

РЕФЕРАТ

Выпускная квалификационная работа содержит: 61 страницу, 17 рисунков, 4 таблицы, список использованных источников содержит 14 позиций, приводится 2 приложения.

DOCKER, KUBERNETES, МИКРОСЕРВИСЫ, CLOUD COMPUTING
ВЫСОКОНАГРУЖЕННЫЕ СИСТЕМЫ, ВИЗУАЛИЗАЦИЯ ДАННЫХ,
СЕРВИС ДЛЯ УПРАВЛЕНИЯ И МОНИТОРИНГА ПРОЕКТОВ,
POSTGRESQL, REACT, DJANGO.

Выпускная квалификационная работа посвящена разработке программного обеспечения для управления и мониторинга проектов на основе виртуальной интерактивной доски (на примере учебных проектов МАИ) с использованием микросервисной архитектуры, средств контейнеризации и современных шаблонов проектирования.

В теоретической части исследуются выбранные средства и шаблоны проектирования, библиотеки и инструменты для разработки, средства и шаблоны проектирования высоконагруженных веб-приложений. На основе проведённого анализа сделан выбор набора технологий, оптимально подходящих для разработки данного сервиса.

В практической части представленной работы описаны выбранная микросервисная архитектура, способ её реализации на базе Docker контейнеров и системы оркестрации Kubernetes, представлены этапы разработки приложения с приложенными схемами и диаграммами, а также пользовательский интерфейс для пользователей сервиса.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
ОСНОВНАЯ ЧАСТЬ	7
1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ	8
1.1 Постановка задачи	8
1.2 Микросервисная архитектура	9
1.2.1 Микросервисы	9
1.2.2 REST	11
1.3 Технология контейнеризации	14
1.3.1 Docker	18
1.3.2 Kubernetes	19
1.4 Frontend	21
1.4.1 Язык разметки HTML	22
1.4.2 Таблицы стилей CSS	23
1.4.3 Язык программирования JavaScript	24
1.5 Backend	26
1.5.1 Программный интерфейс приложения	28
1.5.2 Схема MVC	28
1.5.3 Технология ORM	31
1.5.4 Web-сервер	32
1.6 Базы данных	34
1.6.1 Основные понятия	34
1.6.2 Модели данных	36
2 ПРАКТИЧЕСКАЯ ЧАСТЬ	39
2.1 Проектирование архитектуры	39
2.1.1 Use-Case диаграмма	40
2.1.2 Архитектура приложения	41
2.1.3 Схема базы данных	44
2.2 Разработка практических решений	47

2.2.1 Frontend	48
2.2.2 Backend	50
2.2.3 Размещение на вычислительной машине	53
ЗАКЛЮЧЕНИЕ	55
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	56
ПРИЛОЖЕНИЯ	57
ПРИЛОЖЕНИЕ А	58
ПРИЛОЖЕНИЕ Б	61

ВВЕДЕНИЕ

Выпускная квалификационная работа посвящена разработке сервиса для управления и мониторинга проектов на основе виртуальной интерактивной доски. Объектом исследования является выбор архитектурного комплекса средств для поддержания прозрачности и простоты использования, легкости поддержки и модификации, а также с использованием современных шаблонов проектирования.

В настоящее время сложно представить свою жизнь без использования всемирной сети Интернет, которая проникла в повседневную жизнь каждого человека, постоянно увеличивая долю проведённого в ней времени. Такое повышенное внимание к этой технологии не остаётся без внимания огромного количества крупных IT компаний и отдельных разработчиков, внедряющих новые технологии и продукты для удовлетворения информационных потребностей пользователей, тем самым делая их жизнь комфортнее.

Таким образом программные продукты успешно внедрены в такие сферы, как образование, государственные услуги, медицина, военная промышленность и многие другие. Однако многие проблемы информационных технологий ожидают своего решения.

Во многих образовательных учреждениях (школы, ВУЗы и др.) внедрены электронные системы учёта успеваемости и посещаемости обучающихся. Наиболее актуальны сейчас платформы для дистанционного обучения, общения между преподавателем и студентом, что позволяет в условиях ограниченных возможностей человека также продолжать успешно обучаться и получать высшее образование.

Однако зачастую подобные программные продукты является весьма не тривиальными в использовании как для преподавателей, так и для студентов.

Следовательно, все преимущества таких систем, а именно простота использования, скорость и информативность, сводятся к минимуму.

Для того, чтобы избежать таких неудобств разрабатывают новые технологии совместной работы, интерфейсы для взаимодействия пользователей и способы восприятия информации человеком.

Технологии в наше время быстро развиваются и меняются, поэтому перед разработчиками стоит сложная задача – быстро и качественно модифицировать свой продукт и подстраиваться под современные реалии. В связи с такими условиями работы были придуманы различные методы разделения приложения на составляющие (модульность), также способы быстрого развёртывания приложений (контейнеризация). В совокупности эти методы представляют современный и жизнеспособный шаблон разработки и поддержки приложений.

Целью данной работы является разработка облачного сервиса для управления и мониторинга проектов на основе виртуальной интерактивной доски, используя подходы микросервисной архитектуры и контейнеризации. Такой сервис поможет студентам продуктивнее вести свои проекты и выполнять задания, а преподавателям выдавать и контролировать выполнение проектов (лабораторные работы, курсовые проекты и т.д.).

ОСНОВНАЯ ЧАСТЬ

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Постановка задачи

В рамках данной выпускной квалификационной работы предполагается разработать сервис (веб-приложение) для управления и мониторинга проектов на основе виртуальной интерактивной доски (на примере учебных проектов МАИ).

Работа реализована в соответствии концепции микросервисной архитектуры. В рамках такого типа архитектуры чаще всего используются средства контейнеризации отдельных модулей (сервисов) приложения, что делает сервис автономным, устойчивым и легко масштабируемым. Популярным средством реализации технологии контейнеризации является инструмент Docker. Также в дополнение к нему используется система-оркестратор Kubernetes.

Само веб-приложение, как крупная система взаимодействия компонентов, состоит из следующих модулей:

1. API (Application Program Interface) – программный интерфейс приложения. Выполняет функцию связи клиентской стороны пользовательского интерфейса (Frontend) и программно-аппаратной части приложения (Backend), служит способом их взаимодействия.
2. Backend – программно-аппаратная часть приложения. Данная часть отвечает за операции с данными, аутентификацию пользователей и разделение их по ролям. В данном случае используется фреймворк Django, подразумевающие реализацию логики на языке программирования Python.
3. База данных. Этот компонент отвечает за хранение и формирование выборки данных для приложения. Важно, чтобы он обладал следующими качествами: надёжность, быстроедействие, всевозможные дополнительные функциональные улучшения,

встроенные инструменты администрирования и мониторинга состояния. Выбор пал на свободную объектно-реляционную систему управления базами данных PostgreSQL.

4. Frontend – клиентская сторона пользовательского интерфейса. Отвечает за взаимодействие пользователя с программно-аппаратной частью приложения. В моём случае это интерфейс, выполненный в стиле виртуальной интерактивной доски для достижения большей простоты и интуитивности в использовании. Для реализации этой идеи был выбран язык программирования JavaScript с использованием библиотеки React и фреймворком для управления потоком данных MobX.

1.2 Микросервисная архитектура

1.2.1 Микросервисы

В данной выпускной квалификационной работе я придерживался концепции микросервисной архитектуры. Микросервисная архитектура – такой вариант сервис-ориентированной архитектуры программного обеспечения, который направлен на взаимодействие наиболее раздробленных, небольших, слабосвязанных и легко модифицируемых модулей, которые называются микросервисами [1]. Этот метод организации архитектуры получил наибольшее распространение в середине 2010-х годов, поскольку параллельно развивались методологии гибкой разработки и философия DevOps.

Отличие такого подхода от традиционного варианта сервис-ориентированных архитектур заключается в том, что в привычных способах зачастую используются тяжеловесные модули, включающие в себя большое количество программно-аппаратных элементов разной направленности [2]. Подобные системы принято связывать

стандартизированными тяжеловесными протоколами (такими, как SOAP, XML-RPC). Дdiamетрально противоположной философией обладает микросервисный подход. Компоненты делят максимально мелко, чтобы каждый из них выполнял собственную обособленную задачу, а в качестве протоколов стараются использовать максимально экономичные, например, в стиле REST(Representational State Transfer) или gRPC (Google Remote Procedure Calls), используя JSON [3].

У такого подхода есть ряд достоинств и недостатков. Для начала рассмотрим преимущества:

1. Модули организованы на основе одной условно элементарной выполняемой функции, что позволяет удобно логически их разделять.
2. Появляется возможность использовать абсолютно различные наборы технологий для каждого из модулей. Таким образом выполняемая модулем задача может быть выполнена максимально эффективно с использованием подходящего языка программирования, фреймворка, может выполняться под управлением разных операционных систем, а также аппаратных платформ.
3. Модули могут быть легко модифицированы, заменены и выведены из эксплуатации в случае ненадобности, максимально избегая ошибок и других неприятных неожиданностей.
4. Микросервисная архитектура является полностью симметричной, а не иерархически устроенной, имеют место только одноранговые зависимости между сервисами.

Наиболее популярным способом организации работы микросервисной архитектуры являются системы управления контейнеризированными приложениями (Kubernetes, Docker Swarm, OpenShift, Apache Mesos). В этом случае каждый компонент изолируется в отдельный контейнер или группы контейнеров, управляемых средой системой оркестрации, которая в свою

очередь гарантирует поддержание отказоустойчивости и балансировку нагрузки.

Как и в любом другом подходе, нельзя обойтись и без недостатков. Критики такой организации архитектуры излагают следующие суждения:

1. Поскольку все компоненты общаются между собой при помощи какого-либо сетевого протокола и вообще расположены в сети, то могут возникать сетевые задержки, сказывающиеся на производительности всего приложения в целом. К тому же, технологии дополнительной виртуализации накладывают дополнительные уровни сложности для взаимодействия модулей, что также увеличивает сетевые задержки.
2. Все взаимодействующие между собой компоненты должны иметь утверждённый, формализованный формат сообщений, что накладывает дополнительные трудности к отладке и может привести к потенциальным ошибкам.
3. Очень важно правильно распределить нагрузку на компоненты, что далеко не всегда является тривиальной задачей, особенно, когда их большое количество. Также необходимо обеспечение инструментом отказоустойчивости, так как незапланированное выведение из строя одного сервиса может негативно отразиться на работе всех остальных.

1.2.2 REST

Для успешной реализации микросервисной архитектуры чаще всего прибегают к способу взаимодействия компонентов при помощи REST (Representational State Transfer – «передача состояния представления»). REST подразумевает архитектурный стиль распределённого в сети приложения. Стандарт включает набор ограничений, которым нужно следовать в

проектировании системы, чтобы достичь максимальной производительности и упрощения архитектуры [4].

Вызовом какого-то действия компонента может являться обыкновенный HTTP-запрос (чаще всего GET или POST), а необходимая информация находится в его параметрах. Такой запрос называют «REST-запрос». Веб-приложения, следующие ограничениям REST, называют «RESTful».

