DOI: 10.15514/ISPRAS-2025-37(4)-8

Реализация гибкой системы управления доступом в приложении

Т.И. Васильев, ORCID: 0009-0008-6671-3026 <timurvasilev@gmail.com>
Paybis USA LTD, США, Талса, Бостон Авеню 321.

Аннотация. В статье рассматривается развитие механизма конфигурации гибкой ролевой системы для приложений, которые требуют динамического разграничения прав пользователей в зависимости от бизнес-контекста или различных условий. Современные приложения становятся все больше и сложнее, что приводит к необходимости реализации эффективных механизмов контроля доступа. Описанный подход предполагает использование собственной ролевой модели, которая позволит точно настраивать права пользователей для различных сущностей в базе данных. Основное внимание уделено гибкости и масштабируемости модели, возможности настраивать доступ в зависимости от статусов объектов, ролей пользователей и нужд бизнеса. В качестве примера рассмотрена конфигурация интернет-магазина, с разграничением по ролям пользователей и учетом различий в статусах товара. Подробно описан процесс настройки доступов, приведён пример кода для проверки разрешений и динамического отображения полей. В заключение обсуждаются преимущества данного механизма и его недостатки.

Ключевые слова: гибкая система управления доступом; ролевая модель; контроль доступа; права пользователей; информационная безопасность; JSON-конфигурации; интернет-магазин; управление доступом; разграничение прав доступа; масштабируемость информационных систем.

Для цитирования: Васильев Т.И. Реализация гибкой системы управления доступом в приложении. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 147–160. DOI: 10.15514/ISPRAS-2025-37(4)-8.

Implementation of a Flexible Access Control System in the Application

T.I. Vasilev, ORCID: 0009-0008-6671-3026 <timurvasilev@gmail.com>
Paybis USA LTD, USA, Tulsa, 321 S Boston Ave Downtown.

Abstract. The article considers the development of a flexible role system configuration mechanism for applications that require dynamic differentiation of user rights depending on the business context or various conditions. Modern applications are becoming more and more complex, which leads to the need to implement effective access control mechanisms. The described approach involves the use of its own role model, which will allow you to fine-tune user rights for various entities in the database. The focus is on the flexibility and scalability of the model, which allows you to customize access depending on the statuses of objects, user roles and business needs. As an example, the configuration of an online store is considered, with differentiation by user roles and various product statuses. The access configuration process is also described in detail and an example code is provided for checking permissions and dynamically displaying fields. In conclusion, the advantages of this mechanism and its disadvantages are discussed.

Keywords: flexible access control system; role model; access control; user rights; information security; JSON configurations; online store; access control; access rights differentiation; scalability of information systems.

For citation: Vasilev T.I. Implementation of a flexible access control system in the application, vol. 37, issue 4, part 1, 2025 , pp. 147-160 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-8.

1. Введение

В современном мире информационные системы становятся все более объёмными и сложными, требуя надёжного и гибкого подхода к управлению доступом. Важным аспектом безопасности данных в веб-приложениях является контроль над тем, какие пользователи системы могут обращаться к той или иной информации или имеют право на выполнение того или иного действия. Для достижения наибольшей гибкости и удобства системы в управлении правами доступа возникает необходимость построения собственного механизма ролевой системы, с помощью которой будет возможно конфигурировать права пользователей к разным аспектам системы в зависимости от текущих условий и контекста использования. Данный механизм должен быть масштабируем и легко применим в возможных сценариях работы системы без значительных архитектурных изменений в приложениях. В статье описывается подход к реализации системы управления доступом, основанной на гибкой ролевой модели в веб-приложениях с использованием реляционной базы данных MySQL или Postgres [1]. Предложенный механизм был применён в нескольких проектах и продемонстрировал свою эффективность в разнообразных сценариях. Целью работы является представление разработанного механизма, а также анализ его возможностей в сравнении с альтернативными решениями. Основной целью проведенной работы по адаптации известных подходов являлось повышение их гибкости, что особенно актуально для веб-приложений, работающих с реляционными базами данных.

2. Обзор существующих решений

Устоявшимися и часто используемыми ролевыми моделями доступа в веб-приложениях являются модели RBAC и ABAC [2-3]. Ролевая модель доступа RBAC основывается на присваивании пользователям ролей с фиксированным набором прав. Положительными чертами данного подхода являются:

- Простота реализации RBAC позволяет быстро установить роли и назначить их пользователям, что существенно упрощает администрирование.
- Предсказуемость поведения системы, так как количество ролей и их комбинация ограничены.

- Простое управление доступом для больших групп пользователей за счёт предоставления или ограничения доступа на основе роли.

В качестве минусов можно выделить:

- Ограничение гибкости системы. Для роли фиксируется набор прав, который часто может не соответствовать набору необходимых разрешений для работы в системе.
- Сложность при изменении ролей. В больших системах с множеством ролей и пользователей достаточно сложно управлять изменениями ролевой системы.

