Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого

LET'S GO PROGRAMMING

КУРСОВОЕ ПРОЕКТИРОВАНИЕ

Учебное пособие

Авторы:
Александрова О.В
Андрианов А.А
Денисенко Б. А.
Дутова А. А.
Клочкова Л. И.
Королев Д. О.
Котлярова Л. П.
Никифоров И. В.
Ромашов В. А.
Тянутов М. В.

Санкт-Петербург
2023

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ

ACID принципы	— Atomicity, Consistency, Isolation, Durability — набор требований к транзакционной системе, обеспечивающий наиболее надёжную и предсказуемую её работу
Angular CLI	— Angular Command Line Interface — инструмент интерфейса командной строки, который используется для поддержки приложений Angular непосредственно из командной оболочки.
API	— Application Program Interface — программный интерфейс приложения
CAP теорема	— Consistency, Availability, Partition tolerance — теорема, утверждающая, что в любой реализации распределенных вычислений возможно обеспечить не более двух из трех следующих свойств: согласованность данных, доступность, устойчивость к разделению
CQRS	— Command and Query Responsibility Segregation — это стиль архитектуры, в котором операции чтения отделены от операций записи
CRUD	— Create, Read, Update, Delete — четыре базовые функции, используемые при работе с базами данных.
ETL	— Extract, Transform, Load — процесс миграции данных из одного источника в другой
HATEOAS	— Hypermedia As The Engine Of Application State — архитектурные ограничения для REST-приложений
HLD	— High Level design — верхнеуровневое описание архитектуры системы
HTTP	— HyperText Transfer Protocol

	— широко распространенный протокол передачи данных
JSON	— JavaScript Object Notation — текстовый формат обмена данными
MVC	— Model-View-Controller — схема разделения данных приложения и управляющей логики
NoSQL	— Not only SQL — термин относится к нереляционным типам баз данных, данные в которых хранятся в формате, отличном от реляционных таблиц. Такие БД используют средства, отличные от SQL.
ORM	— Object-Relational Mapping — технология программирования, суть которой заключается в создании «виртуальной объектной базы данных»
REST	— Representational State Transfer — архитектурный стиль взаимодействия компонентов распределенного приложения в сети
UI	— User Interface — пользовательский интерфейс
UML	— Unified Modeling Language — язык графического описания для объектного моделирования в области разработки программного обеспечения
URI	— Uniform Resource Identifier — последовательность символов, идентифицирующая абстрактный или физический ресурс
URL	— Uniform Resource Locator — адрес, который выдан уникальному ресурсу в интернете
UX	— User Experience

	— опыт, который получает пользователь при взаимодействии с сайтом или приложением
WAL	— Write-ahead logging — Журнал предзаписи - стандартный метод обеспечения целостности данных
YML/ YAML	— Yet Another Markup Language — язык сериализации данных
БД	— База Данных
ОС	— Операционная Система

ВВЕДЕНИЕ

Целью курсового проектирования является формирование навыков разработки промышленных клиент-серверных приложений с применением принципов микросервисной архитектуры, языков программирования Golang, фрейморка Angular и TypeScript.

В разделе «Введение» приведено описание цели курсового проекта и названия инструментов, используемых при его реализации.

Раздел «База данных», «Коннектор JIRA», «Серверная часть», «Пользовательский интерфейс» содержит методические указания и рекомендации к выполнению каждого из компонентов проекта.

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

В разделе приведено краткое описание и высокоуровневая постановка задачи для создания программного средства.

Программное средство должно быть представлено в виде клиент-серверного приложения.

Входные данные:

адрес JIRA репозитория(ев)

Выходные данные:

web-страничка, позволяющая посмотреть аналитические данные по проекту.

Примеры задач аналитики, которые позволят сравнить проекты между собой и выявить наиболее активные/живые/сильные/слабые/т.д.:

- построение гистограммы, отражающей время, которое задача провела в открытом состоянии (от момента создания до момента закрытия). По оси абсцисс – расположить время. По оси ординат – расположить суммарное количество задач, которое было в открытом виде соответствующее время. В расчет брать только закрытые задачи по проекту.
- построить диаграммы, которые показывают распределение времени по состояниям задачи. По оси абсцисс – расположить время. По оси ординат – расположить суммарное количество задач, которое было в открытом виде соответствующее время. В расчет брать только закрытые задачи по проекту. Для каждого состояния должна строиться своя диаграмма
- построить график, показывающий количество заведенных и закрытых задач в день. График должен отражать не только информацию о количестве задач в день, но и накопительный итог по задачам. По оси абсцисс – расположить календарные дни. По оси ординат – расположить суммарное количество задач для этого дня (открытые, закрытые и для каждого – накопительный итог);
- построить гистограмму, отражающую время, которое затратил пользователь на ее выполнение на основе залогированного времени. По оси абсцисс – расположить время. По оси ординат – расположить

суммарное количество задач, которое соответствует этому времени. В расчет брать только закрытые задачи по проекту;
- построить график, выражающий количество задач по степени серьезности.

Функциональные требования

- программная реализация должна позволять указывать один или несколько проектов для проведения аналитики;
- отображение на странице данных по проекту должно быть сделано в формате сравнения товаров в интернет магазине;
- должна быть предусмотрена мониторинговая система опроса проектов для динамичного отображения изменения информации и графиков (псевдо real-time). Время повторного опроса настраивается, м.д. раз в 1 минуту, может быть 1 раз в час;
- в пользовательском интерфейсе должно быть меню с настройками администратора инструмента, списком всех проектов и их сухой статистикой по открытым задачам JIRA, страница сравнения результатов, страница мониторинга изменения в задачах JIRA.

Технические требования

- ETL-процесс для выгрузки данных из JIRA;
- база данных для offline хранения информации о JIRA задачах;
- backend с REST-интерфейсом;
- front-end для взаимодействия пользователя;
- микросервисная архитектура;
- автоматизация тестирования.

ТЕХНИЧЕСКИЕ ТРЕБОВАНИЯ

В разделе приводятся детальные требования на компоненты программной системы, которые позволяют понять фактически создаваемое поведение программного комплекса.

Общие требования на систему

Функциональные требования

Таблица 1. Общие функциональные требования на приложение

№ требования	Описание функциональности
ОФ1	Программное обеспечение должно быть представлено в виде распределенного микросервисного приложения, состоящего из компонент: ● коннектор к системам трекинга задач; ● база данных; ● серверная часть (backend); ● пользовательский интерфейс (web-приложение).
ОФ1.1	Формат дистрибутива для распространения: zip
ОФ1.2	В дистрибутив входят: ● скрипт установки; ● скрипт запуска коннектора; ● скрипт запуска сервисов БД; ● скрипт запуска backend; ● скрипт запуска webapp; ● конфигурационные файлы; ● библиотеки и lib; ● директорию logs; ● временные директории (temp); ● директорию results; ● docker images; ● readme с инструкцией установки и запуска всех сервисов на отдельно стоящем ПК (локальный запуск) (ОФ1.2.1); ● директория с документаций.
ОФ1.2.1	Локальный запуск подразумевает запуск сервисов дистрибутива на персональном компьютере пользователя с указанием всех требований к системе и зависимостей
ОФ1.3	В директории doc находится UML-диаграмма классов модели данных инструмента

№ требования	Описание функциональности
ОФ2	Проектируемая система должна решать следующие аналитические задачи
ОФ2.1	Построение гистограммы, отражающей время, которое задача провела в открытом состоянии (от момента создания до момента закрытия). По оси абсцисс – расположить время. По оси ординат – расположить суммарное количество задач, которое было в открытом виде соответствующее время. В расчет брать только закрытые задачи по проекту
ОФ2.1.1	В случае отсутствия закрытых задач необходимо сообщить пользователю о невозможности построения гистограммы
ОФ2.1.2	В настройках проекта отображать следующую задачу с названием: “Гистограмма, отражающая время, которое задачи провели в открытом состоянии”
ОФ2.1.3	Максимальное количество столбцов в гистограмме 75, минимальное 1
ОФ2.1.4	Значение времени по оси абсцисс имеют следующий вид: - задачи, на выполнение которых ушло меньше дня, разбиваются по часам и отображаются от “0h” (задачи на выполнение которых ушло меньше часа) до “23h” - задачи, на которые ушел день и больше, разбиваются по дням от “1day” до “30day” - задачи, на которые ушел месяц, разбиваются аналогично. От 1 до 11. - задачи, на решение которых ушел год и более, разбиваются аналогично.
ОФ2.1.4.1	Максимальное значение для разбиения - 7 лет. Если задача провела в состоянии больше этого времени она будет внесена в столбец с названием “8+year”
ОФ2.2	Построить диаграммы, которые показывают распределение времени по состояниям задачи. По оси абсцисс – расположить время. По оси ординат – расположить суммарное количество задач, которое было в данном состоянии соответствующее время. В расчет брать только закрытые задачи по проекту.
ОФ2.2.1	Для каждого состояния должна строиться своя диаграмма.
ОФ2.2.2	В настройках проекта отображать следующую задачу с названием: “Диаграммы, которые показывают распределение времени по состоянием задач”

№ требования	Описание функциональности
ОФ2.3	Построить график, показывающий количество заведенных и закрытых задач в день. График должен отражать не только информацию о количестве задач в день, но и накопительный итог по задачам. По оси абсцисс – расположить календарные дни. По оси ординат – расположить суммарное количество задач для этого дня (открытые, закрытые и для каждого – накопительный итог);
ОФ2.3.1	В настройках проекта отображать следующую задачу с названием: “График активности по задачам”
ОФ2.4	Построить гистограмму, отражающую время, которое затратил пользователь на ее выполнение на основе залогированного времени. По оси абсцисс – расположить время. По оси ординат – расположить суммарное количество задач, которое соответствует этому времени. В расчет брать только закрытые задачи по проекту;
ОФ2.4.1	В настройках проекта отображать следующую задачу с названием: “График сложности задач”
ОФ2.5	Построить график, выражающий количество задач по степени серьезности.
ОФ2.5.1	В настройках проекта отображать следующую задачу с названием: “График, отражающий приоритетность всех задач”
ОФ2.5.2	Именование столбцов будет совпадать с именованием приоритетов задач существующих в базе проекта
ОФ3	Инструмент поддерживает работу в следующих операционных системах: ● Microsoft Windows 10; ● Ubuntu.