Для RESTful API не существует официального утверждённого стандарта, что и отличает его от веб-сервисов, использующих SOAP. Это происходит из-за того, что SOAP является протоколом, когда REST является архитектурным стилем. Однако, подавляющее большинство реализаций REST используют стандарты JSON, URL, HTTP.

Всего существует шесть обязательных условий, при выполнении которых распределённая система приобретает право называться RESTful. В совокупности все эти требования помогают системе приобрести такие преимущества как: масштабируемость, простота, производительность, надёжность, переносимость и отслеживаемость.

1. Модель клиент-сервер. В основе данного ограничения лежит принцип разграничения потребностей. В данном случае мы разделяем потребности клиентской стороны пользовательского интерфейса от потребностей программно-аппаратной реализации, хранящей данные. Это позволяет упростить серверную часть и улучшить её масштабируемость, а также повысить переносимость кода интерфейса на почти платформы. Также это позволяет двум частям разрабатываться и улучшаться независимо друг от друга.
2. Отсутствие состояния. Данное требование накладывает запрет на хранение информации о состоянии клиента на сервере (stateless protocol – «протокол без сохранения состояния»). Запросы от клиента должны быть сформированы таким образом, чтобы сервер получал всё необходимую о клиенте информацию при обработке получаемого

запроса. Таким образом состояние текущей сессии хранится на стороне клиента. Данные состояния сессии могут быть делегированы серверной частью другим сервисам (например, базе данных, которая поддерживает устойчивое состояние). Когда клиент готов, инициируется отправка запроса с переходом в новое состояние.

3. Кэширование. Могут возникать ситуации, когда большое количество клиентов запрашивают одну и ту же информацию, в таком случае ресурсы серверной части используются повторно для повторения одинаковых действий. Чтобы предотвратить подобное, используют кэширование ответов сервера, при этом важно помечать ответы, как кэшированные или нет, чтобы не передавать пользователю устаревшую информацию. При соблюдении этого условия повышается производительность и масштабируемость системы.
4. Единообразие интерфейса. Ключевым требованием к RESTful системам является унификация интерфейса, которая позволяет отдельным компонентам развиваться отдельно. Существует четыре условия для соответствия унифицированному интерфейсу:

- 1) Идентификация ресурсов. Они идентифицируются в запросах при помощи URI, например. Представления, которые отправляются клиентам, концептуально отделены от ресурсов. Способ хранения информации в базе данных может отличаться от формата отправляемых клиенту данных, например, в виде JSON.
- 2) Манипуляция ресурсами через представление. Для удаления или модификации ресурса клиенту достаточно обладать информацией о представлении этого ресурса, в том числе и метаданные.

- 3) «Самоописываемые» сообщения. Каждый запрос обладает достаточным количеством информации для определения сервером способа его обработки.
- 4) Гипермедиа как средство изменения состояния приложения. Пользователи могут изменять состояние сервиса только с помощью тех действий, которые динамически определены в гипермедиа на сервере. Пользователь не может предположить доступны какие-то действия на каком-либо ресурсом, если не получал об этом информацию ранее, исключая элементарные точки входа в сервис.
5. Слои. От клиента скрыта информация о промежуточных слоях распределённой системы, таких как: сервера балансировки нагрузки и кэширования. Они повышают масштабируемость системы.
6. Код по требованию. Данное условие не является обязательным, однако REST может повысить производительность и быстродействие на стороне клиента за счёт отправки кода с сервера (например, в виде сценариев или апплетов).

Если все вышеописанные условия соблюдаются, тогда распределённая система получит следующие преимущества:

- Надёжность;
- Масштабируемость;
- Производительность;
- Простота интерфейсов;
- Лёгкость внесения изменений;

1.3 Технология контейнеризации

Виртуализация – это набор или логическое объединение вычислительных ресурсов, отделённое от аппаратной реализации. Она

обеспечивает изоляцию вычислительных процессов, выполняемых на одном физическом устройстве, друг от друга.

Частой практикой является использование нескольких операционных систем, развёрнутых на одном физическом устройстве и обладающих собственным набором логических ресурсов таких, как: оперативная память, пространство для хранения данных и эксплуатация процессора. Эти ресурсы предоставляются из общедоступного пространства, которое является доступным на уровне устройства. Управляет ими операционная система, находящаяся на хосте – гипервизор. Такая практика является явным примером использования виртуализации.

Контейнеризация – виртуализация на уровне операционной системы или также контейнерная виртуализация. Это такой метод виртуализации, при котором несколько изолированных экземпляров рабочего пространства пользователя поддерживаются ядром операционной системы вместо одного. Как правило, такие экземпляры называются контейнерами или реже зонами. Они абсолютно идентичны отдельно взятому экземпляру операционной системы. В подавляющем большинстве Unix-подобных операционных систем существует встроенный механизм, который называется chroot. Его улучшенная реализация как раз похожа на технологию контейнеризации. Процессы отдельных контейнеров не могут воздействовать друг на друга, так как ядро операционной системы поддерживает их абсолютную изолированность.

В контейнере предоставляется возможным запустить экземпляр операционной системы только с таким же ядром, как и операционной системы, установленной на хосте в отличие от аппаратной виртуализации, которая эмулирует аппаратные компоненты и позволяет запускать множество других операционных систем. Огромным плюсом технологии контейнеризации является тот факт, что лишние вычислительные ресурсы не

расходуются на поддержание виртуального оборудования и полноценной операционной системы.

Существуют различные варианты реализации подобной технологии. Начиная с тех, которые создают практически полноценные экземпляры операционных систем (например, Solaris Containers, OpenVZ, Virtuozzo), и заканчивая теми, цель которых заключается в изоляции отдельно взятых сервисов с минимально возможным операционным окружением (например, Docker, jail).

Сравнение таких реализаций приведено в таблице 1.1.

Таблица 1.1 Сравнение реализаций Docker

Механизм	ОС	Изоляция файловой системы	Квоты на пространство хранения	Лимиты на ввод-вывод	Лимиты на память	Квоты ЦПУ	Изоляция Сети	Живая миграция
chroot	Unix	Частично	-	-	-	-	-	-
Docker	Linux, FreeBSD, Windows, MacOS	+	+	+	+	+	+	-
Solaris Containers	Solaris, Open Solaris	+	+	-	+	+	+	-
OpenVZ	Linux	+	+	+	+	+	+	+
Virtuozzo	Linux, Windows	+	+	+	+	+	+	+
FreeBSD jail	FreeBSD	+	+	-	+	Частично	+	-
sysjail	AIX	+	+	+	+	+	+	+

О том, что технология контейнеризация очень популярно говорит статистика – примерно 25% компаний лидеров IT индустрии используют её в своих «боевых» продуктах. К тому же, ещё столько планируют внедрить это решение в свои рабочие процессы.

Естественно это происходит не просто так, а из-за большого количества преимуществ, которое предоставляет данная технология, а именно:

- Легковесность;
- Быстродействие;
- Возможность работать на высоком уровне абстракции;

Подобные факторы позволяют трудозатраты по разработке и эксплуатации приложений, что положительно сказывается и на привлекательности для бизнеса.

Одним из главных преимуществ является возможность упаковать приложение вместе с его зависимостями, что решает проблему запуска

программ в разных окружениях. К примеру, различие в версии установленных библиотек может привести к тому, что на одной машине приложение функционирует, а на другой нет.

Для успешной работы контейнеров не достаточно их собрать и запустить. Компания RedHat достаточно доступно и подробно изложила 7 ключевых принципов контейнеризированных приложений:

1. 1 контейнер – один сервис. Контейнер должен выполнять одну максимально элементарную функцию. Это позволит добиться большей переиспользуемости и масштабируемости.
2. Неизменность образа. Все изменения содержимого контейнера должно проводиться до сборки окончательного образа, это поможет подстраховаться от потери данных при удалении контейнера.
3. Утилизируемость контейнеров. Данная концепция подразумевает, что любой контейнер может быть уничтожен в любой момент времени и заменён на другой без остановки работы всего приложения. Получается, что нужно быть готовым к ротации контейнеров при сбое состояния какого-то единичного.
4. Отчётность. Следует логировать состояния и действия приложения, а также иметь возможность проверки жизнеспособности отдельных компонентов.
5. Управляемость. Контейнер должен контролироваться внешним процессом, чтобы исключить возможность потери пользовательских данных и некорректного завершения работы.
6. Самодостаточность. Сам контейнер должен содержать в себе все необходимые данные для самостоятельной работы: код приложения и библиотеки, зависимости, конфигурационные файлы. Другие сервисы не должны находиться внутри, так как это бы противоречило правилу описанному выше – 1 контейнер – сервис.

7. Ограничение ресурсов. Необходимо контролировать потребление ресурсов (загрузка CPU и RAM) контейнерами. Это позволит избежать их избыточного использования.

Как было сказано ранее, на рынке существует множество реализующих технологию контейнеризации решений, однако 80% контейнеров выполняется в среде Docker, а для оркестрации более 50% используют Kubernetes от IT гиганта Google.

1.3.1 Docker

Из-за своей популярности в текущих реалиях IT open-source продукт Docker стал практически синонимом слова «контейнер».

В самом продукте Docker можно выделить следующий набор сущностей:

- **Dockerfile.** Это текстовый конфигурационный файл, который используется для описания и создания образа будущего контейнера. Внутри себя Dockerfile содержит ссылку на родительский (базовый) образ, который является фундаментом нового образа, и набор различных инструкций, таких как установка необходимых зависимостей приложения, объявление переменных окружения, сборка приложения и копирование других конфигурационных файлов. Также он содержит команду, выполняемую при запуске контейнера, служащую его входной точкой.
- **Image.** По сути это готовая к использованию, сформированная по инструкциям в конфигурационном Dockerfile файле, файловая система. Она в свою очередь служит прообразом для запускаемых контейнеров.
- **Instance.** Это как раз и есть контейнер, работающий экземпляр образа (image). Он является минимальной единицей развёртывания в Docker.

- Volume. Том является подключаемой к контейнеру файловой системой. Он существует независимо от самого контейнера, а его существование, вообще говоря, не является обязательным. При помощи томов представляется возможным сохранять на локальном хосте данные, находящиеся внутри контейнера после его уничтожения.
- Registry. Это удалённый репозиторий, в котором хранятся Docker Images (образы). Репозиторий может быть доступен как и всем, так и ограниченному числу лиц, имеющим соответствующие права доступа.

Процесс разработки и использования выглядит следующим образом. Сначала разработчик формирует Dockerfile. Там он прописывает, базовый образ, зависимости, монтирует местоположения хранения файлов, сетевые порты, запуск каких-то команд или скриптов. После этого из данного файла-описания формируется Docker образ (image), который является неизменным и может быть далее перемещён на другие вычислительные машины. Данный образ загружается соответствующим заранее установленным программным обеспечением (Docker), а затем на его основе запускают сам контейнер. На другие вычислительные машины не накладывается никаких специфических ограничений, и никаких дополнительных действий по установке зависимостей не требуется, так как требуемые уже находятся внутри контейнера [5].

1.3.2 Kubernetes

Kubernetes (K8s) – также open-source (с открытым программным кодом) продукт, предназначенный для автоматизации развёртывания, масштабирования и управления связанными сущностями (контейнерами).