Модель управления доступом ABAC использует атрибуты пользователей и объектов для динамического принятия решений о доступе. Основным отличием этого подхода в том, что каждая ситуация оценивается не с точки зрения роли пользователя и действия, которое он хочет совершить, а с точки зрения атрибутов, которые к ним относятся. Подробнее с данным механизмом можно ознакомиться в работе [4].

У модели ABAC имеются несколько достоинств:

- Данная модель позволяет динамично изменять политики доступа, что делает систему гибкой и масштабируемой.
- Благодаря тонкой настройке прав, доступ может быть настроен с учётом различных атрибутов.
- Модель ABAC особенно полезна в сценариях, где необходимо учитывать контекст запроса, такие как географическое местоположение или тип устройства.

В качестве недостатков можно выделить следующие:

- Сложность реализации модели и ее интеграции в веб-приложение. Требуется тщательно настроить правила для различных атрибутов и объектов, что значительно усложняет поддержку и разработку.
- При большом количестве атрибутов и правил проверка прав может стать достаточно медленной.

Подход, представленный в данной статье, позволяет более гибко настраивать доступ, но при этом сохраняет простоту и улучшает производительность по сравнению с моделями ABAC и RBAC.

К настоящему времени опубликовано множество научных работ по созданию ролевых моделей в различных программных системах. Исследование [5] предлагает анализ различных решений для построения ролевой модели управления доступом в корпоративных информационных системах. В работе рассмотрены как классические подходы к RBAC, так и их адаптации для сложных корпоративных сред. Ключевым преимуществом данной работы является структурированный подход к внедрению ролевой модели с учётом корпоративных требований. Однако в работе отсутствует описание практической реализации и примеры конкретных сценариев использования, что делает её сложной для использования в реальных проектах. В публикации [6] акцент сделан на использовании ролевой модели управления доступом в специализированной системе – системе конференций. Основное внимание уделяется тому, как модель RBAC может быть адаптирована для управления большими потоками участников, модераторов и организаторов. Положительной стороной является описание специфики доступа в контексте временных событий, таких как конференции. Однако данный подход сложно адаптировать для систем, не связанных с временными процессами, и отсутствуют универсальные рекомендации. Исследование [7] рассматривает применение модели RBAC в системах архивирования данных. Положительной стороной является учёт специфики работы с архивами, включая долгосрочное хранение и доступ к историческим данным. Недостатком является ограниченность подхода для использования в системах с более динамичным характером данных, например, в веб-приложениях. В работе [8] описана реализация гибкой системы управления доступом с акцентом на роли и возможности настройки уровня доступа в информационных системах. Основное

преимущество работы заключается в детализированном описании механизма доступа, который ориентирован на адаптивность под различные сценарии. Однако в работе отсутствует конкретика в применении технологий – отсутствует привязка к реальным инструментам или программным средствам, что может затруднить практическую реализацию.

Рассмотренные работы представляют различные подходы к реализации систем управления доступом. Тем не менее каждый из них имеет свои ограничения. Предложенное решение сочетает лучшие черты существующих подходов и предлагает гибкость, масштабируемость и простоту настройки, что делает его универсальным инструментом для современных веб-приложений.

3. Построение гибкой ролевой системы

Рассмотрим реализацию механизма на примере упрощённой версии интернет-магазина. Проектируемый интернет-магазин будет иметь дело со следующими объектами: товар и пользователь, остальные объекты, которых в реальной жизни много, в статье намерено опущены для упрощения представления реализации. Для хранения данных интернет-магазин использует реляционную SQL базу данных, что позволяет организовать объекты и их взаимосвязи в виде таблиц. В данном интернет-магазине будет использован следующий набор ролей:

- User – Роль для клиента интернет-магазина.
- Courier – Роль работника склада, осуществляющего доставку товара.

Товар будет иметь следующий набор статусов:

- Available – Доступен к заказу.
- NotAvailable – Закончился на складе и в данный момент недоступен.

При создании механизма сначала необходимо определить способ хранения конфигураций и значений параметров объектов. Суть механизма конфигураций в том, что у каждого типа объектов может существовать несколько описаний (конфигураций), где для каждого описания определяются наборы параметров и правила доступа к ним. Например, для типа «Товар» можно задать две разные конфигурации – «Книга» и «Телевизор». Конфигурация «Книга» включает, например, параметр «Автор» и «Количество страниц», а конфигурация «Телевизор» – «Диагональ экрана», «Разрешение» и другие характеристики.

Значения параметров объекта – это конкретные данные, которые присущи каждому экземпляру объекта. Например, один телевизор может иметь диагональ 40 дюймов, а другой – 50, но оба относятся к конфигурации «Телевизор» и типу «Товар». Аналогично, в случае «Книги» у отдельных экземпляров могут отличаться автор, количество страниц и другие характеристики.

Далее будут рассмотрены различные способы хранения конфигураций и значений параметров объектов, а также их преимущества и недостатки.

