Санкт-Петербургский государственный университет

Петров Владимир Сергеевич

Выпускная квалификационная работа

Реализация динамических атрибутов в сервисе хранения номеров

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5162.2020 «Технологии программирования»*

Научный руководитель:
старший преподаватель кафедры СП, к. т. н. Ю.В. Литвинов

Консультант:
руководитель группы, ООО «Яндекс.Технологии» К.С. Сазонов

Рецензент:
руководитель группы, ООО «Яндекс.Технологии», к. т. н. Г.А. Кириченко

Санкт-Петербург
2024

Saint Petersburg State University

Vladimir Petrov

Bachelor's Thesis

Implementation of dynamic attributes in number storage service

Education level: bachelor

Speciality *02.03.03 «Software and Administration of Information Systems»*

Programme *CB.5162.2020 «Programming Technologies»*

Scientific supervisor:
C.Sc., senior lecturer Y.V. Litvinov

Consultant:
Group leader at "Yandex.Technologies" K.S. Sazonov

Reviewer:
Group leader at "Yandex.Technologies", C.Sc. G.A. Kirichenko

Saint Petersburg
2024

Оглавление

1. Введение

«Единая база номеров» (ЕБН) — внутренний сервис компании «Яндекс» для хранения данных о подключенных телефонных номерах и информации об их физических и логических подключениях. Данный сервис используется инженерами компании для аудита номеров и автоматизации рабочих процессов телефонии.

ЕБН хранит информацию о различных объектах реального мира. Каждый объект имеет свой тип (телефонный номер, договор, партнер и т.д.), который определяет атрибуты, описывающие свойства данного объекта.

В последнее время значительно увеличился рост количества задач по добавлению новых атрибутов к существующим типам объектов приложения. Такие задачи решались разработчиками данного сервиса путем добавления атрибутов в схему базы данных, расширения исходного кода и вывода добавленных атрибутов в API[1] сервиса. Но таких задач стало возникать очень много, и соответствующая работа занимает у разработчиков ощутимое количество времени. Усугубляет это положение тот факт, что данные задачи будут и дальше возникать достаточно активно. Связано это с тем, что сервис был запущен относительно недавно и сейчас его начинают подключать в процессы автоматического конфигурирования телефонных систем. При этом сервис используется как инвентарная система, по данным которого конфигурируются другие устройства. Количество и набор атрибутов, используемых в этом процессе, зависит от типа устройства. Из-за этого при подключении новых устройств данная проблема будет возникать снова и снова.

Таким образом была поставлена задача по реализации подсистемы автоматизированного добавления новых атрибутов в данный сервис. Суть задачи заключается в том, чтобы появилась возможность добавлять новые атрибуты и управлять их значениями через API приложения без ручной доработки исходного кода.

[1]API (application programming interface) – программный интерфейс, то есть описание способов взаимодействия одной компьютерной программы с другими.

2. Постановка задачи

Целью работы является упрощение добавления новых атрибутов в приложение для администраторов и разработчиков сервиса Яндекс “Единая база номеров”.
Для её достижения были поставлены следующие задачи.

- Выполнить обзор решений для реализации динамических атрибутов.
- Спроектировать подсистему автоматизированного добавления новых атрибутов, включая обработчики, схему хранения и валидации данных динамических атрибутов.
- Реализовать спроектированную подсистему.
- Расширить существующие обработчики динамическими атрибутами.
- Написать пользовательскую документацию для администраторов ЕБН.
- Выполнить апробацию разработанной подсистемы, получив список атрибутов и автоматически добавив их в сервис.

3. Обзор

3.1. Технические детали проекта ЕБН

ЕБН — это REST API[2] сервис, написанный на языке Go и использующий в качестве базы данных PostgreSQL[3].

Данные в ЕБН представлены в виде некоторого набора сущностей, связь между которыми образует дерево объектов. Каждое такое дерево несет информацию о том, какие номера подключались компанией, с каким оператором связи заключались договора, кем заключались договора, описание договоров и сетевых особенностей подключений данных номеров.

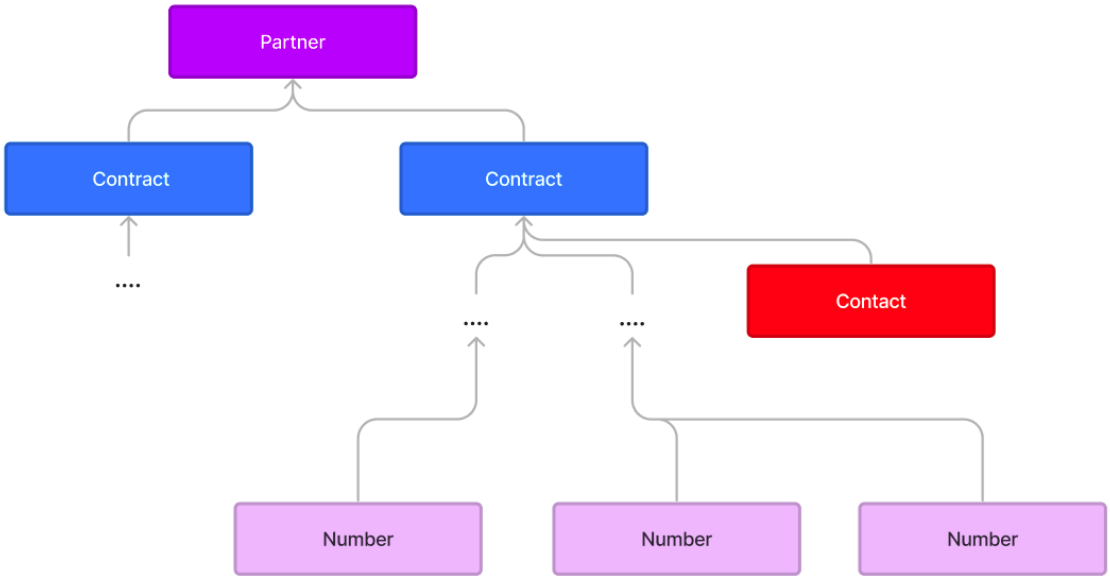

Рис. 1: Иллюстрация взаимосвязей сущностей в ЕБН.

Всего в ЕБН насчитывается 12 типов сущностей, входящих в эти деревья взаимосвязей, некоторые из них представлены на рисунке 1 (Number, Partner и т.д.).

[2]REST API – архитектурный стиль разработки API веб-приложений или компонентов распределённого приложения, используя протокол HTTP.

[3]PostgreSQL – свободная объектно-реляционная система управления базами данных (СУБД).

Данные об этих деревьях хранятся в базе данных в виде нескольких таблиц.

Таблица **object_tree** содержит информацию о структуре всех деревьев. Каждая запись в этой таблице содержит следующие атрибуты:

- tree_id — идентификатор дерева объектов;
- object_id — идентификатор объекта;
- object_type_id — внешний ключ на таблицу object_type;
- parent_id — идентификатор объекта, являющегося родителем для данного объекта в этом дереве.