Важно, что одну логическую единицу может представлять не только один контейнер, но и группа контейнеров, выполняющих одну функцию –

один сервис. Таким образом поддерживается масштабируемость и отказоустойчивость приложения. Например, порождение нескольких балансировщиков нагрузки поможет правильно распределить нагрузку и на них, и поддержит систему в рабочем состоянии даже в случае отказа работы одного экземпляра. К тому же, Kubernetes может автоматически запускать и останавливать контейнеры по мере надобности или в случае неожиданных отказов других [6].

В действительно это совокупность сервисов, являющихся распределённым вычислительным кластером, выполняющим функции оркестрации. Kubernetes не является альтернативой Docker, наоборот, он сильно расширяет спектр его возможностей, предоставляя удобные механизмы по развёртыванию, сетевой маршрутизации, использованием ресурсов, распределением нагрузки и поддержанием отказоустойчивости приложения.

В самом продукте Kubernetes можно выделить следующий набор сущностей:

- Node (master, slave). Это узлы, которые в совокупности формируют вычислительный кластер Kubernetes. Master-узел выполняет контроль над всем кластером при помощи планировщика и менеджера контроллеров, является интерфейсом взаимодействия с пользователями, а также является хранилищем etcd, где располагаются метаданные, конфигурационные настройки кластера и статусы его сущностей. Slave-узел используется исключительно для запуска контейнеров и имеет два сервиса: агент планировщика и сетевой маршрутизатор.
- Namespace. Это структурный объект, который позволяет разграничивать ресурсы кластера по ролям (между пользователями и командами).

- Pod. Представляет минимальную единицу развёртывания в кластере Kubernetes – группу из одного или нескольких контейнеров, собранных вместе логически для совместного развёртывания на узле. Собрать в группы контейнеры разного назначения имеет смысл только в том случае, когда они сильно связаны между собой и должны быть на одном узле. Также это позволяет улучшить время отклика при их совместной работе.
- ReplicaSet. Объект, который содержит описание и контролирует количество экземпляров Pod. Наличие нескольких экземпляров Pod позволяет повысить отказоустойчивость и масштабирование приложения. Общепринятой практикой является создание ReplicaSet при помощи Deployment.
- Deployment. Объект, декларативно описывающий экземпляры Pod, их количество и, в случае обновления параметров, стратегию замены реплик Pod.
- StatefulSet. По сути является ReplicaSet, но с возможностью описывать сетевые адреса Pod и их дисковое пространство.
- DaemonSet. Объект, отвечающий за то, что на каждом узле (или ряда выбранных) будет выполняться по экземпляру указанного Pod.
- Job, CronJob. Объекты, запускающие Pod однократно и регулярно по расписанию соответственно. Также они отслеживают результат завершения его работы.
- Label, Selector. Текстовые метки, позволяющие описывать ресурсы для упрощения групповых действий с ними.
- Service. Инструмент, выполняющий развёртывание приложения как сетевого сервиса. Также он выполняет распределение нагрузки между разными Pod.

После представленного выше описания становится ещё более понятна разница между продуктами Docker и Kubernetes. Справедливо утверждать,

что Docker управляет экземплярами контейнеров, а Kubernetes управляет самим Docker.

1.4 Frontend

Frontend или, иначе говоря, клиентская часть, пользовательский интерфейс является одним из важнейших компонентов успешного приложения, сервиса и вообще любого продукта. Именно эта часть связывает конечного пользователя и внутреннюю логику приложения. Если интерфейс будет слишком сложен, засорён лишним наполнением или недостаточно быстр, то обыкновенный пользователь не будет им пользоваться.

Сама клиентская часть является довольно сложным, объёмным механизмом, использующим внушительное количество технологий, которые делают путь становления Web-разработчика долгим и полным разнообразных инструментов. Сама сфера быстро развивается и меняется, что добавляет дополнительной сложности в поддержании успешных проектов. Можно перечислить список основных инструментов при Web-разработке: HTML, CSS, JavaScript с бесчисленным количеством библиотек, фреймворков и шаблонов проектирования (React, JQuery, AJAX, SPA, Webpack). Предлагаю остановиться на некоторых более подробно.

1.4.1 Язык разметки HTML

HTML (HyperText Markup Language – «язык гипертекстовой разметки») – язык описания разметки документов. После загрузки документа с описанием браузер клиента преобразует его в визуальный формат, наблюдаемый на экранах цифровых устройств.

Также существует более строгий формат языка HTML – XHTML. Его появление обосновано тем, что в первом могут быть перекрытия тегов (описываемых сущностей страницы), которые не интерпретируются браузером как ошибки, однако могут принести разработчику много

неудобств. Для облегчения отладки и формирования HTML описания и представили XHTML. «X» в названии является первой буквой формата XML, к строгости описания которого и стремились разработчики нового стандарта.

Как было сказано выше, HTML – теговый язык описания документов. Такие документы состоят из элементов, которые в начале и конце помечаются специальными метками – тегами. Эти элементы могут быть и пустыми, однако в будущем наполнены. Они могут обладать собственными уникальными или общими свойствами для определения внешнего вида и наполнения [7].

В конце прошлого века производителями наиболее популярных браузеров начали внедряться собственные модификации тегов HTML, что привело к неоднозначности при интерпретации одной и той же страницы в разных браузерах. Разработчикам на JavaScript приходилось, да и сейчас приходится, прибегать к различного рода хитростям, чтобы получить на выходе приложение, поддерживаемое большим количеством браузеров.

Однако сейчас ситуация в этом направлении стала менее напряжённой в последнее время, так как производители браузеров стараются придерживаться современного стандарта W3C, который позволяет упростить разработку кросс-браузерных приложений.

1.4.2 Таблицы стилей CSS

CSS (Cascading Style Sheets – «каскадные таблицы стилей») – язык описания визуальной составляющей документа (страницы), написанного, в основном, при помощи языка разметки HTML.

Каскадные таблицы стилей используются Web-разработчиками для описания внешнего вида отдельных элементов, задания шрифтов, определения цветов и других тонкостей визуализации информации на странице. Основная идея такого подхода заключается в том, чтобы разделить логическую структуру Web-страницы от описания её внешнего вида. Это

позволяет большую гибкость в разработке и модификации этих частей отдельно, а также уменьшает сложность и повторяемость кода.

Помимо этого CSS позволяет представлять один документ в разных способах подачи информации: на экране, в печатном виде, чтение голосом или даже шрифтом Брайля при наличии соответствующих дополнительных устройств.

Стили CSS могут подключать к документу несколькими способами. Можно добавлять формальное описание CSS прямо в HTML документ, а можно хранить отдельный файл или несколько файлов с описанием и в основном HTML документе прописывать путь к ним.

Использование CSS построено на двух основных принципах – наследование и каскадирование.

Принцип наследование заключается в наследуемости элементами-потомками свойств и атрибутов от элементов-предков.

Принцип каскадирования заключается в том, что одному CSS элементу может соответствовать несколько описанных CSS правил (стилей). В таком случае происходит конфликт этих правил, который разрешается следующими правилами приоритета (расположены в порядке возрастания приоритета):

- Стандартные стили используемого браузера;
- Стили, заданные пользователем браузера в его настройках;
- Стили, заданные автором документа (HTML страницы):
 - Стили, наследуемые от предков;
 - Стили, подключаемые из внешних CSS файлов;
 - Стили, заданные в контейнере стилей непосредственно внутри документа;
- Стили, помеченные сопроводительным правилом !important;

1.4.3 Язык программирования JavaScript

JavaScript – язык программирования, поддерживающий использование множества парадигм программирования. Поддерживаемыми стилями являются: функциональный, объектно-ориентированный и императивный. Чаще всего используется для придания интерактивности и «оживлённости» страницам Web-приложений.

Как было сказано выше, JavaScript является объектно-ориентированным языком программирования, однако отличием от традиционного представления ООП является используемое в нём прототипирование, что подразумевает отсутствие как такового понятия класса, а наследование происходит путём копирования объекта-родителя. Помимо этого JavaScript обладает свойствами, традиционными для функциональных языков программирования: функции являются объектами, объекты представлены списками, каррирование (представление функции многих аргументов, как множества функций одного аргумента), существование анонимных функций, наличие замыканий. Все эти свойства добавляют ещё больше гибкости этому языку.

Код языка очень легко встраивается в HTML документы. Достаточно включить в контейнер `script` код на вышеупомянутом языке или указать ссылку на внешний документ с кодом в атрибутах контейнера.

Синтаксис JavaScript во многом схож с синтаксисом языка Си, однако имеет очень важные отличия от него:

- Поддержка объектов с возможностью интроспекции (получения типа и структуры объекта во время выполнения программы);
- Функции представлены объектами;
- Поддержка автоматической очистки неиспользуемых ячеек памяти;
- Автоматическое приведение типов;
- Поддержка анонимных функций;

Большую популярность язык получил вместе с развитием технологии AJAX, позволяющей выполнять «фоновый» асинхронный обмен данными клиентского приложения с Web-сервером. В результате такого обмена не требуется перезагружать текущую браузерную страницу, что упрощает и ускоряет процесс взаимодействия со страницей и улучшает пользовательский опыт, делая Web-приложение более привлекательным для пользователей.

Также существуют, так называемые, пользовательские скрипты, позволяющие выполнять автоматическое заполнение выдаваемых пользователю форм, изменять форматирование страницы, скрывать нежелательное содержимое, добавлять дополнительные элементы управления на страницу и многое другое. Они написаны на JavaScript и выполняются в браузере при загрузке страницы.

Язык широко знаменит из-за большого количества существующих для него библиотек, которые позволяют автоматизировать и упрощать большое количество действий. Например, библиотека JQuery предоставляет возможность AJAX запросов.

Очень популярной практикой сейчас является разработка SPA (одностраничных) Web-приложений. В основе такого подхода лежит использование лишь единственного HTML документа, как оболочку для всех остальных страниц. Работа с таким приложением реализуется при помощи динамически подгружаемых HTML, CSS и JavaScript объектов, как правило с использованием технологии AJAX, описанной выше.

Основными сущностями в таких Web-приложениях являются:

- JavaScript фреймворк, реализующий приложение и его интерактивное окружение;
- Routing (маршрутизация) – навигация между представлениями (view) в клиентской части;
- HTML шаблоны, служащие основой для динамически построенных страниц;

- API для взаимодействия с серверной частью (как правило REST);
- Технология AJAX

Самыми популярными JavaScript фреймворками или библиотеками для реализации SPA приложений являются React.js, Angular и Vue.js.

1.5 Backend

Backend или, иначе говоря, программно-аппаратная часть сервиса также является одним из важнейших компонентов успешного приложения, сервиса и вообще любого продукта. Именно эта часть связывает выполняет внутреннюю логику приложения, реализующую запросы и требования клиента. Если программно-аппаратная часть приложения будет слишком сложной, имеющей сложно распознаваемую кодовую базу или будет недостаточно оптимизированной, то обыкновенному пользователю придётся тратить лишнее время на ожидания ответа от сервера, что побудит его отказаться от использования приложения или просто испортит опыт эксплуатации. Эти факторы не позволят сервису успешно развиваться и быть востребованным.