Нефункциональные требования

Таблица 2. Общие нефункциональные требования на приложение

№ требования	Описание функциональности
ОНФ1	Проектируемая программная система должна поддерживать горизонтальное масштабирование
ОНФ1.1	Должна быть представлена возможность увеличения количества реплик микросервисов
ОНФ2	Проектируемая система должна быть отказоустойчивой
ОНФ2.1	В случае выхода из строя узла система должна восстановить свое состояние
ОНФ2.1.1	Реплики stateful компонентов должны быть подняты согласно последней версии компонента, данные которого лежат на примонтированном к серверу томе памяти
ОНФ2.1.2	В случае падения stateless приложений система не должна понести потерь данных и должна восстановится в течение 5 минут
ОНФ3	Проектируемая система должна реплицировать данные для обеспечения надежности хранения информации
ОНФ3.1	Stateful компоненты при изменении данных должны сохранять свое текущее состояние на смонтированном к серверу томе памяти

Требования к модулю коннектора

Функциональные требования

Таблица 3. Функциональные требования на модуль коннектора

№ требования	Описание функциональности
КФ1	Модуль коннектора должен реализовывать загрузку данных с Jira
КФ1.1	Загрузка должна осуществляться при помощи JIRA REST API
КФ1.1.1	Должна быть реализована возможность загрузки всех полей Issues включая Changelog

№ требования	Описание функциональности
КФ1.1.2	Должна быть реализована система повторных запросов в случае закрытия соединения
КФ1.1.2.1	Если соединение оказалось закрыто, тогда необходимо подождать время, указанное в конфиг-файле (КФ2). В случае повторной неудачи необходимо экспоненциально повышать это время ожидания и повторять запрос, пока это время не превысит максимальное время ожидание, указанное в конфиг-файле (КФ2)
КФ1.1.3	Для загрузки данных должна быть реализована многопоточность
КФ1.1.3.1	При загрузке данных из JIRA в БД должна поддерживаться атомарность, то есть, если при скачивании части данных произошла ошибка, то никакие данные не будут записаны в БД (все или ничего)
КФ1.1.3.2	Если в одном из потоков произошла ошибка, тогда все остальные потоки должны завершиться досрочно
КФ1.2	Загруженные данные должны сохраняться в локальную базу данных (КФ3).
КФ2	Модуль должен иметь возможность конфигурации настроек
КФ2.1	Параметры конфигурации хранятся в файле конфига в формате YML
КФ2.2	Конфиг содержит настройки для подключения базы данных и настройки самого коннектора

КФ2.2.1	Конфигурационный файл должен содержать следующую информацию:
	- Настройки базы данных
	  - имя пользователя PostgreSQL
	  - пароль пользователя PostgreSQL
	  - хост PostgreSQL
	  - порт PostgreSQL
	  - название базы данных
	- Настройки модуля
	  - Порт для локального http сервера
	  - URL сайта JIRA
	  - Количество Issues, загружаемые из Jira за 1 запрос
	  - Количество потоков при выгрузке из Jira
	  - Максимальное время ожидания для повторной отправки запроса в Jira
	  - Начальное время ожидания для повторной отправки запроса в Jira

КФ3	Данные должны выгружаться в базу данных PostgreSQL

№ требования	Описание функциональности
КФ3.1	Подключение к базе данных должно осуществляться через драйвер database/sql
КФ3.1.1	Параметры подключения должны задаваться с помощью конфиг файла (КФ2)
КФ4	Модуль должен предоставлять REST API для взаимодействия с ним
КФ4.1	REST API должен включать запрос для выгрузки всех issue из проекта
КФ4.1.1	Проект указывается в качестве параметра запроса
КФ4.1.2	Если проект пустой, тогда в БД заносится только название проекта
КФ4.2	REST API должен включать запрос на получение проектов из Jira
КФ4.2.1	Запрос должен возвращать ограниченное число проектов. Какие проекты необходимо получить - указывается через параметры.
КФ4.2.1.1	Параметр limit - сколько всего проектов необходимо вернуть
КФ4.2.1.2	Параметр page - порядковый номер страницы, который необходимо вернуть
КФ4.2.1.3	Параметр search - фильтр, который накладывается на название и ключ проекта. То есть запрос должен возвращать только те проекты, названия или ключ которых содержит значение параметра search
КФ4.2.2	В случае удачного выполнения запроса должен быть возвращен JSON, который содержит массив проектов и общее количество страниц при данном параметре limit
КФ4.2.3	Возвращаемые проекты должны содержать следующие поля: ● ключ проекта ● имя проекта ● url проекта
КФ4.4	http-сервер для обработки запросов должен запускаться на отдельном порте
КФ4.1	Порт для запуска http-сервера указывается в конфиг-файле (КФ2)
КФ5	Модуль должен иметь систему логирования
КФ5.1	Логи должны сохраняться в 2 файла: logs.log и err_logs.log. В первом сохраняются все логи, а во втором уровня WARNING или выше.
КФ5.2	Логи уровня WARNING и выше должны выводиться в консоль

Нефункциональные требования

Таблица 4. Нефункциональные требования на модуль коннектора

№ требования	Описание функциональности
КНФ1	Коннектор должен быть производительным
КНФ1.1	Загрузка данных должна происходить многопоточно
КНФ1.1.1	Пользователь должен иметь возможность настройки многопоточности (выбирать количество горутин, в которых будет происходить загрузка проекта)
КНФ1.2	Процесс загрузки данных должен адаптироваться под ограничения JIRA
КНФ2	Загрузка должна осуществляться при помощи “голого” JIRA REST API без использования библиотек-оберток

Требования к базе данных

Функциональные требования

Таблица 5. Функциональные требования к базе данных

№ требования	Описание функциональности

БДФ1	В качестве основного хранилища используется реляционная база данных PostgreSQL
БДФ1.1	Используемая версия 13.8
БДФ1.2	Внутри базы данных определены три основных пользователя базы данных
БДФ1.2.1	Пользователь postgres – пользователь администратор, от имени которого меняются конфигурационные файлы
БДФ1.2.2	Пользователь pguser – который работает с базой данных
БДФ1.2.2.1	Пароль пользователя - pgpwd
БДФ1.2.2.2	У pguser есть доступ к таблицам "project", "issues", "author", "statusChange"
БДФ1.2.2.3	У pguser есть доступ к первичным ключам таблиц "project", "issues", "author",
БДФ1.2.3	Пользователь replicator – который необходим для настройки потоковой репликации
БДФ1.2.3.1	Пароль пользователя - postgres
БДФ1.3	Используемая база данных называется testdb
БДФ1.3.1	База данных содержит таблицы "project", "issues", "author", "statusChange"
БДФ1.3.2	Все публичные ключи таблиц соответствуют ключам, изъятым из JIRA
БДФ1.3.1	Время хранится в timestamp без добавления временной зоны и соответствует типу времени определенной в UNIX системах
БДФ1.4	Для поднятия базы данных в тестовом окружение используется Docker контейнер
БДФ1.4.1	Скрипт для поднятия контейнера в тестовом окружении написан как docker-compose.yaml файл
БДФ1.4.2	Docker-compose содержит образ базы данных и pgadmin

БДФ1.4.2.1	Тестовой окружение использует подготовленный разработчиками базы данных образ postgres версиии определенной в БДФ1.1
БДФ1.4.2.1.1	Скрипты для создания базы данных монтируются в папку docker-entrypoint-initdb.d
БДФ1.4.2.1.1	База данных запускается на стандартном порту 5432
БДФ1.4.2.1.1	Для тестов выставлен лимит на используемые ресурсы – 4G памяти
БДФ1.4.2.2	Тестовой окружение использует подготовленный разработчиками образ dpage/pgadmin4
БДФ1.4.2.5.1	Используемая версия - 5.7
БДФ1.4.2.5.2	Используемый порт – 5050
БДФ1.4.2.5.3	Стандартный пользователь доступен по email - pguser@mail.ru
БДФ1.4.2.5.4	Пароль для стандартного пользователя - pgadmin
БДФ1.4.2.5.5	Для тестов выставлен лимит на используемые ресурсы – 1G памяти
БДФ1.4.2.5.6	Для тестов выставлен лимит на используемые ядра – 0,5 minicors
БДФ1.4.3	Не предполагается, что для тестового окружения необходима консистентность данных.
БДФ1.4.3.1	При каждом новом запуском контейнера база данных поднимается с нуля
БДФ1.4.4	Тип сети между контейнерами – мост
БДФ1.5	База данных должна иметь возможность запуска внутри систем оркестрации (kubernetes)
БДФ1.5.1	В качестве тестового окружения используется Docker Desktop standalone Kubernetes server.
БДФ1.5.1.1	Данное решение позволяет работать только с одно нодным кластером
БДФ1.5.2	Для поднятия базы данных были созданы Service, Config-Map, StatefullSet абстракции
БДФ1.5.2.1	Сервис предоставляет механизм балансировки между контейнерами