Таблица **object_type** хранит информацию о существующих типах сущностей. В ней можно выделить два атрибута, которые представляют идентификатор типа объекта и его название.

Для каждого типа сущности создана своя таблица, которая хранит атрибуты, описывающие объекты данного типа. Эта таблица связана с object_tree внешним ключом по object_id.

Основные операции, предоставляемые ЕБН, являются CRUD[4] для всех типов сущностей, поиск, массовая загрузка номеров.

Для поиска указывается шаблон строки, который в конечном итоге передается в аргументы хранимой процедуры в базе данных. Эта процедура обходит записи таблиц для всех типов сущностей, проверяя, удовлетворяет ли хотя бы один атрибут данному шаблону. Если атрибут записи удовлетворяет данному шаблону, то данный объект входит в результат поиска.

Массовая загрузка номеров предназначена для загрузки большого числа номеров из файла формата csv в базу данных. Данный файл с помощью специального обработчика загружается в базу данных в сыром формате. Далее демон-процессы кусками берут данные о номерах

[4]CRUD — акроним, обозначающий четыре базовые функции, используемые при работе с базами данных и объектами приложений: создание (create), чтение (read), модификация (update), удаление (delete).

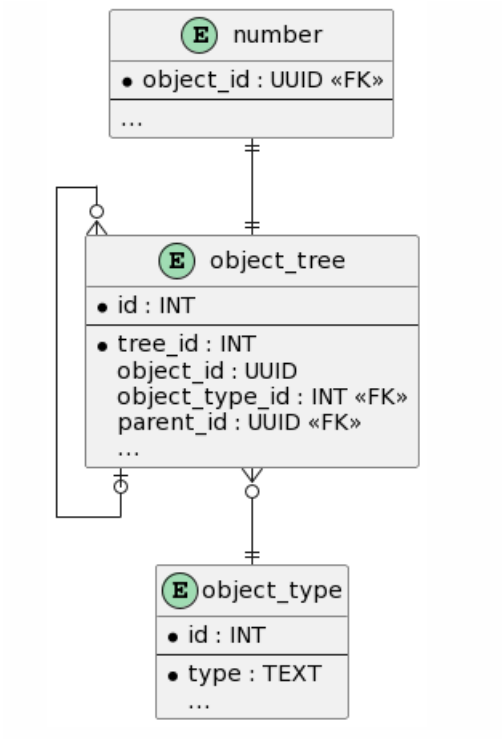

Рис. 2: Представление данных о деревьях в базе данных с таблицей number в качестве пример таблицы определенного типа сущности.

из файла, валидируют их и загружают в таблицу номеров записи с полученными данными.

Архитектуру приложения можно разделить на три слоя.

- Слой обработчиков — слой для сбора входящих данных.
- Слой работы с базой данных — слой, реализующий шаблон «репозиторий»[5]. Его основной задачей является преобразовать полученные данные с предыдущего слоя в формат SQL-запроса. Данный SQL-запрос вызывает хранимую процедуру в базе данных, передавая туда полученные данные в качестве аргументов.
- Слой хранимых процедур в базе данных — выполняет всю бизнес-логику, манипулируя входными данными и данными из таблиц.

[5]Шаблон «репозиторий» – это шаблон проектирования программного обеспечения, который действует как промежуточный уровень между бизнес-логикой приложения и хранилищем данных.

Как можно заметить, весь бизнес-код работает в хранимых процедурах, то есть бизнес-логики, написанной на SQL, довольно много.

3.2. Способы реализации динамических атрибутов

Важно отметить, что выбираемый подход реализации динамических атрибутов должен не только поддерживать хранение динамического множества пар ключ-значение. Подход также должен позволять задавать четкую структуру того, какие атрибуты есть у сущности, какие у них типы и ограничения областей значений. По этой структуре будут проводиться проверки соблюдения всех условий у каждого объекта сущности. Поэтому при выборе подхода важно понимать следующее.

- Как хранить пары атрибут-значение для объекта сущности.
- Как описывается и хранится схема атрибутов для каждой сущности (какие атрибуты доступны для сущности, какие типы и ограничения для них).
- Как проверять, что атрибуты и их значения для объекта удовлетворяют схеме атрибутов.

3.2.1. Использование HStore

HStore [8] — это расширение для PostgreSQL, разработанное в 2003 году. Поставляется как отдельный тип данных, представляющий хранилище ключ-значение. Ключи и значения хранятся в виде строк, есть поддержка индексов. Несмотря на то, что данный тип данных позволяет хранить динамическое множество атрибутов, для него не существует стандартного подхода по описанию его структуры. Можно использовать подход, аналогичный методу EAV [3.2.2]. Но так как значения хранятся в виде строк, то база данных никак не сможет помочь в проверке корректности типов данных у атрибутов. Поэтому это будет только усложнение данного метода. Остается использовать стандартные инструменты описания структуры документов, примерами которых могут

служить JSON-схема [1] или XSD[6]. Перед внесением новых данных в базу данных, атрибуты придется представить в виде соответствущего документа, провалидировав его с помощью специальных валидаторов.

3.2.2. EAV

Подход EAV (Entity Attribute Value) позволяет построить информационную систему, схема данных которой управляется через пользовательский интерфейс без доработки исходного кода приложения. Особенно он распространился в медицине, где изменения в схеме данных происходят часто [13].

Данный подход использует несколько управляющих таблиц, описывающих сущности и их атрибуты вместо того, чтобы зафиксировать их в схеме явно. Значения атрибутов хранятся в специальной таблице, в которой указывается идентификатор объекта, идентификатор атрибута сущности и значение данного атрибута.

На рисунке 3 представлен пример организации схемы данных в модели EAV. Таблицы EntityType и EntityAttribute описывают, какие есть сущности и какие у них могут быть атрибуты. Таблица EntityAttributeValue описывает значения атрибутов у объектов. В примере все значения хранятся в текстовом виде, но обычно для каждого типа данных создают свою таблицу [11] EntityAttributeValue, чтобы за типизацией значений атрибутов следила СУБД. Таким образом к этой схеме нужно добавить еще несколько таблиц для каждого типа данных. Также для каждой такой таблицы значений нужно создавать триггеры[7], которые проверяют, что entity_id может иметь атрибут entity_attribute_id и что его значение валидно. Также для реализации возможности ограничивать значения атрибутов нужно расширять таблицу EntityAttribute соответствующим опциональным атрибутом (например regexp), который описывает, какое ограничение применяется, и с помощью триггера проверять, что ограничения выполняются.

[6]XSD (XML Schema Definition) схема – это документ, который определяет структуру и ограничения для XML-данных.

[7]Триггер – хранимая процедура особого типа, исполнение которой обусловлено действием по модификации данных.