Сама внутренняя логика приложения является довольно сложным, тонким механизмом, требующим от разработчика построения правильного потока данных, циркулирующих внутри сервиса [8]. Данные приходят, обрабатываются, запрашиваются из базы данных, в отдельных случаях формируется страница клиента прямо во внутренней логике сервиса. Для оптимальной манипуляции этими потоками существуют и регулярно используются специальные инструменты, фреймворки, моделируются шаблоны поведения. Всё вышеперечисленное требует от Backend-разработчика предельной внимательности и осознанности во всех решениях. Сфера развивается и меняется, появляются новые шаблоны поведения, оттачиваются старые, проверенные, «боевые» механизмы. Можно

перечислить список основных инструментов и шаблонов при Backend-разработке:

- API (Application Program Interface – «программный интерфейс приложения») – совокупность способов взаимодействия клиентской части приложения и внутренней логики;
- Часто Backend реализуется на следующих языках программирования: Python, Java, C#, Go, JavaScript, PHP, Ruby и другие;
- Требуются фреймворки для автоматического тестирования: PyTest, PyUnit, XUnit, TestNG, Jasmine и многие другие;
- Средства доступа к базам данных (ORM – Object-Relational Mapping), об этом будет рассказано подробнее;
- Средства администрирования баз данных, например, pgAdmin для PostgreSQL;
- Средства аутентификации пользователей, например, стандартные средства Python-фреймворка Django;
- Шаблоны манипуляции данными и фреймворки их реализующие, например, Django;
- Настройка Web-сервера;

Предлагаю остановиться на некоторых более подробно.

1.5.1 Программный интерфейс приложения

Как было сказано выше, API это совокупность методов и способов, с помощью которых общаются две программные единицы. В данном случае это интерфейс приложения (сторона клиента) и программно-аппаратная часть (серверная сторона).

Если попытаться простым способом описать, выполняемую работу, то можно получить схему, представленную на рисунке 1.1.

API участвует на двух этапах: при получении запросы от клиентской части и при отправке ей ответа. Как правило, он представлен как множество

пар URL запросов и контроллеров, отвечающих за их обработку. Поступающий запрос разбирается, определяется соответствующий ему контроллер, которому затем передаются данные. Контроллер выполняет запрограммированные манипуляции с данными, возможно, обращаясь при этом к базе данных, и отправляет результат своих действий в клиентскую часть приложения. API можно считать ещё одним уровнем абстракции над традиционными протоколами передачи данных во всемирной сети (в нашем случае HTTP/HTTPS). Разработчики приложения с двух сторон должны заранее договориться о стандартизированных методах обмена информации между частями сервиса.

1.5.2 Схема MVC

Посмотрим, какие задачи должно решать Web-приложение:

- Маршрутизация запроса. Как было сказано выше, по входящему URL нужно понять какую именно функцию вызвать для обработки данного запроса;
- Разбор заголовков и параметров запроса. Например. Разобрать заголовки Cookie и извлечь конкретное значение;
- Хранения состояния (сессии) пользователя. Протокол HTTP не имеет состояний, то есть каждый новый запрос может приходить к разным серверам, поэтому мы не можем связывать пользователя с каким-то сетевым соединением, поэтому требуется хранить какое-то состояние пользователя;
- Выполнение бизнес логики. При выполнении бизнес логики приложение может делать какие-то вычисления, однако по большей части приложения это прослойки для доступа к данным;
- Работа с базами данных;
- Генерация HTML страницы или JSON (или другого формата) ответа;

Как можно заметить, поставленных задач достаточно большое количество, и решать их каждый раз в каждом отдельном приложении очень трудоёмко, поэтому при разработке Web-приложений в последнее время используются какие-либо фреймворки.

MVC (Model-View-Controller – «Модель-Представление-Контроллер») – архитектурный шаблон, который задаёт структуру Web-приложения, представленную на рисунке 1.1. Он определяет из каких частей должно состоять Web-приложение, и как они должны взаимодействовать.

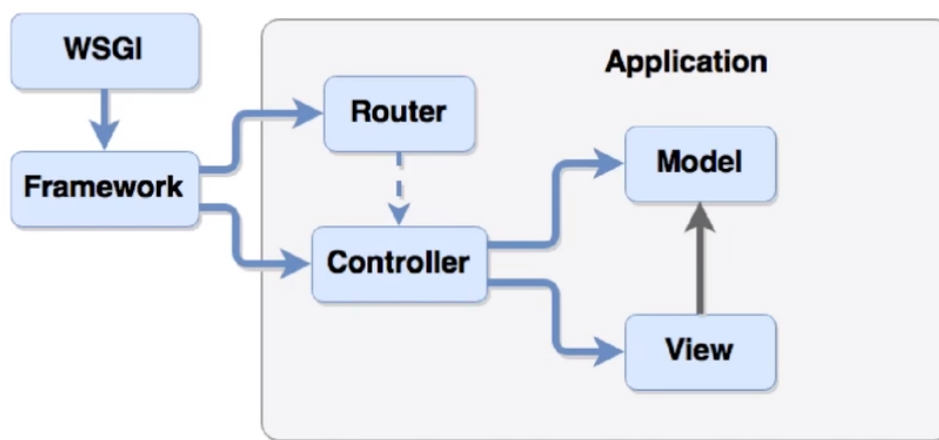


Рисунок 1.1 - Архитектура MVC приложения

Требуется определить какую функцию для какого запроса вызвать, то есть определить точку входа в приложение. Для этого используется Router (маршрутизатор). Это компонент Web-приложения, выполняющий отображение входящего запроса на его функцию-обработчика. Он является скорее декларативным описанием того, какой запрос должен приводить к какой функции.

Далее управление передаётся контроллеру (Controller). В приложении он скорее всего не один, а их существует некоторое множество. Контроллер делает вызовы к модели (Model), она в свою очередь реализует полностью бизнес-логику: обращение к базам данных, взаимодействие с внешними сервисами, какие-то вычисления и другие. Модель возвращает контроллеру готовые для отображения данные. Он в свою очередь передаёт их в

компонент View (представление). Его роль заключается в формировании HTML странички (JSON, XML) из имеющихся данных. В результате полученная страница (данные) возвращается клиенту.

На сегодняшний день существует большое количество MVC фреймворков для различных языков программирования: Django (Python), Yii (PHP), Ruby on rails (Ruby) и другие. У фреймворков есть немало преимуществ:

- Готовая архитектура. Облегчает разработку для программиста, избавляет от потенциальных ошибок, и архитектура проекта получается лучше;
- Повторное использование кода. Использование готовых библиотек и функций для популярных задач;
- Экономия ресурсов. Отчасти является следствием предыдущего пункта;
- Проще найти программистов. Поскольку фреймворки популярны, то количество людей знающих их большое;
- Проще обучать программистов. Популярные фреймворки идут с хорошей документацией;

1.5.3 Технология ORM

ORM (Object-Relational Mapping – «объектно-реляционное отображение») – инструмент, используемый в программирования для связывания между собой сущности в базе данных с сущностями в объектно-ориентированных языках программирования.

Изначальная идея заключалась в том, чтобы избавить разработчика от необходимости писать SQL запросы для взаимодействия с СУБД. Данная технология позволяет использовать виртуальную объектную базу данных в Backend части приложения, а именно: изменять данные в самой базе, получать их из базы, оптимизировать манипуляции, проводимые с

сущностями и их содержимым. Также это помогает избежать повторяющихся рутинных действий для написания таких запросов и уменьшает количество потенциальных ошибок, которые может допустить разработчик.

Определённые реализации ORM имеют возможность кэшировать результаты запросов, и тогда при следующем подобном запросе Backend не будет запрашивать данные у СУБД, а возьмёт их с кэша, что может существенно положительно отразиться на быстродействии сервиса. Некоторые реализации позволяют автоматически синхронизировать данные сущностей в памяти приложения с базой данных, что позволяет ускорить разработку приложения, убирая этот механический пункт из списка задач Backend-программиста. С его стороны он просто работает с традиционными объектами, свойственными ООП языка, а они автоматически сохраняются в базе данных.

Однако существуют и недостатки подобного подхода. Из их числа следует отметить потенциально большее использование памяти, а также невысокая скорость работы в определённых случаях. Правда большинство ORM инструментов на данный момент даёт разработчику возможность самостоятельно утвердить код SQL запросов.

1.5.4 Web-сервер

Web-сервер – сетевой сервер, который занимается обработкой протокола HTTP. В Unix системах Web-сервер является демоном (Daemon) – программой, не связанной с консолью или графическим интерфейсом, которая постоянно запущена, находится в памяти и обрабатывает данные, которые приходят по сети.

При запуске web-сервер читает файл конфигурации, в котором описаны атрибуты и параметры его работы. В случае неправильного конфигурационного файла сервер не запустится. Также он пишет логи –

текстовые файлы, в которых отображается ход его работы (например, сообщения об ошибках).

Web-сервер выделяет два вида процессов, запущенных на уровне операционной системы: master-процесс и worker-процессы. Master-процесс – тот процесс, который запускается при запуске сервера. Он, как правило, один, и в его задачи входит чтение конфигурационного файла, открытие сокетов для логов и запуск дочерних процессов – worker-процессов. После запуска дочерних процессов он переходит в режим мониторинга и занимается поддержанием определённого количества worker-процессов. В задачи этих процессов входит циклическая обработка входящих запросов.

Рассмотрим упрощённый цикл работы worker-процесса:

1. Чтение HTTP-запроса;
2. Определяется virtual host – часть конфигурационного файла, отвечающая за обработку определённого домена;
3. Выбор location. Location (локация, путь) – более мелкая часть конфигурационного файла, отвечающая за обработку конкретной группы URL;
4. Проверка доступа к файлу в ОС и, возможно, авторизации пользователя;
5. Чтение файла;
6. Применение фильтров, установка соответствующих HTTP заголовков;
7. Отправка ответа;

Частой практикой является использование Web-сервера для отдачи статических файлов. Например, пользователь первый раз заходит на сайт, соответственно ему не обязательно задействовать внутреннюю логику приложения. Гораздо удобнее и проще отдать ему сначала статический файл с HTTP страницей.

Это единственный тип запросов, который обслуживается Web-сервером самостоятельно.

