

министерство науки и высшего образование РФ

Федеральное государственное бюджетное  
образовательное учреждение  
высшего учебного образование  
«нижегородский государственный технический  
университет Р.Е. Алексеева»

Кафедра «Вычислительные системы и технологии»

**Сервис ориентированные системы  
лабораторные работы**

Нижний Новгород 2023

Составители: **Д.В. Жевнерчук, В.О. Сапожников**

Лабораторные работы по дисциплине Сервис ориентированные системы для студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» по профилю подготовки «Программное обеспечение средств вычислительной техники и автоматизированных систем» очной и очно-заочной формы обучения.

# Оглавление

Лабораторная работа 1	3
Лабораторная работа 2	21
Лабораторная работа 3	23
Лабораторная работа 4	35
Оформление курсовой работы	37
Приложение 1. Академическая лицензия JMX	39
Приложение 2. Реляционные базы данных	41

# Лабораторная работа 1

## Спецификация OpenAPI 3.0

### Цель работы:

1. Изучить спецификацию OpenAPI 3.0
2. Научиться проектировать конечные точки<sup>1</sup>, разрабатываемых сервисов.
3. Научиться документировать API разрабатываемого сервиса по спецификации OpenAPI 3.0 в графическом редакторе.

### Задание к лабораторной работе

Согласно варианту курсовой работы продумать конечные точки разрабатываемого сервиса и модель данных. Задокументировать API согласно спецификации OpenAPI 3.0 в графическом редакторе [Swagger Editor](#)

### Минимальные требования

Для минимального выполнения данной лабораторная работа необходимо задействовать все основные блоки спецификации OpenAPI 3.0 при документировании API.

- Описать основную информацию о сервисе в **InfoObject**: title, description, version.
- Описать модель данных, используемых при работе сервиса - **ComponentsObject**.
- Описать конечные точки CRUD операция над разработанной моделью данных:
  - создание элемента;
  - получение всех элементов;
  - получение элемента по id;
  - обновление элемента по id;
  - удаление элемента по id.

---

<sup>1</sup>**Endpoint API** (конечная точка) - интерфейс взаимодействия с сервером, поддерживающая набор HTTP операций.

## Содержание отчета

1. Титульный лист.
2. Номер варианта курсовой работы.
3. Краткое описание разрабатываемого сервиса.
4. Разработанная модель данных.
5. Основные блоки написанные вами документации.
6. Скриншоты из графического дизайнера.

## Теория

OpenAPI представляет собой независимую от ЯП<sup>2</sup> спецификацию для описания RESTful интерфейсов. Данный формат является понятным для человеческой и машинной обработке, позволяя взаимодействовать с интерфейсом ничего не зная о его реализации.

Спецификация OpenAPI может использоваться для визуализации и отображения документации API, инструментами генерации серверных и клиентских частей приложения.

## История развития

В начале 2010 года компания SmartBear начала разработку фреймворка Swagger и спецификации для него, для внутренних нужд компании Wordnik.

В марте 2015 году SamrtBear выкупила права на спецификацию Swagger и перевала развитие данной технологии в open source.

В декабре 2015 года спецификация Swagger была передана новой организацией под эгидой Linux Foundation: OpenAPI Initiative. С данного времени Swagger и спецификация OpenAPI стали раздельными понятиями. В 2017 году вышла версия OpenAPI 3.0.

OpenAPI развивается своим отдельным сообществом, в то время время как Swagger, пусть и самым популярный, но не единственным инструментом для работы со спецификацией OpenAPI.

На данный момент спецификация OpenAPI является одной из популярнейших. Возможные альтернативы: [API Blueprint](#) и [RAML](#).

## Альтернативы

### API Blueprint

API Blueprint так же, как и OpenAPI определяет спецификацию для описания REST API.

Одним из ключевых факторов не успеха API Blueprint является спецификация основанная на синтаксисе Markdown. MD формат удобен для "верстки" текста, но сложен для машинной обработки - парсинга и выделения отдельных блоков.

---

<sup>2</sup>Язык программирования

## RAML

Потенциально сильный инструмент для документирования API. RAML - надстройка над YAML, поддерживающая переиспользование переменных и наследование. В последнее время развитие данной технологии как open source прекратилось.

RAML имеет высокий порог входа и поддерживает только First Design API паттерн, не позволяя генерировать документацию из кода. Так же для работы данной технологии необходимо развертывание отдельных сервисов для работы с ней. Взамен указанным затратам RAML предоставляет большие возможности по переиспользованию элементов и автогенерации тестов API.

На данный момент RAML все ещё можно встретить на крупных предприятиях, использующих одну из его дорогостоящих корпоративных реализаций.

## Формат

Документация, составленная согласно спецификации OpenAPI, представляет собой JSON объект, который может быть представлен в формате JSON или YAML документа.

- **JSON** (*англ. JavaScript Object Notation*) - текстовый формат обмена данными, основанный на JavaScript. Но при этом формат независим от JS и может использоваться в любом ЯП.
- **YAML** (*англ. YAML Ain't Markup Language — «YAML — не язык разметки»*). Задумывался как удобный для человека формат данных. В спецификации OpenAPI используется как более удобное для человека представления JSON.

## JSON

**JSON-объект** — это неупорядоченное множество пар «ключ : значение».

В данной работе JSON всегда рассматривается как текстовый формат передачи данных. Важно разделять понятие JSON-объекта, как формата данных и объектов ЯП JavaScript.

- **Ключ** - всегда строка, указывающаяся в кавычках. Порядок ключей неважен.
- **Значение** может представлено одним из следующих типов:
  - JSON-объект
  - Массив
  - Число - целое или вещественное
  - Литералы булевого значения - **true/false**
  - null
  - Символьная строка

## Массивы в JSON

Массив в JSON так же должен храниться по ключу, заключается в [ ] и является уже упорядоченным множеством. У элементов массива уже нет ключей, есть только индекс. Массив в JSON не имеет типа и может содержать любые значения.

## Пример JSON

```
{
  "name": "Jack",
  "surname": "Dawson",
  "age": 20,
  "girlfriend": {
    "name": "Rose",
    "surname": "DeWitt Bukater",
    "age": 17
  }
}
```

## Типы данных

Примитивные типы данных в спецификации OpenAPI основаны на типах, поддерживаемых [JSON Schema Specification Wright Draft 00](#).

Примитивные типы имеют **необязательное** свойство-модификатор: `format`. Данное поле является открытым строковым свойством и может иметь любое значение. `Format` используется для уточнения типов.

Таблица 1: Типы данных OpenAPI

Type	Format	Комментарий
integer	int32	32-битное целочисленное - Int
integer	int64	64-битное целочисленное - Long
number	float	вещественное одинарной точности
number	double	вещественное двойной точности
string		
string	byte	символы в кодировке base64
string	binary	любая последовательность октетов
boolean		
string	date	full-date - <a href="#">RFC3339</a>
string	date-time	date-time - <a href="#">RFC3339</a>
string	password	подсказка пользовательским интерфейсам о необходимости скрытия ввода

Данное, странное на первый взгляд, представление типов данных связано с независимостью спецификации OpenAPI от ЯП.

Все ЯП умеют работать с 64-битными вещественными и целыми числами, но везде это реализовано по-своему. Например, в JavaScript все числа являются один и тем же классом [Number](#).

Отдельное внимание так же стоит уделить не указанному в таблице типу `Object`. Через данный тип и его дополнительные параметры, как `format` определяются структуры данных и пользовательские типы.

## Пример пользовательского типа данных

Пользовательский тип данных `Dog` с полями имя, порода и возраст может быть представлено в следующем виде.

Type	Format	Комментарий
object	Dog	пользовательский тип данных

```
{
  "type": "object",
  "format": "Dog"
}
```

Сам же тип `Dog` будет расписан в отдельном необязательном объекте схемы - **Components Object**.

## Структуры данных

Иногда Endpoint может принимать в качестве параметров или возвращать в ответе какие-то структуры данных.