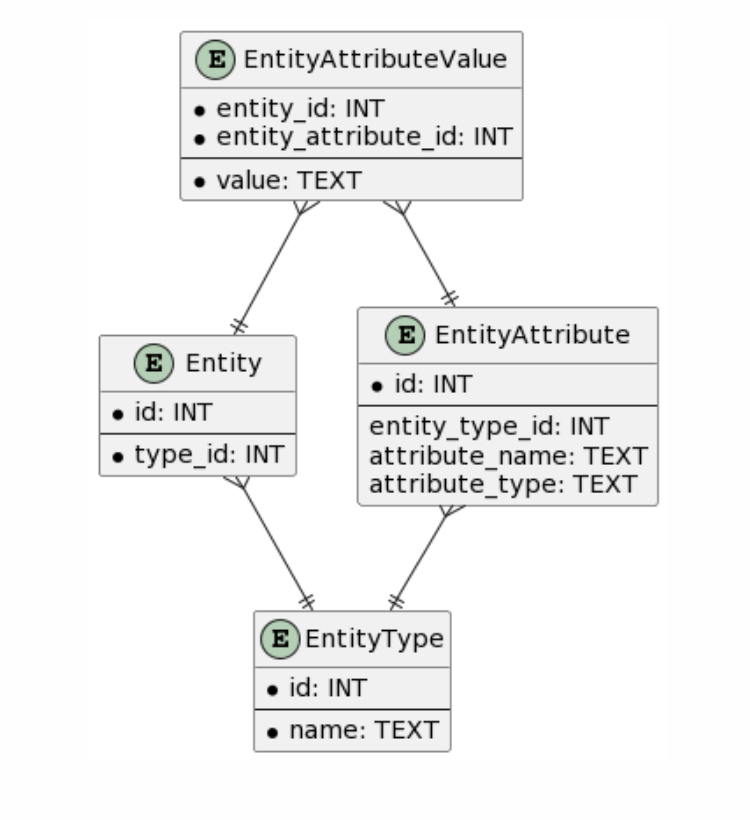

Рис. 3: Пример организации таблиц в модели EAV.

3.2.3. Использование JSONB и JSON-схемы

JSONB [7] — тип данных в PostgreSQL, представляющий JSON, хранящийся в бинарном формате, что обеспечивает его эффективную работу. Представленный тип данных в последнее время активно развивается в PostgreSQL, у него есть поддержка ряда функций для обработки, создания JSONB-данных и построения к ним эффективных запросов.

JSONB можно использовать, как хранилище «ключ-значение» для динамических атрибутов. В качестве описания схемы атрибутов у сущностей будет разумно использовать JSON-схему [1] — также JSON-документ, который описывает структуру JSON-документа. С помощью данной схемы можно будет не только описать, какие у сущности есть атрибуты и какие у них типы, но и еще различные ограничения на значения данных атрибутов. Спецификация JSON-схемы предполагает валидацию JSON-документа по ней. Существуют валидаторы для разных языков программирования, Go и PostgreSQL ими обладают. То есть

существуют готовые инструменты для того, чтобы проверять, что значения атрибутов у экземпляров сущностей соответствуют JSON-схеме данной сущности.

3.2.4. Итоги

Использование HStore для реализации динамических атрибутов не выглядит правильным выбором только из-за того, что не имеет стандартного подхода для описания схемы атрибутов и валидации по ней самих атрибутов. Построение данной функциональности вокруг HStore ведет к подходам, схожим с EAV, где EAV выигрывает из-за возможности проверки типов данных силами СУБД, или с использованием JSON-схемы, где использование JSONB выглядит естественнее.

Подход EAV удовлетворяет всем требованиям и даже имеет преимущество в том, что все проверки и валидации атрибутов проводятся на уровне СУБД, а не приложения. Но данный подход выглядит слишком громоздким, так как порождает множество таблиц, триггеров и SQL кода. Это ведет к излишней сложности системы.

Подход с использованием JSONB и JSON-схемы также удовлетворяет всем требованиям, но в сравнении с другими подходами обладает стандартными инструментами для описания схемы атрибутов и валидации их значений по ней. Также в сравнении с подходом EAV он порождает меньше сущностей и SQL кода, перекладывая всю разработку на уровень приложения. Как итог был выбран этот метод.

3.3. Обзор валидаторов JSON-схемы

Так как выбранный подход реализации динамических атрибутов подразумевает использование валидаторов JSON-схемы, то стоит рассмотреть решения, которые находятся в открытом доступе.

Валидацию JSON-документа на основе схемы можно проводить, как на уровне приложения, так и на уровне базы данных.

Для PostgreSQL были рассмотрены два самых популярных расширения предоставляющие функцию для проведения валидации:

- gavinwahl/postgres-json-schema [6];
- supabase/pg_jsonschema [5].

Первый проект написан на PLpgSQL, поддерживает спецификацию четвертой версии. Второй проект написан на языке Rust. Поддерживает спецификации до седьмой версии включительно. Оба проекта имеют практически одинаковые интерфейсы и протестированы на официальном наборе тестов [10], что дает уверенность в корректности их работы. Но, основываясь на замерах производительности, выложенном в GitHub репозитории [5], supabase/pg_jsonschema имеет более высокую скорость исполнения валидации. Таким образом для валидации на уровне базы данных данный проект выглядит лучше.

Для Go были рассмотрены следующие проекты:

- xeipuuv/gojsonschema [4];
- santhosh-tekuri/jsonschema [3];
- qri-io/jsonschema [2].

Данные проекты представлены на официальной странице JSON-Schema в разделе реализации на языке Go [12]. xeipuuv/gojsonschema и qri-io/jsonschema поддерживают седьмую версию спецификации JSON схемы, santhosh-tekuri/jsonschema поддерживает все версии спецификации. В открытом GitHub репозитории [9] проведены замеры производительности и тесты на корректность работы данных валидаторов. Основываясь на нем, qri-io/jsonschema имеет большое количество ошибок в валидации схемы. При этом самым же быстрым оказался santhosh-tekuri/jsonschema, также он развивается активнее всего, на это указывает количество выходящих релизов и решаемых проблем на GitHub. Как итог, для Go данный валидатор является лучшим решением из-за корректности работы и лучшей производительности.

Валидация на уровне приложения выглядит более разумным решением, так как это позволит писать меньше SQL кода (что для проекта

важно), позволит писать более точечные тесты, будет легче в разработке и поддержке, а также не возникнет проблем с обратной совместимостью расширений для новых версий PostgreSQL.

Таким образом был выбран валидатор santhosh-tekuri/jsonschema для Go.

