Санкт-Петербургский государственный университет

Кафедра системного програмирования

Группа 22.Б15-мм

Проектирование и реализация архитектуры нового веб-сервера Desbordante

Нурмухаметов Рафик

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры информационно-аналитических систем, Чернышев Г.А.

Санкт-Петербург
2024

Оглавление

Введение

Архитектура программных систем играет ключевую роль в их устойчивости, масштабируемости и адаптивности. Правильно организованная архитектура минимизирует зависимости между компонентами, улучшая модульность и упрощая процесс внесения изменений. Она также существенно улучшает процесс тестирования: хорошо спроектированные модули легко изолировать для проведения модульных тестов, что позволяет проверять корректность работы отдельных частей системы независимо друг от друга. Это снижает риск регрессий и ускоряет выявление ошибок, что особенно важно в условиях динамичного развития и постоянных обновлений системы.

Ошибки, допущенные на этапе прототипирования и разработки серверной части веб-версии Desbordante — наукоёмкого профилировщика данных, написанного на C++, — привели к множеству проблем, связанных с неудачной архитектурной организацией. Приложение представляло собой комплекс взаимозависимых компонентов, реализованных на различных языках программирования и с использованием разных технологий. Такая гетерогенная, сильно связанная архитектура системы осложняла процесс разработки: любое изменение в одной части требовало корректировок в других, что увеличивало риск ошибок, снижало гибкость системы и повышало трудозатраты.

В связи с замедлением развития веб-версии было принято решение о полной переработке серверной части и её переписывании с нуля. Основным языком для нового сервера выбран Python, что обоснованно существованием Python-модуля, являющегося обвязкой для основного приложения Desbordante на C++. Это позволит выполнять задачи профилирования на Python, улучшив тем самым интеграцию с вычислительным ядром. На момент начала работы уже была реализована минимально жизнеспособная версия нового сервера, поддерживающая инициализацию и выполнение задач профилирования. Однако, так как это лишь начальная версия, четкой архитектурной организации у нового сервера не было. Из-за чего сохранилась проблема сильной зависимо-

сти между компонентами, что по-прежнему приводило к сложностям при внесении изменений, проблемам с тестированием, увеличению времени на доработку и повышению вероятности возникновения ошибок.

Целью данной работы является разработка и внедрение новой архитектуры серверной части, которая устранит ключевые недостатки предыдущей реализации, такие как взаимозависимость компонентов и отсутствие модульности. Для этого предполагается использовать современные архитектурные подходы и паттерны проектирования, что позволит создать более гибкую и легко тестируемую систему. Это упростит добавление новых функций, сократит технический долг и повысит устойчивость к изменениям. Таким образом, данная работа стремится заложить прочный фундамент для последующей разработки и роста проекта, обеспечивая его стабильность и адаптивность в будущем.

1. Постановка задачи

Целью работы является проектирование и реализация архитектуры нового веб-сервера Desbordante. Для её выполнения были поставлены следующие задачи:

1. Провести анализ архитектур предыдущих решений с целью выявления ключевых недостатков
2. Определить требования к новой архитектуре
3. Сделать обзор основных подходов к проектированию систем
4. Спроектировать и реализовать новую архитектуру
5. Провести тестирование компонентов системы как по отдельности, так и в совокупности

2. Обзор и анализ существующих решений

2.1. Устройство старого сервера

Серверная часть веб-приложения Desbordante построенна с помощью нескольких, в разной степени взаимосвязанных компонент, для разработки и поддержки каждой из которых испольуются различные языки и технологии. На рисунке 1 показано общее устройство старой версии веб-приложения Desbordante.

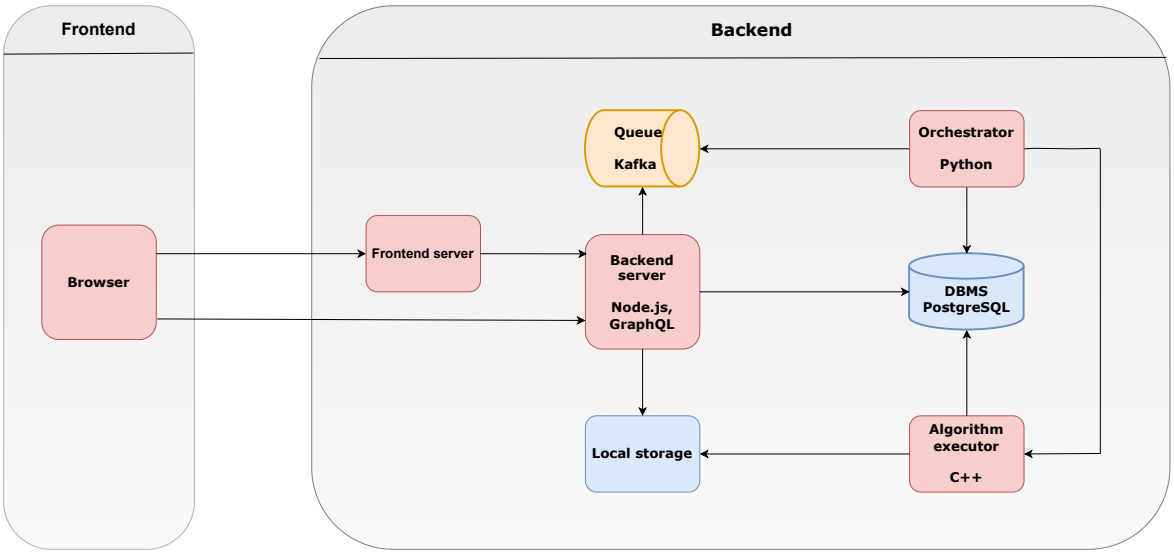

Рис. 1: Устройство старого веб-приложения Desbordante.

2.1.1. Основное приложение

Основной компонентой серверной части является сервис, предоставляющий интерфейс для взаимодействия с клиентом, разработанный на платформе Node.js [7]. Первостепенной функцией данного сервиса является создание и управление задачами, которые в контексте Desbordante представляют собой профилирование определённого набора данных. Этот процесс охватывает инициализацию новых задач и мониторинг их выполнения. Кроме того, сервис обеспечивает управление данными пользователей и контроль прав доступа к системным функциям. Он также отвечает за сохранение информации о загруженных файлах и

их конфигурациях.

Взаимодействие с клиентской частью осуществляется через GraphQL [5], что обеспечивает высокую степень гибкости в обработке запросов и оптимизации работы системы. Данный подход позволяет клиенту запрашивать только те данные, которые необходимы для выполнения конкретных операций, что минимизирует объем передаваемой информации и сокращает время отклика.

Этот компонент, как и другие элементы системы, полностью зависит от выбранной системы управления базами данных (далее СУБД) и, забегая вперед, от библиотеки, предоставляющей объектно-реляционное отображение. В случае необходимости перехода на другую СУБД или смены типа хранилища, например, от реляционных баз данных к альтернативным решениям, потребуется частичное или полное переписывание данного компонента, а также всех связанных с ним элементов системы.

Подобные сервисы, предоставляющие интерфейсы для клиента, не предназначены для выполнения сложных вычислений, поскольку их основная задача заключается в обработке запросов. Задача профилирования данных является сложным процессом, требующим значительных временных и ресурсных затрат. Поэтому, в контексте данного сервиса, под “созданием задачи” подразумевается лишь её инициализация и отправка в специализированную очередь, где будет производиться дальнейшая обработка с использованием выделенных вычислительных ресурсов.

2.1.2. Очередь задач

Инициализированные задачи из веб-сервиса, предоставляющего интерфейс для взаимодействия с клиентом(далее в данном пункте просто “веб-сервис”), поступают в очередь задач, организованную с использованием распределённого брокера сообщений Apache Kafka [1]. Оркестратор, являющийся ещё одним элементом системы, извлекает задачи из этой очереди. Таким образом, Apache Kafka выполняет роль посредника между веб-сервисом и оркестратором.

Apache Kafka использует модель производитель-потребитель, которая подразумевает, что производитель отправляет новые события в очередь, а потребитель обрабатывает их по мере поступления. Когда потребитель получает событие из очереди, оно не считается обработанным до тех пор, пока не произойдёт его коммит — процесс, который отмечает, что событие успешно обработано и в следующий раз потребитель получит следующее событие из очереди. Коммит может происходить автоматически сразу после получения события или вручную после завершения обработки.

В текущей реализации используется автоматический коммит — событие помечается как обработанное сразу же после его поступления в потребитель. Это упрощает реализацию, но создаёт серьёзные риски: если потребитель по какой-то причине аварийно завершится до завершения обработки задачи, событие уже закоммичено, и его невозможно будет повторно обработать. В результате, задача будет потеряна. При этом Apache Kafka, написанная на Java, требует существенных вычислительных ресурсов для работы, и при минимальном использовании её функций система становится неоправданно тяжёлой. С таким же успехом можно было бы использовать любую СУБД.

На первый взгляд, несмотря на проблему с автоматическим коммитом, архитектура выглядит продуманной, однако как веб-сервис, так и оркестратор непосредственно взаимодействуют с Apache Kafka, что создаёт значительную зависимость от данного компонента. Это означает, что любые изменения, которые могут понадобиться в структуре или функциональности очереди задач, потребуют частичного или полного переписывания кода как веб-сервиса, так и оркестратора. Эта зависимость усложняет процесс сопровождения системы, увеличивая риски возникновения ошибок и затраты на поддержку.