### Массив

Массивы в спецификации OpenAPI описываются так же как и остальные типы. Для описания типа массива используется особый объект `items`. Спецификация поддерживает массивы следующих видов:

- Массив фиксированного типа.
- Массив одного из возможных типов.
- Массив любого (любой комбинации) из возможных типов.
- Массив произвольного типа.

### Массив фиксированного типа

Массив, который может содержать **только один** указанный тип.

```
{
  "type": "array",
  "items": {
    "type": "integer",
    "format": "int64"
  }
}
```



### Массив одного из типов

Массив, который может содержать **только один** из указанных тип. Список возможных типов указывается в массиве `oneOf`.

```
{
  "type": "array",
  "items": {
    "oneOf": [
      {
        "type": "integer",
        "format": "int64"
      },
      {
        "type": "number",
        "format": "double"
      }
    ]
  }
}
```

### Массив любого из возможных типов

Массив, который может содержать **любую** комбинацию из указанных типов. Список возможных типов указывается в массиве `anyOf`.

```
{
  "type": "array",
  "items": {
    "anyOf": [
      {
        "type": "integer",
        "format": "int64"
      },
      {
        "type": "number",
        "format": "double"
      }
    ]
  }
}
```

### Массив произвольного типа

Массив, который может содержать **любую** комбинацию из определенных в данной схеме типов.

```
{
  "type": "array",
  "items": {}
}
```

```
}
```

Данная упрощенная запись эквивалентна записи через `anyOf` с перечислением всех типов схемы.

## Ассоциативный словарь

Поскольку формат JSON сам по себе является не упорядоченным множеством пар «ключ : значение», то ассоциативный словарь представляется через `object`. В отдельном дополнительном объекте `additionalProperties` указывается тип значений.

```
{
  "type": "object",
  "additionalProperties": {
    "type": "number",
    "format": "float"
  }
}
```

## Уникальное множество

В спецификации OpenAPI не предусмотрено отдельного обозначения для множества уникальных значений. В качестве него просто указывается массив, а преобразование массива в множества и обратно уже реализуется внутренней логикой сервиса.

У массива так же есть не обязательный булевый параметр `uniqueItems`, который по умолчанию `false`. Но даже если он `true`, то это не обозначает структуру данных типа `Set`.

## Схема

В данной лабораторной работе рассматривается лишь базовый, необходимый набор элементов схемы, который необходимо задействовать.

Больше информации можно получить на [GitHub репозитории OpenAPI](#).

Элементы спецификации представляют собой JSON-объекты, с фиксированным набором обязательных и необязательных полей. Если поле явно не помечено **обязательно**, оно может считаться необязательным.

## OpenAPI Object

Главный корневой объект документа OpenAPI.

Таблица 2: Поля OpenAPI Object

Поле	Тип	Описание
openapi	string	<b>обязательно.</b> Семантический номер используемой версии спецификации OpenAPI.
info	Info Object	<b>обязательно.</b> Предоставление метаданных об API.
servers	[Server Object]	Массив объектов <b>Server Object</b> , предоставляющих информацию о подключении к целевому серверу. Если поле servers не указано или является пустым массивом, то значением по умолчанию будет объект <b>Server Object</b> со значением url, равный "\" .
paths	Paths Object	<b>обязательно.</b> Доступные пути и операции с API.
components	Components Object	Элемент для хранения различных схем для спецификации. Здесь же указываются пользовательские типы данных.

## Info Object

Объект, содержащий метаданные об API. Метаданные могут использоваться клиентскими приложениями, а так же инструментами для работы со спецификацией.

Таблица 3: Поля Info Object

Поле	Тип	Описание
title	string	<b>обязательно.</b> Название приложения.
description	string	Краткое описание приложения
version	string	<b>обязательно.</b> Версия OpenAPI документа. Может отличаться от версии реализации API.

## Пример Info Object

```
{
  "title": "example.com",
  "description": "Сервис Ориентированные Системы. Лабораторная работа №1.
                  Пример спецификации OpenAPI, написанный в Swagger Editor.",
  "version": "1.0.1"
}
```

## Server Object

Объект, предоставляющий информацию о подключении к целевому серверу.

Таблица 4: Поля Server Object

Поле	Тип	Описание
url	string	<b>обязательно.</b> URL целевого хоста.
description	string	Описание хоста, указанного в URL

### Пример Server Object

Server Object указываются в массиве по ключу "servers" .

```
[
  {
    "url": "sos.lrl.example",
    "description": "Продуктовый сервер"
  }
]
```

### Path Object

Содержит относительные пути к отдельным конечным точкам и их операциями. Путь добавляется к URL-адресу из Server Object для создания полного адреса.

Таблица 5: Поля Path Object

Поле	Тип	Описание
{path}	Path Item Object	Относительный путь к отдельной конечной точке. Имя должна начинаться с "\" .

### Path Item Object

Описывает операции, доступные по данному пути.

Таблица 6: Поля Path Item Object

Поле	Тип	Описание
HTTP метод	Operation Object	Определение операции по данному HTTP методу.

### Operation Object

Описывает одну API операцию по указанному пути.

Таблица 7: Поля Operation Object

Поле	Тип	Описание
summary	string	Краткое описание того, что делает операция.
operationId	string	Строка, используемая для идентификации операции. Идентификатор должен быть уникальным среди всех операций описанных в API. Чувствителен к регистру.
parameters	[Parameter Object   Reference Object]	Список параметров, применимых для этой операции.
responses	Response Object	<b>обязательно.</b> Список возможных ответов, возвращаемых в результате выполнения операции.

### Пример Operation Object

```
{
  "summary": "Получение информации оу указанном студенте",
  "operationId": "getStudentById",
  "parameters": [
    {
      "name": "studentId",
      "in": "path",
      "required": true,
      "description": "Id студента для получения информации",
      "schema": {
        "type": "string"
      }
    }
  ],
  "responses": {
    "200": {
      "description": "Список полученных студентов",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Student"
          }
        }
      }
    }
  }
}
```

```

    }
}

```

## Responses Object

Контейнер для ожидаемых ответов. Сопоставляет код ответа HTTP с ожидаемым ответом - **Response Object**.

Для всех не рассматриваемых кодов можно использоваться **default**.

Должен содержать хотя бы один успешный ответ.

Таблица 8: Поля Responses Object

Поле	Тип	Описание
HTTP status code	Response Object   Reference Object	В качестве ключа можно использовать

## Пример Responses Object

```

{
  "200": {
    "description": "Список полученных студентов",
    "content": {
      "application/json": {
        "schema": {
          "$ref": "#/components/schemas/Student"
        }
      }
    }
  }
}

```

## Response Object

Описывает один ответ от операции.

Таблица 9: Поля Response Object

Поле	Тип	Описание
description	string	<b>обязательно.</b> Краткое описание ответа.
content	Map[string, MediaTypeObject]	Ключ - тип носителя, значение - описание ответа.

## Пример Response Object

```
{
  "description": "Список полученных студентов",
  "content": {
    "application/json": {
      "shema": {
        "$ref": "#/components/schemas/Student"
      }
    }
  }
}
```

## Parametr Object

Описывает один параметр операции.

Возможные значения параметра **in**:

- **path** - используется в шаблоне пути. Значения параметра фактически является частью URL-адреса операции. Например, в `/items/{itemId}` параметр пути равен `itemId`.
- **query** - параметры, добавляемые к URL-адресу. Например, в `/items?id=###` параметр запроса - `id`.
- **header** - пользовательские заголовки, ожидаемые как часть запроса.
- **cookie** - используется для передачи API определенного значения cookie.

Таблица 10: Поля Parameter Object

Поле	Тип	Описание
name	string	<b>обязательно.</b> Имя параметра. Чувствителен к регистру.
in	string	<b>обязательно.</b> Местоположение параметра. Возможные значения: «query», «header», «path», «cookie».
description	string	Краткое описание параметра.
required	boolean	Определяет является ли параметр обязательным. Если параметр располагается в <b>path</b> , то это свойство <b>обязательно</b> и должно быть <b>true</b> .