Первый способ хранения конфигураций изображен на рис. 1. Конфигурация представлена в виде JSON-файлов, которые хранятся в репозитории вместе с кодом приложения. Изменения в конфигурации выполняются программистами путем создания новых или редактирования существующих JSON-файлов с последующим подтверждением изменений в системе контроля версий.

Преимуществом такого подхода является возможность совместной работы команды программистов: изменения в конфигурации могут быть проверены через систему контроля версий, что обеспечивает их прозрачность и контроль качества. Кроме того, тестирование изменений на тестовых окружениях перед выпуском в рабочую среду помогает минимизировать риски, связанные с некорректной конфигурацией.

В процессе разработки и последующего использования системы, которая использовала данный подход к хранению конфигураций, могут возникать следующие проблемы:

- Скорость реагирования на сбои в рабочей среде может оказаться слишком низкой из-за того, что любые конфигурационные правки требуют полноценного развертывания, что во многих приложениях занимает существенное время.
- Заметная часть времени программистов может уходить на настройку конфигураций, а не на написание кода.

Таким образом, данный подход обеспечивает прозрачность и тестируемость изменений, но накладывает ограничения на оперативное управление конфигурациями в условиях реальной эксплуатации.

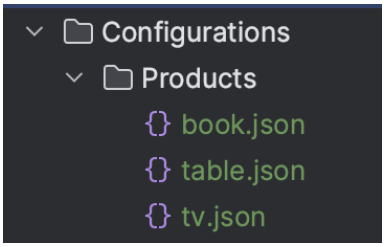

Рис. 1. Хранение конфигураций в коде приложения.
Fig. 1. Storing configurations in the application code.

Другим способом хранения конфигураций объектов является их размещение в базе данных приложения (рис. 2). В этом случае каждая конфигурация представлена отдельной записью в таблице. Данный подход даёт возможность разрабатывать административную панель, которая будет взаимодействовать непосредственно с базой данных и позволять вносить изменения в конфигурации через удобный пользовательский интерфейс, без участия программистов.

Чтобы понять разницу предложенных способов, рассмотрим общий процесс работы с конфигурациями при файловом хранении:

1. Программист изменяет конфигурационный файл в локальном репозитории и отправляет изменения в систему контроля версий.
2. Система сборки (CI/CD) при необходимости запускает тесты, формирует новую сборку приложения и подготавливает файлы для развёртывания.
3. Развёртывание в целевое окружение (тестовое, производственное и т.д.).

Если конфигурации размещены в файлах, чтобы внести даже незначительную правку (например, добавить новое поле или изменить права для роли пользователя), придётся снова проходить весь цикл: создать новую версию, выполнить сборку, а затем развернуть новые файлы на сервере. Возможен и упрощённый вариант действий, при котором в систему можно загрузить только изменённые файлы конфигурации без полноценной пересборки. Однако в этом случае пропускается этап автоматизированного тестирования, что увеличивает риск появления ошибок или несовместимых настроек, которые бы выявились при прохождении полного цикла сборки.

В случае же хранения конфигураций в базе данных вся цепочка может быть упрощена, если использовать административную панель:

1. Администратор с помощью веб-интерфейса получает доступ к списку существующих конфигураций.
2. Изменение конфигураций происходит путём прямого редактирования соответствующих записей в базе данных.
3. Сохранение правок сразу отражается в приложении, без пересборки и развёртывания всего проекта.

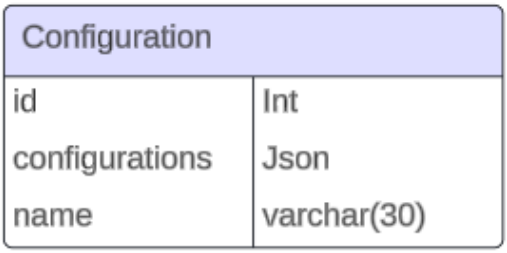

Рис.2. Хранение конфигураций в базе данных.
Fig. 2. Storing configurations in a database.

Преимуществами данного подхода можно считать следующие:

- Администраторы, которые не обладают навыками программирования или не умеют работать с системой контроля версий, могут изменять настройки через удобный интерфейс, без необходимости редактирования JSON-файлов в коде приложения.
- Все изменения в конфигурациях могут быть применены оперативно через панель администрирования, что исключает необходимость пересборки приложения или развертывания.

Подход имеет и недостатки:

- Ручное внесение данных через веб-интерфейс повышает риск опечаток и неправильных значений, особенно если нет механизмов валидации или она недостаточно жёсткая. Такой риск может быть снижен продуманными строгими проверками на уровне приложения, однако полностью исключить человеческий фактор нельзя.
- Если изменения конфигурации вносятся индивидуально через панель, остальным участникам команды сложнее оперативно узнать, кто внес изменения и когда это произошло. В отличие от запросов (pull-request) или подтверждений (commit) в системе контроля версий, панель администрирования обычно не имеет встроенного механизма обсуждения и согласования правок.
- Создание и поддержка полноценной панели администрирования требует времени и ресурсов. Необходимо разрабатывать интерфейс, учитывать разные сценарии использования и права доступа, внедрять механизмы аудита изменений. Для небольших проектов это может оказаться избыточным.