2.1.3. Оркестратор

Задачи из очереди обрабатываются оркестратором, реализованным на языке программирования Python. Этот компонент системы выполняет подготовку к исполнению задач и мониторит их выполнение. Ор-

кестратор запускает Docker-контейнеры, в которых происходит выполнение задач, и передает исполнителю алгоритма все необходимые данные, включая конфигурацию. Он также отслеживает состояние контейнеров и процесс выполнения, и при возникновении ошибок в работе алгоритма обновляет соответствующую информацию о задаче в базе данных.

Как было отмеченно ранее, данный компонент, как и все остальные зависит от выбранной СУБД и библиотеки, предоставляющей объектно-реляционное отображение.

2.1.4. Исполнитель алгоритма

Исполнение основной задачи, то есть непосредственное выполнение алгоритма, осуществляется специализированным модулем, реализованным на C++, в Docker-контейнере, запущенном оркестратором. Этот модуль получает необходимые конфигурации и входные данные из базы данных, выполняет вычисления, а затем сохраняет результаты обратно в базу.

2.1.5. База данных

В веб-приложении Desbordante для хранения и управления данными используется реляционная система управления базами данных PostgreSQL [9]. Основным инструментом для работы с этой базой данных являются библиотека Sequelize [12], предоставляющая объектно-реляционное отображение (Object-relational mapping, ORM), что позволяет разработчикам взаимодействовать с базой данных, представляя её как набор объектов и их связей, а не просто как таблицы и SQL-запросы.

База данных Desbordante имеет архитектурное разделение на три ключевые области. Первая область включает таблицы, которые содержат информацию о пользователях, их устройствах, активных сеансах и прочую информацию. Вторая область сосредоточена на таблицах, хранящих встроенные и загруженные пользователями данные для профи-

лирования. Третья область охватывает таблицы, в которых фиксируются активные и завершенные задачи, их подзадачи, настройки и результаты выполнения. Тем не менее, последняя область данных страдает от необоснованной избыточности таблиц и как следствие неэффективным хранением данных. Каждая конфигурация и результат работы различных примитивов имеют свои отдельные таблицы, что усложняет структуру и затрудняет ее понимание, делая её неоправданно сложной.

2.2. Устройство нового сервера

К началу данной работы был инициирован процесс полной переработки серверной части на языке программирования Python, обусловленный выявленными недостатками предыдущей реализации. Выбор Python был сделан ввиду возможности интеграции с новым Python-модулем, обеспечивающим взаимодействие с основным приложением Desbordante, разработанным на языке C++. На данном этапе была уже создана минимально жизнеспособная версия нового сервера, предоставляющая функциональность для инициализации и выполнения задач профилирования.

Несмотря на частичное устранение проблем предыдущей реализации, таких как сложность и продолжительность процесса добавления новых примитивов и их алгоритмов, архитектурные недостатки остались до конца нерешёнными. Поскольку была реализована лишь минимально жизнеспособная версия продукта, она не обладала полноценной архитектурной организацией. В результате компоненты системы продолжали оставаться тесно связанными, что препятствовало созданию модульной и гибкой архитектуры. На рисунке 2 показано общее устройство нового веб-версера Desbordante.

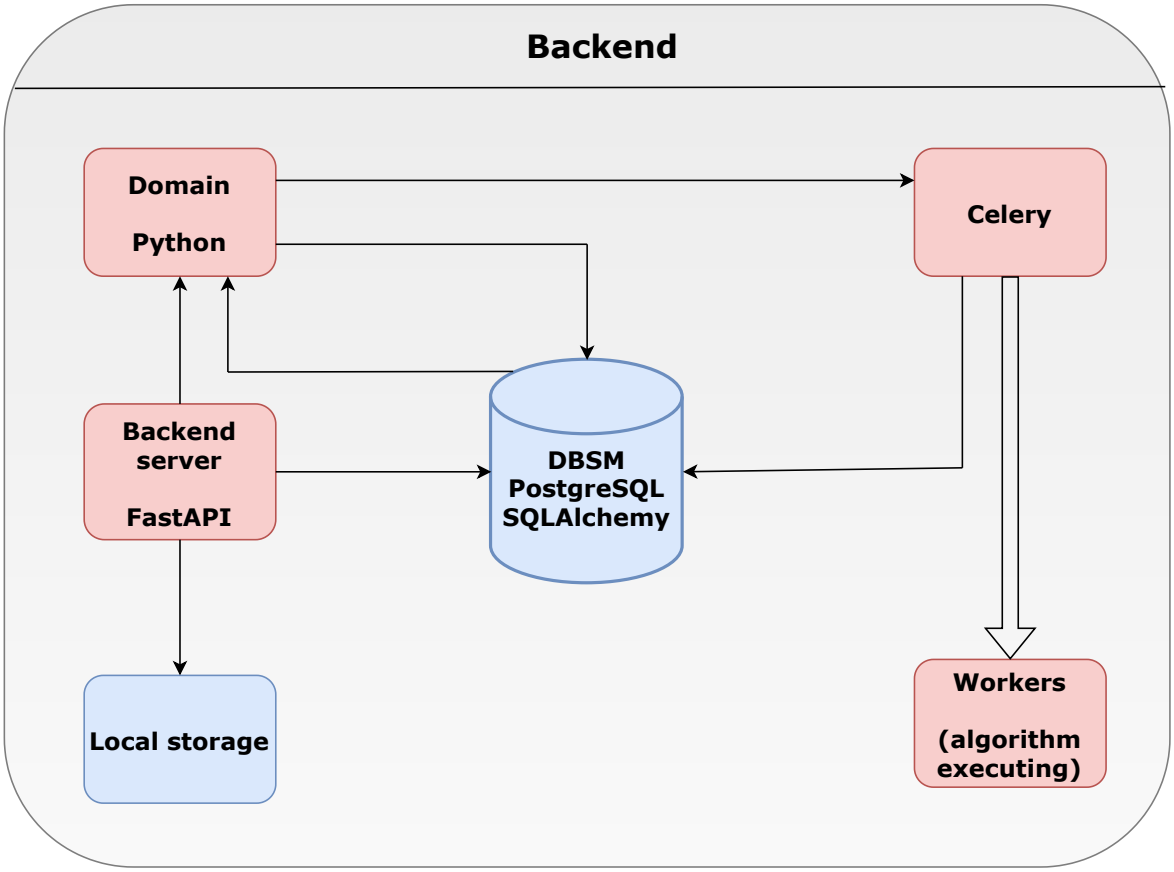

Рис. 2: Устройство нового веб-сервера Desbordante

2.2.1. Взаимодействие с клиентом

Для взаимодействия с клиентом в системе был выбран подход на основе REST API(REpresentational State Transfer) [8], который предоставляет возможность обмена данными между клиентом и сервером через стандартные HTTP-запросы. Этот архитектурный стиль позволяет разделить логику обработки данных и пользовательский интерфейс.

В качестве инструмента для реализации REST API был выбран фреймворк FastAPI [4]. Он позволяет достаточно быстро и эффективно разрабатывать API, относительно других веб-фреймворков на Python. Одним из ключевых преимуществ FastAPI является автоматическая генерация документации на основе аннотаций типов, что упрощает процесс разработки, тестирования и взаимодействия с другими разработчиками.

2.2.2. Фоновые задачи

Для выполнения задач профилирования данных в системе применяется Celery [2] — система для распределённой обработки задач. Celery позволяет выносить трудоёмкие или продолжительные операции в отдельные очереди, обеспечивая их выполнение в фоновом режиме. Благодаря Celery обработка данных отделяется от основного потока приложения, что не только повышает производительность системы, но и улучшает её отзывчивость, позволяя основному приложению продолжать работу без задержек.

2.2.3. База данных

База данных была значительно изменена в отношении таблиц, связанных с выполнением задач профилирования [13].

2.3. Требования к новой архитектуре

Важно, чтобы новая архитектура устраняла проблемы предыдущей реализации, такие как высокая взаимозависимость компонентов и сложность внесения изменений, откуда, как следствие, вытекает плохая тестируемость. Для этого ставятся следующие требования:

1. Разделение ответственности. Архитектура должна строго разделять логику приложения, позволяя каждому компоненту выполнять свою специфическую задачу без вмешательства в работу других модулей

2. Минимизация связности. Необходимо снизить зависимость между компонентами, чтобы изменение одного модуля минимально влияло на другие. Это повысит гибкость системы

3. Устойчивость к изменениям. Архитектура должна позволять легко интегрировать новые функциональные возможности без необходимости кардинального изменения существующих модулей

4. Тестируемость. Архитектура должна облегчать процесс тестирования, предоставляя возможность создавать независимые и изолированные тесты для различных компонентов системы. Это будет способствовать повышению надёжности системы

5. Производительность. Архитектура должна сохранять текущую производительность при выполнении задач профилирования

2.4. Обзор основных подходов к проектированию систем

Выбор архитектурного подхода представляет собой сложную задачу, учитывающую разнообразие существующих вариантов. В данном подразделе представлен обзор основных принципов и методологий, применяемых для проектирования архитектуры систем, а также различных паттернов, используемых для реализации функциональных возможностей. Это позволит глубже понять подходы к организации систем и сделать обоснованный выбор для решения поставленных задач.

2.4.1. MVC

Паттерн проектирования Model-View-Controller (MVC), впервые предложенный Трюгве Реенскаугом в 1979 году [11], представляет собой одну из старейших и наиболее известных архитектурных концепций. Его основная идея заключается в разделении приложения на три ключевых компонента: модель, представление и контроллер. Это разделение служит для повышения модульности, а также для отделения бизнес-логики от интерфейса пользователя, что способствует упрощению разработки, тестирования и сопровождения программного обеспечения.