## Пример Parametr Object

```
{
  "name": "studentId",
  "in": "path",
  "required": true,
  "description": "Id студента для получения"
```

```

        информации",
    "schema": {
        "type": "string"
    }
}

```

## Reference Object

Объект, позволяющий ссылаться на другие компоненты спецификации как внутри, так и снаружи.

Таблица 11: Поля Reference Object

Поле	Тип	Описание
\$ref	string	<b>обязательно.</b> Ссылочная строка.

## Пример Reference Object

```

{
    "ref": "#/components/schemas/Student"
}

```

## Response Object

## Components Object

Содержит набор повторно используемых объектов для различных аспектов спецификации. Объекты, определенные в **Components Object** не влияют на API, если на них явно не ссылаются свойства вне объекта компонентов.

Таблица 12: Поля Components Object

Поле	Тип	Описание
schemas	Map[string, Schema Object   Reference Object]	Объект для хранения повторно используемых Schema Object.

## Пример Components Object

```

{
    "schemas": {
        "Student": {
            "type": "object",
            "required": [
                "id",

```



```

        "name",
        "age"
    ],
    "properties": {
        "id": {
            "type": "string",
            "format": "uuid"
        },
        "name": {
            "type": "string"
        },
        "age": {
            "type": "integer",
            "format": "int32"
        }
    }
}
}
}
}

```

## Пример

```

{
    "openapi": "3.0.2",
    "info": {
        "title": "example.com",
        "description": "Сервис Ориентированные Системы.  
Лабораторная работа №1. Пример спецификации,  
написанный в Swagger Editor.",
        "version": "1.0.1"
    },
    "servers": [
        {
            "url": "sos.lr1.example",
            "description": "Продуктовый сервер"
        }
    ],
    "paths": {
        "/students": {
            "get": {
                "summary": "Получение всех студентов",
                "operationId": "getAllStudents",
                "parameters": [
                    {
                        "name": "limit",
                        "in": "query",
                        "description": "Кол-во возвращаемых объектов. По умолчанию 100",

```

```

        "required": false,
        "schema": {
            "type": "integer",
            "format": "int64"
        }
    },
    {
        "name": "offset",
        "in": "query",
        "description": "Сколько объектов от начала необходимо пропустить",
        "required": false,
        "schema": {
            "type": "integer",
            "format": "int64"
        }
    }
],
"responses": {
    "200": {
        "description": "Список полученных студентов",
        "content": {
            "application/json": {
                "schema": {
                    "$ref": "#/components/schemas/Student"
                }
            }
        }
    },
    "default": {
        "description": "Непредвиденная ошибка",
        "content": {
            "application/json": {
                "schema": {
                    "$ref": "#/components/schemas/Error"
                }
            }
        }
    }
}
},
"components": {
    "schemas": {
        "Student": {
            "type": "object",
            "required": [

```



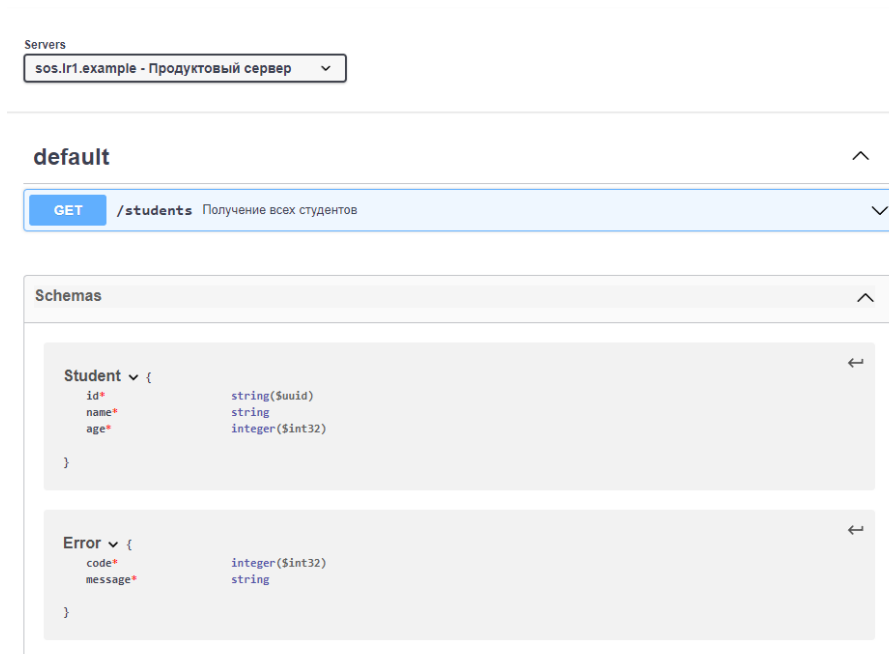


Рис. 1: Скриншот полученной документации.

## Вопросы

1. Альтернативы спецификации OpenAPI.
2. Форматы JSON и YAML.
3. Примитивные и сложные типы данных. Особенности представления, свойство-модификатор format.
4. Представление основных структур данных, согласно спецификации OpenAPI 3.0:
  - Массив. Виды массивов.
  - Ассоциативный массив/словарь - Map.
  - Уникальное множество - Set.
5. Основные блоки документа спецификации OpenAPI 3.0.

## Источники

1. A brief history of the OpenAPI Specification [Электронный ресурс] - <https://dev.to/mikeralphson/a-brief-history-of-the-openapi-specification-3g27>
2. OpenAPI Specification version 3.0.2 [Электронный ресурс]. - <https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.2.md>

3. OpenAPI Specification 3.0. examples [Электронный ресурс] - <https://github.com/OAI/OpenAPI-Specification/tree/main/examples/v3.0>
4. Курс по документированию API [Электронный ресурс] - <https://starkovden.github.io/index.html>
5. API Blueprint [Электронный ресурс] - <https://apiblueprint.org/>
6. RAML [Электронный ресурс] - <https://raml.org/>

# Лабораторная работа 2

## План аудиторного занятия

1. Краткая информация о технологиях, на которых базируется JMIХ:
  - (a) Java 17+
  - (b) Spring
  - (c) Spring Boot
  - (d) Vaadin
2. Объяснение ЛР2:
  - (a) Обзор фреймворка на котором будет вестись разработка.
  - (b) Разработка БД - **Приложение 2**.
3. Знакомство с JMIХ - воссоздание и разбор [примера](#).

## Цель работы:

1. Обосновать выбор фреймворка для разработки.
2. Произвести краткий обзор используемых модулей выбранного фреймворка.
3. Разработать базовую структуру БД.

## Задание к лабораторной работе

Обосновать выбор фреймворка для разработки и произвести краткий обзор используемых модулей. Разработать базовую структуру БД согласно теме вашей курсовой.

Студентам, планирующим вести разработку на ЯП **Java** предлагается при использовать фреймворк **JMIХ** при выполнении лабораторных и курсовых работ.

**Работу с БД можно осуществлять следующими способами:**

1. Написание собственного **SQL**-скрипта.
2. Воспользоваться любым **граф. дизайнером** для выбранной СУБД.
3. С использованием **ORM**.

Студент в праве выбрать любой тип БД, подходящий для его задачи: *реляционные, документоориентированные, графовые и тд.*

Перед разработкой БД рекомендуется ознакомиться с **приложением 2**, где описаны нормальные формы и виды связи в реляционных БД.

## Минимальные требования

Для минимального выполнения лабораторной работы необходимо:

Произвести обзор фреймворка, на котором будет вестись дальнейшая разработка - вплоть до сдачи курсовой работы.

Разработать структуру БД согласно вашему варианту курсовой работы:

- Описание каждой используемой в работе таблицы и связей между ними.
- Прикрепить диаграмму разработанной БД.
- Описать процесс создания БД в зависимости от выбранного способа:
  - При написании собственного SQL-скрипта - прикрепить его.
  - При использовании граф. дизайнера - описать основные функции дизайнера (что и куда жать, чтобы создать таблицу и тд. и тп.) *По возможности прикрепить сгенерированный SQL-скрипт создания БД.*
  - При использовании ORM - прикрепить классы, описывающие сущности с комментариями. *По возможности прикрепить сгенерированный SQL-скрипт создания БД.*

## Содержание отчета

1. Титульный лист.
2. Номер варианта курсовой работы.
3. Краткое описание разрабатываемого сервиса.
4. Описание выбранного фреймворка и его модулей
5. Описание таблиц и связей БД.
6. Диаграмма разработанной БД.
7. *По возможности:* SQL-скрипт создания БД.