4. Проектирование

4.1. Проектирование схемы данных

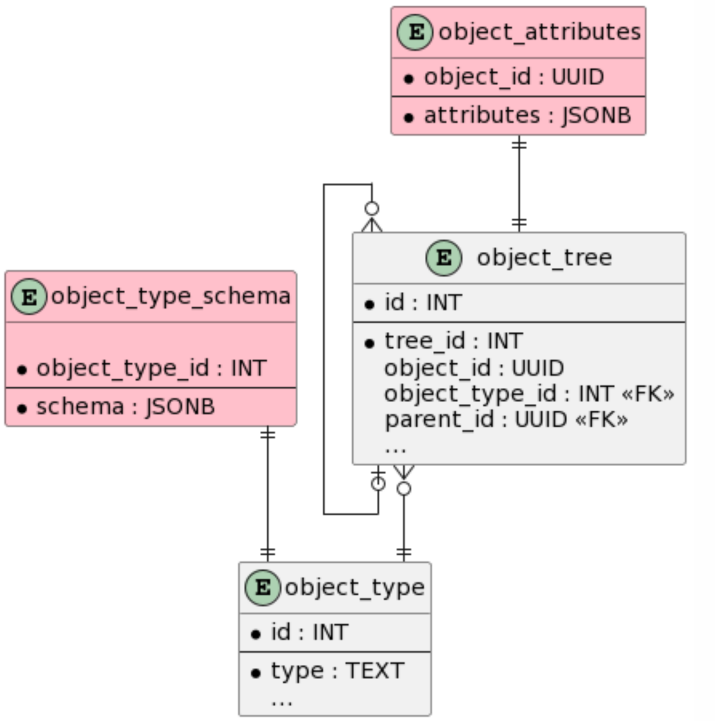

Рис. 4: Схема хранения данных для динамических атрибутов.

На рисунке 4 изображена схема хранения данных динамических атрибутов, розовым отмечены добавленные таблицы.

Схема атрибутов хранится в отдельной таблице object_type_schema, где каждому типу сущности сопоставляется JSON схема, описывающая структуру атрибутов.

Атрибуты хранятся в таблице object_attributes, где каждому объекту сопоставляется JSON-документ, в котором хранятся атрибуты. Здесь рассматривался вариант расширения колонкой JSONB таблиц каждого типа сущности. Но данная схема имеет преимущество в том, что при добавлении нового типа сущности, для него автоматически будут доступны динамические атрибуты без дополнительного написания кода.

4.2. Структура JSON-схемы атрибутов сущности

Изначальная структура JSON-схемы для каждого типа сущности представлена на листинге 1.

```
{
    ''type'' : ''object'',
    ''properties'' : {},
    ''additionalProperties'': false,
    ''required'': []
}
```

Листинг 1: Базовая структура JSON-схемы.

Данная схема описывает документ типа object, то есть документ ключ-значение. Допустимые имена ключей, типы и ограничения их значений описываются в свойстве «properties». Таким образом в этом свойстве будут описываться атрибуты, изначально никакие атрибуты не объявлены.

```
{
    ''type'' : ''object'',
    ''properties'' :
    {
        ''number'':
        {
            ''type'': ''string''
        },
        ''code'':
        {
            ''type'': ''integer''
        }
    },
    ''additionalProperties'': false,
    ''required'': [''number'', ''code'']
}
```

Листинг 2: Пример схемы атрибутов у сущности.

Атрибут «additionalProperties», выставленный в false, говорит, что в документе не могут находиться атрибуты, не описанные в «properties». Атрибут «required» содержит имена атрибутов, которые должны быть обязательно представлены в документе (то есть нахождение атрибута

в «properties» не обязывает документ иметь эти атрибуты). В качестве примера на листинге 2 представлена схема для сущности, у которой должно быть два атрибута: строка number и число code.

4.3. Описание API динамических атрибутов

Были выделены четыре обработчика, управляющие схемой атрибутов у сущностей, и два, управляющие значениями динамических атрибутов у объектов сущностей.

4.3.1. Обработчик, добавляющий атрибут к схеме сущности

`POST attributes/schema/`

Формат тела запроса и ответа представлены в листинге 3.

```
{
    ''object_type'': ''number'',
    ''attribute_name'': ''name'',
    ''attribute_type'': ''string'',
    ''description'': ''description'',
    ''required'': true,
    ''default_value'': {},
    ''enum'': [],
    ''pattern'': ''.*'',
    ''min_length'': 1,
    ''max_length'': 1,
    ''minimum'': 1,
    ''maximum'': 1
}
```

Листинг 3: Тело ответа и запроса на добавление атрибута в схему сущности.

В запросе нужно указывать обязательные параметры: имя сущности, имя атрибута, тип атрибута и его обязательность. Остальные параметры являются опциональными. Параметр default_value указывается, если атрибут обязательный. В поле enum указываются значения, которые может принимать атрибут. Параметры pattern, min_length, max_length нужны для ограничения значений строкового атрибута.

Параметры minimum и maximum нужны для ограничения значения числового атрибута.

4.3.2. Обработчик, изменяющий конфигурацию атрибута у сущности

`PUT attributes/schema/`

Тело и ответ на запрос имеют тот же формат, что и при создании атрибута.

4.3.3. Получение схемы атрибутов у конкретной сущности

`GET attributes/schema?object_type=number`

Тело ответа представлено на листинге 4.

```
[
    {
        ''attribute_name'': ''name'',
        ''attribute_type'': ''string'',
        ''description'': ''description'',
        ''required'': true,
        ''default_value'': {},
        ''enum'': [],
        ''pattern'': ''.*'',
        ''min_length'': 1,
        ''max_length'': 1,
        ''minimum'': 1,
        ''maximum'': 1
    }
]
```

Листинг 4: Тело ответа на получение схемы атрибутов.

4.3.4. Обработчик для удаления атрибута из схемы сущности.

`DELETE attributes/schema?object_type=number&attribute_name=name`

Тело ответа такое же, как и у обработчика добавления атрибута.

4.3.5. Обработчик, изменяющий значения атрибутов у конкретных объектов

`POST attributes/values/`

Тело запроса представлено в листинге 5.

```
{
    ''object_id'' : ''uuid'',
    ''attributes'' :
    {
        ''attribute1'': ''value'',
        ''attribute2'': 1,
        ''attribute3'': null
    }
}
```

Листинг 5: Тело запроса на изменение значений динамических атрибутов.

В поле attributes выписываются изменяемые атрибуты. Если значения атрибута не нулевое, то данное значение у атрибута сохраняется. Если значение null, то атрибут со своим значением удаляется.

4.3.6. Обработчик для получения динамических атрибутов у объекта

`GET attributes/values?ojbect_uuid=uuid`

Тело ответа такое же, как и у обработчика изменения значений.

5. Реализация

5.1. Реализация на уровне базы данных

Были созданы таблицы из блока про проектирование [4.1]. В проекте обращение к таблицам происходят через вызов хранимых процедур, было принято решение сохранить данный стиль.

