storage Patterns in Microservices Archicture. In: *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE).* Budapest, Hungary; 2020. p. 373-380. (In Eng.) DOI: https://doi.org/10.1109/SoSE50414.2020.9130516

[23] Zhong Y., Li W., Wang J. Using Event Sourcing and CQRS to Build a High Performance Point Trading System. In: *Proceedings of the 2019 5th International Conference on E-Business and Applications (ICEBA 2019).* Association for Computing Machinery, New York, NY, USA; 2019. p. 16-19. (In Eng.) DOI: https://doi.org/10.1145/3317614.3317632

[24] Meißner D., Erb B., Kargl F., Tichy M. Retro-λ: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS '18).* Association for Computing Machinery, New York, NY, USA; 2018. p. 76-87. (In Eng.) DOI: https://doi.org/10.1145/3210284.3210285

[25] Villaça L.H.N., Azevedo L.G., Baião F. Query strategies on polyglot persistence in microservices. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18).* Association for Computing Machinery, New York, NY, USA; 2018. p. 1725-1732. (In Eng.) DOI: https://doi.org/10.1145/3167132.3167316

*Поступила 05.07.2020; принята к публикации 26.08.2020; опубликована онлайн 30.09.2020.*
*Submitted 05.07.2020; revised 26.08.2020; published online 30.09.2020.*

Об авторе:

**Балес Александр Иннокентьевич**, магистрант факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Россия, г. Москва, ГСП-1, Ленинские горы, д. 1), ORCID: http://orcid.org/0000-0001-8641-9600, aleks_bales@mail.ru

**Благодарности:** автор выражает особую благодарность кандидату физико-математических наук, старшему научному сотруднику факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова Дмитрию Евгеньевичу Намиоту за полезные рекомендации и ценные советы в подготовке материалов статьи.

*Автор прочитал и одобрил окончательный вариант рукописи.*

About the author:

**Alexander I. Bales**, undergraduate student of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1, Leninskie gory, Moscow 119991, Russia), ORCID: http://orcid.org/0000-0001-8641-9600, aleks_bales@mail.ru

**Acknowledgement:** The author expresses special gratitude to Dmitry E. Namiot, Candidate of Physical and Mathematical Sciences, Senior Researcher of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University, for useful recommendations and valuable advice in the preparation of the materials of the article.

*The author has read and approved the final manuscript.*