## Вопросы

1. Виды связей в реляционных БД.
2. Нормальные формы в реляционных БД.
3. Вопросы по разработанным при выполнении работы таблицам в БД.

**Замечание:** вне зависимости от выбранного типа БД студенты 4-ого курса бакалавриата направления 09.03.01 обязаны уметь работать с реляционными БД :)

# Лабораторная работа 3

## Разработка и реализация REST-сервиса

### Цель работы:

1. Изучить фреймворк для разработки REST-сервиса согласно выбранному технологическому стеку.
2. Научиться проектировать конечные точки и внешние интерфейсы сервисов.
3. Научиться разделять слои работы с данными.
4. Изучить шаблон проектирования DAO.
5. Научиться документировать разработанный API.

### Задание к лабораторной работе

При помощи выбранного фреймворка разработать и реализовать REST-сервис для доступа к БД из ЛР2. Задokumentировать разработанный API.

### Минимальные требования

Для минимального выполнения лабораторной работы необходимо:

- Реализовать взаимодействие сервиса с БД.
- Реализовать разные слои работы с данными.
- Разработать точки доступа к сервису для выполнения CRUD-операций с данными:
  - Создание нового элемента.
  - Получение множества элементов.
  - Получение одного элемента по id.
  - Обновление уже имеющегося элемента.
  - Удаление элемента.
- Задokumentировать разработанный API согласно спецификации OpenAPI 3.0.

*Можно использовать подход Code First - документирование из кода.*



## Должны быть соблюдены следующие требования

- **REST**-архитектура сервиса.
- Функции работы с данными должны быть **чистыми**.
- Отображения данных должны быть **иммутабельными объектами**.

## По возможности

- Используйте иммутабельные коллекции.
- Используйте потоковую обработку данных и коллекций.

Данные ограничения являются не обязательными, но если на вашем стеке технологий они легко достижимы - то можете попробовать.

Например в Java есть странные `immutable` и `unmodifiable` реализации коллекций с которыми сложно работать, в то время как в Scala и Kotlin изначально заложены два пакета коллекций - `mutable` и `immutable`. По умолчанию в обоих языках используются `immutable` коллекции, тогда как для вызова `mutable` необходимо явно прописать импорт из нужного пакета.

## Содержание отчета

1. Титульный лист.
2. Краткое описание разрабатываемого сервиса.
3. Диаграмма БД.
4. Описание слоев разделения данных - как реализуется слой и его предназначение.
5. Описание разработанных точек доступа к сервису.
6. Скриншоты работы сервиса.
7. *При наличии front-end'a:*
  - Описание выбранной технологии.
  - Пример и описание механизма получения данных от back-end'a.
8. Скриншоты документации API согласно спецификации OpenAPI 3.0
9. Листинг только с реализацией конечных точек. Всё остальное прикреплять не надо.

## Замечания

- Наличие полноценного front-end'a не обязательно. При отсутствии front-end'a для демонстрации работы сервиса можно воспользоваться POSTMAN или другой схожей программой.
- Наличие подсистемы регистрации, авторизации и аутентификации не обязательно.

*Данная лабораторная работа показывает лишь один из подходов к разработке и реализации REST-сервисов.*

## Теория

### "Функциональное программирование"

В связи с развитием технологий, языки программирования и подходы к разработке начинают двигаться в сторону функционального стиля, перенимая хорошие практики. Например: *иммутабельные объекты, иммутабельные коллекции и чистые функции*. Данные практики упрощают разработку, поддержку кода, распараллеливание.

На первый взгляд может показаться, что постоянное переключивание данных и создание новых объектов могут создать большую нагрузку, однако современные железо и сборщики мусора позволяют делать это без видимых потерь.

#### Иммутабельный объект

Объект является **иммутабельным**, если он удовлетворяет следующим требованиям:

- Все поля класса должны быть приватными в соответствии с принципами инкапсуляции.
- Для корректного создания класса должен использоваться параметризованный конструктор.
- Для исключения возможности изменения состояния после инстанцирования, в классе не должно быть сеттеров.
- Для полей-коллекций необходимо делать глубокие копии, чтобы гарантировать их неизменность.

Если появляется необходимость "изменить" такой объект, то создается новый объект на основе данных из старого с внесением нужных изменений.

#### Иммутабельная коллекция

Коллекция является **иммутабельной**, если нельзя изменить тип и кол-во её элементов. Если появляется необходимость "изменить" такую коллекцию, то создается новая коллекция на основе данных из старой с внесением нужных изменений.

## Чистая функция

Функция является **чистой**, если она удовлетворяет следующим требованиям:

- Каждый раз вызывает одинаковый результат, когда она вызывается с тем же набором аргументов.
- Не имеет побочных эффектов (*изменение входных значений, HTTP-вызовы, запись на диск, вывод на экран и тд.*).

## Отображения данных

*Данные обозначения и имена классов лишь одна из возможных вариаций. Для данных вещей существуют некоторые стилистические правила и соглашения внутри команды разработки.*

Разделение отображений данных необходимо для достижений большей гибкости и удобства разработки.

Не все данные заполняются пользователем, некоторые данные добавляются лишь на стороне сервера или БД. Например: дата регистрации.

Также "перекладывание" данных из одной оболочки в другую позволяет изменить отображение данных в более удобный формат.

В качестве примера будет рассматриваться упрощенный **сервис для работы с пользователями**, реализованный средствами языка **Kotlin** и фреймворка **DropWizard**. *Представленные фрагменты кода являются урезанными, для упрощения восприятия и передачи сути. Например у классов отображений данных удалены аннотации для связи с таблицами в БД и аннотации документации OpenApi 3.0.*

### Таблица в БД

USERS	
id	UUID
first_name	varchar(50)
last_name	varchar(50)
age	integer
login	varchar(50)
email	varchar(255)
registration_date	date

## Сущность

**Сущность** -это прямое отображение данных из таблицы. Объекты именно этого класса участвуют в общении с БД - при записи и получении.

```
data class UserEntity(  
    val id: UUID,  
    val firstName: String,  
    val lastName: String,  
    val age: Int,  
    val login: String,  
    val email: String,  
    val registrationDate: LocalDate  
)
```

## Бизнес модель

**Бизнес модель** - это отображение данных используемое во внутренней логике сервиса.

При необходимости класс может содержать дополнительные вспомогательные методы и поля, а типы данных не должны 1 к 1 соответствовать типам сущности, но должны мапиться в них.

```
data class User(  
    val id: UUID,  
    val firstName: String,  
    val lastName: String,  
    val age: Int,  
    val login: String,  
    val email: String,  
    val registrationDate: LocalDate  
)
```

## DTO - Data Transfer Object

**DTO** - один из шаблонов проектирования, используется для передачи данных между подсистемами приложения. **DTO** - это так называемый value-object на стороне сервера, который хранит данные, используемые в слое представления.

```
data class UserDTO(  
    val firstName: String,  
    val lastName: String,  
    val age: Int,  
    val login: String,  
    val email: String  
)
```

**DTO** используется не всегда, приложение может хорошо работать и без него. Это лишь один из шаблонов проектирования. Также в зависимости от ситуации могут использоваться несколько разных типов DTO для использования в запросах/ответах или для

разных типов запросов/ответов. Всё это зависит от подхода и договоренностей в команде разработки.

В данном случае **DTO** используется в запросах к серверу и содержит лишь ту часть информации, которую может заполнить/изменить пользователь. Отсутствуют id и дата создания - пользователь не может влиять на данные параметры.