Таким образом, переход на хранение конфигураций в базе данных и создание административной панели значительно упрощают процесс управления настройками приложения, но требуют внедрения дополнительных механизмов контроля и синхронизации данных для минимизации рисков.

Кроме традиционного хранения конфигураций в базе данных, можно использовать возможности PostgreSQL по работе с внешними таблицами (foreign tables). С помощью такого подхода данные могут фактически оставаться во внешнем источнике, но при этом выглядеть и обрабатываться в PostgreSQL, словно это обычные таблицы. Одним из примеров реализации этой концепции является расширение S3 FDW (Simple Storage Service Foreign-Data Wrapper), позволяющее хранить и читать данные из облачного сервиса Amazon S3 [9].

Это может быть полезно, если требуется гибкость и масштабируемость облачного хранилища, но при этом удобно пользоваться SQL-запросами и механизмами PostgreSQL для поиска, фильтрации и анализа данных. Такое решение частично примиряет идею «файлов в облаке» и «данных в СУБД», поскольку администратор может продолжать работать с конфигурациями через SQL-интерфейс, а сами файлы с настройками физически

размещаются на S3. В результате не требуется ручного копирования или синхронизации – система PostgreSQL сама обеспечивает связность и доступ к внешнему хранилищу.

В итоге, у каждого из рассмотренных подходов – будь то хранение конфигураций в файловой системе, в базе данных или с помощью внешних таблиц (к примеру, через S3 FDW) – есть свои преимущества и недостатки. Файлы проще в использовании и подходят для небольших проектов с редкими изменениями. Базы данных удобны для построения административных панелей, быстрой правки. Внешние таблицы совмещают гибкость базы данных и масштабируемость облачных хранилищ.

Выбор конкретного варианта зависит в первую очередь от потребностей проекта, сложности инфраструктуры, частоты и объёмов изменений, а также требований к безопасности и удобству эксплуатации. Лучше всего предварительно оценить все факторы и выбрать то решение, которое оптимально впишется в архитектуру и процессы команды.

Далее будут рассмотрены несколько подходов к реализации хранения параметров объекта.

На рис. 3 продемонстрирована схема базы данных с реализацией хранения значений параметров объекта в виде колонок в таблице. В данном подходе существенным минусом является необходимость изменения структуры таблицы базы данных при изменении конфигурации, когда при удалении или добавлении нового поля в конфигурации приходится также проводить миграцию данных с изменением структуры таблицы. Частота таких операций зависит от особенностей проекта, для систем с большим набором параметров объектов изменения могут быть ресурсоемкими и затруднять эксплуатацию. В небольших проектах с редкими изменениями конфигураций этот недостаток может оказаться незначительным.

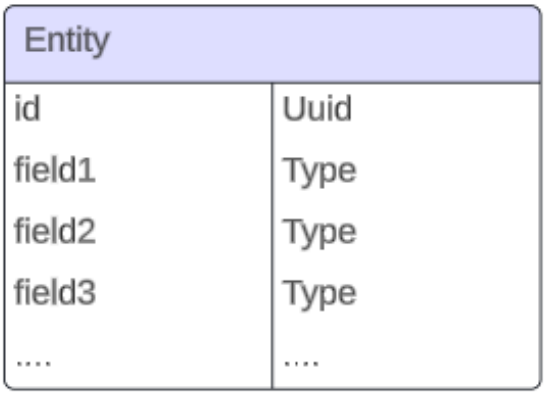

Рис. 3. Схема базы данных гибкой ролевой модели с сохранением данных в колонках.
Fig. 3. A flexible role model database schema with data stored in columns.

Другой способ реализации хранения значений параметров объекта для гибкой ролевой модели изображён на рис. 4, где каждое поле объекта вынесено как запись в отдельную таблицу. Используя данный подход, таблица с данными будет заполняться очень быстро, ведь на каждую запись в таблице Entity будет приходиться n записей (количество полей в наборе параметров объекта) в таблице Data. А также при получении параметров объекта необходимо будет осуществлять дополнительный запрос или JOIN к таблице Data, что существенно замедлит систему. Подробнее о скорости работы JOIN операций можно ознакомиться в работах [10-11].

Другой способ хранения значений параметров объекта продемонстрирован на рис. 5. На данной схеме базы данных параметры хранятся в колонке data с типом JSON, таким образом, данные объекта можно будет получать без дополнительных запросов или JOIN операций, что существенно ускорит работу механизма. Данный подход также имеет и минусы: для

извлечения данных из JSON может потребоваться дополнительная обработка на уровне приложения, особенно если данные хранятся в сложных вложенных структурах.

Полнотекстовые или сложные индексы на основе JSON могут быть недоступны или работать медленно в зависимости от используемой базы данных; JSON-документы могут занимать больше места, чем структурированные данные в колонках, особенно при дублировании параметров.