Были созданы следующие хранимые процедуры для таблицы object_attributes:

- v1_fetch_attributes(object_type) — функция, которая возвращает атрибуты объектов, принадлежащих сущности object_type.
- v1_select_object_attributes(object_id) — возвращает атрибуты объекта с идентификатором object_id.
- v1_insert_or_update_object_attributes(object_id, attributes jsonb) — для объекта object_id добавляет в таблицу запись с атрибутами (если записи еще нет) или обновляет существующую запись новыми атрибутами.
- v1_update_object_attributes_batch(object_type, attribute_name, attribute_value) — всем объектам типа object_type выставляет атрибуту attribute_name значение attribute_value, если у объекта не было выставлено значение данного атрибута.

Для таблицы object_type_schema были созданы следующие процедуры:

- v1_select_attributes_schema(who, object_type) — возвращает JSON схему атрибутов для указанной сущности.
- v1_select_attributes_schema_by_object_id(who, object_id) — возвращает схему атрибутов для объекта.
- v1_insert_or_update_attributes_schema(object_type, schema) — для сущности object_type добавляет в таблицу запись со схемой

(если записи еще нет) или обновляет существующую запись новой схемой.

5.2. Доработки в уровне работы с базой данных в коде приложения

Уровень работы с базой данных выполнен по паттерну «репозиторий». На этом уровне были добавлены методы, которые вызывают добавленные хранимые процедуры.

Также старая реализация работы с базой данных позволяла делать вызовы только в режиме автоматической фиксации[8]. Но реализация динамических атрибутов требует явного использования транзакций.

В каждом методе репозитория вызываются функции getReadableDB или getWriteableDB, которые возвращают структуру sql.DB[9], позволяющую отправлять запросы в мастер или реплику базы данных. Был создан интерфейс DBQueryer, который реализуют sql.DB и sql.Tx[10]. Методы getReadableDB и getWriteableDB были доработаны так, чтобы они возвращали данный интерфейс. Они возвращают транзакцию, если она содержится в контексте запроса. Чтобы транзакция появилась в контексте запроса, для репозитория были реализованы методы WithROTx и WithRWTx, которые начинают транзакцию и кладут её в контекст. Также они возвращают функцию, которая позволяет откатывать или фиксировать транзакцию. Метод WithROTx начинает транзакцию с уровнем «повторное чтение», WithRWTx с уровнем «сериализуемость».

5.3. Работа с JSON-схемой

Для JSON-схемы была создана своя структура для Go. Это позволило преобразовать сырой JSON со схемой в эту структуру, изменять схему в коде и преобразовать это в JSON обратно. В этой структуре

[8]Автоматическая фиксация (autocommit) – это транзакция, зафиксированная при выполнении. То есть это исполнение SQL инструкции без надобности указания начала или конца транзакции.

[9]https://pkg.go.dev/database/sql#DB

[10]https://pkg.go.dev/database/sql#Tx

было объявлено дополнительное поле TypeSlug, чтобы указывать тип атрибута, который будет указываться в API приложения.

Чтобы абстрагироваться от работы с JSON-схемой была создана структура AttributesSchema. На рисунке представлена данная структура.

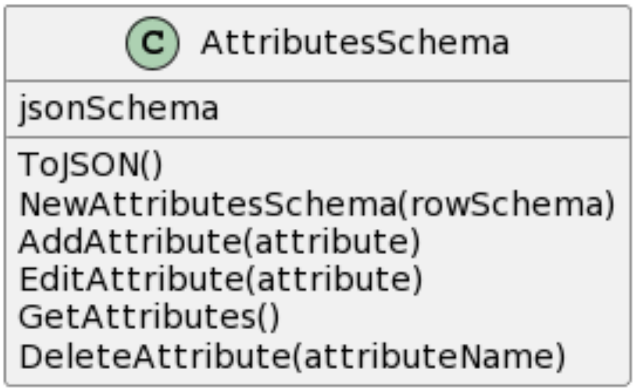

Рис. 5: Структура AttributesSchema.

Она хранит структуру с JSON-схемой, которую нужно передать в конструктор и которую можно получить с помощью метода ToJSON. Метод GetAttributes обходит JSON-схему, формируя список атрибутов и их конфигурацию. Метод DeleteAttribute удаляет из properties JSON схемы ключ с именем атрибута, также он удаляется из поля required, если он там находится.

Методы AddAttribute и EditAttribute очень похожи. Вначале они проверяют существование атрибута в схеме. Далее они формируют новую JSON схему (она описывает конфигурацию атрибута), которая будет указываться в properties итоговой схемы. JSON схема атрибута формируется с помощью интерфейса AttributeSchemaProvider. По типу входного атрибута выбирается нужная реализация данного интерфейса. Интерфейс и его реализации представлены на рисунке 6.

Структура **IntAttributeProvider** собирает JSON схему для числового атрибута. На листинге 6 представлен пример такой схемы.

Структура **StringAttributeProvider** собирает JSON схему для текстового атрибута. На листинге 7 представлен пример такой схемы.

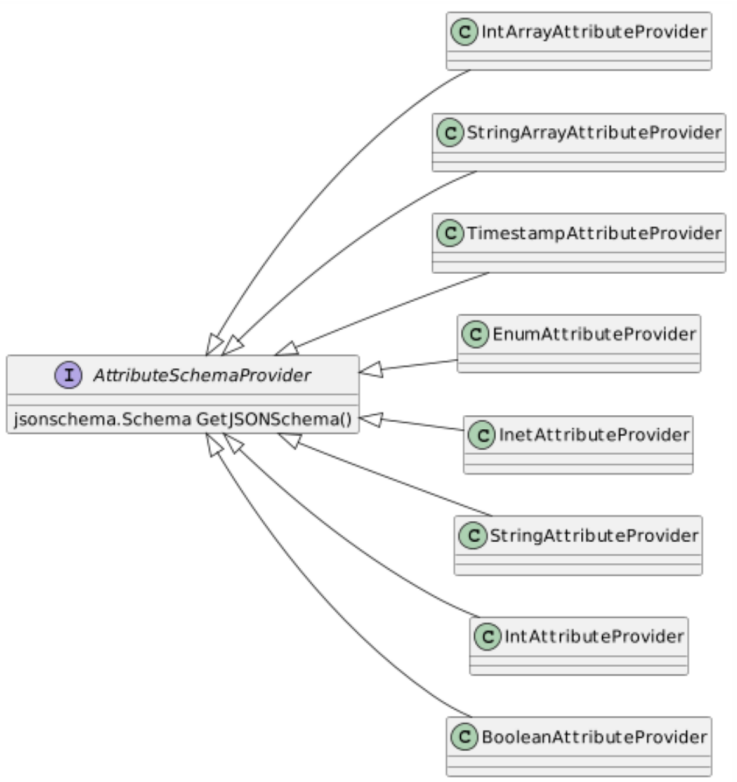

Рис. 6: Интерфейс AttributeSchemaProvider и его реализации.

```
{
    ''type'' : ''integer'',
    ''type_slug'' : ''integer'',
    ''description'' : ''числовой атрибут'',
    ''default'' : 1,
    ''minimum'' : 0,
    ''maximum'' : 10
}
```

Листинг 6: JSON-схема числового атрибута.