## Mapper'ы

Для перекладывания данных из сущностей в бизнес модель в DTO и обратно используются объекты **Mapper'ы**. Также для выполнения данных операций могут подключаться дополнительные библиотеки. Именно внутри функций **Mapper'a**, при перекладывании данных происходит дополнение данных или изменение их представления.

```
interface IUserMapper{
    fun model2Entity(user: User): UserEntity
    fun entity2Model(entity: UserEntity): User
    fun dto2model(id: UUID, dto: UserDTO,  registrationDate: LocalDate): UserEntity
}
```

В данном случае добавление id и даты регистрации происходит на стороне сервера в момент получения данных из DTO при создании или обновлении.

Объекты мапперы могут содержать разные методы преобразования данных между слоями, все зависит от необходимости. Наличие именно таких методов будет раскрыто дальше, при описании end-point'ов.

```
class UserMapper : IUserMapper {
    override fun model2Entity(user: User): UserEntity {
        return UserEntity(
            user.id,
            user.firstName,
            user.lastName,
            user.age,
            user.login,
            user.email,
            user.registrationDate
        )
    }

    override fun entity2Model(entity: UserEntity): User {
        return User(
            entity.id,
            entity.firstName,
            entity.lastName,
            entity.age,
            entity.login,
            entity.email,
            entity.registrationDate
        )
    }
}
```

```

    }

    override fun dto2entity(id: UUID, dto: UserDTO,
        registrationDate: LocalDate): UserEntity {
        return UserEntity(
            id,
            dto.firstName,
            dto.lastName,
            dto.age,
            dto.login,
            dto.email,
            registrationDate
        )
    }
}

```

## DAO

**DAO** - это шаблон проектирования применимый ко множеству языков программирования, большей части программного обеспечения, нуждающемуся в хранении информации, и к большей части баз данных.

**Data Access Object (DAO)** - абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения.

Простыми словами **DTO** - объект для общения в БД.

В данном примере для работы с БД используется библиотека **JDBI**. Обратите внимание на формирование **sql-запросов**, **передаваемые** и **принимаемые параметры**, а так же наличие **параметров по умолчанию**.

### Получение множества записей

При запросе на получение множества должны быть возможность задание всех параметров, а также наличие параметров по умолчанию.

```

interface UserDAO {
    @SqlQuery(
        """
            SELECT * FROM USERS
            ORDER BY <orderBy> <sort>
            LIMIT :limit
            OFFSET :offset
        """
    )
    fun findAllUser(limit: Int = 25, offset: Int = 0,
        @Define("orderBy") orderBy: String = "id",
        @Define("sort") sort: String = "ASC"): List<UserEntity>
}

```

## Получение записи по id

В качестве параметра передается id записи для поиска. В данном случае возвращается специальная обертка, которая может быть пустой или содержать объект - способ избежать работы с null.

```
@SqlQuery(
    """
        SELECT * FROM USERS
        WHERE id = :id
    """
)
fun findUserById(id: UUID): Optional<UserEntity>
```

## Добавление новой записи

В качестве параметра передается сущность для записи в БД.

```
@SqlUpdate(
    """
        INSERT INTO USERS (id, first_name, last_name, age, login,
        email, registration_date)
        VALUES (:id, :firstName, :lastName, :age, :login,
        :email, :registrationDate)
    """
)
fun insertUser(@BindBean user: UserEntity)
```

## Обновление записи

В качестве параметра передается обновленная сущность. Происходит перезапись всех параметров.

```
@SqlUpdate(
    """
        UPDATE USERS SET (first_name, last_name, age, login, email) =
        (:firstName, :lastName, :age, :login, :email)
        WHERE id = :id
    """
)
fun updateUser(@BindBean user: UserEntity)
```

## Удаление записи по id

В качестве параметра передается id записи для удаления.

```
@SqlUpdate(
    """
        DELETE FROM USERS
        WHERE id = :id
    """
)
fun deleteById(id: UUID)
}
```

## Оформление конечных точек

Обычно все методы конечных точек относящиеся к одной сущности описывают в одном месте - класс контроллера или ресурса или тд. и тп.

В данном примере часть пути уже определена на уровне класса:

```
@Path("/user")
class UserResource(
    private val userMapperImpl: IUserMapper,
    private val userDao: UserDao,
)
```

## Создание новой записи

Для создания записи использует HTTP-метод **POST**.

Входные параметры:

- **DTO** с данными введенными пользователем - **UserDTO**.

Выходные значения:

- созданная бизнес модель - **User**.

```
@POST
fun createNewUser(@Valid newUser: UserDTO): Response {
    val userEntity = userMapperImpl.dto2entity(UUID.randomUUID(),
        newUser, LocalDate.now())

    try {
        userDao.insertUser(userEntity)
    } catch (exception: UnableToExecuteStatementException) {
        return Response.status(Response.Status.CONFLICT).build()
    }

    return Response.ok(userMapperImpl.entity2model(userEntity)).build()
}
```



## Получение множества записей

Для получения записи используется HTTP-метод **GET**.

Входные параметры:

- **limit** - кол-во записи в выборке
- **offset** - отступ
- **orderBy** - поля для сортировки
- **sort** - порядок сортировки

Выходные параметры:

- коллекция бизнес моделей - `List<User>`

@GET

```
fun getAllUsers(limit: IntParam, offset: IntParam,
orderBy: String, sort: String): Response {
    val userList: List<UserEntity> = userDao.findAllUser(
        limit.get(), offset.get(), orderBy, sort)

    return Response.ok(
        userList.map user -> userMapperImpl.entity2model(user)
    ).build()
}
```

## Получение записи по id

Для получения записи используется HTTP-метод **GET**. Путь доопределяется параметром {id}

Входные параметры:

- **id** - идентификатор записи, получаемый из **пути запроса**.

Выходные параметры:

- бизнес модель найденной записи
- статус NOT\_FOUND если такой записи нет

@GET

@Path("/{id}")

```
fun getUser(@PathParam("id") id: UUID): Response {
    val user = userDao.findUserById(id)

    return if (user.isEmpty) {
        Response.status(Response.Status.NOT_FOUND).build()
    }
    else Response.ok(userMapperImpl.entity2model(user.get())).build()
}
```

## Обновление записи по id

Для обновления записи используется HTTP-метод **PUT**. Путь доопределяется параметром {id}.

Перед обновлением записи надо проверить существует ли запись с таким id.

Входные параметры:

- **id** - идентификатор записи, получаемый из пути запроса
- **DTO** - с обновленными данными

Выходные параметры:

- если запись **не найдена** - статус NOT\_FOUND
- если запись **найденa** - бизнес модель с обновленными параметрами

@PUT

@Path("id")

```
fun updateUser(@PathParam("id") id: UUID, @Valid userDto: UserDTO): Response {  
    val userEntityOpt = userDao.findUserById(id)  
  
    if (userEntityOpt.isEmpty) {  
        return Response.status(Response.Status.NOT_FOUND).build()  
    }  
  
    val userEntity = userMapperImpl.dto2entity(  
        userEntityOpt.get().id,  
        userDto,  
        userEntityOpt.get().registrationDate  
    )  
    try {  
        userDao.updateUser(userEntity)  
    }  
    catch (exception: UnableToExecuteStatementException) {  
        return Response.status(Response.Status.CONFLICT).build()  
    }  
  
    return Response.ok(userMapperImpl.entity2model(userEntity)).build()  
}
```

## Удаление записи по id

Для обновления записи используется HTTP-метод **DELETE**. Путь доопределяется параметром {id}.

Перед удаление записи надо проверить существует ли запись с таким id.

Входные параметры:

- **id** - идентификатор записи, получаемый из пути

Выходные параметры:

- если запись **не найдена** - статус NOT\_FOUND
- если запись **найденa** - бизнес модель удаленной записи

```
@DELETE
@Path("id")
fun deleteUser(@PathParam("id") id: UUID): Response {
    val userEntityOpt = userDao.findUserById(id)

    if (userEntityOpt.isEmpty) {
        return Response.status(Response.Status.NOT_FOUND).build()
    }

    userDao.deleteById(id)
    return Response.ok(userMapperImpl.entity2model(userEntityOpt.get())).build()
}
```

## Контрольные вопросы

1. Определение иммутабельного объекта.
2. Определение чистой функции.
3. Виды отображений данных и их предназначение.
4. Шаблон DTO.
5. Шаблон DAO.