Способы хранения конфигураций и значений параметров объектов, описанные выше, имеют свои плюсы и минусы, и все подходят для реализации механизма управления доступами. В ходе эксплуатации механизма хранение конфигураций осуществлялось в базе данных для настройки через панель администратора (рис. 2.). А хранение значений параметров объектов было реализовано с помощью JSON колонки из-за большого количества различных параметров и типов объектов (рис. 5.). Финальная схема структуры базы данных изображена на рис. 6.

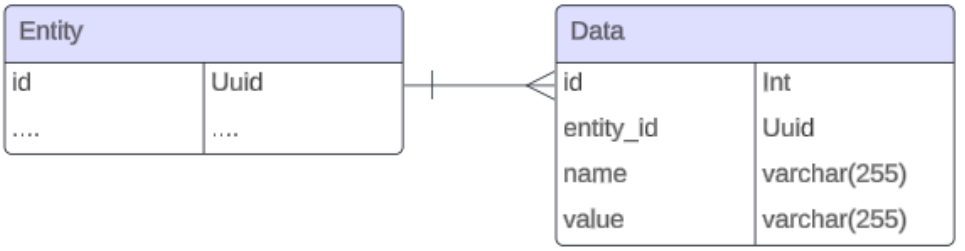

Рис. 4. Схема базы данных гибкой ролевой модели.
Fig. 4. The schema of the flexible role model database.

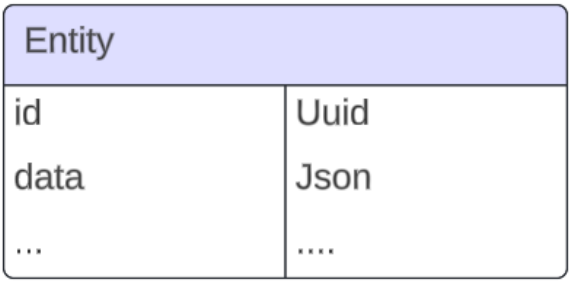

Рис. 5. Схема базы данных гибкой ролевой модели с использованием JSON поля.
Fig. 5. A flexible role model database schema using a JSON field.

Разберем назначение таблиц и полей в них.

User – Реализует сущность “Пользователь”. Хранит учетные записи пользователей и их набор полей. Назначение полей:

- id – Идентификатор пользователя. Тип Uuid.
- name – Имя пользователя.
- phone – Мобильный номер пользователя.
- created_at – Дата создания пользователя.
- role – Роль пользователя. Для упрощения понимания механизма список возможных ролей сокращен до 1 роли.

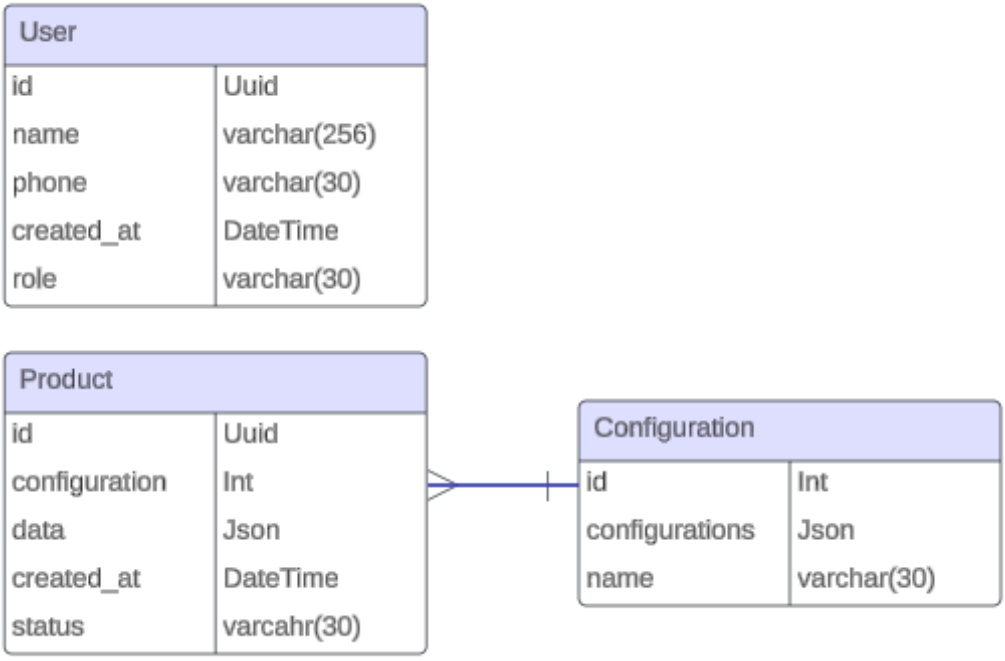

Рис. 6. Схема базы данных гибкой ролевой модели.
Fig. 6. Flexible Role Model database schema.

Product – Реализует сущность “Товар”. Хранит данные товаров, доступных на сайте. Назначение полей:

- id – Идентификатор товара. Тип Uuid.
- configuration – Тип конфигурации. Int ссылка на таблицу Configuration.
- data – Набор данных товара. JSON string.
- status – Текущий статус товара.
- created_at – Дата создания пользователя.