Структура **BooleanAttributeProvider** собирает JSON схему для булевого атрибута. На листинге 8 представлен пример такой схемы.

Структура **InetAttributeProvider** собирает JSON схему для атрибута формата ip адреса. На листинге 9 представлен пример такой схемы.

Структуры **IntArrayAttributeProvider** и **StringArrayAttributeProvider** собирают JSON схему для массивов строк и чисел. На листинге 10 представлен пример такой схемы.

```
{
    ''type'' : ''string'',
    ''type_slug'' : ''string'',
    ''description'' : ''текстовый атрибут'',
    ''default'' : ''value'',
    ''min_length'' : 3,
    ''pattern'' : ''[a-z]+''
}
```

Листинг 7: JSON-схема текстового атрибута.

```
{
    ''type'' : ''boolean'',
    ''type_slug'' : ''boolean'',
    ''description'' : ''булевый атрибут''
}
```

Листинг 8: JSON-схема булевого атрибута.

```
{
  ''type_slug'' : ''inet'',
  ''oneOf'': [
    { ''type'': ''string'', ''format'': ''ipv4'' },
    { ''type'': ''string'', ''format'': ''ipv6'' }
  ]
}
```

Листинг 9: JSON-схема атрибута формата ip адреса.

```
{
 ''type'': ''array'',
 ''type_slug'' : ''text_array'',
 ''items'': { ''type'': ''string'' }
}
```

Листинг 10: JSON-схема массива строк.

Структура **EnumAttributeProvider** собирает JSON схему для enum атрибута. На листинге 11 представлен пример такой схемы.

```
{
  ''type_slug'' : ''enum'',
  ''enum'' : [''value1'', ''value2'', ''value3'']
}
```

Листинг 11: JSON-схема enum атрибута.

5.4. Реализация обработчиков

Ранее код приложения был разделен на два слоя: слой обработчиков и слой репозитория работы с базой данных. Чтобы не нагружать слой обработчиков бизнес-логикой динамических атрибутов, было принято решение ввести слой бизнес-логики. Для динамических атрибутов он называется «сервис атрибутов».

5.4.1. Реализация обработчика добавления атрибута к схеме сущности

На рисунке 7 представлен алгоритм работы обработчика при добавлении обязательного атрибута к схеме сущности.

После того как пользователь отправил запрос, обработчик запроса обрабатывает его, достает нужные данные и передает их в сервис атрибутов. Сервис атрибутов запрашивает схему атрибутов из базы данных для указанной сущности. Далее атрибут добавляется в схему. Так как атрибут является обязательным, то его значение должно быть у каждого объекта данной сущности. Поэтому отправляется запрос на добавление значения данного атрибута к каждому объекту сущности. После этого измененная схема атрибутов сохраняется в базу данных.

5.4.2. Реализация обработчика, изменяющего конфигурацию атрибута у сущности

На рисунке 8 представлен алгоритм работы обработчика, изменяющего конфигурацию атрибута у сущности.

В сервисе атрибутов сначала загружается схема атрибутов из базы данных. Меняется конфигурация атрибута в ней. После чего, если

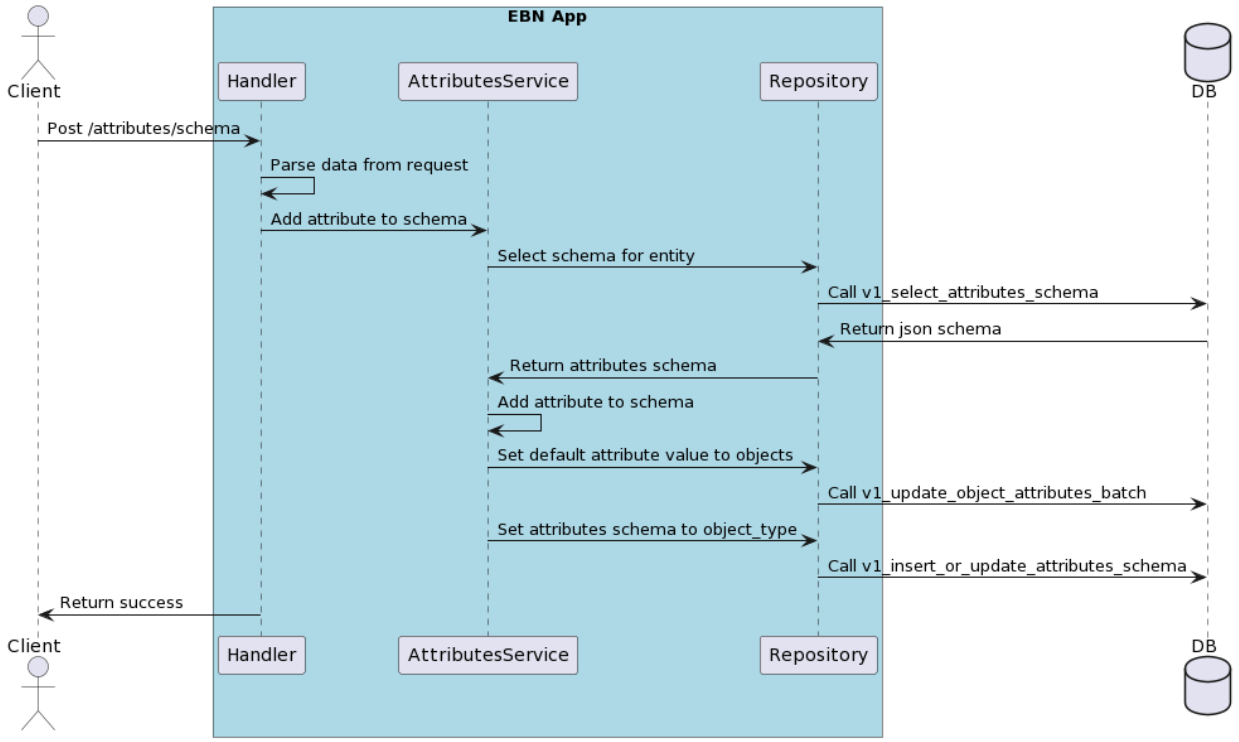

Рис. 7: Схема работы обработчика по добавлению атрибута к схеме сущности.

атрибут является обязательным, то его значение по умолчанию присваивается всем объектам данной сущности. После этого все атрибуты объектов данной сущности загружаются и проверяется, что они соответствуют новой схеме. Если все предыдущие шаги прошли успешно, то новая схема загружается в базу данных.

5.4.3. Реализация обработчика удаления атрибута из схемы сущности

В сервисе атрибутов происходят следующие шаги:

- Проверяется, что не существует объектов содержащих удаляемый атрибут, иначе ошибка.
- Загружается схема из базы данных.
- В ней удаляется конфигурация атрибута.
- Новая схема сохраняется в базу данных.