Существуют различные модификации и производные от паттерна MVC, такие как MVP (Model-View-Presenter), MVVM (Model-View-ViewModel), MVA (Model-View-Adapter) и другие. Несмотря на отличия подходов к взаимодействию между компонентами и способом организа-

ции представления, они все преследуют одну цель — разделить бизнес-логику от представления.

На рисунке 3 показана схема одной из вариаций MVC.

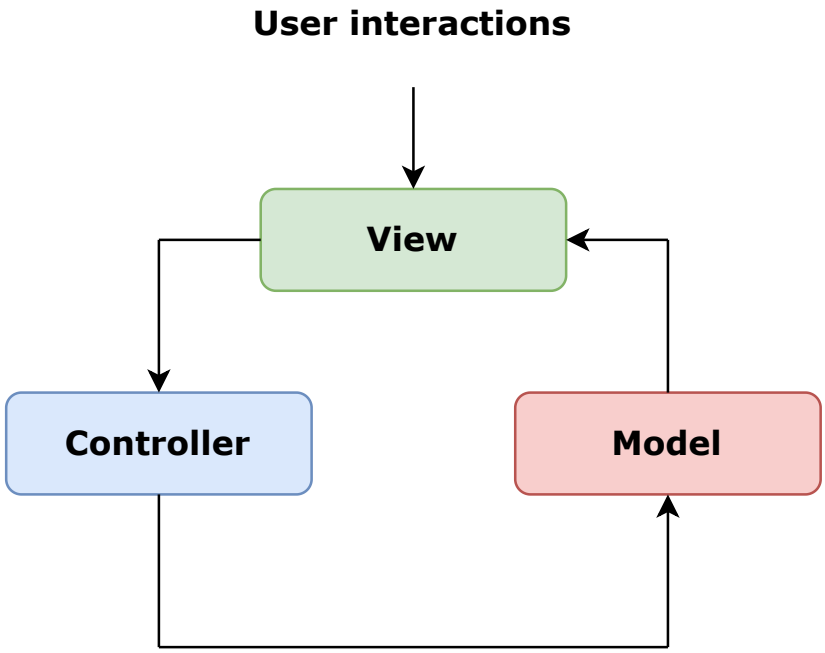

Рис. 3: Схема MVC

На момент написания работы паттерн MVC продолжает активно использоваться, сохраняя свою базовую структуру с момента разработки в 1979 году. Однако произошли некоторые изменения:

1. Упрощение представления. С течением времени процесс создания представлений стал проще благодаря развитию технологий и инструментов для разработки интерфейсов.

2. Упрощение контроллеров: Современные контроллеры стремятся к минимальной цикломатической сложности (идеально — равной единице), выступая в роли посредников между интерфейсом и внутренней логикой.

3. Усложнение моделей: Модели стали более сложными в ответ на возрастающие требования бизнеса.

2.4.2. Трехслойная архитектура

Трёхслойная архитектура, предложенная в 2002 году [10], представляет собой усовершенствованный подход к организации программных систем. В отличие от паттерна MVC, где возможны циклические зависимости между компонентами, в трёхслойной архитектуре слои строго разделены, что обеспечивает чёткую иерархию и отсутствие циклических связей. Приложение делится на три независимых слоя: пользовательский интерфейс, бизнес-логика и доступ к данным, каждый из которых использует собственные тактические шаблоны проектирования, что добавляет дополнительную сложность по сравнению с классическим MVC.

Ключевым нововведением трёхслойной архитектуры является чёткое отделение бизнес-логики от уровня работы с базой данных. Это разделение позволяет снизить зависимость между слоями. На рисунке 4 представлена схема слоев в трёхслойной архитектуре.

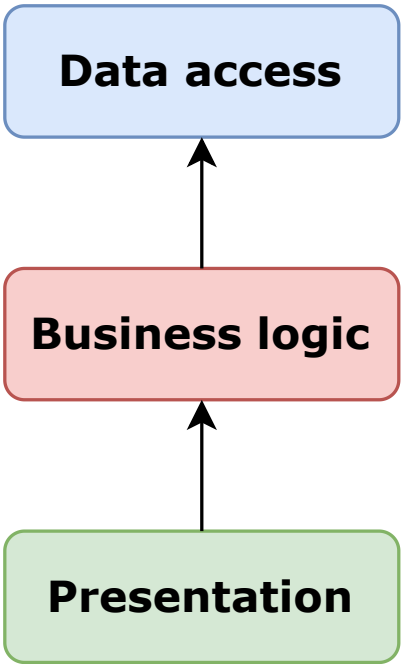

Рис. 4: Схема трёхслойной архитектуры

2.4.3. Порты и адаптеры

Архитектура портов и адаптеров, также известная как гексагональная архитектура, была представлена в 2005 году Альбертом Олифантом [3]. В рамках этой архитектуры впервые было чётко выделено понятие домена — центрального ядра системы, функционирующего в полной изоляции от технических деталей. Ключевая идея заключается в разделении бизнес-логики и инфраструктурных элементов, что способствует лучшей модульности и тестируемости системы.

В отличие от традиционных архитектур с иерархией слоев, гексагональная архитектура строится на принципе последовательного вызова: от пользовательского интерфейса, через бизнес-логику к внешним компонентам, необходимым для выполнения бизнес-операций. Одним из важных аспектов этой архитектуры является инверсия зависимости между бизнес-логикой и слоем доступа к данным. Считается, что уровень доступа к данным — это всего лишь инфраструктурная деталь, которая не должна напрямую влиять на бизнес-логику.

Центральное положение бизнес-логики требует взаимодействия с внешним миром, что достигается через порты — интерфейсы, декларируемые бизнес-логикой для взаимодействия с внешними системами. Внешние компоненты реализуют адаптеры, подключающиеся к этим портам, обеспечивая необходимую интеграцию. Поскольку количество портов может варьироваться, архитектура часто изображается в виде шестиугольника, где каждая сторона представляет интерфейс взаимодействия между бизнес-логикой и внешними системами. На рисунке 5 изображена подобная иллюстрация данной архитектуры.

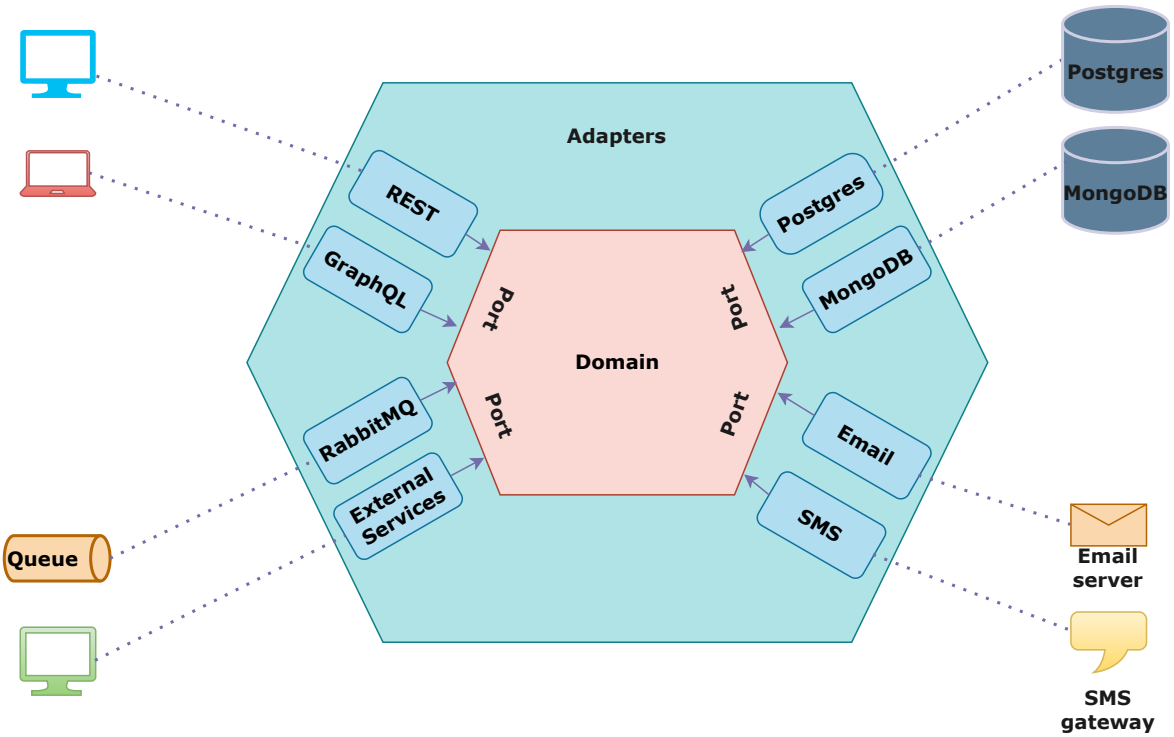

Рис. 5: Схема архитектуры портов и адаптеров.

2.4.4. Чистая архитектура

Чистая архитектура — это архитектурный стиль, предложенный американским инженером и программистом Робертом Мартином в своей книге “Clean Architecture: A Craftsman’s Guide to Software Structure and Design” [6], направленный на создание гибких и легко поддерживаемых приложений. Основная идея заключается в строгом разделении приложения на независимые слои, чтобы изменения в одной части системы минимально затрагивали другие. Это позволяет улучшить тестируемость, масштабируемость и долговечность системы, а также упрощает внесение изменений и модернизацию кода. Чистая архитектура является продолжением идеи трёхслойной архитектуру и архитектуры портов и адаптеров, однако отличается от них дополнительными слоями и некоторыми деталями (см. ниже).