Configuration – Реализует сущность “Конфигурация”. Хранит данные о конфигурации сущности, требующей гибких настроек прав доступа, в данном примере будут демонстрироваться конфигурации только товаров. Назначение полей:

- id – Идентификатор конфигурации. Тип Int автоикремент.
- configurations – Набор конфигураций. JSON string.
- name – наименование конфигурации.

Таким образом каждый товар в системе имеет ссылку на конфигурацию и набор полей в формате JSON в поле data.

Формат поля Configurations для сущности “Конфигурация” показан на рис. 7, на котором можно видеть несколько блоков. Блок data отвечает за существующий набор полей доступных для сущности, их тип и способы валидации (constraints), а также значение по умолчанию (value). Тип полей и способы их валидации могут быть как самостоятельно написанными реализациями в проекте, так и в виде распространяемых библиотек, например, Symfony Forms [12] и Symfony Constraints [13]. Блок view отвечает за права работы с полями сущности в конкретном ее статусе (STATUS), для конкретной роли пользователя (USER_ROLE) и полем (FIELD_NAME) и имеет 2 возможных разрешения view и edit. Блок permissions отвечает за список доступных действий пользователя в конкретном статусе

сущности (Status), для конкретной роли пользователя (USER_ROLE) и определённому действию (PERMISSION_NAME).

Рассмотрим несколько примеров описания конфигураций для нескольких разновидностей товаров и их статусов. Допустим в рассматриваемом интернет-магазине существует 2 вида товаров: книги и телевизоры. Книги имеют следующий набор полей: автор, цена, количество. Телевизор имеет совершенно другой набор полей: диагональ, цена, размер, количество.

Для книг конфигурация будет иметь вид, показанный на рис. 8.

```
{ "data": { "FIELD_NAME": { "type": "text",
                            "constraints": [],
                            "value": null
            }
  },
  "view": { "STATUS": { "USER_ROLE": { "FIELD_NAME": [ "view","edit" ]
                      }
            }
  },
  "permissions": { "STATUS": { "USER_ROLE": [ "PERMISSION_NAME" ]
                 }
  }
}
```

Рис. 7. Формат поля Configurations для сущности “Конфигурация”.
Fig. 7. Configurations field format for “Configuration” entity.

```
{ "data": { "author": { "type": "text",
                        "constraints": [ “NotEmpty” ],
                        "value": null
            },
            "count":  { "type": "int",
                        "constraints": [],
                        "value": 0
            },
            "price":  { "type": "int",
                        "constraints": [ “min:1” ],
                        "value": 1
            }
  },
  "view": { "Available": { "User":    { "author": [ "view" ],
                                        "price":  [ "view" ],
                                        "count":  [ "view" ]        },
                           "Courier": { "count":  [ "view","edit" ] }
            },
          "NotAvailable": {
                           "User":    { "author": [ "view" ] },
                           "Courier": { "count":  [ "view","edit" ] }
            }
  },
  "permissions": { "Available": { "User":          [ "buy" ],
                                  "Courier":       [ "deliver" ]     }
  }
}
```

Рис. 8. Формат поля Configurations для сущности “Книга”.
Fig. 8. Configurations field format for “Book” entity.

Таким образом, тип товара книга доступен для пользователя в статусе Available для просмотра всех полей и возможности заказать, курьер не видит данных ненужных ему о книге, но может изменить доступное количество. В статусе NotAvailable пользователь видит только наименование автора и не может заказать себе товар.

Для товара телевизор отличием конфигурации будет исключительно в блоке view (см. рис. 9), то есть отличием будет исключительно набор данных, доступных для сохранения или отображения пользователю.

```
"view": { "Available": { "User": { "diagonal": [ "view" ],
                                   "price":    [ "view" ],
                                   "count":    [ "view" ]
                                 },
                         "Courier": { "count": [ "view", "edit" ],
                                   "size":  [ "view" ]
                                 }
          },
          "NotAvailable": {
                            "User": { "author": [ "view" ] },
                         "Courier": { "count":  [ "view", "edit" ] }
          }
}
```

Рис. 9. Формат поля Configurations для сущности “Телевизор”.
Fig. 9. Configurations field format for “TV set” entity.

Пример реализации работы с механизмом конфигурации продемонстрирован на языке программирования высокого уровня PHP, приведенный далее код достаточно просто адаптировать для других языков программирования, например, с помощью сервиса [14]. Получение данных, доступных пользователю с помощью блока view, проиллюстрировано на рис. 10. Проверка доступности операции с сущностью с помощью блока permissions показана на рис.11.

```
<?php
/**
          $data – данные из поля data в сущности
**/
function getDataByConfiguration(array $data, array $configurations,
                                string $userRole, string $status):array
{ $returnData = [];
  $configurations = $configurations['view'][$status][$userRole] ?? [];
// Получаем список конфигураций по статусу сущности и роли пользователя
  if (empty($configurations)) {
      return []; // В случае, если отсутствуют данные для роли и статуса
  }
  $returnData = [];
  foreach ($configurations as $key => $configuration) {
    if (in_array('view', $configuration)) {
        $returnData[$key] = $data[$key] ?? null;
// если пользователю доступен view данного поля, то добавляем в ответ
    }
  }
  return $returnData;
}
```

Рис. 10. Описание функции получения данных блока view на языке PHP.
Fig. 10. Get data function in PHP.