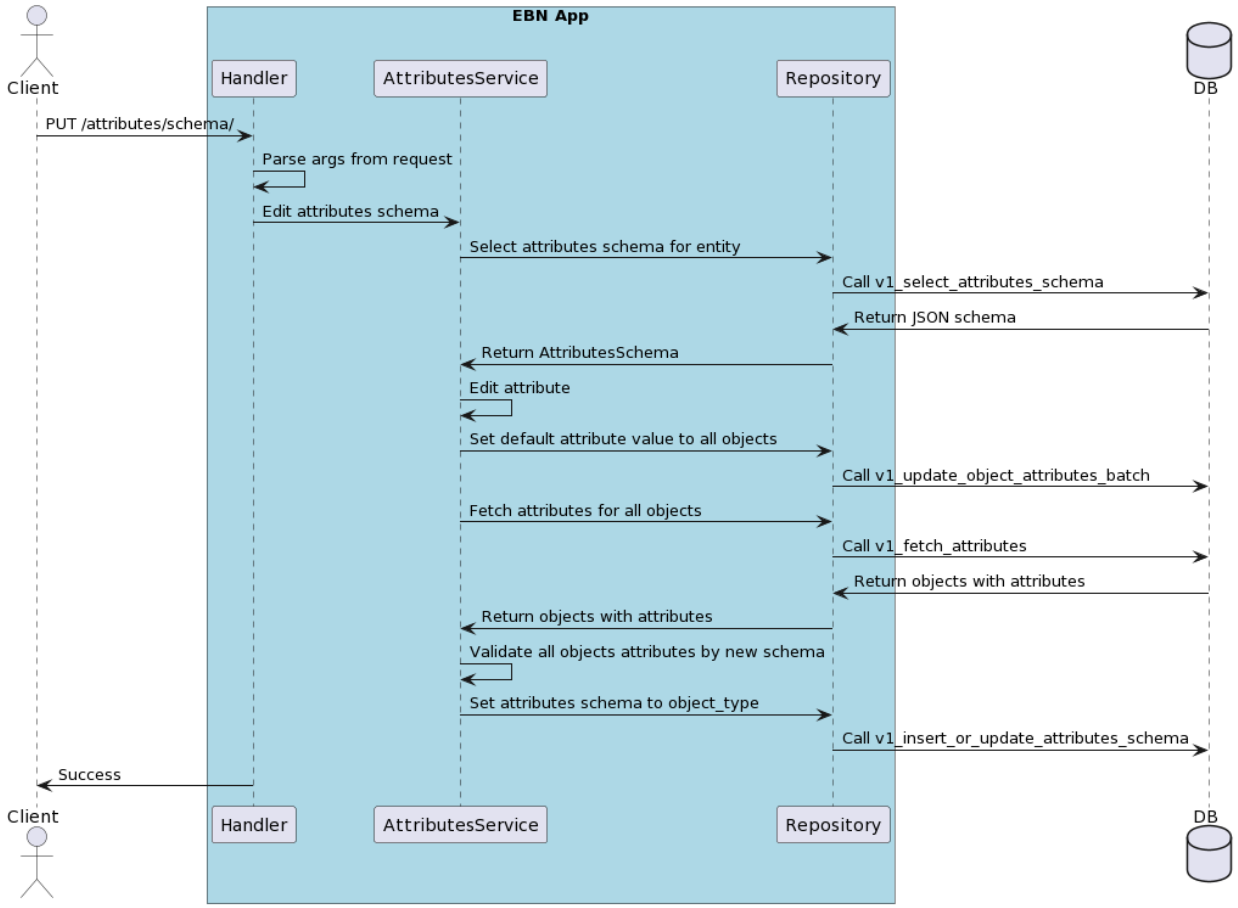

Рис. 8: Схема работы обработчика, изменяющего конфигурацию атрибута у сущности.

5.4.4. Реализация обработчика получения схемы атрибутов сущности

Здесь алгоритм достаточно простой.

- Выкачивается схема из базы данных.
- Вызывается метод GetAttributes() у AttributesSchema, которой отдает схему атрибутов в нужном формате.
- Полученные данные отдаются в обработчик запроса, который отдает их клиенту.

5.4.5. Реализация обработчика изменяющего значения атрибутов у сущностей

На рисунке 9 представлен алгоритм работы данного обработчика.

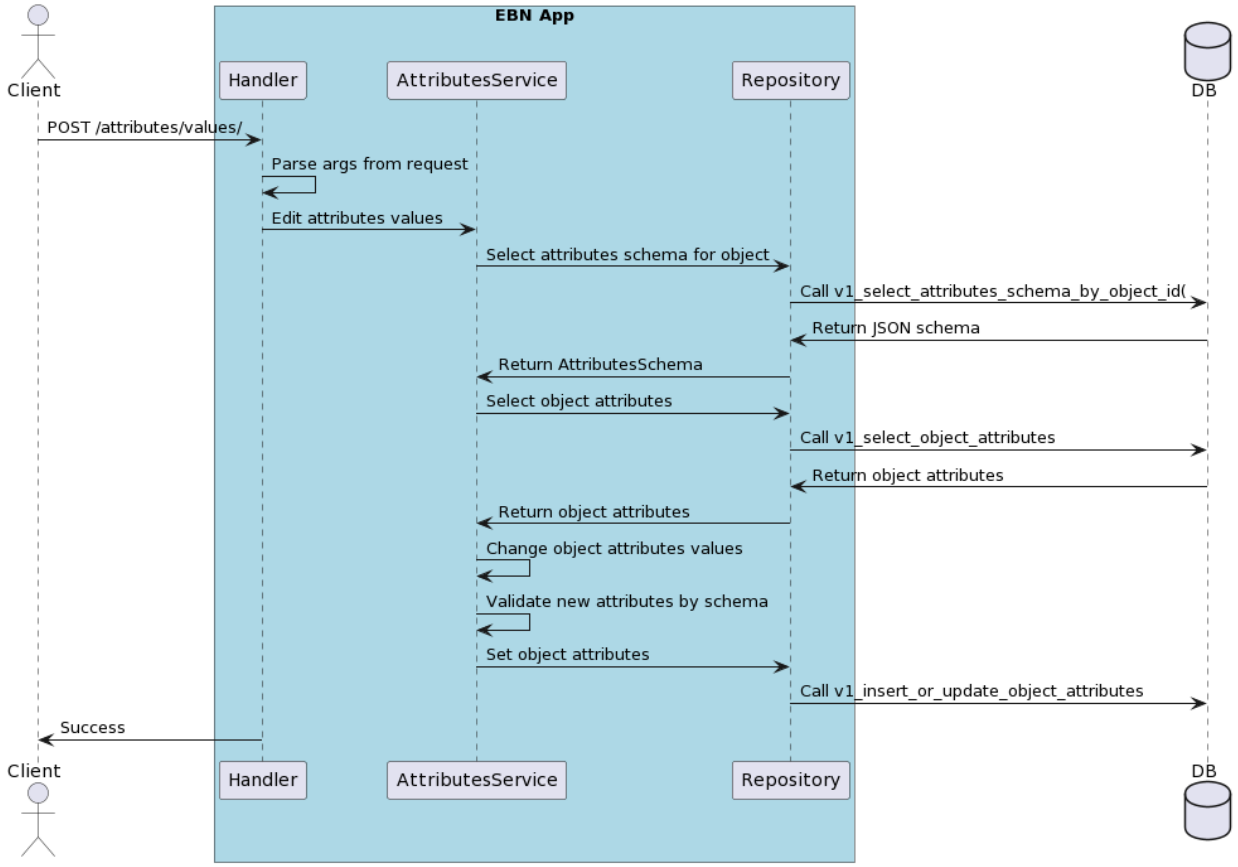

Рис. 9: Схема работы обработчика изменяющего значения атрибутов у объекта.