Основные принципы чистой архитектуры включают инкапсуляцию бизнес-логики, независимость от фреймворков и деталей инфраструктуры, разделение на слои, инверсию зависимостей и лёгкость тестиро-

вания. Эти принципы обеспечивают создание архитектуры, где бизнес-логика не зависит от внешних технических решений, таких как базы данных или пользовательский интерфейс, что позволяет адаптировать приложение к изменяющимся технологиям без затрагивания основной логики.

На рисунке 6 представлена схема чистой архитектуры.

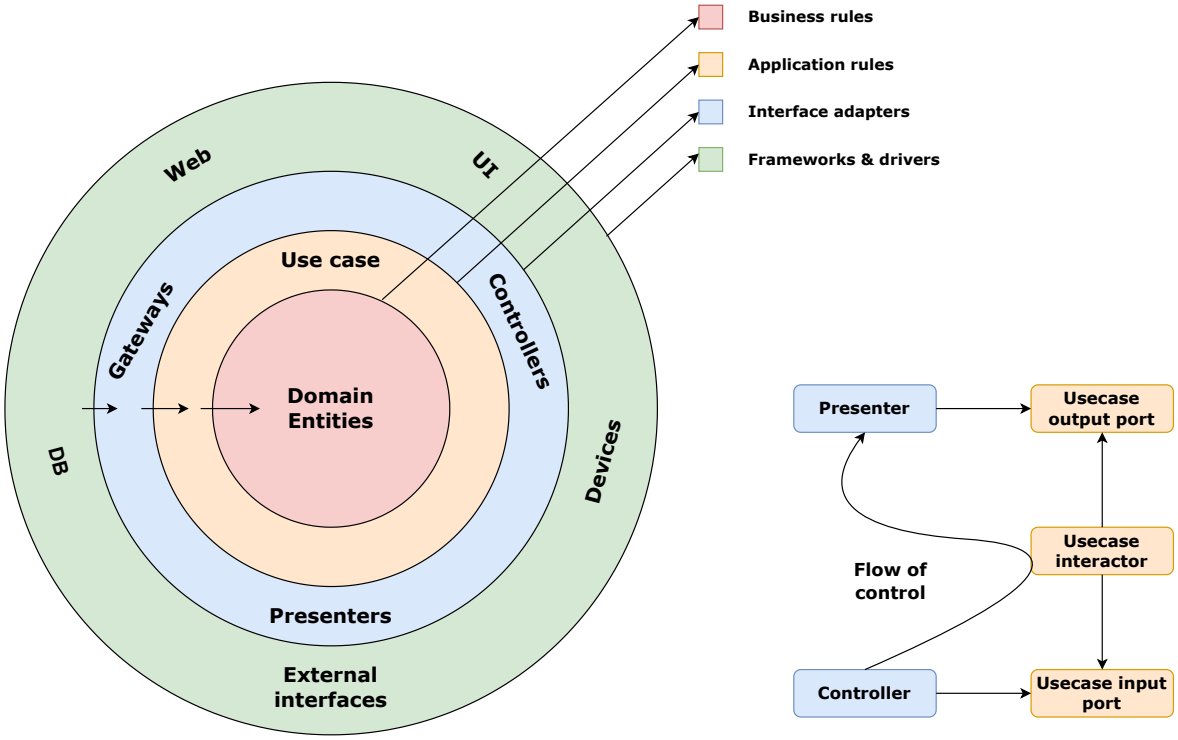

Рис. 6: Схема чистой архитектуры

Чистая архитектура делится на несколько слоев. В центре находятся сущности (entities), которые представляют бизнес-правила и могут существовать независимо от технологий. Вокруг них располагается слой сценариев использования (use cases), который определяет, как бизнес-логика взаимодействует с внешними системами. Далее следуют интерфейсы (interface adapters), которые служат мостом между бизнес-логикой и внешними системами, такими как базы данных и пользовательские интерфейсы. На внешнем слое находятся инфраструктура и фреймворки, которые предоставляют средства для работы с файлами,

базами данных и сетевыми взаимодействиями.

Одним из ключевых аспектов чистой архитектуры является инверсия зависимостей. Внутренние слои (сущности и сценарии использования) не зависят от внешних слоев, наоборот, внешние слои зависят от внутренних через интерфейсы. Это позволяет сохранить бизнес-логику неизменной даже при изменении технических деталей. Такой подход часто называют чистой зависимостью, так как он позволяет изолировать ядро системы от внешних факторов.

Одним из ключевых преимуществ чистой архитектуры является улучшенная тестируемость. Поскольку каждый слой архитектуры изолирован от других, можно легко тестировать компоненты системы независимо друг от друга. Внутренние слои, такие как бизнес-логика, не зависят от внешних систем, что позволяет использовать заглушки или мок-объекты для имитации их работы во время тестирования. Это упрощает написание модульных тестов и повышает их эффективность, так как тестирование, например, бизнес-логики и сценариев, не зависит от деталей реализации, таких как база данных или API.

Чистая архитектура покрывает все требования предъявленные к новой архитектуре — разделение ответственности, минимизация связанности, тестируемость, устойчивость к изменениям, производительность. Однако у неё также есть недостатки. Самый существенный — это сложность на этапе проектирования и разработки, особенно для небольших приложений.

2.4.5. Репозиторий

Репозиторий — это паттерн проектирования, который используется для абстракции доступа к данным в системе. Его основная задача — инкапсулировать логику взаимодействия с источниками данных, такими как базы данных, файлы или сторонние сервисы, и предоставить единый интерфейс для управления данными. Например, вместо того, чтобы напрямую взаимодействовать с SQL-запросами или API, бизнес-логика обращается к методам репозитория, у которого может быть любая реализация, но нам это в контексте работы с бизнес-логикой должно

быть неважно.

Таким образом, репозиторий скрывает детали реализации хранилища, будь то реляционная база данных, NoSQL или файловая система. Благодаря этому бизнес-логика не зависит от конкретного типа хранилища, что делает код гибким и легко тестируемым. Это позволяет заменять или модифицировать хранилище данных без влияния на основную логику приложения.

2.4.6. Data Transfer Object

Data Transfer Object (DTO) — это паттерн проектирования, используемый для передачи данных между различными слоями или компонентами архитектуры приложения. Его основная задача — служить контейнером для данных, который перемещается между слоями системы, не нарушая их изоляции и ответственности. DTO помогает отделить внутренние сущности от внешних представлений данных, например, при взаимодействии с пользовательским интерфейсом, внешними сервисами или базами данных.

Важной характеристикой DTO является отсутствие у него любой бизнес-логики. Это просто объект с данными, который служит мостом между слоями, такими как слой представления и слой бизнес-логики, или между бизнес-логикой и инфраструктурными компонентами (например, базой данных или API). В рамках архитектуры DTO упрощает процесс передачи данных, минимизирует количество зависимостей и позволяет передавать несколько полей сразу, объединяя их в один объект, что улучшает производительность системы.

2.4.7. Unit of work

Unit of work (UoW, Единица работы) — это паттерн проектирования, который используется для управления транзакциями в системе, собирая изменения, произведенные бизнес-логикой, и гарантируя их единое завершение или откат. Основная цель этого паттерна — координировать работу с несколькими репозиториями или сущностями в рамках одной

транзакции, минимизируя количество обращений к системе хранения данных и контролируя их атомарность.

В паттерне Unit of work отслеживаются все изменения в данных (создание, обновление, удаление) за определённый промежуток времени. Затем, по завершению работы, все изменения фиксируются или отменяются в зависимости от успешности выполнения операции. Например, если в бизнес-сценарии используются два и более репозитория, возможно с разной реализацией для разных систем хранения данных, и есть необходимость выполнить несколько изменений через эти репозитории целостно — с этой целью и используется UoW.

Таким образом, Unit of Work гарантирует целостность данных, помогая избежать частичных изменений и проблем с согласованностью, которые могут возникнуть при работе с множеством объектов и источников данных. На рисунке 7 продемонстрирована схема работы с хранилищем данных без репозитория и с ним, а также с репозиторием и UoW.

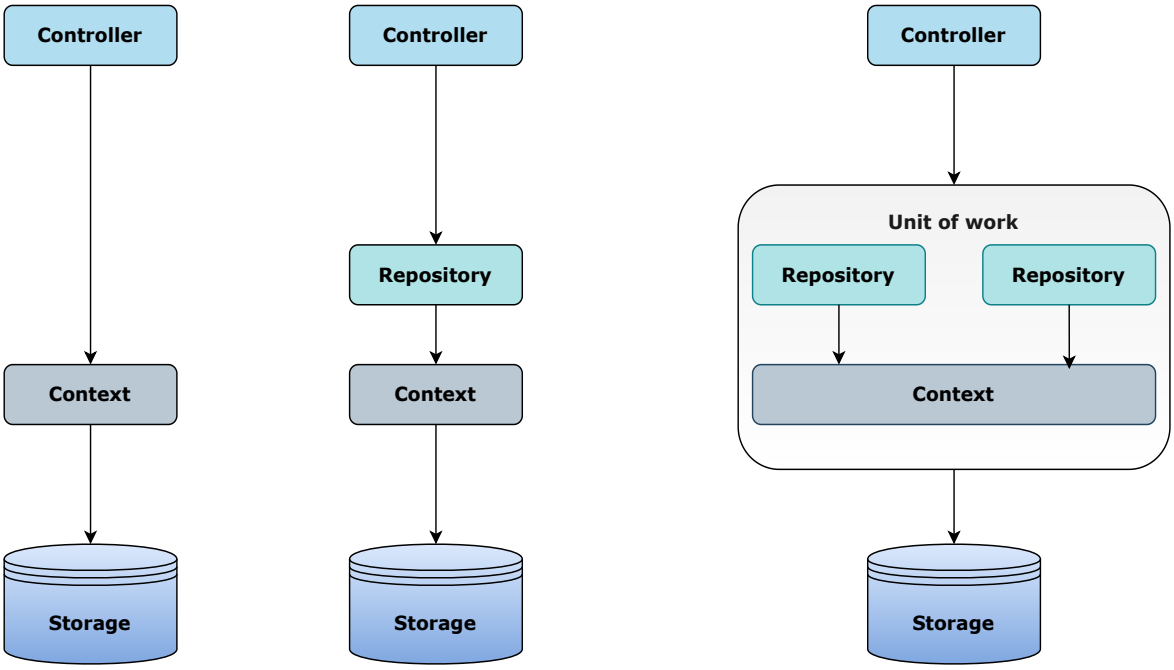

Рис. 7: Работа с хранилищем данных в трех вариантах: без репозитория, с репозиторием, с репозиторием и unit of work.