Работа с гибкой ролевой моделью затрагивает не только серверную частью приложения (backend), но и клиентскую (frontend). Необходимо решить проблему отображения полей и доступности действий, например, нет необходимости отображать поле author для роли Courier так, как оно ему недоступно. Есть несколько решений данной проблемы.

```
<?php
function isAllowed(string $rule, array $configurations,
                   string $userRole, string $status): bool
{ $returnData = [];
  $configurations = $configurations['view'][$status][$userRole] ?? [];
  if (empty($configurations)) {
      return [];
  }
  return $configurations[$rule] ?? false;
// Проверяем доступна ли операция
}
```

Рис. 11. Описание функции проверки доступности операции на языке PHP.
Fig. 11. isAllowed function in PHP.

Очевидным решением является передача набора конфигураций на клиентскую часть приложения, таким образом, клиент или мобильное приложение в зависимости от настроек сможет генерировать форму с данными. Для этого необходимо подготовить API и передавать список доступных действий пользователя и список доступных ему полей. Плюс данного подхода в простоте реализации и единообразии обработки конфигураций на клиентской и серверной частях приложения. Минусом является передача конфигурационных настроек клиенту, чем могут воспользоваться злоумышленники для поиска уязвимостей и взлома системы.

Для исключения передачи конфигураций на клиентскую часть приложения можно применить следующую реализацию. Сервер приложения будет передавать только те поля, которые доступны пользователю для его роли. Также можно добавить новый блок настроек, например, с названием guiActions (рис. 12).

Зная конфигурацию, можно возвращать клиенту определённый набор действий, доступных пользователю по данному объекту, благодаря чему на клиентской стороне могут отображаться доступные пользователю блоки или кнопки.

```
"guiActions ": { "STATUS": { "USER_ROLE": [ "ACTION1", "ACTION2", ] }
}
```

Рис. 12. Пример блока настроек.
Fig. 12. Example of the settings block.

Существенным минусом механизма гибкой системы управления доступами, который предлагается автором статьи, является большой объём конфигураций и у клиента, и у сервера. В ходе эксплуатации механизма было выявлено, что получение файла конфигураций по объектам с большим количеством настроек занимало значительный объём оперативной памяти. Для минимизации потребления оперативной памяти приходилось разрабатывать дополнительные механизмы обработки и получения конфигураций, что также сказывалось на быстродействии системы.

Несомненным плюсом механизма гибкого системы управления доступами, который описан в данной статье, является возможность добавлять множество дополнительных настроек или расширений. Например, в приложении появилась потребность добавить условия перехода сущности по статусам. В качестве решения можно описать новый блок statuses (рис. 13).

На рис. 13 FROM_STATUS – это статус, с которого будет осуществляться переход, а TO_STATUS_1 и TO_STATUS_2 статусы, на которые возможен переход со статуса

FROM_STATUS. Наименования REASON_1 и REASON_2 – это условия перехода, это может быть как наименование класса в системе, так и код, который должен быть исполнен для проверки возможности перехода.

```
"statuses": { "FROM_STATUS": { "TO_STATUS_1": [ "REASON_1" ],
                               "TO_STATUS_2": [ "REASON_2" ]
            }
}
```

Рис. 13. Пример блока настроек.
Fig. 13. Example of the settings block.

Разработанный механизм гибкой настройки прав доступа был успешно внедрён в нескольких проектах, связанных с управлением сложными ролевыми моделями. Его использование позволило сократить время на адаптацию системы под новые бизнес-требования, а также упростить процесс добавления новых сущностей и ролей. Например, часто в проект добавлялись новые роли, но при использовании данного механизма достаточно было обновить конфиги для работы нового типа пользователя.

Практика показала, что использование JSON-структур для хранения данных, а также чётко описанных конфигураций ролей и их прав, позволило избежать избыточности и сделать систему более производительной и масштабируемой.

4. Заключение

Предложенный механизм гибкой системы управления доступом объединяет преимущества моделей RBAC и ABAC, позволяя добиться одновременно гибкости и простоты настройки. Рассмотренные подходы к хранению конфигураций и параметров объектов в базе данных демонстрируют возможности для адаптации под различные сценарии использования, от простых интернет-магазинов до сложных корпоративных приложений.

Основным преимуществом системы является использование гибкой структуры для хранения параметров объектов, что обеспечивает скорость работы и удобство конфигурации, особенно при масштабировании. В то же время, гибкость достигается за счёт разделения настроек на права доступа к полям и действий по определенному статусу объекта, что позволяет учитывать роли пользователей, статусы объектов и конкретные операции.