БДФ1.5.2.1.1	Сервис привязан к приложениям, содержащим лейбл – “postgres-db-test”
БДФ1.5.2.2	Config-Map содержит информацию о дефолтном пользователе, пароле и стартовой базы данных. Определенных в БДФ1.2.3.1, БДФ1.2.2.1, БДФ1.3
БДФ1.5.2.3	StatefullSet определяет основные характеристики запускаемого приложения
БДФ1.5.2.3.1	Для запуска используется версия базы данных определенная в БДФ1.1 и образ определенный в БДФ1.4.2.1
БДФ1.5.2.3.2	Количество секунд необходимых для поднятия пустой базы – 20 секунд
БДФ1.5.2.3.3	Дополнительное время (отсрочка) после не поднятии базы – 20 секунд
БДФ1.5.2.3.4	Для хранения данных создается отдельная абстракция – запрос на получения постоянного тома (PVC)
БДФ1.5.2.3.4.1	PVC запрашивает хранилище размером в 5G для каждой реплики
БДФ1.5.2.3.4.2	Доступ к данного хранилищу определен как "ReadWriteOnce"
БДФ1.5.2.3.5	Настроены проверки на жизнеспособность и готовность
БДФ1.5.2.3.5.1	Проверка на жизнеспособность обращается к базе данных каждые 5 секунд
БДФ1.5.2.3.5.2	Проверка на жизнеспособность начинает работать через 10 секунд после поднятия приложения внутри оркестратора
БДФ1.5.2.3.5.3	Проверка на готовность начинает работать через 10 секунд после запуска инициализационного файла
БДФ1.5.2.3.5.4	Проверка на готовность повторяется каждые 5 секунд пока под не поднимется
БДФ1.5.3	Внутри k8s база данных имеет стандартный порт 5432
БДФ2	Для хранения логов используется TSDB

Нефункциональные требования

Таблица 6. Нефункциональные требования к базе данных

№ требования	Описание функциональности
БДНФ1	Для поддержки горизонтального масштабирования база данных поддерживает возможность управления количествами реплик внутри kubernetes
БДНФ1.1	Количество реплик задается внутри StatefullSet скрипта
БДНФ1.1.1	Минимальное количество реплик 2
БДНФ1.1.2	Максимальное количество реплик определяется затратами на ресурсы
БДНФ2	Проектируемое хранилище должно быть отказоустойчивым
БДНФ2.1	Для поддержки отказоустойчивости предлагается прибегнуть к потоковой репликации с поддержкой мульти мастеров
БДНФ2.2	Для обеспечения потоковой репликации был изменен ряд параметров конфигурационных файлов базы данных для мастера
БДНФ2.2.1	Был изменен файл pg_hba
БДНФ2.2.1.1	Была добавлена возможность подключения пользователя replicator ко всем хостам базы
БДНФ2.2.2	Были внесены изменения в postgresql.conf
БДНФ2.2.2.1	"archive_mode = on" параметр необходимый для отправки WAL сегмента в активное хранилище при помощи archive_command
БДНФ2.2.2.2	"archive_command = '/bin/true'" - устанавливает команду оболочки, которая будет вызываться для архивирования файла WAL.
БДНФ2.2.2.3	"archive_timeout = 0" - принудительно переключается на следующий файл WAL, если новый файл не был запущен в течение N секунд.
БДНФ2.2.2.4	"max_wal_senders = 8" - устанавливает максимальное количество одновременно запущенных процессов-отправителей WAL.
БДНФ2.2.2.5	"wal_keep_segments = 32" - устанавливает количество файлов WAL, хранящихся для резервных серверов.

БДНФ2.2.2.6	"wal_level = replica"- устанавливает уровень информации, записываемой в WAL. Значение по умолчанию — replica, которое записывает достаточно данных для поддержки архивирования и репликации WAL, включая выполнение запросов только для чтения на резервном сервере.
БДНФ2.2.2.7	"hot_standby = on" - разрешает подключения и запросы во время восстановления
БДНФ2.3	Для обеспечения репликации на стороне слушателя необходимо было настроить бэкап конфигурация

Требования на REST интерфейс

Функциональные требования

Таблица 7. Функциональные требования к backend модулю

№ требования	**Описание функциональности**
РФ1	Модуль “Resource” REST интерфейса должен удовлетворять 3 уровню Richardson Maturity Model
РФ1.1	Для работы с ресурсами должны поддерживаться методы GET/POST/PUT/DELETE
РФ1.2	Каждая ссылка HATEOAS должна содержать: ● Имя ресурса ● Ключ href и ссылку
РФ1.3	Список ссылок HATEOAS должен содержать ссылку на endpoint, который был вызван
РФ2	Из домашней конечной точки должны быть доступны ссылки на все конечные точки всех модулей, кроме модуля “Gateway”
РФ3	Все конфигурационные переменные должны быть определены в конфигурационном файле REST API интерфейса
РФ3.1	Настройка значения порта должна определяться в конфигурационном файле
РФ3.1.1	Настройка должна называться “port”
РФ3.1.2	По умолчанию значение настройки должно быть “8000”

№ требования	Описание функциональности
РФ3.2	Настройка значения времени ожидания ответа от сервера для запросов, связанных с ресурсами, которые могут быть получены с помощью CRUD операций, должна быть определена в конфигурационном файле. Эти запросы обрабатываются конечными точками вида /api/{номер версии}/{имя ресурса}/*, например, /api/v1/projects/*.
РФ3.2.1	Настройка должна называться “resourseTimeout”
РФ3.2.2	По умолчанию значение настройки должно равняться 5 с.
РФ3.2.3	REST API должен отправить ответ в течении количества секунд, указанного в настройке “resourseTimeout”, на категории запросов, прописанных в пункте РФ3.2
РФ3.2.4	Если ответ не был получен в течении количества секунд, указанного в настройке “resourseTimeout”, пользователь должен получить ответ с кодом ошибки 408 и сообщением “Request Timeout”
РФ3.3	Настройка значения времени ожидания ответа от сервера для запросов, связанных с коннектором или аналитическими задачами, должна быть определена в конфигурационном файле. Эти запросы обрабатываются конечными точками вида /api/v1/connector/*, /api/v1/graph/*
РФ3.3.1	Настройка должна называться “analyticsTimeout”
РФ3.3.2	По умолчанию значение настройки должно равняться 15 с.
РФ3.3.3	REST API должен отправить ответ в течении количества секунд, указанного в настройке “analyticsTimeout”, на категории запросов, прописанных в пункте РФ3.3
РФ3.3.4	Если ответ не был получен в течении количества секунд, указанного в настройке “analyticsTimeout”, пользователь должен получить ответ с кодом ошибки 408 и сообщением “Request Timeout”
РФ3.4	Настройки, связанные с базой данных, также должны быть определены в конфигурационном файле в следующем составе: ● dbUser - имя пользователя PostgreSQL ● dbPassword - пароль пользователя PostgreSQL ● dbHost - хост PostgreSQL ● dbPort - порт PostgreSQL ● dbName - название базы данных

№ требования	Описание функциональности
РФ3.5	Если при запуске REST API интерфейса в конфигурационном файле отсутствует какая-либо из описанных в требованиях настроек, запуск должен завершится, и пользователю должна быть возвращена ошибка “The {имя настройки} is not configured”. В случае если отсутствуют несколько настроек, данная ошибка должна быть выведена для каждой из них
РФ4	При отправке запроса на получение проектов из коннектора то, какие проекты необходимо получить, указывается через параметры
РФ4.1	Параметр limit - сколько проектов должно быть получено за один запрос
РФ4.2	Параметр page - порядковый номер страницы, которую необходимо получить (отсчет страниц начинается с 1)
РФ4.3	В случае успешного выполнения запроса должен быть возвращен JSON, который содержит массив проектов, и поле pageInfo, которое содержит: ● currentPage - текущая страница ● projectsCount - количество проектов ● projectsCount - количество страниц
РФ5	Аутентификация должна быть реализована на основе JSON Web Token
РФ6	REST интерфейс должен реализовывать микросервисную архитектуру. Должны быть представлены следующие сервисы: ● сервис аутентификации; ● сервис для работы с ресурсами; ● сервис для запуска аналитических задач и визуализации; ● сервис для работы с коннектором;
РФ7	В REST интерфейсе должно быть реализовано логирование
РФ7.1	Для логирования необходимо использовать библиотеку logrus
РФ7.2	Логи должны храниться в двух файлах: logs.log и err_logs.log
РФ7.4.1	В файле logs.log должны храниться все логи
РФ7.4.2	В файле err_logs.log должны храниться логи уровня WARNING или выше.
РФ7.5	Логи уровня WARNING и выше должны выводиться в консоль
РФ8	Запросы для CRUD операций должны выполняться синхронно, а длительные задачи (запросы, связанные с коннектором, и аналитические запросы) должны выполняться асинхронно

№ требования	Описание функциональности
РФ9	Должна быть реализована функциональность получения данных в формате Gzip

Нефункциональные требования

Таблица 8. Нефункциональные требования к backend модулю

№ требования	Описание функциональности
РНФ1	Документация на REST интерфейс должна быть реализована в виде описания архитектуры и всех конечных точек
РНФ2	Для поддержки горизонтального масштабирования и отказоустойчивости серверной части должна быть доступна возможность управления количествами реплик внутри kubernetes

Требования к web-UI

Функциональные требования

Таблица 9. Функциональные требования к web модулю

№ требования	Описание функциональности
ИФ1	Модуль web-UI должен быть реализован с помощью фреймворка Angular со страницами, написанными на языке TypeScript
ИФ1.1	Модуль должен поддерживать паттерн MVC
ИФ2	Модуль должен иметь возможность конфигурации настроек
ИФ2.1	Параметры конфигурации хранятся в файле конфига в формате YML
ИФ2.2	Конфигурационный файл должен содержать следующую информацию: ⬇ Настройки модуля o host o port
ИФ3	Модуль должен быть после сборки загружен на Tomcat
ИФ4	Меню модуля
ИФ4.1	Проекты