Под “контекстом” на рисунке 7 понимается некоторая сущность, че-

рез которую возможно осуществление взаимодействия с системой хранения данных. Например, в первом случае, где происходит прямое взаимодействие с системой хранения данных — это может быть просто объект из ORM, предоставляющий соединение с базой данных. Если хранилище данных предоставляет возможность использования транзакций, то их реализация как раз и находится в контексте.

2.4.8. Выбор подходов для проектирования новой архитектуры

Чистая архитектура и архитектура портов и адаптеров (гексагональная архитектура) представляются наиболее подходящими решениями для нового сервера Desbordante, поскольку, в отличие от паттернов семейства “MV*”, они разделяют не только бизнес-логику от представления, но и обеспечивают разделение от инфраструктурных деталей, таких как база данных или сторонние сервисы. В отличие от трехслойной архитектуры, которая отделяет бизнес-логику от представления, но сохраняет зависимость от базы данных, данные подходы делают бизнес-логику центральным элементом системы и независимой от внешних систем и технологических деталей. Это обеспечивает высокую тестируемость, расширяемость и устойчивость к изменениям.

Чистая архитектура и гексагональная архитектура действительно очень похожи по концепциям. Однако, было принято решение использовать чистую архитектуру, так как она представляет собой более гибкую и универсальную модель, которая расширяет и уточняет идеи гексагональной архитектуры. В то время как гексагональная архитектура концентрируется на отделении внешних интерфейсов от внутренней логики, чистая архитектура добавляет дополнительные уровни абстракции и четкое разграничение между доменными сущностями, кейсами использования и внешними компонентами системы. Это делает чистую архитектуру более пригодной для сложных систем, которые требуют модульности и расширяемости.

Для эффективной работы с чистой архитектурой будут использованы такие паттерны, как DTO для транспортировки данных между

слоями, репозитории для инкапсуляции доступа к данным, и Unit of Work для управления транзакциями и контекстом работы с хранилищем данных.

3. Описание решения

3.1. Доменный слой

Доменный слой системы включает в себя ключевую бизнес-логику и разделен на три поддомена: задача профилирования, файл и пользователь. Каждый из поддоменов содержит свои объекты-значения и сущности, которые отражают специфику соответствующих бизнес-процессов.

3.1.1. Задачи профилирования

В поддомене задач профилирования выделяются два типа объектов: объекты-значения и сущности. Объекты-значения, как это следует из названия, не заключают в себе логики и характеризуют параметры задач. Примером таких объектов может быть конфигурация алгоритма или примитива, которая используется для настройки конкретного профилирующего процесса. Сущности в данном поддомене — это классы, реализующие основную логику выполнения задач профилирования. По сути, сущности здесь представляют собой примитивы профилирования. В данной реализации сущности задач соответствуют применяемым сущностям в предыдущей версии проекта. Однако ключевым отличием является более четкое разделение и изоляция бизнес-логики от внешних систем.

Для того, чтобы создать сущность для нового примитива необходимо сначала реализовать объекты-значения, представляющие конфигурацию и вид результата для данного примитива, конфигурацию алгоритма и прочие объекты, интерфесы которых описаны в подмодуле объектов-значений. Затем, класс новой сущности наследуется от абстрактного класса, приведенного в листинге 1. При этом, обязательно, должны быть реализованы все его абстрактные методы и переданы реализованные типы для параметризованных в классе типов.

Листинг 1: Абстрактная сущность примитива.

```
from abc import ABC, abstractmethod
import desbordante
```

```
import pandas
from internal.domain.task.value_objects import TaskConfig, TaskResult

class Task[A: desbordante.Algorithm, C: TaskConfig, R: TaskResult](ABC):
    @abstractmethod
    def _match_algo_by_name(self, algo_name: str) -> A: ...

    @abstractmethod
    def _collect_result(self, algo: A) -> R: ...

    def execute(self, table: pandas.DataFrame, task_config: C) -> R:
        algo_config = task_config.config
        options = algo_config.model_dump(
            exclude_unset=True,
            exclude={"algo_name"}
        )
        algo = self._match_algo_by_name(algo_config.algo_name)
        algo.load_data(table=table)
        algo.execute(**options)
        return self._collect_result(algo)
```

В листинге 2 приведен пример реализованной сущности для примитива AFD.

Листинг 2: Сущность примитива AFD.

```
from desbordante.fd import FdAlgorithm  # This is not a typo
from desbordante.afd.algorithms import Pyro, Tane
from internal.domain.task.entities.task import Task
from internal.domain.task.value_objects import PrimitiveName
from internal.domain.task.value_objects.afd import AfdTaskResult, AfdTaskConfig
from internal.domain.task.value_objects.afd import (
    AfdAlgoName,
    AfdAlgoResult,
    FdModel,
    IncorrectAFDAlgorithmName,
)

class AfdTask(Task[FdAlgorithm, AfdTaskConfig, AfdTaskResult]):

    def _collect_result(self, algo: FdAlgorithm) -> AfdTaskResult:
        fds = algo.get_fds()
        algo_result = AfdAlgoResult(fds=list(map(FdModel.from_fd, fds)))
        return AfdTaskResult(primitive_name=PrimitiveName.afd, result=algo_result)

    def _match_algo_by_name(self, algo_name: str) -> FdAlgorithm:
        match algo_name:
```

```
            case AfdAlgoName.Pyro:
                return Pyro()
            case AfdAlgoName.Tane:
                return Tane()
            case _:
                raise IncorrectAFDAlgorithmName(algo_name)
```

3.1.2. Файл

Поддомен файлов в текущей версии системы реализует одну ключевую бизнес-логику — генерацию имени файла. Основная задача этой логики — обеспечить уникальность имен файлов, хранимых в системе. В настоящее время эта задача решается с помощью генерации универсальных уникальных идентификаторов (UUID), что гарантирует отсутствие конфликтов при именовании файлов в хранилище. Дальнейшее развитие поддомена может предусматривать более сложные схемы именования, включающие дополнительные атрибуты, такие как метаданные или типы файлов. В листинге 3 приведена реализация сущности файла.

Листинг 3: Сущность файла.

```
from uuid import uuid4, UUID

class File:

    def __init__(self):
        self._name = uuid4()

    @property
    def name(self) -> str:
        return str(self._name)

    @property
    def name_as_uuid(self) -> UUID:
        return self._name
```

3.1.3. Пользователь

На текущий момент поддомен пользователя не реализован, однако он тоже будет разделен на объекты-значения и сущности, аналогично

поддомену задач профилирования. Объектами-значениями могут стать классы, описывающие различные состояния и параметры пользователя, например, тип “статус аккаунта”. Сущностями в этом поддомене будут ключевые компоненты, такие как пользователь, сессия, менеджер по безопасности, менеджер по токенам и пользовательские устройства. Эти сущности будут ответственны за управление взаимодействием пользователя с системой, включая авторизацию, аутентификацию и управление сессиями.

3.2. Контекст хранилища данных

Контекст хранилища данных — объект, который играет роль “накопителя изменений”, которые применяются к системе хранения данных только после завершения транзакции. Если для данной системы хранения транзакции не поддерживаются или не реализованы, то изменения из контекста передаются в систему сразу же. Это своеобразный мост между системой хранения данных и, забегая вперед, репозиторием.

В системе описан интерфейс DataStorageContext, с которым можно ознакомиться в листинге 4:

Листинг 4: Контекст хранилища данных.

```
from typing import Protocol

class DataStorageContext(Protocol):

    def commit(self) -> None: ... # Успешно.

    def flush(self) -> None: ... # Применить.

    def rollback(self) -> None: ... # Откат.

    def close(self) -> None: ... # Закрытие.
```

Данный интерфейс подразумевает, что его конкретная реализация должна включать все четыре метода. На самом деле, обязательно должен реализовываться метод закрытия контекста, после которого использовать его будет не возможно. Остальные методы реализуются с

заглушками, если система хранения данных, для которой реализуется контекст не поддерживает транзакции.

Очевидно, что реализация данного интерфейса должна не только предоставлять методы для управления транзакциями, но и обеспечивать доступ к системе хранения данных. Это включает в себя логику внесения изменений, которые будут зафиксированы после вызова метода “commit”. Без этой функциональности реализация, хоть и будет удовлетворять интерфейсу формально, на самом деле не будет иметь практического смысла.

Для дальнейшей работы необходим объект, называемый ”фабрикой контекстов” — это функция или генератор, способнный создавать и возвращать новые контексты.