Практическая реализация показала, что использование JSON для хранения данных может быть оптимальным решением для систем с большим количеством параметров и сложной логикой доступа. Однако важно учитывать потенциальные сложности, связанные с обработкой JSON на уровне базы данных и поддержанием целостности данных.

Список литературы / References

[1]. Что такое реляционная база данных. Доступно по адресу: https://selectel.ru/blog/relational-database, дата обращения: 27.03.2025. / What is a relational database. Available at: https://selectel.ru/blog/relational-database (in Russian), accessed 27.03.2025.
[2]. Sandhu, R. Role-based access control models. IEEE Computer Society Press, 1996, pp. 1-21.
[3]. Ferraiolo, D. F., Sandhu, R. Assessment of Role-based Access Control Systems. IEEE Transactions on Software Engineering, 2007, pp. 1-11.
[4]. Калимолдаев М.Н., Бияшев Р.Г., Рог О.А. Анализ методов атрибутного разграничения доступов. Прикладная дискретная математика, 2019, стр. 43-56. / Kalimoldaev M.N., Biyashev R.G., Rog O.A. Analysis of methods of attribute access differentiation. Applied discrete mathematics, 2019, pp. 43-56 (in Russian). DOI: 10.17223/20710410/44/4.
[5]. Амелина А.А. Исследование решений для реализации ролевой модели системы контроля доступа к корпоративной информационной системе. Перспективы развития и применения современных технологий, 2021, стр. 32-38. / Amelina A.A. Research of solutions for the implementation of the role

model of the access control system to the corporate information system. Prospects for the development and application of modern technologies, 2021, pp. 32-38 (in Russian).

[6]. Кальдина Д.И. Ролевая модель управления доступом в системе конференций. ИТ. НАУКА. КРЕАТИВ. Материалы I международного форума: в 5-ти томах. т. 4. Молодёжь. Наука. Творчество, 2024, стр. 315-320. / Kaldina D.I. Role model of access control in the conference system. Materials of the international forum Youth. Science. Creativity: in 5 volumes, vol. 4, 2024, pp. 315-320 (in Russian).

[7]. Мухаммад А., Миков Д.А. Ролевая модель управления доступом в системе динамического архивирования. Инновационные научные исследования, 2021, с. 109-203. / Muhammad A., Mikov D.A. A role-based access control model in a dynamic archiving system. Innovative Scientific Research, 2021, pp. 109-203 (in Russian).

[8]. Иргашева Д.Я., Усманов А.К. Разработка ролевой модели с зональным разграничением доступа. Наука и мир, 2016, стр. 35-40. / Irgasheva D.Ya., Usmanov A.K. Development of a role model with zonal access control. Nauka i Mir, 2016, pp. 35-40 (in Russian).

[9]. Документация инструмента AWS S3. Доступно по адресу: https://aws.amazon.com/ru/s3. Дата обращения: 27.03.2025. / Documentation of the AWS S3 tool. Available at: https://aws.amazon.com/ru/s3, accessed: 27.03.2025.

[10]. Исследуем производительность JOIN в MySQL. Доступно по адресу: https://habr.com/ru/articles/122210/. Дата обращения: 30.11.2024. / Let's explore the performance of JOIN in MySQL. Available at: https://habr.com/ru/articles/122210, accessed: 30.11.2024.

[11]. Как реляционная СУБД делает JOIN? Доступно по адресу: https://habr.com/ru/articles/560834/. Дата обращения: 30.11.2024. / How does a relational database do a JOIN? Available at: https://habr.com/ru/articles/560834, accessed: 30.11.2024.

[12]. Веб-сайт Symfony Forms. Доступно по адресу: https://symfony.com/doc/current/forms.html. Дата обращения: 06.11.2024. / Symfony Forms website. Available at: https://symfony.com/doc/current/forms.html, accessed: 06.11.2024.

[13]. Веб-сайт Symfony Constraints. Доступно по адресу: https://symfony.com/doc/current/reference/constraints.html. Дата обращения: 06.11.2024. / Symfony Constraints website. Available at: https://symfony.com/doc/current/reference/constraints.html, accessed: 06.11.2024.

[14]. Онлайн конвертер кода. Доступно по адресу: https://syntha.ai/. Дата обращения: 04.11.2024. / Online code converter. Available at: https://syntha.ai, accessed: 04.11.2024.

Информация об авторах / Information about authors

Тимур Игоревич ВАСИЛЬЕВ – выпускник Казанского национального исследовательского технического университета – КАИ имени Туполева в городе Казани, где получил степень магистра по специальности "Информатика и вычислительная техника". В настоящее время является ведущим разработчиком в финтех компании Paybis.com.

Timur Igorevich VASILIEV got his Master degree in Computer Science and Engineering from the Tupolev Kazan State Technical University in Russia. Currently, he is a leading developer in a fintech company Paybis.com.