№ требования	Описание функциональности
ИФ4.2	Задачи
ИФ4.2.1	На странице Задачи располагаются все задачи выбранных проектов
ИФ4.3	Сравнение
ИФ4.4	Мои проекты
ИФ5	Страница Проекты должна содержать список всех проектов
ИФ5.1	Пользователь может кнопкой “Добавить” добавлять проект к себе на страницу
ИФ5.2	Должна быть фильтрация минимум по названию проекта
ИФ5.3	Присутствует пагинация по проектам
ИФ6.1	Страница Мои проекты содержит сухую статистику задач по каждому из проектов
ИФ6.1.1	Статистика должна содержать: ● Общее кол-во задач ● Кол-во открытых задач ● Кол-во закрытых задач ● Среднее время выполнение задачи (часы) ● Среднее кол-во заведенных задач в день за последнюю неделю
ИФ6.2	У каждого проекта должна быть возможность выбора аналитических задач
ИФ6.2.1	Аналитические задачи: ● Гистограмма, отражающая время, которое задачи провели в открытом состоянии ● Диаграммы, которые показывают распределение времени по состоянием задач ● График активности по задачам ● График сложности задач ● График, отражающий приоритетность всех задач ● График, отражающий приоритетность закрытых задач
ИФ6.2.2	Кнопка “Обработать” неактивна, если не выбрана ни одна из задач
ИФ7	При нажатии кнопки происходит сравнение выбранных проектов (от 2-х до 3-х)
ИФ7.1	Сравнение происходит как в интернет магазинах
ИФ7.1.1	Если выбрано меньше одного или больше трех проектов, кнопка сравнить не открывает новую страницу

№ требования	Описание функциональности
ИФ7.1.2	Выводится предупреждение
ИФ7.2	Вывод на отдельной странице
ИФ7.2.1	Вывод сводных таблиц для сравниваемых проектов

БАЗА ДАННЫХ

Хранение данных, полученных в ходе работы приложения, предлагается осуществлять в базе данных.

Существует теорема CAP, являющаяся аббревиатурой от сокращения трех свойств: "Consistency" (C) - согласованность, "Availability" (A) - доступность и "Partition Tolerance" (P) - устойчивость к расщеплению. Сама теорема гласит, что ни одна система хранения данными не может удовлетворять одновременно всем трем свойствам. Из теоремы вытекает то, что все базы данных могут быть разделены на 3 типа. Первый тип, это тип, удовлетворяющий свойствам "C" и "A" (CA), второй тип - "A" и "P" (AP), третий тип - "C" и "P" (CP). На рисунке 1 представлено разбиение типов хранилищ на классы в соответствии с теоремой CAP.

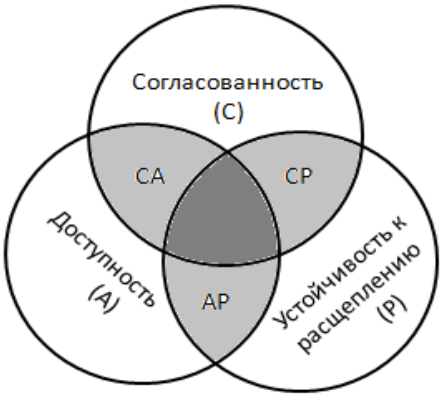

Рисунок 1 - Классификация

Традиционные реляционные базы данных удовлетворяют свойствам CA. Так же они строго следуют принципам ACID (Atomicity, Consistency, Isolation, Durability — атомарность, непротиворечивость, изолированность, долговечность), которые позволяют обеспечить целостность данных. Такие базы данных надежны, легко администрируются и обладают высокой скоростью на чтение данных, ввиду их структурированности. Но в тоже время, они обладают посредственной масштабируемостью на высоких нагрузках. Чаще всего такие базы можно масштабировать только по вертикали наращивая вычислительную мощность машины на которой она запущена или увеличивая объем накопителя.

Если вы полагаете, что вам потребуется хранить большие объемы неструктурированной информации, вам стоит обратится к NoSQL-базам, которые не накладывает ограничений на типы хранимых данных. Такие базы удовлетворяют свойствам AP (доступность и устойчивость к расщеплению) и CP (согласованность и устойчивость к расщеплению) по теореме CAP. Так же такие базы обеспечат высокую скорость разработки проектов структура, которых подвержена частым изменениям. Однако они потребуют от вас:

- реализовать собственный механизм транзакций;
- самостоятельно контролировать целостность данных;

Стоит также упомянуть, что NoSQL-базы не используют язык запросов SQL.

В ходе выполнения курсовой работы обучающемуся предлагается взять СУБД PostgreSQL. Может показаться, что в данном случае могла бы подойти другое решение для управления БД но важным является именно получение навыков практической работы с данным продуктом.

Одним из ключевых требований к базе данных является масштабируемость и консистентность данных после отказа одного из узлов. Масштабирование делится на вертикальное и горизонтальное. Вертикальное подразумевает под собой наращивание мощностей сервера. Основным преимуществом метода является его простота, так как оно требует управления одним крупным сервером. Минус такого подхода в очевидном аппаратном ограничении и высокой стоимости такого решения. Горизонтальное масштабирование означает увеличение производительности за счёт разделения данных на множество серверов. Такой способ предполагает увеличение производительности без снижения отказоустойчивости. Существует два основных типа горизонтального масштабирования.

- шардирование - это принцип проектирования базы данных, при котором части таблиц хранятся раздельно, на разных физических серверах. Большие таблицы в таком случае также разбиваются на таблицы меньших размеров согласно какому-то принципу. Такое решение также даст выигрыш в производительности так как теперь

мы не будем искать по меньшему количеству данных. Шардинг является наиболее приемлемым решением для масштабных проектов, особенно он эффективен в связке с репликацией.

- репликация - это процесс, под которым понимается копирование данных из одного источника на другой (или на множество других) и наоборот. При репликации изменения, сделанные в одной копии объекта, могут быть распространены в другие копии. При использовании такого метода выделяют два типа серверов: master и slave. Мастер используется для записи или изменения информации, слейвы — для копирования информации с мастера и её чтения. Чаще всего используется один мастер и несколько слейвов, так как обычно запросов на чтение больше, чем запросов на изменение. Главное преимущество репликации — большое количество копий данных. Так, если даже головной сервер выходит из строя, любой другой сможет его заменить. Однако как механизм масштабирования репликация не слишком удобна. Причина тому — рассинхронизация и задержки при передаче данных между серверами. Чаще всего репликация используется как средство для обеспечения отказоустойчивости вместе с другими методами масштабирования.

В ходе выполнения работы вам предлагается ознакомится с методом репликации. Итак, PostgreSQL поддерживает несколько видов репликации:

- Потоковая репликация (Streaming Replication) - это репликация, при которой от основного сервера PostgreSQL на реплики передается WAL. И каждая реплика затем по этому журналу изменяет свои данные. Для настройки такой репликации все серверы должны быть одной версии, работать на одной ОС и архитектуре. Потоковая репликация в Postgres бывает двух видов — асинхронная и синхронная.
    - Асинхронная репликация. В этом случае PostgreSQL сначала применит изменения на основном узле и только потом отправит записи из WAL на реплики. Преимущество такого способа — быстрое подтверждение транзакции, т.к. не нужно ждать пока все реплики применят изменения. Недостаток в том, что при падении основного сервера часть данных на репликах может потеряться, так как изменения не успели продублироваться.

    - Синхронная репликация. В этом случае изменения сначала записываются в WAL хотя бы одной реплики и только после этого фиксируются на основном сервере. Преимущество — более надежный способ, при котором сложнее потерять данные. Недостаток — операции выполняются медленнее, потому что прежде чем подтвердить транзакцию, нужно сначала продублировать ее на реплике.
- Логическая репликация (Logical Replication). Логическая репликация оперирует записями в таблицах PostgreSQL. Этим она отличается от потоковой репликации, которая оперирует физическим уровнем данных: биты, байты, и адреса блоков на диске. Возможность настройки логической репликации появилась в PostgreSQL 10. Этот вид репликации построен на механизме публикации/подписки: один сервер публикует изменения, другой подписывается на них. При этом подписываться можно не на все изменения, а выборочно. Например, на основном сервере 50 таблиц: 25 из них могут копироваться на одну реплику, а 25 — на другую. Также есть несколько ограничений, главное из которых — нельзя реплицировать изменения структуры БД. То есть если на основном сервере добавится новая таблица или столбец — эти изменения не попадут в реплики автоматически, их нужно применять отдельно. Также в отличие от потоковой репликации, логическая может работать между разными версиями PostgreSQL, ОС и архитектурами.

Вам предстоит самим сделать выбор между тем или иным видом репликации. Также предстоит предусмотреть систему мониторинга для master версии. В таком случае, если придерживаться рекомендуемой архитектуры, одна из slave версий должна перенять на себя роль “ведущего игрока”. Существует ряд решений которые можно использовать для получения на ведомых компонентах сообщений о падении master версии.

Еще одна проблема с который вы можете столкнуться заключается в необходимости разделения запросов на чтение и на изменение. Самый простой способ этого добиться сразу подстроить архитектуру своего приложения под паттерн CQRS.

CQRS расшифровывается как Command Query Responsibility Segregation (разделение ответственности на команды и запросы). В 1980 Бертран Мейер сформулировал термин CQS. В начале двухтысячных Грег

Янг расширил и популяризовал эту концепцию к CQRS. CQRS предлагает разделять операции чтения и записи на отдельные типы операций Query и Commands.

- Command ориентированы на задачи, а не на данные. ("Забронировать номер в отеле", а не установить для ReservationStatus значение "зарезервировано").
- Command может помещаться в очередь для асинхронной обработки, а не обрабатываться синхронно.
- Query никогда не должен изменять базу данных. Query возвращает DTO, который не инкапсулирует знания предметной области.

Такую архитектуру в упрощенном виде можно представить в виде схемы на рисунке 2.