3.3. Репозиторий

Как уже было упомянуто в обзоре, репозиторий представляет собой слой абстракции, который инкапсулирует взаимодействие с системой хранения данных. Изначально интерфейсы репозиториев описываются с помощью протоколов (Protocol) непосредственно в файлах с соответствующими сценариями использования.

В контексте нашей реализации, все методы репозиториев, кроме параметров, специфичных для конкретной задачи, также должны принимать объект контекста хранилища данных. Контекст обеспечивает управление транзакциями: репозиторий передает информацию об изменениях в контекст, который фиксирует их в случае успешного выполнения транзакции или откатывает их в случае ошибки. Такой подход повышает гибкость системы и гарантирует целостность данных при работе с различными системами хранения.

В листинге 5 приведен пример интерфейса-репозитория, описывающего функцию для добавления нового датасета в некоторое хранилище.

Листинг 5: Пример интерфейса репозитория.

```
from internal.dto.repository.file import DatasetCreateSchema, DatasetResponseSchema
from internal.uow import DataStorageContext
```

```
class DatasetRepo(Protocol):

    def create(
        self, dataset_info: DatasetCreateSchema, context: DataStorageContext
    ) -> DatasetResponseSchema: ...
```

3.4. Worker

Аналогично репозиториям, “worker” представляет собой слой абстракции, но предназначеный не для взаимодействия с системами хранения данных, а для постановки и управления фоновыми задачами. Они также описываются с помощью протоколов в файлах с конкретными сценариями использования, и включают только те методы, которые необходимы для данного сценария.

В листинге 6 приведен пример интерфейса-работника, описывающего функцию для установления задачи профилирования в очередь.

Листинг 6: Пример интерфейса “worker”.

```
from internal.dto.worker.task import ProfilingTaskCreateSchema

class ProfilingTaskWorker(Protocol):

    def set(self, task_info: ProfilingTaskCreateSchema) -> None: ...
```

3.5. DTO

Паттерн DTO реализован с использованием библиотеки Pydantic, что обеспечивает удобство работы с валидацией и сериализацией данных.

Для реализации паттерна DTO был создан отдельный модуль, включающий описание всех необходимых структур для взаимодействия с репозиториями и “работниками”. Это обеспечивает единообразие форматов данных и упрощает взаимодействие между слоями системы. В этом же модуле определены исключения, которые могут возникнуть

при работе с репозиториями и “работниками”, что позволяет эффективно управлять ошибками и повышает надежность системы.

Используется данная реализация по следующей схеме: конкретный сценарий использования получает данные в некотором виде (каждый сценарий описывает этот вид самостоятельно) и преобразует их в формат, понятный репозиториям и “работникам” — это и есть объект DTO. После преобразования эти данные передаются соответствующим интерфейсам репозиториев и “работников”. Такой подход обеспечивает четкое разграничение данных между слоями системы и упрощает их передачу.

В листинге 7 показан пример схемы для создания датасета в системе хранения данных через некоторый репозиторий.

Листинг 7: Схема создания датасета для репозитория.

```
from pydantic import BaseModel, ConfigDict

class DatasetCreateSchema(BaseModel):
    file_id: UUID
    separator: str
    header: list[int]
    is_built_in: bool = False

    model_config = ConfigDict(from_attributes=True)
```

3.6. Unit of work

В модуле uow реализован паттерн Unit of work, представленный в виде класса, конструктор которого принимает фабрику контекстов какого-либо хранилища данных (это интерфейс, описывающий возвращаемую реализацию интерфейсов DataStorageContext). Данный класс реализует методы, необходимые для использования UoW в качестве контекстного менеджера. При входе в контекстный менеджер UoW возвращает новый контекст, а при выходе из него происходит сначала фиксация всех изменений, сделанных внутри контекстного менеджера, а затем закрытие контекста. Если в теле контекстного менеджера произошла ошибка, все изменения в контексте будут откатаны, а ошибка будет передана на уровень сценария использования, где она будет об-

работана. В листинге 8 представлена реализация UoW.

Листинг 8: Абстрактный Unit of Work.

```
class DataStorageContextMaker(Protocol):
    def __call__(self) -> DataStorageContext: ...


class UnitOfWork:
    def __init__(self, context_maker: DataStorageContextMaker):
        self._context_maker: DataStorageContextMaker = context_maker
        self._context: DataStorageContext | None = None

    def __enter__(self) -> DataStorageContext:
        self._context = self._context_maker()
        return self._context

    def __exit__(self, exc_type, exc_val, exc_tb) -> None:
        if self._context is not None:
            if exc_type:
                self._context.rollback()
            else:
                self._context.commit()
            self._context.close()
            self._context = None
```

Перед выполнением сценариев использования в них передаётся конкретный экземпляр Unit of work. С помощью контекстного менеджера создаётся новый экземпляр контекста, который передаётся в репозитории для выполнения операций с данными. После завершения операций и выхода за пределы контекста система либо фиксирует изменения и закрывает контекст, если транзакция прошла успешно, либо выполняет откат всех изменений и освобождает ресурсы, сопровождая это выбросом исключения в случае ошибки. Пример использования паттерна Unit of Work внутри сценария представлен в листинге 9 (все импорты, инициализации классов и переменных опущены).

Листинг 9: Пример использования реализации Unit of work.

```
class SaveFile:
    def __init__(self, unit_of_work: AbstractUnitOfWork, file_repo: FileRepo,
        file_metadata_repo: FileMetadataRepo):
        self.unit_of_work = unit_of_work
        self.file_repo = file_repo
```

```
        self.file_metadata_repo = file_metadata_repo

    async def __call__(self, *, upload_file: File):

        create_file_schema = ...
        file_metadata_create_schema = ...

        with self.unit_of_work as context:
            self.file_metadata_repo.create(file_metadata_create_schema, context)
            await self.file_repo.create(upload_file, create_file_schema, context)
```

Насколько корректно будет работать UoW со всеми репозиториями зависит от реализации контекста. Подробнее про это в следующих разделах.

3.7. Сценарии использования

Как было описано раннее, сценарии использования необходимы для того, чтобы описывать и управлять конкретными действиями и процессами в системе, которые связаны с бизнес-логикой. В рамках архитектуры они служат связующим звеном между пользовательскими запросами и доменными сущностями, реализуя требования и правила, заложенные в бизнес-процессах.

Для сценариев использования создан отдельный модуль, разделенный на подмодули: сценарии для работы с задачами, файлами и пользователями. Большинство сценариев, как это и должно быть, напрямую зависимы от доменного слоя.

Взаимодействие с “внешним миром” реализовано через интерфейсы репозиториев и “работников”, которые были представленны выше. Эти интерфейсы описываются при помощи от класса “Protocol”, что позволяет классам соответствовать интерфейсу, реализуя необходимые методы и поля, без необходимости прямого наследования. Благодаря этому сценарии могут работать с различными реализациями репозиториев или сервисов, не зная деталей их внутренней реализации, что способствует гибкости и расширяемости системы.

С внешним миром, включая репозитории и “работников”, сценарии

использования обмениваются данными через объекты, реализованные с использованием паттерна DTO. Сами сценарии для работы на вход принимают строго определенный набор параметров и всегда возвращают строго определенный объект.

Примеры сценариев использования — загрузка и чтение файла, проверка файла на корректность, установка задачи, выполнение задачи и прочие.

3.8. Инфраструктурный слой

Инфраструктурный слой в контексте чистой архитектуры представляет собой внешний уровень системы, отвечающий за реализацию технических аспектов взаимодействия приложения с внешними системами и сервисами. Инфраструктурный слой реализует детали, такие как подключение к базам данных, отправка запросов через сеть, управление файловой системой и взаимодействие с другими внешними ресурсами, не включая в себя при этом бизнес-логики.

В предлагаемой реализации инфраструктурный слой представлен несколькими подмодулями.

3.8.1. Системы хранения данных

В качестве основной системы хранения данных была выбрана PostgreSQL. В ней будет храниться вся информация о пользователях, файлах и задачах в соответствии с форматом, который уже был введен на новом сервере.

Работа с PostgreSQL осуществляется через библиотеку SQLAlchemy, предоставляющую ORM для взаимодействия с реляционными базами данных. Миграции базы данных осуществляются при помощи инструмента Alembic. Для описания структуры базы данных используется декларативный стиль создания моделей. В качестве примера можно привести модель для таблицы датасетов, как показано в листинге 10.

Листинг 10: Модель таблицы датасетов.

```
from sqlalchemy import ForeignKey, Integer, ARRAY
```

```
from sqlalchemy.orm import mapped_column, Mapped, relationship
from internal.infrastructure.data_storage.relational.model import ORMBaseModel
from internal.infrastructure.data_storage.relational.model.file.file_metadata import (
    FileMetadataORM,
)
if typing.TYPE_CHECKING:
    from internal.infrastructure.data_storage.relational.model.task import TaskORM

class DatasetORM(ORMBaseModel):
    __tablename__ = "dataset"
    id: Mapped[UUID] = mapped_column(primary_key=True, default=uuid4)

    is_built_in: Mapped[bool] = mapped_column(default=False)
    header: Mapped[list[int]] = mapped_column(ARRAY(Integer), default=[])
    separator: Mapped[str]
    file_id: Mapped[UUID] = mapped_column(
        ForeignKey("file_metadata.id"), nullable=False
    )
    file_metadata: Mapped[FileMetadataORM] = relationship("FileMetadataORM")

    related_tasks: Mapped[list["TaskORM"]] = relationship(
        "TaskORM", back_populates="dataset"
    )
```

Для дальнейшей реализации репозиториев, работающих с данным хранилищем, требуется контекст управления транзакциями и операциями с базой данных. Однако это уже предусмотрено в самой библиотеке SQLAlchemy. Объект “Session” может служить контекстом для любых реляционных баз данных, включая PostgreSQL, а создание таких контекстов осуществляется с помощью фабрики контекстов — “sessionmaker”. Описание реализации этого контекста и фабрики представлено в листинге 11:

Листинг 11: Контекст для реляционной базы данных.

```
type RelationalContextType = Session

type RelationalContextMakerType = sessionmaker
```

Сохранение файла осуществляется в файловой системе, в директории “/uploads”. Для реализации репозиториев, работающих с данным хранилищей были разработаны FlatContext и FlatContextMaker.