## Источники

1. Неизменяемый объект [Электронный ресурс] - <https://ru.wikipedia.org/wiki/>
2. DTO [Электронный ресурс] - <https://ru.wikipedia.org/wiki/DTO>
3. DAO [Электронный ресурс] - [https://ru.wikipedia.org/wiki/Data\\_Access\\_Object](https://ru.wikipedia.org/wiki/Data_Access_Object)
4. Андрей Беляев - DTO: живи быстро, гори ярко [Электронный ресурс] - <https://youtu.be/gD8xUkZW1GU?si=AXUTMVjYpQ6gRh5V>

# Лабораторная работа 4

## Модуль генерации отчетов

### Цель работы:

1. Изучить фреймворк/библиотеку для разработки генерации отчета согласно выбранному технологическому стеку.

### Задание к лабораторной работе

При помощи выбранного фреймворка/библиотеки реализовать составление отчета в одном из допустимых форматов: pdf, doc, docx, odt.

### Минимальные требования

Для минимального выполнения лабораторной работы необходимо:

- Реализовать генерацию отчета по данным из вашей работы.
- Реализовать точку доступа для обращения к модулю генерации отчета.

### Содержание отчета

1. Титульный лист.
2. Краткое описание разрабатываемого сервиса.
3. Обоснование выбора фреймворка/библиотеки для генерации отчета.
4. Краткое описание выбранного фреймворка/библиотеки.
5. Описание генерируемого отчета.
6. Листинг связанный с подключением и использованием фреймворка/библиотеки генерации отчета.
7. Приложение - сгенерированный отчет.

## JMIX

Для студентов выполняющих работы по данному курсу при помощи платформы для разработки **JMIX** предлагается воспользоваться свободно распространяемым модулем **Reports**.

## Пояснения

### Что должно быть в сгенерированном отчете?

Отчет может содержать все что угодно связанное с тематикой разработанного сервиса: данные, сводки, расчеты, графики. В самом худшем случае выводится табличка с сущностями из БД.

### Если вам не нужен данный модуль в ВКР.

Данный модуль обязателен для сдачи ЛР4 и не обязателен для Курсовой работы. Если данный модуль не предусмотрен в вашей ВКР, то при реализации предусмотрите возможность легкого удаления данного модуля (*отдельный класс и контроллер/отдельная ветка GIT*).

# Структура пояснительной записки к курсовой работе

## 1. Титульный лист

## 2. Содержание

## 3. Список сокращений

## 4. Список терминов

## 5. Введение

- Актуальность сервис-ориентированной архитектуры при разработке распределенных программных систем в целом (1,2 абзаца)
- Дать краткую характеристику решаемой задачи, описать особенности, которые обуславливают выбор ее реализации как распределенной программной системы.
- Цель курсовой работы.
- Задачи, решаемые в ходе курсовой работы.

## 6. Постановка задачи

- Приводится текст задачи с указанием варианта.

## 7. Анализ предметной области и проектирование API (ЛР 2)

### (а) Объектно-ориентированная модель ресурсов

- Разработка множества сущностей предметной области и связей между ними. *Сущности и структура БД.*
- Разработка объектно-ориентированной модели ресурсов (с построением диаграммы классов и диаграммы взаимодействия) *(на диаграммах необходимо отразить подмножество классов, к объектам которых необходимо предоставлять доступ из внешних систем)*

### (b) Проектирование REST-API (ЛР 1,3) Разработка интерфейса доступа к ресурсам с обоснованием связи между объектно-ориентированной моделью и множеством URI:

- Описание ресурсов
- Описание групп точек доступа (укажите, по каким признакам выполнялось группирование точек доступа)

- Описание отдельных точек доступа (методы, параметры)
  - Документирование REST-API согласно спецификации Open API 3.0. *(несколько скриншотов из Swagger UI разместить в приложениях)*
8. Реализация сервис-ориентированной системы (платформа и средства разработки выбираются индивидуально)
- (a) Аналитический обзор платформ и средств построения сервис-ориентированных систем (ЛР 2)
- Привести краткую характеристику 3-4 вариантов
  - Описать преимущества и недостатки каждого инструмента
  - Обосновать выбор технологического стека для решения задачи.
- (b) Компонентная модель сервис-ориентированной системы (ЛР 3)
- Описание компонентов системы и их интерфейсов:
    - Внутренних, реализующих функции, к которым нет прямого доступа из внешних систем
    - Внешних
  - Описание UML-диаграммы компонентов (разместить в приложении)
  - Описание групп компонентов по программным модулям и вычислительным узлам (укажите, по каким признакам выполнялось группирование компонентов)
  - Описание UML-диаграммы размещения (разместить в приложении)
- (c) Настройка программного проекта
- Опишите перечень модулей, используемых для разработки (название модуля, версия, краткая характеристика), в приложении приведите секцию зависимостей конфигурационных файлов maven, gradle или иных систем автоматизации сборки проектов)

## 9. Список источников

# Приложение 1. Академическая лицензия JMX

**JMX** - высокоуровневый open source фреймворк, позволяющий в разы ускорить разработку enterprise-приложений на ЯП Java.

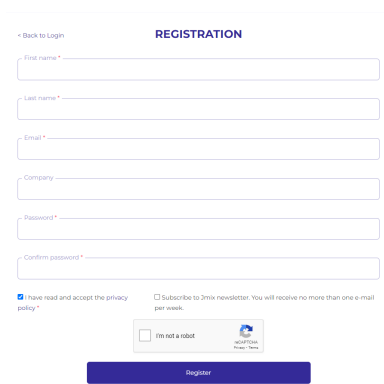
Версия **JMX 2.0** базируется на следующих технологиях:

1. Java 17+
2. Spring Boot 3.1
3. Spring Framework 6.0
4. Spring Security 6.1
5. Vaadin 24.0
6. EclipseLink 4.0
7. Flowable 7.0

В бесплатной версии **отсутствует граф. дизайнер пользовательского интерфейса**, а также урезан функционал.

## Регистрация

Для получения академической лицензии необходимо [завести аккаунт](#).



The screenshot shows a web registration form titled "REGISTRATION". At the top left is a link "< Back to Login". The form contains several input fields: "First name \*", "Last name \*", "Email \*", "Company", "Password \*", and "Confirm password \*". Below these fields are two checkboxes: "I have read and accept the privacy policy \*" (which is checked) and "Subscribe to Jmx newsletter. You will receive no more than one e-mail per week." (which is unchecked). At the bottom left is a checkbox "I'm not a robot" next to a CAPTCHA image. A blue "register" button is at the bottom right.

Рис. 2: Регистрации на сайте JMX.



Также можно войти при помощи сквозной аутентификации следующих сервисов:

1. Google
2. GitHub
3. Facebook

## Получение лицензии

После входа на сайт необходимо перейти к "[покупке](#)" лицензии. Выбираете академическую версию, соглашаетесь с лицензионным соглашением (на академической версии нельзя вести коммерческую и проприетарную разработку), переходите к оформлению лицензии.