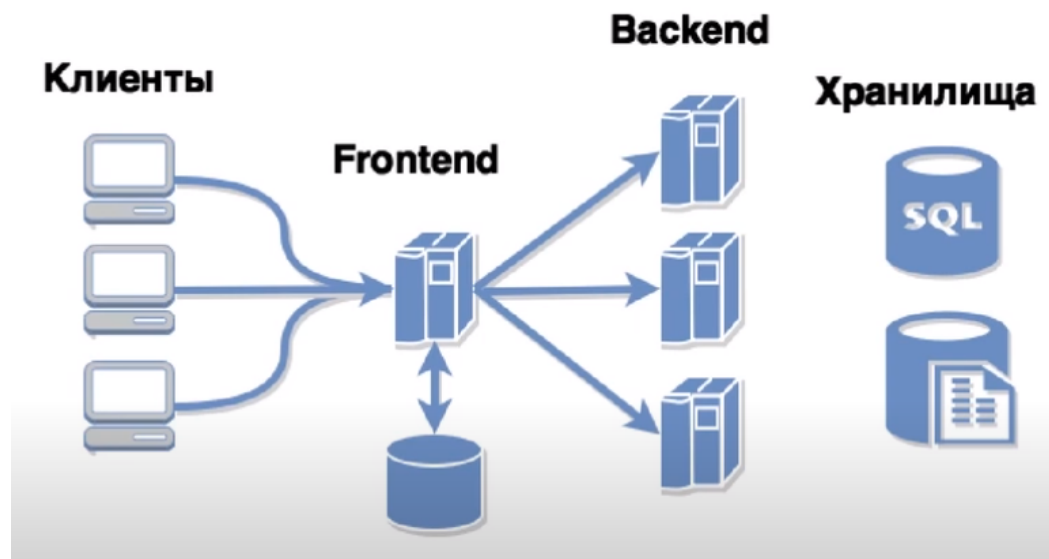


Рисунок 1.2 - Архитектура Frontend-Backend

В контексте схемы, согласно рисунку 1.2, Frontend компонент это наш Web-сервер, а Backend это сервера внутренней логики приложения. Остальные виды запросов (за динамическими страницами, к API сайта, персистентные соединения) сервер делегирует в Backend. Такой процесс называется проксированием.

В задачи Frontend сервера входит:

- Отдача статических документов. То есть все документы, которые не меняются отдаются сервером с диска;
- Проксирование (reverse proxy) – передача запросов на Backend сервера;
- Балансировка нагрузки. Frontend сервер выбирает тот Backend сервер, который в данный момент не нагружен и старается равномерно распределять нагрузку между ними;
- Кэширование. Уменьшается количество обращений к внутренней логике приложения при помощи сохранения результатов запросов;
- Авторизация, сжатие данных;

Из их числа следует отдельно отметить проксирование, так как эта важная часть архитектуры Frontend-Backend. Такие прокси-сервера

позволяют снизить загрузку на Backend-сервера приложения и обрабатывать большее количество пользователей параллельно.

1.6 Базы данных

В 60-х годах прошлого столетия пришло осознание проблемы обработки больших объёмов данных. Сначала были предприняты усилия по созданию и улучшению вычислительных инструментов, а затем информацию нужно было интерпретировать с точки зрения некоторой предметной области. Это побудило к появлению экспертных систем.

1.6.1 Основные понятия

База данных – особая структура организации связанных данных конкретной предметной области, в которой определения и отношений между ними отделены от процедур. Она характеризуется своей методологией, технологией и практикой [9].

Главным отличием баз данных от систем на основе файлов заключается в том, что такие системы подразумевают несколько назначений и несколько представлений о данных, а базы данных – только одно представление о данных. Такой подход обеспечивается специальными средствами доступа к ним.

Система управления базами данных (СУБД) – совокупность специальных программно-аппаратных средств, обеспечивающих доступ к базе данных и управление данными. Следует выделить ряд требований к СУБД:

- Эффективное выполнение функций предметной области;
- Безопасность;
- Минимизация избыточности;
- Предоставление непротиворечивой информации;
- Простота использования;

- Простота модификации;
- Наличие центральной точки управления;

В основе любой базы данных лежат специальные модели, позволяющие смоделировать структуру данных и способы их использования. Модель данных – комплекс методов и средств определения логического представления физических данных какой-либо конкретной предметной области. Она описывается тремя компонентами:

- Правила структурирования данных;
- Совокупность возможных операций, применимых к базе данных;
- Ограничения целостности, которые определяют совокупность возможных состояний БД;

Схема базы данных – описание БД средствами языка определения данных. Схема характеризует свойства базы данных в соответствии с типами данных, хранящихся в ней. Всего различают три типа схем: концептуальная, логическая и физическая.

1. Процесс проектирования данных начинается с определения концептуальных требований, формируется концептуальная модель. Она представляет объекты с их связями не определяя способы физического хранения.
2. Следующим шагом из концептуальной модели формируется логическая в соответствии с выбранной СУБД.
3. В конечном итоге в логической модели определяется расположение данных и методы доступа к ним – физическая модель.

Концептуальная модель оперирует объектами, именуемыми сущностями, которые могут находиться в каком-то отношении друг к другу. Отношение этих сущностей демонстрирует их связь. Моделями «сущность-связь» или по-другому ER-моделями (Entity-Relationship) называются модели, представляющими из себя совокупность сущностей и связей. Они имеют смысловую интерпретацию.

В случае когда одному экземпляру сущности соответствует единственный экземпляр другой – такая связь называется один-к-одному (1:1). Если одному экземпляру соответствует несколько, тогда такая связь называется один-ко-многим (1:M). И наконец если одному экземпляру сущности соответствует множество экземпляров другой и наоборот – такая связь называется многие-ко-многим (M:N).

В основу реляционной модели данных были заложены три основных принципа:

1. Независимость данных на логическом и физическом уровнях;
2. Создание структурно простой модели;
3. Использование языков высокого уровня для описания операций над данными;

Главным мотивом к созданию реляционной модели послужило желание явно разделить логические и физические стороны управления базами данных.

1.6.2 Модели данных

Вообще говоря, реляционная база данных – совокупность таблиц, находящихся в определённом отношении друг другу и содержащих данные об объектах определённого типа. Строка таблицы – данные и атрибуты одного объекта. Столбец – определённая характеристика, имеющаяся у группы объектов. Записи в таблице (строки) имеют одинаковую структуру, все состоят из полей, содержащих атрибуты объекта. Каждая таблица в такой базе должна иметь первичный ключ – поле или совокупность полей (составной ключ), однозначно идентифицирующих каждый объект таблицы.

Таблицы в реляционной базе данных должны отвечать таким требованиям, как: отсутствие дублирования, непротиворечимость данных и стремление к наименьшим трудозатратам на поддержание базы. Такие требования выстраивают нормализацию отношений.

Существует три нормальные формы отношений:

1. Реляционная таблица имеет первую нормальную форму только в том случае, когда ни одна из ее записей (строк) не содержит в любом своем поле более одного значения и ни одно из ее ключевых полей не пусто.
2. Реляционная таблица имеет вторую нормальную форму только в том случае, когда она имеет первую нормальную форму и все её поля, которые не входят в первичный ключ, связаны полной функциональной зависимостью с первичным ключом. Следовательно требуется определить функциональную зависимость полей – зависимость, при которой в объекте определённого значения ключевого реквизита соответствует только одно значение описательного реквизита.
3. Реляционная таблица имеет третью нормальную форму только в том случае, когда она имеет вторую нормальную форму, и ни одно из ее неключевых полей не зависит функционально от любого другого неключевого поля.

Объектно-реляционная база данных – база данных, использующая в атрибутах экземпляров объектов данных методы и типы, свойственные объектно-ориентированному подходу. Также её иногда называют расширенной реляционной, поскольку основная её логика основана на идеях реляционной модели.

Такой подход имеет ряд преимуществ:

- Повторное и совместное использование компонентов. Например, в приложении может понадобиться данные пространственного типа, как точки, линии и многоугольники. В таком случае уже будут существовать функции выполняющие необходимые связанные вычисления.

- В случае грамотного проектирования такой подход позволит воспользоваться достоинствами нового функционала без потери преимуществ уже существующей базы.

Однако, существуют и недостатки:

- Очевидным недостатком является дополнительная сложность проектирования, накладываемая на систему, и, следовательно, дополнительные расходы.
- Теряется простота и ясность присущая реляционной модели.

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Проектирование архитектуры

Мною была поставлена задача реализовать облачный сервис для управления и мониторинга проектов на основе виртуальной интерактивной доски (на примере учебных проектов МАИ). В соответствии с данной задачей можно выделить следующие компоненты и шаги, которые необходимо реализовать, для её успешного выполнения:

1. Описать всевозможные сценарии использования пользователями представленного сервиса для проектирования методов API и схемы БД, а также описать техническое задание, благодаря которому можно формализовать следующие шаги. Сделать удобно при помощи Use-Case диаграммы (диаграммы вариантов использования).
2. Далее стоит смоделировать архитектуру приложения, чтобы наглядно определить компоненты, требующие реализации, и сформировать более конкретное представление о работе приложения согласно концепции микросервисной архитектуры.
3. Следующим шагом необходимо спроектировать базы данных. Она является важным компонентом сервиса, позволяющим пользователю не задумываться о локальном сохранении собственной информации.
4. Следующие два шага можно выполнять параллельно. Это разработка пользовательского интерфейса и разработка программно-аппаратной части Web-приложения.
5. Разработка базы данных будет проходить на этапе реализации внутренней логики приложения, поскольку инструменты разработки позволяют делать это непосредственно из программно-аппаратной части.

Рассмотрим каждый шаг по-отдельности и опишем решения проблем и механизмы их разработки.

2.1.1 Use-Case диаграмма

Как было сказано ранее, Use-Case диаграмма, она же диаграмма вариантов использования, отображает отношения между акторами (группами пользователей приложения) и действиями, которые могут быть произведены в системе [10].

В текущем случае можно выделить три актора:

1. Студент. Имеет возможность смотреть свои задания, создавать собственную карточку, добавлять/удалять задания, отмечать прогресс их выполнения, а также выполнять такие действия со своим аккаунтом как регистрация и смена пароля.
2. Преподаватель. Имеет подобные студенту возможности, а также умеет назначать задания и отмечать статус их проверки. Однако, он не может самостоятельно зарегистрироваться (об этом ниже).
3. Администратор. Помимо того, что администратор имеет доступ ко всем данным сервиса, он регистрирует преподавателей, чтобы исключить случаи подделывания аккаунтов с такой ролью, и добавляет новые доски (рабочие окружения для нового семестра).