FlatContext предоставляет возможность работы с файлами в рамках “транзакций”, что позволяет обрабатывать создание и сохранение файлов атомарно. Это значит, что все операции по сохранению могут быть отменены в случае возникновения ошибок, что имитирует поведение транзакций в системах хранения данных. Реализация данного контекста представлена в листинге 12 (некоторые импорты и классы опущены).

Листинг 12: Контекст для работы с файловой системой.

```
class FlatContext:
    def __init__(self, upload_directory_path: Path):
        self._upload_directory_path = upload_directory_path
        self._is_closed = True
        self._to_add: list[FlatAddModel] = []
        self._added: list[Path] = []

    @property
    def upload_directory_path(self) -> Path:
        return self._upload_directory_path

    async def async_flush(self) -> None:
        for file_model in self._to_add:
            path_to_file = Path.joinpath(
                self.upload_directory_path, str(file_model.file_name)
            )
            async with aiofiles.open(path_to_file, "wb") as out_file:
                while content := await file_model.file.read(CHUNK_SIZE):
                    await out_file.write(content)
            self._added.append(path_to_file)
            self._to_add.remove(file_model)

    def flush(self) -> None:
        for file_model in self._to_add:
            path_to_file = Path.joinpath(
                self.upload_directory_path, str(file_model.file_name)
            )
            with open(path_to_file, "wb") as out_file:
                while content := file_model.file.read(CHUNK_SIZE):
                    out_file.write(content)  # type: ignore
            self._added.append(path_to_file)
            self._to_add.remove(file_model)

    def rollback(self) -> None:
        for file_path in self._added:
```

```
            if file_path.exists():
                os.remove(file_path)
        self._added.clear()
        self._to_add.clear()

    def commit(self) -> None:
        if self._to_add:
            self.flush()
        self._added.clear()

    def close(self) -> None:
        if self._added:
            self.rollback()
        self._is_closed = True

    def add(self, file_model: FlatAddModel) -> None:
        self._to_add.append(file_model)
```

В листинге 12 представлена искусственная реализация транзакций в файловой системе. Репозиторий добавляет файл в контекст через метод add, передавая в него объект, содержащий файл и его имя. Этот объект сохраняется внутри контекста до тех пор, пока не будет вызван метод flush, который сохраняет файл, но не завершает транзакцию. Таким образом, если после вызова метода flush произойдет откат изменений, все изменения, внесенные методом flush, но не зафиксированные методом commit, будут отменены. Кроме того, существует асинхронная версия метода flush, которую рекомендуется использовать для сохранения файлов там, где это возможно.

Описанные контексты для реляционных баз данных и файлов предоставляют возможность работы в рамках одного Unit of Work с репозиториями, связанными с одним хранилищем (для которых реализован предоставляемый UoW контекст), однако не позволяют работать с репозиториями, связанными с разными хранилищами. Для обеспечения такой возможности был реализован “универсальный” контекст, который управляет транзакциями между всеми контекстами, из которых он состоит. Именно фабрика универсального контекста должна передаваться внутрь Unit of work, для того, чтобы было возможно осуществлять транзакции в рамках одного UoW для репозиториев всех типов. В ли-

стинге 13 приведена реализация универсального контекста(импорты и часть функций опущены).

Листинг 13: Универсальный контекст.

```
class Context:
    def __init__(self, postgres_context: RelationalContextType, flat_context: FlatContext):
        self._postgres_context = postgres_context
        self._flat_context = flat_context

    @property
    def flat_context(self):
        return self._flat_context

    @property
    def postgres_context(self):
        return self._postgres_context

    def commit(self):
        self._postgres_context.commit()
        self._flat_context.commit()

    def rollback(self):
        self._postgres_context.rollback()
        self._flat_context.rollback()

    def close(self):
        self._postgres_context.close()
        self._flat_context.close()

    def flush(self):
        self._postgres_context.flush()
        self._flat_context.flush()

    async def async_flush(self):
        self._postgres_context.flush()  # async calling not supported
        await self._flat_context.async_flush()

    def add(self, model: RelationalAddModel | FlatAddModel):
        if isinstance(model, RelationalAddModel): self._postgres_context.add(model)
        if isinstance(model, FlatAddModel): self._flat_context.add(model)

    def delete(self, model: RelationalDeleteModel | FlatDeleteModel):
        if isinstance(model, RelationalDeleteModel): self._postgres_context.delete(model)
        if isinstance(model, FlatDeleteModel): self._flat_context.delete(model)
```

3.8.2. Репозиторий

Реализации репозиториев располагаются в отдельном модуле, который делится на подмодули relational и flat. В подмодуле relational находятся реализации репозиториев для реляционных баз данных, поддерживаемых SQLAlchemy. Эти реализации являются обобщенными, а желаемая система управления базами данных устанавливается внутри описанного ранее RelationalContext, который передается в репозиторий. Все эти репозитории наследуются от реализованного параметризованного класса CRUD, предоставляющего основные операции (создание, чтение, обновление, удаление) для работы с реляционной базой данных.

Все репозитории из слоя сценариев использования реализованы в данном модуле. В сценариях использования репозитории описывались с помощью протоколов, содержащих только те методы, которые нужны конкретному сценарию; здесь же все репозитории одной сущности объединяются в один общий репозиторий. Например, для репозитория, осуществляющего поиск задачи, и репозитория, выполняющего установку задачи, реализуется единый репозиторий с обоими методами.

Единственный репозиторий, который относится к подмодулю flat, — это репозиторий файлов. Он сохраняет файлы в директорию, указанную в настройках (на момент написания работы это “uploads/”).

3.8.3. Фоновые задачи

Фоновые задачи, как и в предыдущей версии, реализованы с использованием очереди задач Celery. Однако, в отличие от предыдущей реализации, Celery теперь не зависит ни от чего, кроме слоя сценариев использования и сущностей. Функции, обрабатывающие задачи, работают напрямую со сценариями использования, которые создаются при помощи механизма инъекции зависимостей, а затем передаются в эти функци.

В листинге 14 представлена функция, выполняющая задачу профилирования.

Листинг 14: Функция, выполняющая задачу профлирования

```
@worker.task(base=ResourceIntensiveTask, ignore_result=True, max_retries=0)
def profiling_task(
    task_id: UUID,
    dataset_id: UUID,
    config: OneOfTaskConfig,
) -> Any:

    profile_task = get_profile_task_use_case()

    result = profile_task(dataset_id=dataset_id, config=config)
    return result
```

При использовании Celery крайне важно обеспечить использование единственного соединения с базой данных, без пула соединений. Хотя в данной реализации Celery напрямую не взаимодействует с базами данных, она осуществляет это опосредованно, через объекты сценариев использования. Именно по этой причине при создании объекта сценария в него с помощью механизма инъекции зависимостей должен передаваться контекст, обеспечивающий соединение с базой данных без использования пула соединений. Такой подход позволяет избежать проблем с конкурентным доступом к базе данных.

3.8.4. Worker

Сами задачи Celery не вызываются напрямую из сценариев использования или внешних компонентов. Вместо этого, как уже было упомянуто, взаимодействие осуществляется через интерфейсы, что позволяет абстрагироваться от конкретной реализации очереди задач. Модуль worker содержит реализацию этих интерфейсов. В листинге 15 приведена реализация интерфейса “ProfilingTaskWorker” для установки задачи профилирования в очередь.

Листинг 15: Реализация “ProfilingTaskWorker”

```
from internal.dto.worker.task import ProfilingTaskCreateSchema
from internal.infrastructure.background_task.celery.task import profiling_task
```

```
class ProfilingTaskWorker:

    def set(self, task_info: ProfilingTaskCreateSchema) -> None:

        profiling_task.delay(
            task_id=task_info.task_id,
            dataset_id=task_info.dataset_id,
            config=task_info.config,
        )
```

3.9. Rest

Основное приложение, с которым взаимодействует клиент, было решено строить на основе архитектурного стиля REST. Для этого, как и в предыдущей версии, используется фреймворк FastAPI.

Реализация взаимодействия организована следующим образом: для каждого маршрута задается формат данных, которые он принимает и возвращает. В каждый маршрут передаются необходимые сценарии использования, с применением механизма инъекции зависимостей. Основная задача каждого маршрута заключается в преобразовании входящих данных в формат, понятный сценариям использования, выполнении этих сценариев и последующем преобразовании результата в формат, требуемый клиентом.

В листинге 16 представлена функция, обрабатывающая маршрут “api/file/csv”, который используется для загрузки датасета. Эта функция принимает от клиента файл и различные параметры, затем проверяет формат файла, сохраняет файл и его метаданные, а после сохраняет информацию о данных внутри этого файла.