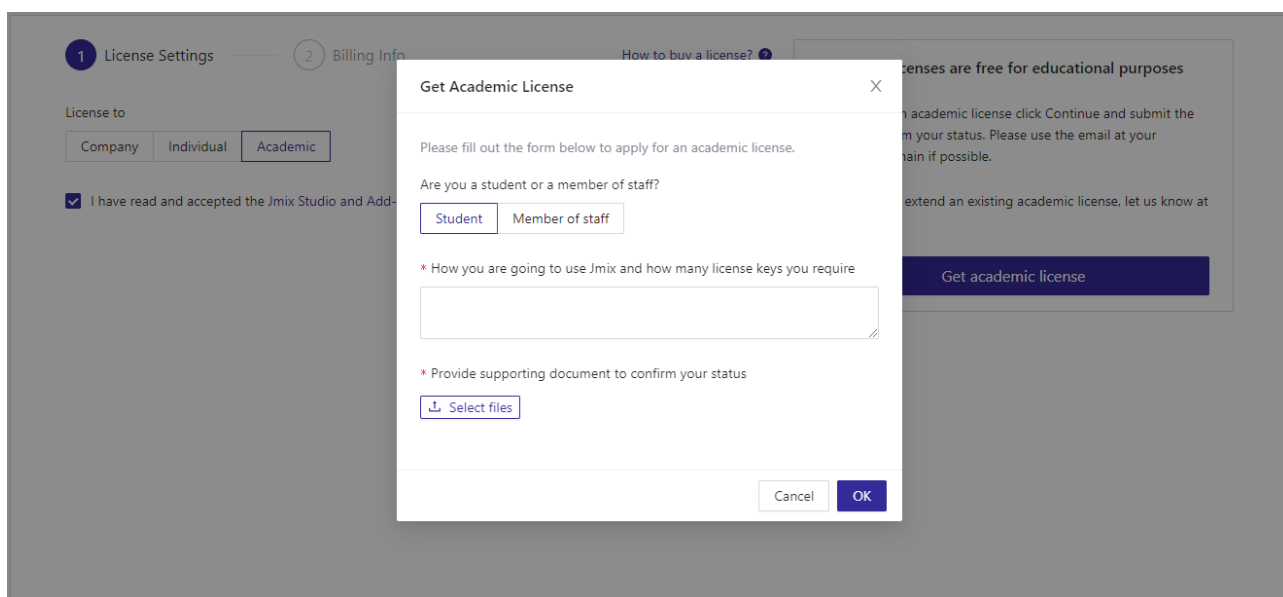


Рис. 3:

В текстовом поле указываете, что вы проходите обучение на программного инженера в ВУЗе и вам нужно 1 лицензия. Прикрепляете документы подтверждающие, что вы студент НГТУ. Лучше всего прикрепить скан студенческого билета и справки из дирекции в форма pdf.

*Заполнить текстовое поле можно на русском языке (Haulmont российская компания).*

## Ждем..

Рассмотрение заявки на выдачу академической лицензии может занять до нескольких недель.

## Приложение 2. Реляционные базы данных

В данном приложении будут рассмотрены **нормальные формы** и **виды связей** таблиц в реляционных базах данных.

### Нормальные формы

**Нормальная форма** — свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, потенциально приводящей к логически ошибочным результатам выборки или изменения данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение.

**Замечание:** пусть данные нормальные формы описаны только для **реляционных бд**, но в той или иной степени они являются справедливыми и для *постреляционных баз данных*: документоориентированных, графовых и тд.

*Теория выделяет 5 нормальных форм, но в практике есть только 3. (Мисевич П.В. - из курса по БД.)*

В интернете можно найти множество нормальных форм (намного больше 5-ти), в том числе и именных, но на практике достигаются лишь 3.

*Основные три нормальные формы у Вас получатся сами собой, если Вы адекватный человек. (Сапожников В.О.)*

Процесс проектирования БД с использованием метода НФ является итерационным и заключается в последовательном переводе отношения из 1НФ в НФ более высокого порядка по определенным правилам. Каждая следующая НФ ограничивается определенным типом функциональных зависимостей и устранением соответствующих аномалий при выполнении операций над отношениями БД, а также сохранении свойств предшествующих НФ.

1 НФ = 4 условия.

2 НФ = 1 НФ + доп.условия

3 НФ = 2 НФ + доп.условия

## 1 Нормальная форма

- Каждая запись должна содержать в своих полях только атомарные значения.
- Не одно из ключевых полей таблицы не пусто.
- Таблица не должна иметь дубликатов.
- В таблице нет повторяющихся групп полей.

**Замечание:** применяется к каждой таблице проекта!

### Пример

Фирма	Модели
BMW	M5, X5M, M1
Nissan	GT-R

Нарушение нормализации 1НФ происходит в моделях BMW, т.к. в одной ячейке содержится список из 3 элементов: M5, X5M, M1, т.е. он не является атомарным. Преобразуем таблицу к 1НФ:

Фирма	Модели
BMW	M5
BMW	X5M
BMW	M1
Nissan	GT-R

## 2 Нормальная форма

- Каждая таблица находит в 1НФ.
- Каждый не ключевой атрибут неприводимо зависит от Первичного Ключа(ПК). **Неприводимость** означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость.

### Пример

Модель	Фирма	Цена	Скидка
M5	BMW	5500000	5%
X5M	BMW	6000000	5%
M1	BMW	2500000	5%
GT-R	Nissan	5000000	10%

Таблица находится в первой нормальной форме, но не во второй. Цена машины зависит от модели и фирмы. Скидка зависит от фирмы, то есть зависимость от первичного ключа неполная. Исправляется это путем декомпозиции на два отношения, в которых не ключевые атрибуты зависят от ПК.

Модель	Фирма	Цена
M5	BMW	5500000
X5M	BMW	6000000
M1	BMW	2500000
GT-R	Nissan	5000000

Фирма	Скидка
BMW	5%
Nissan	10%

### 3 Нормальная форма

- Каждая таблица находит в 2НФ.
- Любое не ключевое поле таблицы не определяется с помощью не первичных ключей.  
*По-другому:* Каждый не ключевой атрибут нетранзитивно зависит от первичного ключа.

#### Пример

Модель	Магазин	Телефон
BMW	Риал-авто	87-33-98
Audi	Риал-авто	87-33-98
Nissan	Некст-Авто	94-54-12

Таблица находится во 2НФ, но не в 3НФ.

В отношении атрибут «Модель» является первичным ключом. Личных телефонов у автомобилей нет, и телефон зависит исключительно от магазина.

Таким образом, в отношении существуют следующие функциональные зависимости: Модель  $\rightarrow$  Магазин, Магазин  $\rightarrow$  Телефон, Модель  $\rightarrow$  Телефон. Зависимость Модель  $\rightarrow$  Телефон является транзитивной, следовательно, отношение не находится в 3НФ.

В результате разделения исходного отношения получаются два отношения, находящиеся в 3НФ:

Магазин	Телефон
Риал-авто	87-33-98
Некст-Авто	94-54-12

Модель	Магазин
BMW	Риал-авто
Audi	Риал-авто
Nissan	Некст-Авто

#### Дополнение

В современных реалиях нормализация БД не является критерием чистоты проекта и часто для того, чтобы проект был более технологически понятен заказчику, производят денормализацию БД. Также в некоторых случаях денормализация БД проводится для оптимизации времени запросов к таблицам.

## Связи таблиц в БД

Связи создаются с помощью внешних ключей (foreign key) - **FK**.

**Внешний ключ** — это атрибут или набор атрибутов, которые ссылаются на **primary key** - **РК** или unique другой таблицы. Другими словами, это что-то вроде указателя на строку другой таблицы.

### Виды связей:

n:n - многие ко многим

1:n - один ко многим

— с обязательной связью

— с необязательной связью

1:1 — с обязательной связью

— с необязательной связью

### Многие ко многим

Представим, что нам нужно написать БД, которая будет хранить работников ИТ-компании. При этом существует некий стандартный набор должностей. При этом:

- Работник может иметь одну и более должностей. Например, некий работник может быть и админом, и программистом.
- Должность может «владеть» одним и более работников. Например, админами является определенный набор работников. Другими словами, к админам относятся некие работники.

Работников представляет таблица «Employee» (id, имя, возраст),

Employee
id
name
age

должности представляет таблица «Position» (id и название должности).

Position
id
name

Как видно, обе эти таблицы связаны между собой по правилу многие ко многим: каждому работнику соответствует одна и больше должностей (многие должности), каждой должности соответствует один и больше работников (многие работники).