Диаграмма, представленная на рисунке 2.1, удовлетворяет стандартам проектирования Use-Case диаграмм, а именно:

- Квадратом отмечено пространство сервиса, он подписан сверху;
- Синими овалами отмечены возможные действия пользователей;
- Действия, которые являются обязательными перед выполнением других, обозначены связью с подписью ««include»».
- Акторы находятся вне квадрата с сервисом;

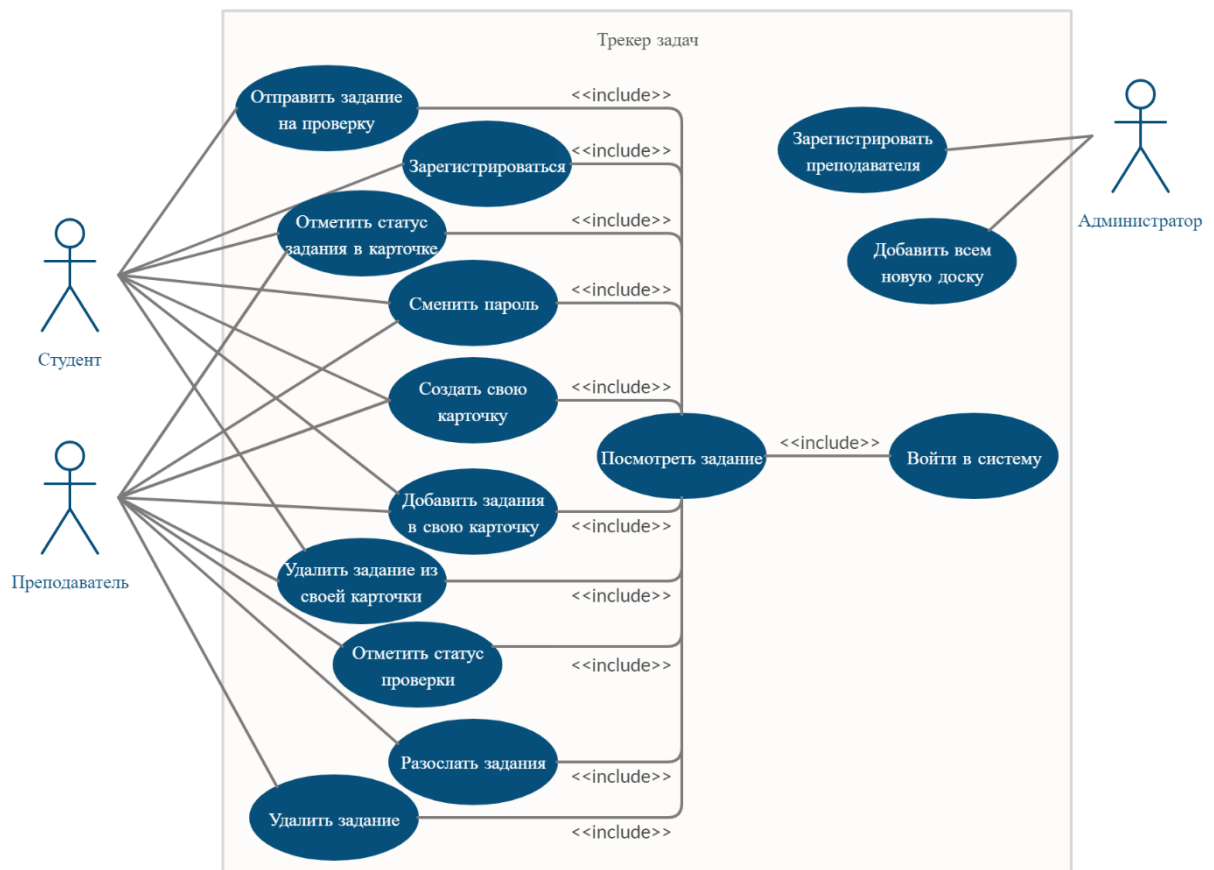


Рисунок 2.1 - Диаграмма вариантов использования проекта

Исходя из этой диаграммы можно сделать заключения о функционале, который требуется реализовать в приложении. Также она помогает сориентироваться в разделении ролей и требований приложения. Может служить визуализацией технического задания для разработчика.

2.1.2 Архитектура приложения

После изучения Use-Case диаграммы можно сделать выводы о необходимых компонентах, и как их скомпоновать. Для наглядности составим диаграмму архитектуры приложения.

Очевидно, что пользователь должен иметь какой-то интерфейс для взаимодействия с внутренней логикой приложения. Для его реализации был выбран JavaScript фреймворк React совместно с фреймворком, реализующим управление потоками данных, MobX. Для определения комплекса

инструментов мною было проведено сравнительное исследование трёх самых популярных JavaScript фреймворков, представленное в таблице 2.1.

Таблица 2.1 Сравнение Frontend фреймворков

Фреймворк	React	Angular	Vue
Удобен для микросервисной архитектуры	+	-	+
Легок в освоении	+	+-	+
Наличие VirtualDom	+	-	+
Наличие хорошей документации	+	+	-

В ходе исследования было выделено четыре параметра для сравнения: удобство для микросервисной архитектуры, легкость в освоении, наличие VirtualDom и наличие хорошей документации.

Нетрудно заметить, что по итогам сравнения приоритет получил React. Говоря о фреймворке для управления потоком данных, был выбран MobX, поскольку является самым легковесным, масштабируемым и простым в использовании.

В качестве прокси сервера был выбран стремительно набирающий популярность инструмент NGINX. Он имеет отличную документацию, сообщество, является лёгким в освоении и достаточно быстр благодаря своей асинхронной архитектуре.

Для реализации внутренней логики приложения был выбран технологический комплекс, включающий язык программирования Python и MVC фреймворк Django. Причины выбора были следующими. Язык Python достаточно прост в использовании и позволяет быстро программировать гибкие приложения, к тому же он имеет большое количество встроенных библиотек, облегчающих процесс разработки. То же касается и фреймворка

Django. Обширный спектр готовых компонентов позволяет оптимальным образом реализовать Web-приложение, использующее шаблон MVC [11].

Во множестве различных инструментов по конструированию и управлению базами данных выделяются следующие две группы: SQL и noSQL. SQL подразумевает традиционный реляционный подход в то время, как идея noSQL заключается в избавлении от ограничений при хранении и использовании данных. Такие БД используют неструктурированный подход и предлагают много эффективных способов работы с информацией в отдельных сценариях. Сравнение таких подходов описано в таблице 2.2.

Таблица 2.2 Сравнение SQL и noSQL подходов

Подход	SQL	noSQL
Скорость	-	+
Масштабируемость	-	+
Надёжность	+	-
Данные	Структурированы жёстко	Неструктурированы Изменчивы
Скорость разработки	-	+

Несмотря на то, что суммарно оба подхода имеют одинаковое количество преимуществ, noSQL подход имеет два существенных недостатка. Поскольку мы имеем дело с учебным процессом, нам очень важны такие параметры как надёжность и неизменчивость. Следовательно, выбор был сделан в пользу SQL.

Существует огромное разнообразие инструментов, реализующих SQL подход. Можно выделить следующие популярные решения MySQL, Oracle, Microsoft SQL Server, PostgreSQL. Для реализации текущего проекта было выбран инструмент PostgreSQL, поскольку он имеет удобные средства для масштабирования, существенной оптимизации работы, реляционно-объектную структуру, позволяющую внедрить дополнительные


users		
	id_user	bigint
	first_name	varchar(20)
	id_group	bigint
	second_name	varchar(20)
	patronymic	varchar(20)
	email	varchar(20)

Рисунок 2.3 - Структура таблицы пользователей

- Группы, представленные на рисунке 2.4. Это таблица-словарь, сопоставляющая id группы её учебному названию. В случае, когда пользователь – преподаватель, учебное название группы – название предмета, который он ведёт.


groups		
	id_group	bigint
	code	varchar(10)

Рисунок 2.4 - Структура таблицы групп

- Доски, представленные на рисунке 2.5. Это рабочие окружения – доски, к которым «прикрепляются» карточки с заданиями. Требуется хранить id пользователя-владельца доски, а также её название.


dashboards		
	id_dashboard	bigint
	id_user	bigint
	name	varchar(20)

Рисунок 2.5 - Структура таблицы досок

- Карточки, представленные на рисунке 2.6. Это направления деятельности, например, предмет обучения. Требуется хранить id предмета (направления) карточки, id владельца, id доски и название самой карточки.


cards		
	id_card	bigint
	id_subject	bigint
	id_user	bigint
	id_dashboard	bigint
	name	varchar(20)

Рисунок 2.6 - Структура таблицы карточек

5. Задачи, представленные на рисунке 2.7. Это задача, которая находится в какой-то карточке пользователя. Требуется хранить id карточки, к которой привязана задача, название задачи, её описание, дата назначения и окончания, а также id преподавателя, который её назначил.


tasks		
	id_task	bigint
	id_card	bigint
	name	varchar(20)
	description	text
	start_date	date
	end_date	date
	id_tutor	bigint
	id_user	bigint

Рисунок 2.7 - Структура таблицы заданий

6. Предметы, представленные на рисунке 2.8. Это таблица-словарь, сопоставляющая id предмета (направления) его названию. Помимо названий предметов существует название, означающее личную карточку.


subjects		
	id_subject	bigint
	name	varchar(40)

Рисунок 2.8 - Структура таблицы предметов

Сама схема представлена на рисунке 2.9:

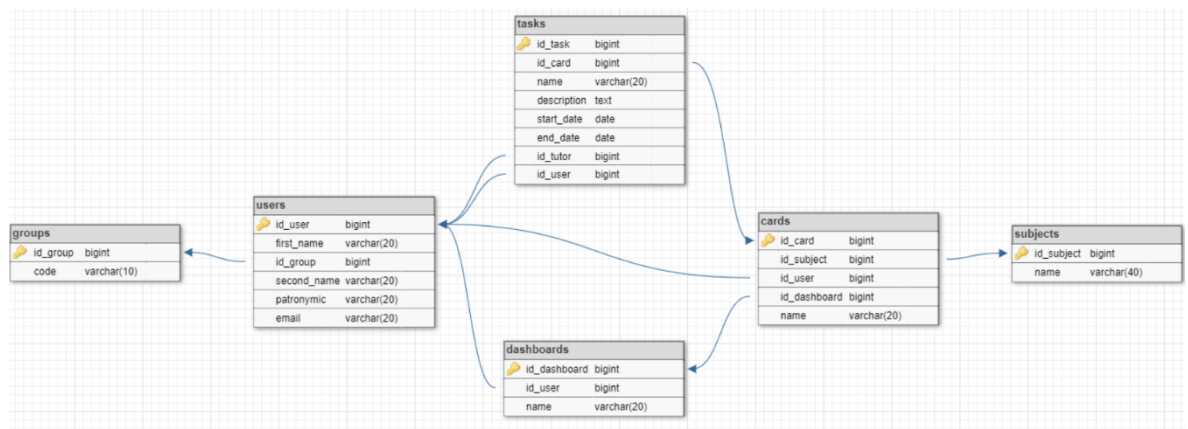


Рисунок 2.9 - Схема базы данных проекта

На схеме отчётливо можно видеть сущности БД, связанные между собой. Присутствуют в основном связи один-ко-многим (1:M), а также две таблицы, выполняющие функцию словаря.

2.2 Разработка практических решений

Итак, суммируя всё сказанное выше, мы имеем необходимую информацию, чтобы начать непосредственно разработку. Однако, следует описать методы взаимодействия пользовательского интерфейса и внутренней логики приложения – API, представленные в таблице 2.3. Используя открытый инструмент Swagger, можно удобно и наглядно спроектировать API. Был выделен ряд категорий запросов по сущностям: доска, карточка, задание и пользователь.