Листинг 16: Обработка маршрута “api/file/csv”

```
from typing import Annotated
from uuid import UUID
from fastapi import Form, UploadFile, Depends, APIRouter
from internal.rest.http.file.di import (
    get_save_file_use_case,
    get_save_dataset_use_case,
    get_check_content_type_use_case,
)
```

```python
from internal.usecase.file import SaveFile, SaveDataset, CheckContentType

router = APIRouter()

@router.post("/csv")
async def upload_csv_dataset(
    file: UploadFile,
    separator: Annotated[str, Form()],  # ?separator=","
    header: Annotated[list[int], Form()],  # ?header=0?header=1?header=2,
    check_content_type: CheckContentType = Depends(get_check_content_type_use_case),
    save_file: SaveFile = Depends(get_save_file_use_case),
    save_dataset: SaveDataset = Depends(get_save_dataset_use_case),
) -> UUID:

    check_content_type(upload_file=file)
    save_file_result = await save_file(upload_file=file)
    save_dataset_result = save_dataset(
        file_id=save_file_result.id,
        separator=separator,
        header=header,
    )
    return save_dataset_result
```

Исключения, которые могут быть выброшены из сценариев использования обрабатываются с помощью встроенных инстурментов FastAPI. Пример представлен в листинге 17.

Листинг 17: Обработка исключений в FastAPI

```python
from fastapi import FastAPI, Request, HTTPException

from internal.usecase.file.exception import IncorrectFileFormatException,
    DatasetNotFoundException

def add_exception_handlers(app: FastAPI):
    @app.exception_handler(IncorrectFileFormatException)
    def incorrect_file_format_exception(request: Request, exc: IncorrectFileFormatException):
        raise HTTPException(status_code=400, detail=str(exc))

    @app.exception_handler(DatasetNotFoundException)
    def dataset_not_found_exception(_, exc: DatasetNotFoundException):
        raise HTTPException(status_code=404, detail=str(exc))
```

3.10. Результаты

В результате была реализована архитектура, полностью соответствующая принципам чистой архитектуры. Схема модулей результата представлена на рисунке 8.

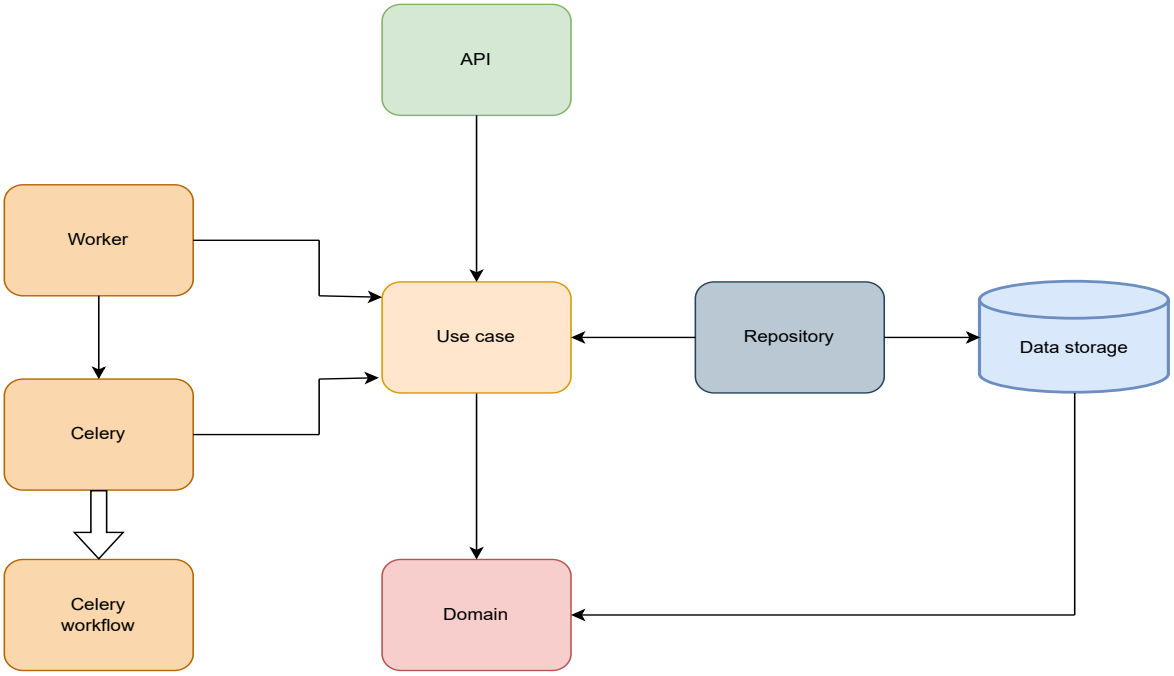

Рис. 8: Схема модулей полученного результата.

Как можно заметить, все зависимиости указаны верно: домен является независимым, сценарии использования зависят только от домена. Репозитории зависят от интерфейсов, описанных в сценариях использования и от конкретных реализаций систем хранения данных. Системы хранения в свою очередь зависят только от домена.

Аналогичная ситуация с левой стороны: worker зависит только от интерфейсов внутри сценариев использования и от конкретной реализации очереди задач для выполнения, которая зависит только от сценариев использования. Большая стрелка, указывающая на рабочие процессы обозначет то, что Celery (конкретная реализация для фоновых задач) следит за выполнением задач профилирования, по окончанию которых с помощью сценариев использования изменит состояния в системе хранения данных.

API зависит только от сценариев использования, экземпляры которых передаются в него вместе с конкретными реализациями репозиториев и рабочих процессов с помощью механизма инъекции зависимостей.

По итогу, полученный результат удовлетворяет первым трем требованиям к новой архитектуре. Пятому требованию, с производительностью, он удовлетворяет тоже, так как сам процесс профилирования и используемые при этом инструменты остались неизменными. Для проверки соответствия четвертому требованию и проврерки работоспособности системы в целом было проведено тестирование системы.

4. Тестирование

Одним из ключевых требований к новой архитектуре приложения являлась высокая тестируемость. Для проверки соответствия этому требованию и обеспечения качества кода были разработаны тесты, охватывающие все ключевые аспекты системы.

4.1. Unit-тестирование

Unit-тесты были разработаны для каждого слоя архитектуры, включая доменные сущности, сценарии использования, конкретные реализации репозиториев, контексты данных, рабочие процессы и Unit of Work. Каждый слой тестировался в изоляции, что позволяло проверять отдельные компоненты без зависимости от других модулей системы. Изоляция достигалась за счет использования заглушек и моков. Общее количество unit-тестов — 74.

4.2. Интеграционное тестирование

Для проверки работы всего приложения в целом были разработаны интеграционные тесты, которые проверяли взаимодействие всех компонентов системы через API. Эти тесты имитировали реальные сценарии использования приложения, обеспечивая проверку корректности работы системы при взаимодействии между различными модулями, включая домен, репозитории, рабочие процессы и пользовательские сценарии. Общее количество интеграционных тестов — 9.

4.3. Тестирование модульности

Были написаны тесты (7 штук), которые проверяют правильность связей между модулями и корректность указания зависимостей. В соответствии с принципами чистой архитектуры все зависимости указывают внутрь системы, что подтверждается написанными тестами.

4.4. Результаты

Тестирование позволило достичь покрытия кода на уровне 93%.

Процесс тестирования системы оказался достаточно простым благодаря соблюдению принципов чистой архитектуры. Все зависимости между модулями были легко изолированы, что позволило без труда применять моки для их замены в тестах. Данный факт подтверждает выполнение требования высокой тестируемости архитектуры.

Заключение

В результате работы были выполнены все поставленные задачи, а именно:

1. Проведен подробный обзор и анализ архитектур предыдущих решений с целью выявления ключевых недостатков;
2. Определены требования к новой архитектуре;
3. Сделан обзор основных подходов к проектированию систем;
4. Спроектирована и реализована новая архитектура;
5. Проведено тестирование компонентов системы как в отдельности, так и в совокупности.

Код реализации доступен в GitHub репозитории[1].

[1] https://github.com/Desbordante/desbordante-server/pull/61

Список литературы

[1] Apache Kafka documentation. — URL: https://kafka.apache.org/documentation/.

[2] Celery documentation. — URL: https://docs.celeryq.dev/en/stable/.

[3] Cockburn Alistair. Ports And Adapters Architecture. — 2006. — URL: http://alistair.cockburn.us/Hexagonal+architecture.

[4] FastAPI documentation. — URL: https://fastapi.tiangolo.com.

[5] GraphQL documentation. — URL: https://graphql.org/learn/.

[6] Martin Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — 1st edition. — USA : Prentice Hall Press, 2017. — ISBN: 0134494164.

[7] Node.js documentation. — URL: https://nodejs.org/docs/latest/api/.

[8] Pautasso Cesare, Wilde Erik. Introduction // REST: From Research to Practice / Ed. by Erik Wilde, Cesare Pautasso. — New York, NY : Springer New York, 2011. — P. 1–18. — ISBN: 978-1-4419-8303-9. — URL: https://doi.org/10.1007/978-1-4419-8303-9_0.

[9] PostgreSQL documentation. — URL: https://www.postgresql.org/docs/.

[10] Ramirez Ariel Ortiz. Three-tier architecture // Linux Journal. — 2000. — Vol. 2000, no. 75es. — P. 7–es.

[11] Reenskaug Trygve. The original MVC reports. — 1979.

[12] Sequalize documentation. — URL: https://sequelize.org.

[13] Рафик Нурмухаметов. Перенос и адаптация моделей базы данных Desbordante на Python с использованием SQLAlchemy. — 2023. —

URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/Database%20refactoring%20-%20Rafik%20Nurmuhametov%20-%202023%20autumn.pdf.