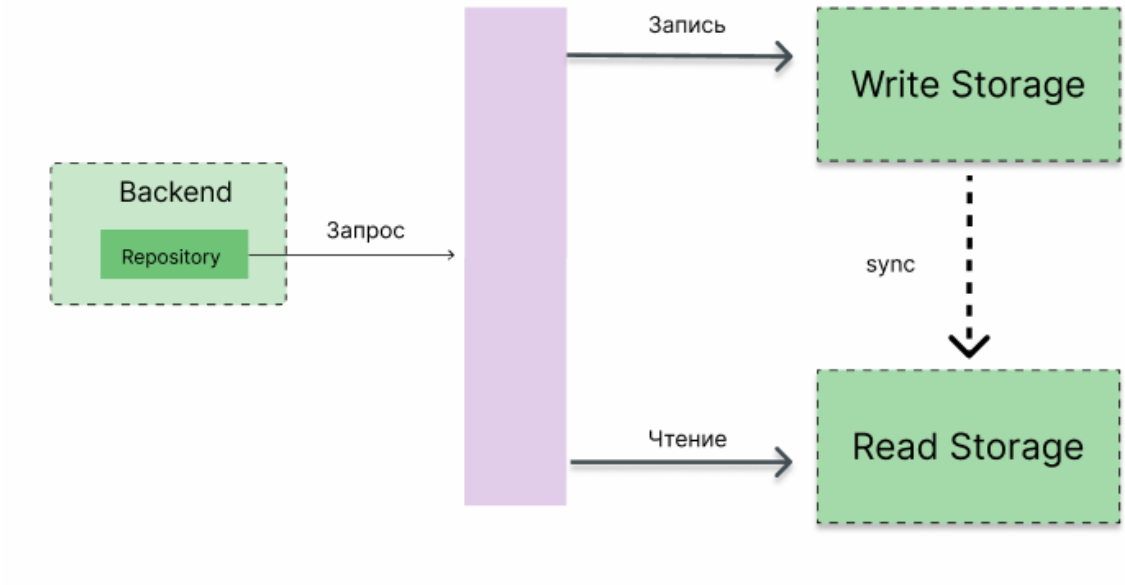

Рисунок 2 - Пример разделения ответственности на команды и запросы

Наличие отдельных моделей запросов и команд упрощает проектирование систем делая их независимыми друг от друга. Однако один из недостатков заключается в том, что код CQRS не может автоматически формироваться из схемы базы данных с помощью ORM или подобных механизмов. Для дополнительной изоляции часто физически разделяют данные для чтения и данные для записи. Как это описано на диаграмме выше. В этом случае в БД для чтения можно оптимизировать ее работу так,

чтобы максимально эффективно выполнять запросы. К примеру использовать *materialized view*, чтобы не обращаться к сложным операциям join'ов или связям. Вы можете использовать в том числе другой тип хранилища данных. Например, база данных для записи останется реляционной, а для чтения вы можете применять NoSQL или наоборот, в зависимости от выбранного вами решения.

КОННЕКТОР JIRA

В рамках курсового проекта исполнителю необходимо реализовать сервис коннектора для получения данных с помощью JIRA REST API. Данный коннектор должен реализовывать ETL-процесс для получения и сохранения информации в локальную базу данных. Также данный сервис должен предоставлять эндпоинты для получения информации о репозитории JIRA. Данный сервис реализуется с помощью языка Golang и стандартных библиотек языка программирования (допускается использование сторонних библиотек, но не для получения данных из JIRA REST API).

Требования к данному модулю описаны в разделе ‘Требования на Jira connector’ документа ‘Технические требования’.

Предлагаемый высокоуровневый дизайн JIRA-коннектора представлен на рисунке 3.

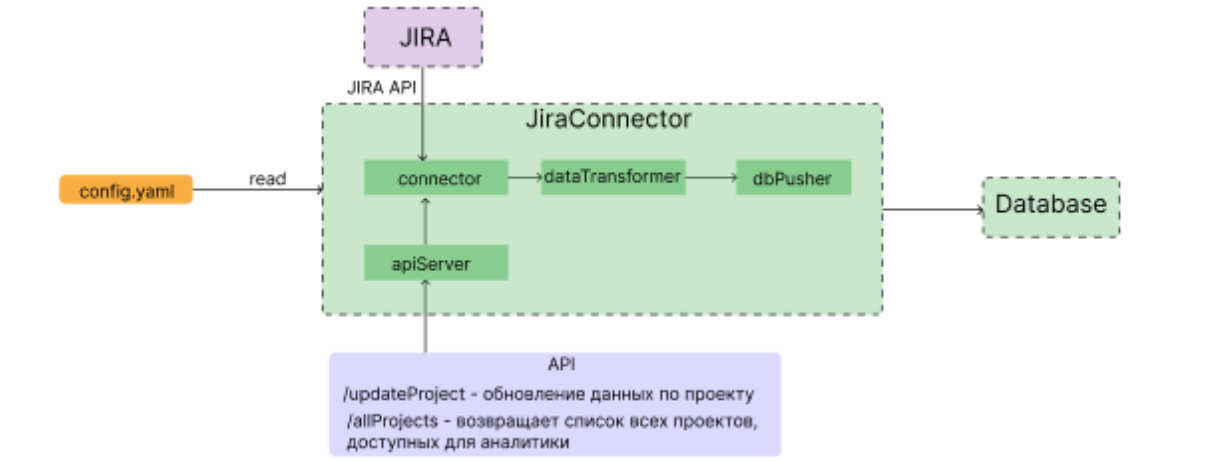

Рисунок 3 - Высокоуровневая архитектура модуля, реализующего модуль коннектора JiraConnector

JIRA - репозиторий JIRA, информацию из которого необходимо получать с помощью JIRA REST API v2

config.yaml - конфиг, в котором указываются параметры подключения к базе данных и параметры работы коннектора

JiraConnector - сам сервис коннектора, реализующий внутри себя connector, который позволяет получить данные из JIRA репозитория с помощью ETL-процесса, а также API сервер, предоставляющий эндпоинты.

ETL-процесс:

- Данные получаются в чистом виде с помощью JIRA REST API
- Полученные данные преобразуются в структуру данных, схожую со структурой БД, лишние данные отбрасываются
- Отформатированные данные загружаются в БД

Процесс загрузки должен быть многопоточным, а пользователь должен иметь возможность настройки этой многопоточности (выбор количества горутин, в которых будет осуществляться загрузка).

В качестве JIRA репозитория рекомендуется брать репозиторий Apache (https://issues.apache.org/jira/), но разрешается любой другой JIRA репозиторий. Выбор репозитория должен осуществляться через файл конфига путем выбора базового url (например “https://issues.apache.org/jira/” или “https://jfrog.atlassian.net/jira”)

Для получения данных из репозиториев необходимо использовать стандартную библиотеку “net/http” и JIRA REST API ver. 2. Полученные данные должны обрабатываться и сохраняться в локальную БД.

Эндпоинты

Сервис должен предоставлять как минимум 2 эндпоинта:

- **/updateProject?project=projectKey**
  Получает (или обновляет) все issues из проекта с ключом 'projectKey' и заносит в базу данных. Что будет происходить - загрузка или обновление - зависит от того, был ли проект сохранен локально ранее.
- **/projects**
  Понадобится для страницы, которая будет отображать все проекты, доступные для загрузки. Из-за того, что эта страница должна поддерживать пагинацию и предоставлять возможность поиска (фильтрации) проектов, необходимы следующие параметры:
  - limit - количество проектов на одной странице (limit > 0)
  - page - номер страницы, с которой необходимо вернуть проекты (page > 0)
  - search - параметр для фильтрации списка проектов. Будут возвращены только те проекты, имя или ключ которых содержат подстроку заданную в этом параметре без учета регистра.

Возвращает JSON, содержащий массив Projects (проекты на странице под номером page) и структуру PageInfo, которая содержит поле PageCount - общее количество страниц при данном параметре limit и search, CurrentPage - номер текущей страницы, ProjectsCount - общее количество проектов при данном параметре search. Значение limit по умолчанию = 20, значение page по умолчанию = 1.

Ограничение JIRA

Важно отметить, что JIRA REST API имеет ограничение на количество загружаемых данных за период времени, поэтому процесс загрузки должен адаптивно подстраиваться под эти ограничения. Проблема заключается в том, что если превысить лимит JIRA на загрузку, то соединение будет временно заблокировано и при очередной попытке запроса данных будет возвращена ошибка.

Решение этой проблемы следующее - необходимо подождать некоторое время и попробовать загрузить данные из этого соединения еще раз. Если соединение все еще будет заблокировано, тогда необходимо увеличить время ожидания в 2 раза и повторить попытку. Необходимо повторять эти действия до тех пор, пока попытка не будет успешной, либо пока очередное время ожидание не превысит максимально допустимое.

Эффективность загрузки

Когда проблема с ограничением JIRA решена - можно начинать думать о том, как увеличить скорость загрузки данных. Так как реализация ограничения JIRA для нас - черный ящик, тогда увеличивать скорость можно изменяя способы или параметры загрузки, отслеживая, как эти изменения повлияют на производительность скачивания. Первый вопрос, который встает перед нами - что лучше использовать - одно соединение для всех горутин или у каждой горутирны свое соединение? Для ответа на этот вопрос сделаем 800 запросов при обоих кейсах и измерим время, за которое эти 800 запросов исполнятся.

Пример кода для кейса с отдельным соединением для каждой горутины приведен в листинге 1.

Листинг 1 - Отдельное соединение для каждой горутины

```
start := time.Now()
config := properties.GetConfig(os.Args[1])
var wg sync.WaitGroup
var m sync.Mutex
init := 0
threadCount := 100
requestCount := 8
after := make([]int, requestCount+1)
for i := 0; i < threadCount; i++ {
   wg.Add(1)
   go func() {
      defer wg.Done()
      m.Lock()
      init++
      m.Unlock()
      JIRAConnector, err :=
connector.GetConnection(config.ProgramSettings.ApacheUrl)
      if err != nil {
      return
      }
      m.Lock()
      after[0]++
      m.Unlock()
      for j := 0; j < requestCount; j++ {
      _, err = JIRAConnector.GetExpandIssuesJSON("AIRAVATA", 0)
      if err != nil {
            return
      }
      m.Lock()
      after[j+1]++
      m.Unlock()
      }
   }()
}
wg.Wait()

fmt.Printf("Init: %d\n", init)
for inx, obj := range after {
   fmt.Printf("After %d: %d\n", inx+1, obj)
}
duration := time.Since(start)
```

```
fmt.Println(duration)
fmt.Printf("Init: %d\n", init)
for inx, obj := range after {
   fmt.Printf("After %d: %d\n", inx+1, obj)
}
```

Пример кода для кейса с одним соединением для всех горутин приведен в листинге 2.