В сервисе атрибутов загружаются атрибуты объекта и схема его сущности. Меняются значения его атрибутов. Перед тем как сохранить новую версию атрибутов, они проверяются на корректность через валидатор.

5.4.6. Реализация обработчика получения значений атрибутов

Значения атрибутов выгружаются из базы данных и проносятся через все слои приложения до клиента.

5.4.7. Расширения существовавших обработчиков

Были расширены CRUD обработчики всех существующих типов объектов. Это было реализовано путем расширения API атрибутом «attributes», который позволяет управлять динамическими атрибутами по аналогии обработчиков динамических атрибутов.

Был расширен поиск объектов системы. Шаблон строки для поиска проверяется и для динамических атрибутов. Поэтому, если какой-то из динамических атрибутов удовлетворяет входному шаблону строки, то объект входит в результат поиска.

Было расширено массовое добавление номеров. Код был дополнен функциональностью, позволяющей обнаруживать динамические атрибуты в CSV-документе и добавляющей найденные атрибуты к создаваемому номеру.

6. Апробация

6.1. Внутреннее ревью

Работа прошла внутреннее дизайн-ревью. К нему были подготовлены документы, содержания которых схожи с главами «Введение», «Обзор», «Проектирование» данной работы. Также была проведена встреча, по итогу которой было дано разрешение на реализацию спроектированной подсистемы.

Код работы прошел внутренний код-ревью и попал в новую версию приложения.

6.2. Тестирование

Было написано более 20 модульных и более 10 интеграционных тестов. Тесты покрывают следущие сценарии.

- Добавление атрибута к типу объекта.
- Редактирование атрибута у типа объекта.
- Удаление атрибута у типа объекта.
- Получения схемы атрибутов у всех типов объектов.
- Добавления значения атрибута к объекту.
- Редактирование значения атрибута у объекта.
- Удаление значения атрибута у объекта.
- Получения значений динамических атрибутов у объектов.

Все написанные файлы с кодом, которые относятся к слоям бизнес-логики и работы с базой данных, имеют тестовое покрытие более 75%.

6.3. Документация

Была написана документация для администраторов ЕБН по использованию разработанной подсистемы. Администраторы были ознакомлены с документацией, замечаний к документации выявлено не было.

6.4. Добавление новых атрибутов

Администраторы ЕБН, используя данную подсистему, добавили более 70 атрибутов к четырем типам объектов.

Значениями добавленных атрибутов были размечены более 10000 объектов сервиса.

7. Заключение

В рамках выполнения данной работы были достигнуты следующие результаты.

- Был выполнен обзор решений реализации динамических атрибутов, в котором рассматривались подход “EAV”, подход с использованием Hstore и подход с использованием JSONB-документа и JSON-схемы.
- Была спроектирована подсистема автоматизированного добавления новых атрибутов, основанная на валидации атрибутов с помощью JSON-схемы на стороне приложения и их хранения в виде JSONB-документа в базе данных.
- Данная подсистема была реализована и протестирована. Было написано более 20 модульных и более 10 интеграционных тестов.
- Динамическими атрибутами были расширены существовавшие обработчики, которые включали CRUD для всех типов объектов, поиск и массовые внесения номеров.
- Реализация попала в новую версию сервиса.
- Была написана документация, с которой ознакомились администраторы сервиса.
- Администраторы ЕБН, используя данную подсистему, добавили более 70 атрибутов к четырем типам объектов.
- Значениями добавленных атрибутов были размечены более 10000 объектов сервиса.

Код работы находится в закрытом репозитории.

Список литературы

[1] JSON-Schema. — URL: https://json-schema.org/ (online; accessed: 2024-01-03).

[2] Валидатор JSON-схемы для Go qri-io/jsonschema. — URL: https://github.com/qri-io/jsonschema (online; accessed: 2024-01-03).

[3] Валидатор JSON-схемы для Go santhosh-tekuri/jsonschema. — URL: https://github.com/santhosh-tekuri/jsonschema (online; accessed: 2024-01-03).

[4] Валидатор JSON-схемы для Go xeipuuv/gojsonschema. — URL: https://github.com/xeipuuv/gojsonschema (online; accessed: 2024-01-03).

[5] Валидатор JSON-схемы для PostgreSQL pg_jsonschema. — URL: https://github.com/supabase/pg_jsonschema (online; accessed: 2024-01-03).

[6] Валидатор JSON-схемы для PostgreSQL postgres-json-schema. — URL: https://github.com/gavinwahl/postgres-json-schema (online; accessed: 2024-01-03).

[7] Документация для JSON и JSONB в PostgreSQL. — URL: https://www.postgresql.org/docs/current/datatype-json.html (online; accessed: 2024-01-03).

[8] Документация для hstore в PostgreSQL. — URL: https://www.postgresql.org/docs/current/hstore.html (online; accessed: 2024-01-03).

[9] Замеры производительности и тесты корректности трех валидаторов JSON-схемы для Go. — URL: https://github.com/swaggest/go-json-schema-bench (online; accessed: 2024-01-03).

[10] Официальный набор тестов для проверки корректности валидатора JSON-схемы. — URL: https://github.com/

json-schema-org/JSON-Schema-Test-Suite (online; accessed: 2024-01-03).

[11] Построение EAV схемы данных с отдельными таблицами под каждый тип данных. — P. 481–482. — URL: https://www.researchgate.net/publication/12720878_Organization_of_Heterogeneous_Scientific_Data_Using_the_EAVCR_Representation (online; accessed: 2024-01-03).

[12] Список реализаций валидаторов JSON-схемы для Go. — URL: https://json-schema.org/implementations#validators-go (online; accessed: 2024-01-03).

[13] Статья об эффективном использовании EAV подхода в медицине. — P. 1–3. — URL: https://www.mdpi.com/2078-2489/9/1/2#B1-information-09-00002 (online; accessed: 2024-01-03).