## Реализация

Мы уже имеем две таблицы, описывающие работника и профессию. Теперь нам нужно установить между ними связь многие ко многим. Для реализации такой связи нам нужен некий посредник между таблицами «Employee» и «Position». В нашем случае это будет некая таблица «EmployeesPositions» (работники и должности). Эта таблица-посредник связывает между собой работника и должность следующим образом:

EmployeesPositions
employee_id
position_id

На рисунке ниже представлена диаграмма, где реализована связь n:n.

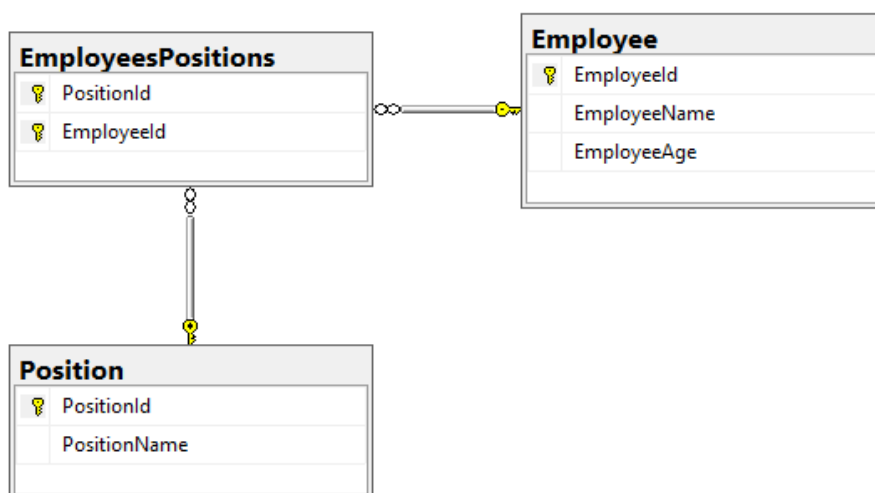


Рис. 4: Диаграмма n:n

**Вывод:** для реализации связи многие ко многим нам нужен некий посредник между двумя рассматриваемыми таблицами. Он должен хранить два внешних ключа, первый из которых ссылается на первую таблицу, а второй — на вторую.

## Один ко многим

Предположим, нам нужно реализовать некую БД, которая ведет учет данных о пользователях. У пользователя есть: имя, фамилия, возраст, номера телефонов. При этом у каждого пользователя может быть от одного и больше номеров телефонов (многие номера телефонов).

В этом случае мы наблюдаем следующее: пользователь может иметь многие номера телефонов, но нельзя сказать, что номеру телефона принадлежит определенный пользователь.

Другими словами, телефон принадлежит только одному пользователю. А пользователю могут принадлежать 1 и более телефонов (многие).

Как мы видим, это отношение **один ко многим**.

## Реализация

Пользователей будет представлять некая таблица «Person» (id, имя, фамилия, возраст),

Person
id
first_name
second_name
age

номера телефонов будет представлять таблица «Phone».

Phone
id
person_id
phone_number

На рисунке ниже представлена диаграмма, где реализована связь 1:n.

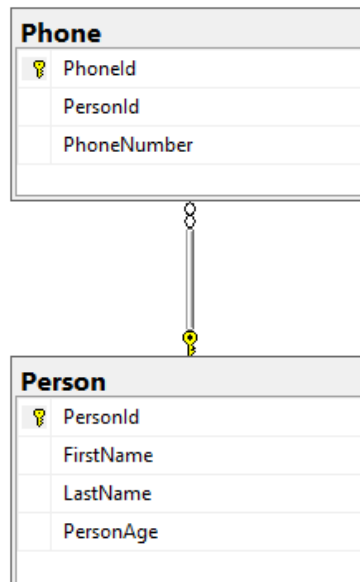


Рис. 5: Диаграмма 1:n

## Примеры обязательных и не обязательных связей

- Один ко многим с обязательной связью:

К одному полку относятся многие бойцы. Один боец относится только к одному полку. Обратите внимание, что любой солдат обязательно принадлежит к одному полку, а полк не может существовать без солдат.

- Один ко многим с необязательной связью:

На планете Земля живут все люди. Каждый человек живет только на Земле. При этом планета может существовать и без человечества. Соответственно, нахождение нас на Земле не является обязательным

**Вывод:** наша таблица Phone хранит всего один внешний ключ. Он ссылается на некоего пользователя (на строку из таблицы Person). Таким образом, мы как бы говорим: «этот пользователь является владельцем данного телефона». Другими словами, телефон знает id своего владельца.

## Один к одному

Связь один к одному является редкой и как ни странно, самой сложной. Сложной с точки зрения обоснования своего использования.

В общем виде две таблицы со связей 1:1 представляют собой *"более крупную, разделенную таблицу"*.

### Сценарии использования связи 1:1

- Не всегда есть необходимость в полных данных сущности, описываемой в таблице.
- Какое-то свойство представляет собой сильно разреженную таблицу.

Представим, что на работе вам дали задание написать БД для учета всех работников для HR. Начальник уверял, что компании нужно знать только об имени, возрасте и телефоне работника. Вы разработали такую БД и поместили в нее всю 1000 работников компании. И тут начальник говорит, что им зачем-то нужно знать о том, является ли работник инвалидом или нет. Наиболее простое, что приходит в голову — это добавить новый столбец типа bool в вашу таблицу. Но это слишком долго вписывать 1000 значений и ведь true вы будете вписывать намного реже, чем false (2% будут true, например).

### Реализация

Более простым решением будет создать новую таблицу, назовем ее «DisabledEmployee». Она будет выглядеть так:

DisabledEmployee
disabled_person_id
employee_id

Но это еще не связь один к одному. Дело в том, что в такую таблицу работник может быть вписан более одного раза, соответственно, мы получили отношение один ко многим: работник может быть несколько раз инвалидом. Нужно сделать так, чтобы работник мог быть вписан в таблицу только один раз, соответственно, мог быть инвалидом только один раз. Для этого нам нужно указать, что столбец EmployeeId может хранить только уникальные значения. Нам нужно просто наложить на столбец EmployeeId ограничение unique. Это ограничение сообщает, что атрибут может принимать только уникальные значения.

Выполнив это мы получили связь один к одному.

На рисунке ниже представлена диаграмма, где реализована связь 1:1.



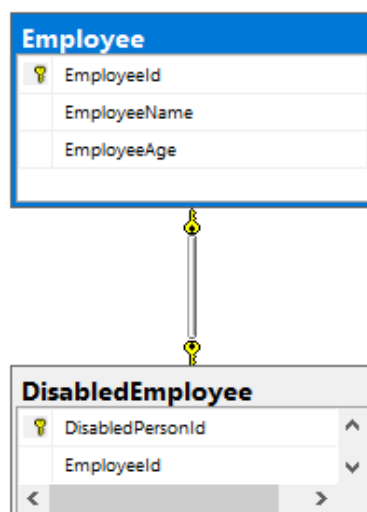


Рис. 6:

Таблица DisabledEmployee имеет атрибут EmployeeId, что является внешним ключом. Он ссылается на атрибут EmployeeId таблицы Employee. Кроме того, этот атрибут имеет ограничение unique, что говорит о том, что в него могут быть записаны только уникальные значения. Соответственно, работник может быть записан в эту таблицу не более одного раза.

### Примеры обязательных и не обязательных связей

- Один к одному с обязательной связью:

У одного гражданина определенной страны обязательно есть только один паспорт этой страны. У одного паспорта есть только один владелец.

- Один к одному с необязательной связью:

У одной страны может быть только одна конституция. Одна конституция принадлежит только одной стране. Но конституция не является обязательной. У страны она может быть, а может и не быть, как, например, у Израиля и Великобритании.

**Вывод:** Можно сказать, что отношение один к одному — это разделение одной и той же таблицы на две.

### Источники

- Курс по базам данных для студентов специальности 09.03.01 «Информатика и вычислительная техника» - Мисевич П.В.
- Хабр: нормализация отношений. Шесть нормальных форм. [Электронный ресурс] - <https://habr.com/ru/articles/254773/>
- Хабр: Связи между таблицами базы данных. [Электронный ресурс] - <https://habr.com/ru/articles/488054/>