Листинг 2 - Одно соединение для всех горутин

```
start := time.Now()

config := properties.GetConfig(os.Args[1])
var wg sync.WaitGroup
var m sync.Mutex
threadCount := 100
requestCount := 8
JIRAConnector, err :=
connector.GetConnection(config.ProgramSettings.ApacheUrl)
if err != nil {
   return
}
doneRequestCount := 0
for i := 0; i < threadCount; i++ {
   wg.Add(1)
   go func() {
      defer wg.Done()
      for j := 0; j < requestCount; j++ {
      _, err = JIRAConnector.GetExpandIssuesJSON("AIRAVATA", 0)
      if err != nil {
            return
      }
      m.Lock()
      doneRequestCount++
      m.Unlock()
      }
   }()
}
wg.Wait()
fmt.Printf("Done request: %d\n", doneRequestCount)

duration := time.Since(start)
fmt.Println(duration)
```

Результаты представлены в таблице 9.

Таблица 10. Время загрузки с двумя способами установки соединения

Множество соединений	Одно соединение
8 мин. 2 сек.	5 мин. 10 сек.

Тогда, судя по результатам, следует использовать одно соединение для всех горутин.

Теперь, когда мы определились со способом организации соединения - необходимо решить, какое количество горутин будет оптимальным (даст наилучшие показатели скорости загрузки). Для этого, во-первых, необходимо реализовать возможность задавать количество горутин, в которых будет осуществляться загрузка, а во-вторых, провести практический эксперимент, который даст ответ на вопрос - какое количество горутин оптимально для вашей машины. Для эксперимента можно использовать код из предыдущего опыта и просто менять количество горутин и количество запросов в одной горутине. Скорее всего, увеличение количества горутин в какой-то момент перестанет давать прирост в скорости, а даже наоборот, приведет к ухудшению результата. Одной из причин этого является то, что мы просто упремся в потолок ограничения JIRA на количество запросов за период времени, и одновременное количество запросов на множестве горутин будет приводить только к скорейшей блокировке соединения, и, как следствие, к более частому ожиданию отката блокировки.

Теперь у нас остается лишь один способ увеличить производительность загрузки - увеличивать количество информации, загружаемой за один запрос. Для этого в запросе к JIRA необходимо указать параметр maxResults, который обозначает количество issues, которые будут загружены за один запрос (по дефолту его значение = 50). Максимальное значение этого параметра - 1000. Но если мы укажем в этом параметре максимальное значение, то это не гарантирует того, что скорость загрузки вырастет, так как, судя по всему, ограничение в JIRA накладывается не на количество запросов за период времени, а на количество загруженной информации. Тем не менее это не мешает нам поэксперементировать и

определить какое значение данного параметра даст наилучшую скорость загрузки. Для того, чтобы сравнить какое из двух значений (например 500 или 1000) этого параметра лучше - необходимо для каждого из значений выбрать такое количество запросов в горутине, чтобы для обоих случаев объем загруженной информации был равен, так как только в этом случае эксперимент можно назвать справедливым. Таким образом, например, если мы хотим сравнить значения 1000 и 500 этого параметра, то в первом случае нам необходимо произвести N запросов, а во втором - N*2, так как объем информации, полученный за один запрос в первом случае - в 2 раза больше, чем во втором.

Конфигурационный файл

Учитывая требование на настраиваемую многопоточность, на систему адаптивной загрузки, а также на возможность изменения объема информации, полученной за один запрос, конфиг файл для JIRA-коннектора должен содержать как минимум следующую информацию:

- DBSettings - настройки для подключения к БД
- ProgramSettings:
    - jiraUrl - url Jira, откуда будут выкачиваться данные.
    - threadCount - количество горутин, которые будут заниматься загрузкой данных с Jira. Минимальное возможное значение - 1.
    - issueInOneRequest - количество задач, загружаемых за один запрос. Минимальное возможное значение - 50, максимальное - 1000.
    - maxTimeSleep - максимальное время ожидания в миллисекундах перед повторной попыткой запроса на загрузку данных с Jira.
    - minTimeSleep - начальное время ожидания в миллисекундах перед повторной попыткой запроса на загрузку данных с Jira.

СЕРВЕРНАЯ ЧАСТЬ

В рамках курсового проекта исполнителю необходимо создать серверную часть в виде REST API, которая будет получать данные из коннектора JIRA и базы данных PostgreSQL и “передавать” их в пользовательский интерфейс. REST API должен быть реализован с применением языка Golang.

Требования к серверной части приведены в подразделе ‘Требования на REST интерфейс’ раздела ‘Технические требования’.

Архитектура серверной части

На рисунке 4 представлена рекомендуемая схема реализации серверной части. Серверная часть получает запросы от UI и формирует ответ для него на основе полученных из база данных и коннектора.

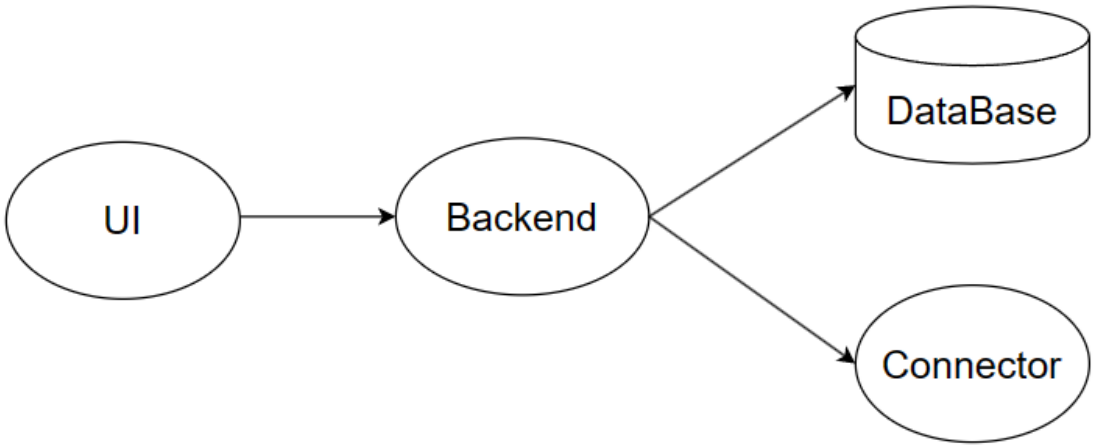

Рисунок 4 - Модульная схема реализации серверной части приложения

Реализуемая серверная часть должна содержать конечные точки для работы с основными ресурсами (добавление их в базу данных и получения из базы):

- issues
- histories
- projects

Также должны быть как минимум реализованы следующие конечные точки, которые отправляют запросы к коннектору JIRA:

- получение списка проектов. Как и в REST интерфейсе коннектора для данной конечной точки должны поддерживаться параметры ограничения количества выводимых проектов в одном запросе и указания текущего номера страницы;
- обновления проекта в базе данных.

Для получения графиков и выполнения аналитических задач необходимо также реализовать конечные точки. Такая конечная точка, которая позволяет получить данные по группам графиков (группы определены согласно техническому заданию), может иметь вид */api/v1/graph/{group:[0-9]}?project={ProjectName}*, где group - номер группы, а ProjectName - название проекта.

Исполнителю также предлагается создать конечную точку, которая будет отображать все доступные конечные точки. Предлагаемый вариант именования - */api/v1/{имя модуля}/services*.

Richardson Maturity Model

Важным требованием к модулю «Resource» является удовлетворение 3 уровню из Richardson Maturity Model[1].

Уровень 3 Richardson Maturity Model характеризуется наличием нескольких URI, каждый из которых поддерживает разные HTTP методы (GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT).

С вводом разных HTTP методов также добавляется необходимость возвращения правильных HTTP статус кодов. Например, на запрос создания ресурса должен быть возвращен код 201 (если ресурс был создан).

Кроме того, на уровне 3 вводится понятие HATEOAS, которое заключается в наличии ссылок на конечные точки сервиса в ответе на запрос, что обеспечивает динамический доступ к сервису. Важным преимуществом такого подхода является лучшее понимание структуры серверной части разработчиками на начальных этапах (ссылки дают

[1] MartinFowler. [Электронный ресурс]. Режим доступа: // https://martinfowler.com/articles/richardsonMaturityModel.html

разработчикам подсказки относительно того, какие конечные точки могут быть вызваны дальше). На рисунке 5 представлено графическое представление уровня 3 из Richardson Maturity Model.