Таблица 2.3 Методы API

Категория	Метод	URL	Описание
Доска dashboard	GET	/dashboard	Получить все доски
	DELETE		Удалить все доски
	GET	/ {userId} /dashboard	Получить доски пользователя
	POST		Добавить доску пользователю
	DELETE		Удалить доски пользователя
	GET	/ {userId} /dashboard /	Получить доску пользователя
	PUT	{dashboardId}	Обновить доску пользователя
	DELETE		Удалить доску пользователя
Карточка card	GET	/card	Получить все карточки
	DELETE		Удалить все карточки
	GET	/ {userId} /card	Получить карточки пользователя
	POST		Добавить карточку пользователю
	DELETE		Удалить карточки пользователя
	GET	/ {userId} /card /	Получить карточку пользователя
	PUT	{cardId}	Обновить карточку пользователя

	DELETE		Удалить карточку пользователя
--	--------	--	-------------------------------

Продолжение таблицы 2.3

Задание task	GET	/task	Получить задания
	DELETE		Удалить задания
	GET	/{userId}/task	Получить задания пользователя
	POST		Добавить задание пользователю
	DELETE		Удалить задания пользователя
	GET	/{userId}/task/ {taskId}	Получить задание пользователя
	PUT		Обновить задание пользователя
	DELETE		Удалить задание пользователя
Пользователь user	POST	/user	Создать пользователя
	POST	/user/createWithList	Создать N пользователей
	GET	/user/login	Войти пользователю
	GET	/user/logout	Выйти пользователю
	GET	/user/{userId}	Получить пользователя
	PUT		Обновить пользователя
	DELETE		Удалить пользователя

Теперь предоставляется возможным вести параллельную разработку интерфейса приложения и его программно-аппаратной части.

2.2.1 Frontend

Используя фреймворк React, мы описываем в программном коде компоненты, из которых будет состоять пользовательский интерфейс, а также маршрутизатор, который отвечает за делегирование запросов (действий) пользователя [12].

Был реализован следующий набор сущностей интерфейса:

1. Задание, представленное на рисунке 2.10.

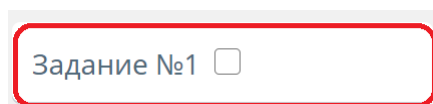


Рисунок 2.10 - Элемент интерфейса – задание

2. Карточка, представленная на рисунке 2.11.

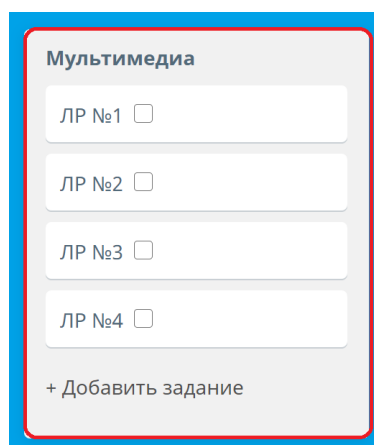


Рисунок 2.11 - Элемент интерфейса - карточка

3. Доска, представленная на рисунке 2.12.

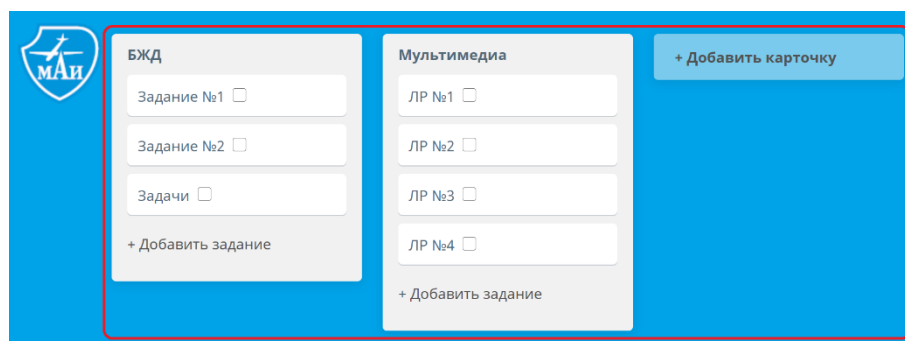


Рисунок 2.12 - Элемент интерфейса - доска

Для формирования внешнего вида пользовательского интерфейса были описаны стили в формате CSS. Файл со стилями указывается в HTML-странице. Он содержит декларативное описание стилей для групп элементов, выбранных по сопоставлению с написанным селектором. Селектор содержит специальное выражение, описывающее класса, номера, теги элементов и другие их атрибуты [13].

В результате проделанной работы получился пользовательский интерфейс, изображение которого находится на рисунке 2.13

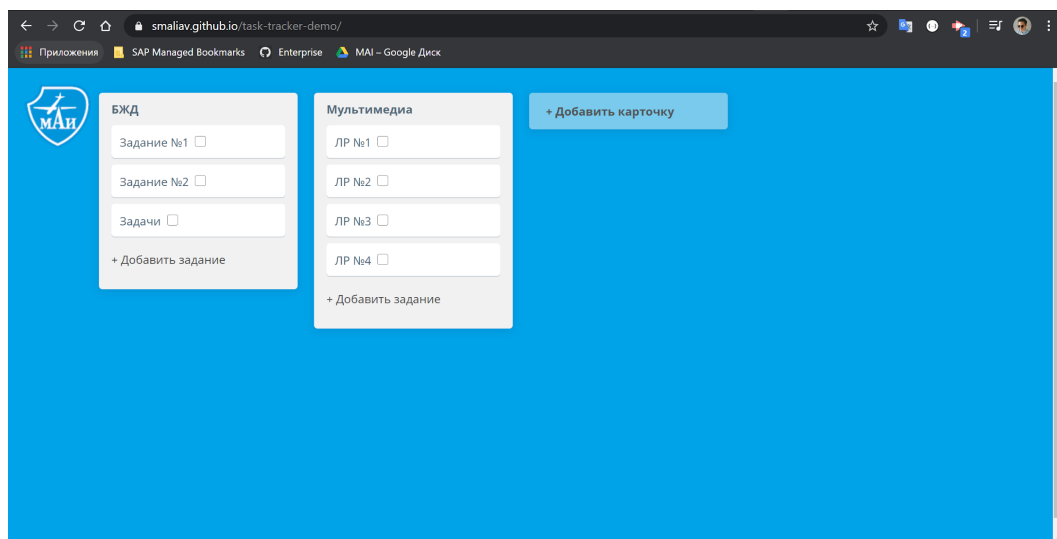


Рисунок 2.13 - Интерфейс приложения

2.2.2 Backend

Используя фреймворк Django, мы описываем в программном коде компоненты, из которых будет состоять внутренняя логика приложения, а именно: маршрутизатор, контроллеры и модели. Такой набор компонентов соответствует шаблону MVC.

В соответствии с описанным API был реализован маршрутизатор (Router), представленный ниже:

```
urlpatterns = [
    re_path(r'^admin/', admin.site.urls),
    re_path(r'^api/v0/', include('api_v0.urls')),
    re_path(r'^', include('main.urls'))
]
```

Маршрутизаторы могут составлять иерархию для более оптимизированного поиска совпадений и разделения логики приложения. Код маршрутизатора нижнего уровня выглядит следующим образом:

```
urlpatterns = [
    # Dashboard urls
    re_path(r'^dashboard/', dashboard, 'dashboard'),
    re_path(r'^(?P<user_id>\d+)/dashboard/', user_dashboards,
    'user-dashboards'),
    re_path(r'^(?P<user_id>\d+)/dashboard/(?P<dashboard_id>\d+)/',
    user_dashboard, 'user-dashboard'),

    # Card urls
    re_path(r'^card/', card, 'card'),
```

```

re_path(r'^(?P<user_id>\d+)/card/', user_cards, 'user-cards'),
re_path(r'^(?P<user_id>\d+)/card/(?P<card_id>\d+)/',
        user_card, 'user-card'),

# Task urls
re_path(r'^task/', task, 'task'),
re_path(r'^(?P<user_id>\d+)/task/', user_tasks, 'user-tasks'),
re_path(r'^(?P<user_id>\d+)/task/(?P<task_id>\d+)/',
        user_task, 'user-task'),

# User urls
re_path(r'^user/', create_user, 'create-user'),
re_path(r'^user/createWithList/', create_users,
'create-users'),
re_path(r'^user/login/', login_user, 'login-user'),
re_path(r'^user/logout/', logout_user, 'logout-user'),
re_path(r'^user/(?P<user_id>\d+)/', user, 'create-user'),
]

```

К каждому запросу сопоставлен контроллер (Controller). В качестве примера в приложении 1 приведён код функций для запросов, касающихся обработки.

Были составлены следующие ORM модели для взаимодействия с базой данных:

4. Пользователи

```

class User(AbstractUser):
    first_name = models.CharField(max_length=20)
    second_name = models.CharField(max_length=20)
    patronymic = models.CharField(max_length=20)
    group = models.ForeignKey(Group, on_delete=models.DO_NOTHING)
    email = models.EmailField()
    manager = UserManager()
    def get_full_name(self):
        full_name = '%s %s %s' % (self.first_name, self.last_name,
self.patronymic)
        return full_name.strip()
    def __unicode__(self):
        return self.get_full_name()

```

5. Группы

```

class Group(models.Model):
    code = models.CharField(max_length=10)
    def __unicode__(self):
        return self.code

```

6. Доски

```

user = models.ForeignKey(User, on_delete=models.CASCADE)

```

```

name = models.CharField(max_length=20)
manager = DashboardManager()
def __unicode__(self):
    return self.name

```

7. Карточки

```

class Card(models.Model):
    dashboard = models.ForeignKey(Dashboard,
on_delete=models.CASCADE)
    subject = models.ForeignKey(Subject,
on_delete=models.DO_NOTHING)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=20)
    manager = CardManager()
    def __unicode__(self):
        return self.name

```

8. Задачи

```

class Task(models.Model):
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    tutor = models.ForeignKey(User, on_delete=models.DO_NOTHING)
    user = models.ForeignKey(User, on_delete=models.DO_NOTHING)
    name = models.CharField(max_length=20)
    description = models.TextField()
    start_date = models.DateTimeField()
    end_date = models.DateTimeField(blank=True)
    manager = TaskManager()
    def __unicode__(self):
        return self.name

```

9. Предметы

```

class Subject(models.Model):
    name = models.CharField(max_length=20)
    def __unicode__(self):
        return self.name

```

Таким образом мы получаем готовую программно-аппаратную часть, написанную на Python с использованием MVC фреймворка Django. К большинству моделей был написан свой класс Manager, реализующий бизнес-логику конкретной модели. Его код указан в приложении 2. Такой подход позволяет повторно использовать код и логически разделять функциональность приложения. К тому же он помогает программисту избавиться от плохого шаблона разработки - Fat Controller, в котором вся логика обработки данных находится в контроллере.

Удобным является то, что нам не требуется непосредственно работать с базой данных и писать SQL запросы. Этим занимается ORM модуль Django. Однако, мы можем следить за её состоянием и набором данных при помощи pgAdmin [14]. Это открытое программное обеспечение, выполняющее роль панели администратора для PostgreSQL, некоторые элементы которой можно наблюдать на рисунках 2.14 и 2.15.