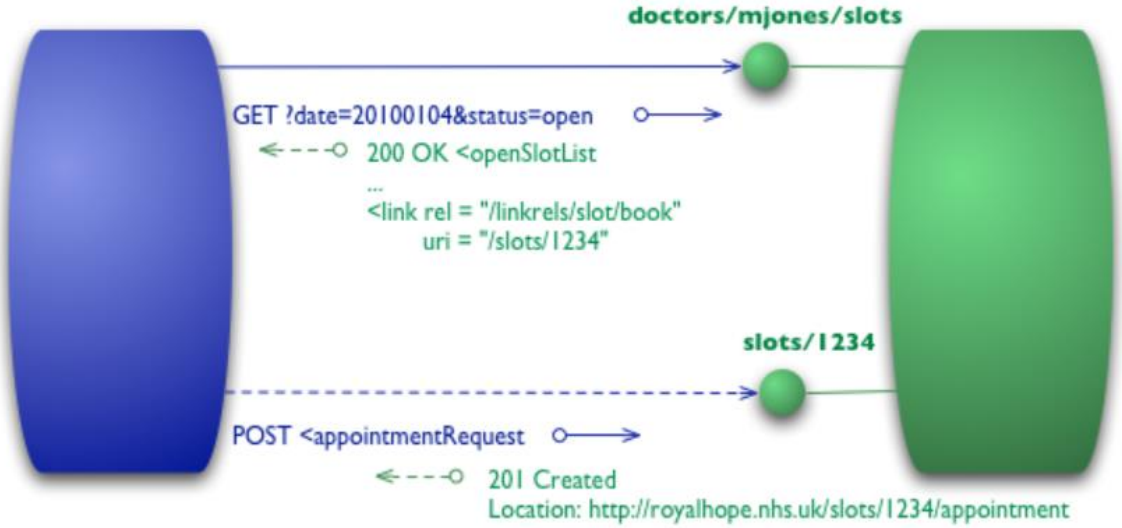

Рисунок 5 - Графическое представление 3 уровня проектирования REST на основе RMM

Пример запроса

В качестве примера запроса рассмотрим GET-запрос получения задачи из базы данных по идентификатору. Этот запрос доступен по адресу *http://localhost:8000/api/v1/resource/issues/{id}*, где id - идентификатор задачи. Так запрос *http://localhost:8000/api/v1/issues/12616737* вернет ответ, приведенный в листинге 3, с кодом состояния 200.

Листинг 3 - Ответ сервера на GET-запрос получения задачи из БД по идентификатору

```
{
  "_links": {
    "issues": {
      "href": "http://localhost:8000/api/v1/issues"
    },
    "projects": {
      "href": "http://localhost:8000/api/v1/projects"
    },
    "histories": {
      "href": "http://localhost:8000/api/v1/histories"
    },
    "self": {
      "href": "http://localhost:8000/api/v1/issues/12616737"
```

```
    }
  },
  "data": {
    "Id": 12616737,
    "Project": {
      "Id": 12310505,
      "Key": "",
      "Name": "Abdera",
      "Url": ""
    },
    "Key": "ABDERA-311",
    "CreatedTime": "2012-11-19T14:11:12Z",
    "ClosedTime": "0001-01-01T00:00:00Z",
    "UpdatedTime": "2012-11-21T23:51:43Z",
    "Summary": "fails to build with java 7",
    "Description": "",
    "Type": "Bug",
    "Priority": "Major",
    "Status": "Resolved",
    "Creator": "gil cattaneo",
    "Assignee": "Christine Koppelt",
    "TimeSpent": 0
  },
  "message": "success",
  "name": "Jira Analyzer REST API Get Issue",
  "status": true
}
```


Как показано выше, ответ содержит поле _links, которое представляет собой набор ссылок на конечные точки сервиса (HATEOAS) в ответе на запрос. Каждый элемент, содержащийся в _links, имеет следующую структуру:

- название ресурса (например, self, issues, histories, projects и т. д.);
- ссылка на конечную точку.

При использовании REST API клиенты или разработчики смогут динамически перемещаться по системе, выбрав одну из ссылок из доступного массива ссылок.

Документирование API

Для удобства разработки исполнитель может использовать различные подходы к документированию серверной части. Одним из таких

подходов является применение инструментов, которые позволяют автоматически описывать API на основе его кода. Одним из наиболее распространенных инструментов является Swagger UI[2]. На рисунке 6 приведен демо пример[3] из документации Swagger UI.

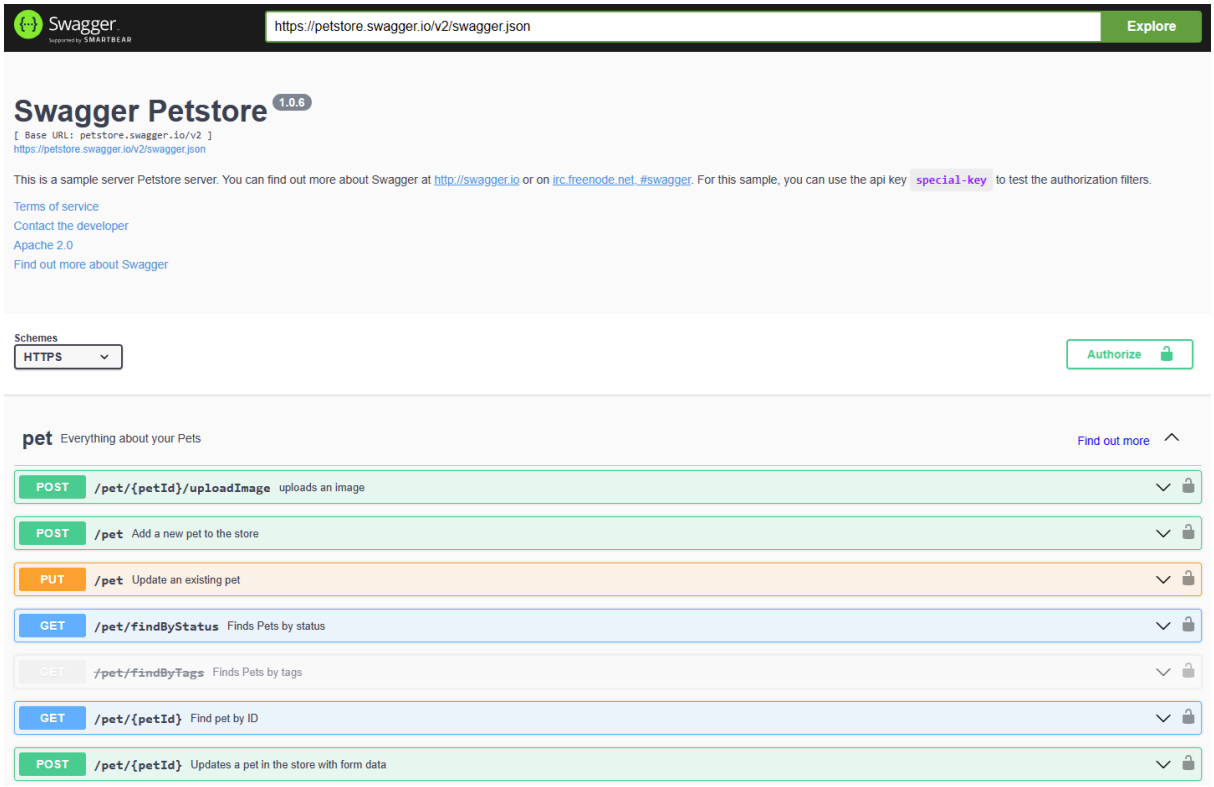

Рисунок 6 - Демонстрационный пример работы с Swagger UI

Другим популярным решением является создание совместного рабочего пространства в инструменте Postman, которое содержит набор запросов к REST API, позволяет выполнять их и получать примеры возвращаемых ответов, заголовков и кодов состояния запроса.

[2] Swagger UI. [Электронный ресурс]. Режим доступа: // https://swagger.io/tools/swagger-ui/

[3] Swagger UI Demo. [Электронный ресурс]. Режим доступа: // https://petstore.swagger.io/?_ga=2.123483843.97597098.1676919673-15568585.1676919673#/

Конфигурационный файл

Все переменные, необходимые для работы сервисной части, должны храниться в конфигурационном файле, который должен в соответствии с требованиями содержать как минимум следующую информацию:

- DBSettings - настройки для подключения к БД
- ProgramSettings:
    - bindAddress - адрес, на котором будет работать серверная часть;
    - bindPort - порт, на котором будет работать серверная часть;
    - resourseTimeout- время ожидания ответа от сервера для запросов, связанных с ресурсами, которые могут быть получены с помощью CRUD операций (модуль «Resource»);
    - analyticsTimeout - время ожидания ответа от сервера для запросов, связанных с коннектором и аналитическими задачами (модуль «APIConnector» и модуль «Analytics»).

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

В рамках курсового проекта исполнителю необходимо создать web-UI (user interface), пример дизайна клиентского приложения приведен в Приложении 1. Необходимо продумать UX для удобства интерфейса при использовании. Данный микросервис призван решать проблему user-friendly отображения информации, взятой JIRA-connector'ом из репозиториев JIRA.

При реализации данного микросервиса необходимо придерживаться паттерна MVC. MVC - схема разделения данных приложения и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо друг от друга.

Минимальные требования к данному модулю описаны в 'Технических требованиях'.

Предлагаемый высокоуровневый дизайн web-UI представлен на рисунке 7.

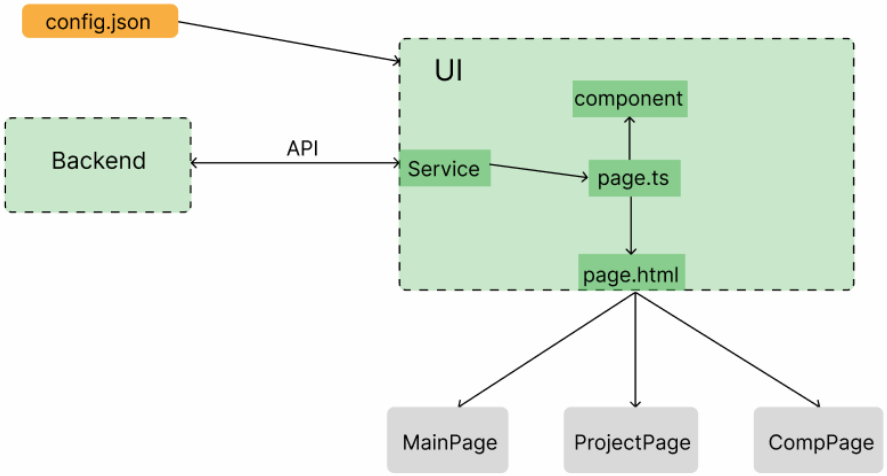

Рисунок 7 - Высокоуровневая архитектура модуля, реализующего пользовательский Web-интерфейс

Backend - серверная часть приложения;

UI - клиентская часть приложения;

config.json - файл конфигурации, в котором указывается host:port для получения данных с REST-API.