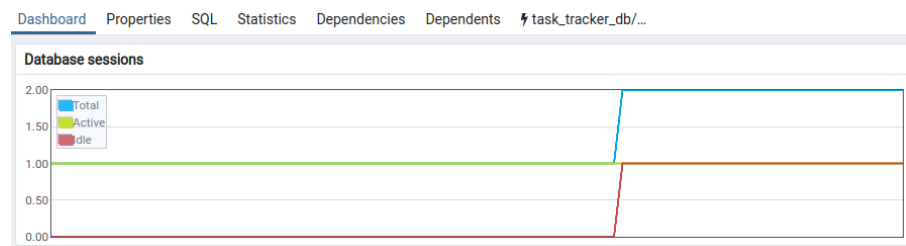


Рисунок 2.14 - Панель количества сессий

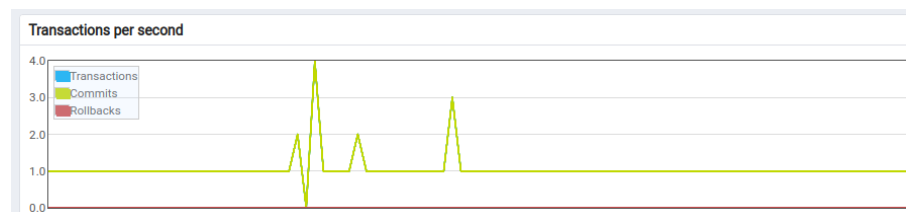


Рисунок 2.15 - Панель количества транзакций

2.2.3 Размещение на вычислительной машине

Для размещения представленного Web-приложения требуется создать образы Docker-контейнеров. Ниже представлены Docker компоненты приложения. Всего их три, по одному на базу данных, внутреннюю логику приложения и прокси-сервер.

1. Dockerfile базы данных. На ресурсе DockerHub (официальное хранилище образов Docker) существует контейнер с PostgreSQL. Однако, его следует правильно сконфигурировать, а именно указать логин, пароль, смонтировать том для хранения данных локально и указать прочие дополнительные параметры.

```
docker run --rm --name pg-docker -e POSTGRES_PASSWORD=postgres
-d -p 5432:5432 -v
$HOME/docker/volumes/postgres:/var/lib/postgresql/data postgres
```


2. Dockerfile программно-аппаратной части. В данном случае требуется наследовать Docker образ от системы Linux, затем переместить необходимые для приложения файлы в образ и написать скрипт, являющийся входной точкой в контейнер и запускающий сам Django сервер.
3. Dockerfile прокси-сервера. В Docker Hub также есть образ NGINX, от которого наследуется новый образ. В него включаются конфигурационные файлы для корректного функционирования.

```
FROM nginx
LABEL authors=smaliyav

# Upload configs
COPY nginx/nginx.conf \
      nginx/.htpasswd \
      /etc/nginx/
```

Для запуска, масштабирования, поддержания контейнеров используется инструмент Kubernetes. Для настройки его работы необходимо сначала описать количество используемых узлов сети Kubernetes, затем их роли. Далее идёт настройка подсети и установка дополнительных параметров. Затем необходимо на каждом узле запустить желаемое количество реплик выбранных контейнеров.

ЗАКЛЮЧЕНИЕ

В результате проделанной работы были выполнены поставленные задачи:

1. Спроектирован программный интерфейс приложения (API), связывающий программный интерфейс (клиентскую часть) и программно-аппаратную часть приложения.
2. Реализована программно-аппаратная часть сервиса, отвечающая за операции с данными и вычислительные механизмы.
3. Разработан программный интерфейс приложения, служащий связующим звеном между приложением и конечным пользователем.
4. Спроектирована база данных, позволяющая хранить данные пользователей и самого приложения и выполнять их обработку.
5. Архитектура приложения разработана с использованием современного подхода микросервисной архитектуры и по возможности максимальна к ней приближена.
6. Используются технологии контейнеризации и оркестрации контейнерами для повышения удобства развёртывания, поддержания и масштабирования приложения.

Представленный программный комплекс позволяет студенту и преподавателю выполнять следующие действия:

- Студент может вести свои задания (заметки) независимо от заданий преподавателя. Он может наблюдать за выданными заданиями и изменять их статус для себя или преподавателя.
- Преподаватель может также вести свои задания (заметки) и назначать задания студентам, к тому же следить за статусом выполнения назначенных заданий.

В будущем планируется улучшать приложение, масштабируя его на несколько серверов и добавляя новые функциональные возможности.

1.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ньюмен С. Создание микросервисов, СПб.: Питер, 2016, 304 с.
2. Christudas B. Practical Microservices Architectural Patterns, Springer Science+Business Media New York, 2019, pp. 916.
3. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка, СПб.: Питер, 2018, 640 с.
4. Patni S. Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS, Springer Science+Business Media New York, 2017, pp. 126.
5. Poulton N. Docker Deep Dive // DockerCon EU, 2017, version 4, pp. 191.
6. Arundel J., Domingus J. Cloud Native DevOps with Kubernetes, O'Reilly Media, 2019, pp. 346
7. Роббинс Д. HTML5, CSS3 и JavaScript. Исчерпывающее руководство. 4-е издание / пер. с англ. М. А. Райтман, М.: Эксмо, 2014, 528 с.
8. Лутц М. Изучаем Python, 4-е издание. / Пер. с англ. СПб.:Символ-Плюс, 2011, 1271 с.
9. Лукин В. Н. Введение в проектирование баз данных, 3-е издание, 2015, 144 с.
10. Розенберг Д., Скотт К. Применение объектного моделирования с использованием UML и анализ прецедентов / Пер. с англ. М.: ДМК Пресс, 2002, .159 с.
11. Меле А. Django 2 в примерах / пер. с англ. Д. В. Плотниковой, М.: ДМК Пресс, 2019, 408 с.
12. Bertoli M. React Design Patterns and Best Practices, Packt Publishing 2017, pp. 332.
13. Хавербеке М. Выразительный JavaScript. Современное веб-программирование. 3-е изд., СПб.: Питер, 2019, 480 с.

14. Васильев А. Ю. Работа с PostgreSQL: настройка и масштабирование, Creative Commons Attribution-Noncommercial 4.0 International, 2017, 288 с.

15.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Программный код контроллеров

```
def task(request):
    if request.method == 'GET':
        tasks = Task.manager.get_all()
        json_res = []
        for task in tasks:
            json_res.append({
                'taskId': task.task_id,
                'cardId': task.card.card_id,
                'user': task.user.name,
                'tutor': task.tutor.name,
                'name': task.name,
                'description': task.description,
                'start_date': task.start_date,
                'end_date': task.end_date,
            })
        return HttpResponse(
            content = json,
            content_type='application/json',
            status = 200
        )
    elif request.method == 'DELETE':
        Task.manager.delete_all()

        return HttpResponse(
            content = '',
            content_type = '',
            status = 200,
        )
    else:
        return HttpResponseBadRequest()
def user_tasks(request, user_id):
    if request.method == 'GET':
        try:
            task = Task.manager.get_all_user(user_id)
            json_res = []
            for task in tasks:
                json_res.append({
                    'taskId': task.task_id,
                    'cardId': task.card.card_id,
                    'user': task.user.name,
                    'tutor': task.tutor.name,
                    'name': task.name,
                    'description': task.description,
                    'start_date': task.start_date,
                    'end_date': task.end_date,
                })
        }
```

```

        except Task.DoesNotExist:
            json_res = []
            return HttpResponse(
                content = json_res,
                content_type='application/json',
                status = 200
            )
        elif request.method == 'POST':
            json_req = json.loads(request.body)
            task = Task.manager.create_card(json_req)      # TODO Check
params

            return HttpResponse(
                content = '',
                content_type='text/plain',
                status = 200
            )
        elif request.method == 'DELETE':
            Task.manager.delete_all_user(user_id)
            return HttpResponse(
                content = '',
                content_type='text/plain',
                status = 200
            )
        else:
            return HttpResponseBadRequest()
def user_task(request,user_id, task_id):
    if request.method == 'GET':
        try:
            tasks = Task.manager.get_user_task(id=task_id)
            json_res = []
            for task in tasks:
                json_res.append({
                    'taskId': task.task_id,
                    'cardId': task.card.card_id,
                    'user': task.user.name,
                    'tutor': task.tutor.name,
                    'name': task.name,
                    'description': task.description,
                    'start_date': task.start_date,
                    'end_date': task.end_date,
                })
        except Task.DoesNotExist:
            json_res = []
            return HttpResponse(
                content = json_res,
                content_type='application/json',
                status = 200
            )
        elif request.method == 'PUT':

```

```
    json_req = json.loads(request.body)
    Task.manager.update_task(json_req.task_id, json_req) # TODO
Check
    return HttpResponse(
        content = '',
        content_type='text/plain',
        status = 200
    )
elif request.method == 'DELETE':
    Task.manager.delete_all_user(user_id)
    return HttpResponse(
        content = '',
        content_type='text/plain',
        status = 200
    )
else:
    return HttpResponseBadRequest()
```


ПРИЛОЖЕНИЕ Б

Программный код классов Manager

```
class TaskManager(models.Manager):
    # /task
    def get_all(self):
        return self.all()
    def delete_all(self):
        return self.all().delete()

    # /{userId}/task
    def get_all_user(self, user_id):
        return self.filter(user=user_id)
    def create_task(self, **kwargs):
        return self.create(**kwargs)
    def delete_all_user(self, user_id):
        return self.filter(user=user_id).delete()

    # /{userId}/task/{taskId}
    def get_user_task(self, task_id):
        return self.filter(task_id=task_id)
    def update_task(self, task_id, **kwargs):
        return self.filter(task_id=task_id).update(**kwargs)
    def delete_task(self, task_id):
        return self.filter(task_id=task_id).delete()

class CardManager(models.Manager):
    # /card
    def get_all(self):
        return self.all()
    def delete_all(self):
        return self.all().delete()

    # /{userId}/card
    def get_all_user(self, user_id):
        return self.filter(user=user_id)
    def create_card(self, **kwargs):
        return self.create(**kwargs)
    def delete_all_user(self, user_id):
        return self.filter(user=user_id).delete()

    # /{userId}/card/{cardId}
    def get_user_card(self, card_id):
        return self.filter(card_id=card_id)
    def update_card(self, card_id, **kwargs):
        return self.filter(card_id=card_id).update(**kwargs)
    def delete_card(self, card_id):
        return self.filter(card_id=card_id).delete()
```

```

class UserManager(models.Manager):
    # /user
    def get_users(self):
        return self.all()

    # /user/{userId}
    def get_user(self, user_id):
        return self.filter(user_id=user_id)
    def update_user(self, user_id, **kwargs):
        return self.filter(user_id=user_id).update(**kwargs)
    def delete_user(self, user_id):
        return self.filter(user_id=user_id).delete()
class DashboardManager(models.Manager):
    # /dashboard
    def get_all(self):
        return self.all()
    def delete_all(self):
        return self.all().delete()

    # /{userId}/dashboard
    def get_all_user(self, user_id):
        return self.filter(user=user_id)
    def create_dashboard(self, **kwargs):
        return self.create(**kwargs)
    def delete_all_user(self, user_id):
        return self.filter(user=user_id).delete()

    # /{userId}/dashboard/{dashboardId}
    def get_user_dashboard(self, dashboard_id):
        return self.filter(dashboard_id=dashboard_id)
    def update_dashboard(self, dashboard_id, **kwargs):
        return self.filter(dashboard_id=dashboard_id).update(**kwargs)
    def delete_dashboard(self, dashboard_id):
        return self.filter(dashboard_id=dashboard_id).delete()

```