Архитектура приложения

Вся видимая часть Angular реализуется с помощью компонентов. Компоненты - часть интерфейса приложения с собственной логикой. Пример архитектуры Angular-приложения приведен на рисунке 8.

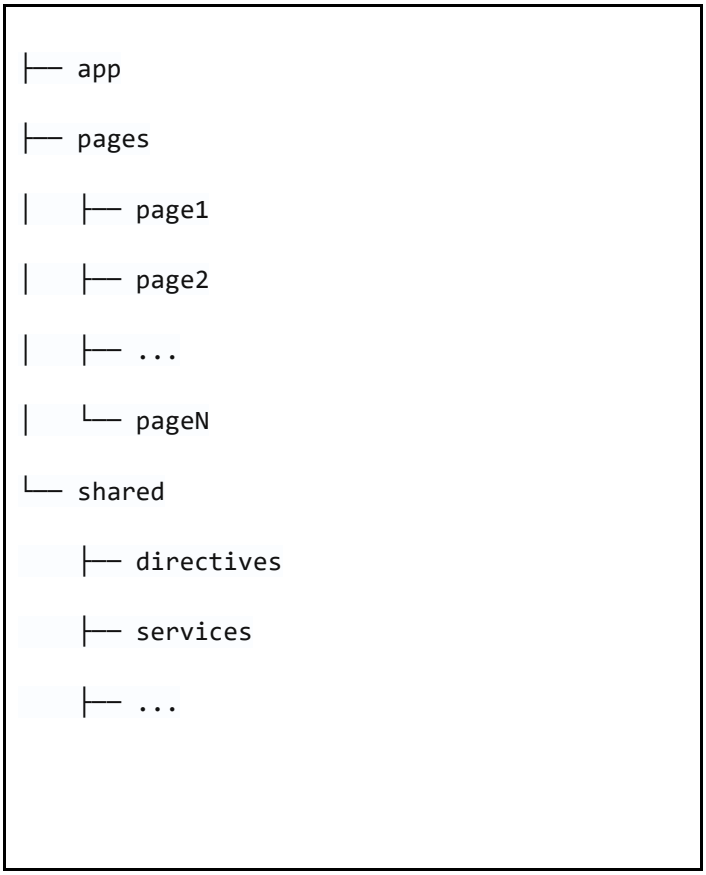

Рисунок 8 - Архитектура Angular-приложения

Pages - страницы приложения, на которых, собственно, и отображаются компоненты. Shared - сущности общего использования. Таковыми являются, например, сервисы, которые предоставляют данные компонентам, или же директивы, которые преобразовывают DOM элементы.

Для создания компонента в терминале можно прописать следующую команду:

↔→ → ▯ ↥↔⇨↘⇨↧

↘→ ↴ →⇨↔⇨△⇨◀⇨⇤

Для создания сервиса в терминале можно прописать следующую команду:

↔→ → ▽ ↥↔⇨↘⇨↧

Примеры запросов к REST-API

Отправка HTTP-запросов серверу реализовывается в сервисах. Для получения данных с сервера отправляем GET-запрос, пример которого реализован в листинге 4.

Листинг 4 - Запросы к REST-API

```
getAll(page: number, searchName: String) {
 return this.http
   .get<{ data: IRequest, page: Page, success: boolean }>(
     environment.API_URL.concat("/api/v1/connector")
       .concat("/projects?")
       .concat("limit=10&page=")
       .concat(page)
       .concat("&search=")
       .concat(searchName)
   ).pipe(
     map((responseData) => responseData.data)
   );
}
```

Пример использования сервиса в компонентах

Для получения данных с сервиса необходимо “подписаться” на соответствующий метод их получения. Пример “подписки” продемонстрирован в листинге 5.

Листинг 5 - Запросы к REST-API

```
constructor(private projectService: ProjectService){}

ngOnInit(): void {
 this.projectService.getAll(this.start_page, this.searchName).subscribe({
   next: projects => {
     this.projects = projects.data
     this.pageInfo = projects.pageInfo
   },
   error: error => {
     console.log(error);
```

```
      alert("При получении проектов возникла
ошибка!\n\n".concat(error["error"]["message"]));
    }
  })
}
```

Конфигурационный файл

Все переменные, необходимые для работы клиентской части приложения, должны храниться в конфигурационном файле. Данный файл должен, в соответствии с требованиями, содержать, как минимум, следующую информацию:

- host - адрес сервера, на который будут отправляться запросы для получения информации;
- port - порт сервера.

Для упрощения задачи, можно использовать файл *./src/environments/environment.ts*, который генерирует *Angular CLI* при создании нового проекта. Информацию он должен содержать ту же, что и обычный файл config.json

УЧЕБНАЯ ЛИТЕРАТУРА

1. Макгаврен Д. Head First. Изучаем Go. — СПб.: Изд-во Питер, 2020. — 544 с.
2. Kozyra N., Mastering Concurrency in Go, Packt Publishing, 2014 — 328 с.
3. Fain Y., Moiseev A., Angular Development with TypeScript. — 2-е изд. — Manning, 2018. — 560 с.
4. Poulton N., The Kubernetes Book, 2017 —191.
5. Shaun M. Thomas, PostgreSQL High Availability Cookbook. — 2-е изд. — Packt Publishing, 2017. — 536 с.
6. Subramanian H., Raj P., Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs, Packt Publishing, 2019 — 378 с.
7. Vanderkam D., Effective TypeScript: 62 Specific Ways to Improve Your TypeScript. — 1-е изд. — O'Reilly Media, 2019. — 264 с.
8. Youngman N., Peppe R., Get Programming with Go. — 1-е изд. — Manning, 2018. — 360 с.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

ПРИМЕР ДИЗАЙНА КЛИЕНТСКОГО ПРИЛОЖЕНИЯ

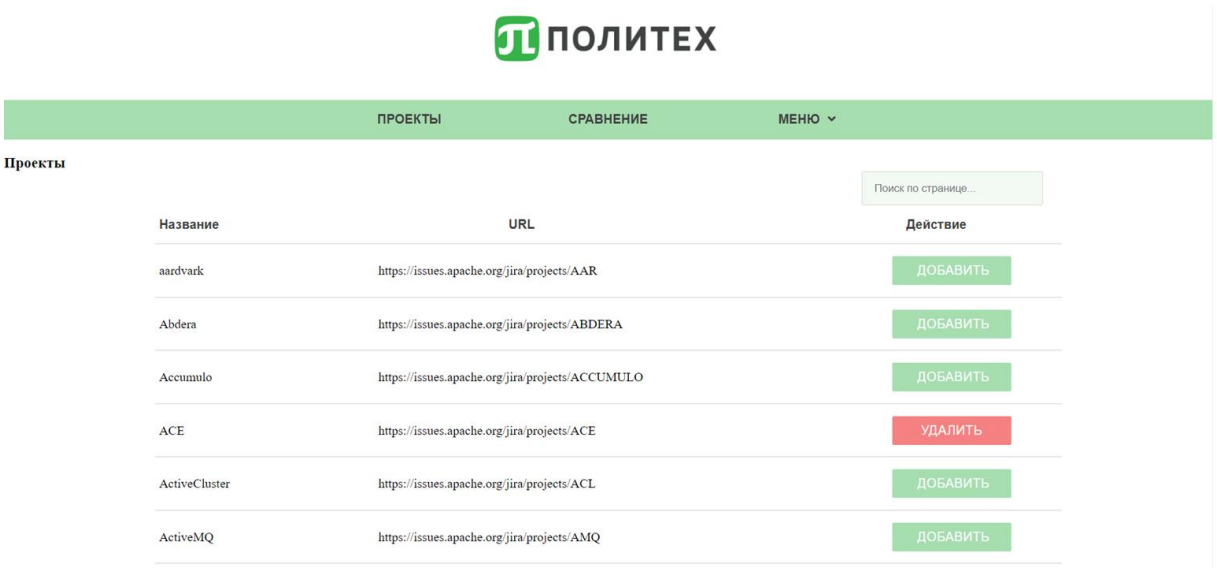

Рисунок 9 - Страница со всеми проектами JIRA

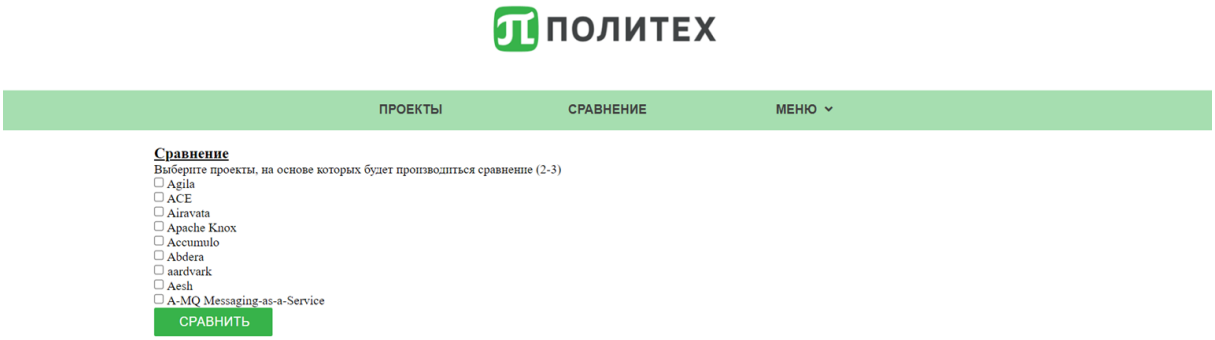

Рисунок 10 - Страница с выбором проектов для сравнения

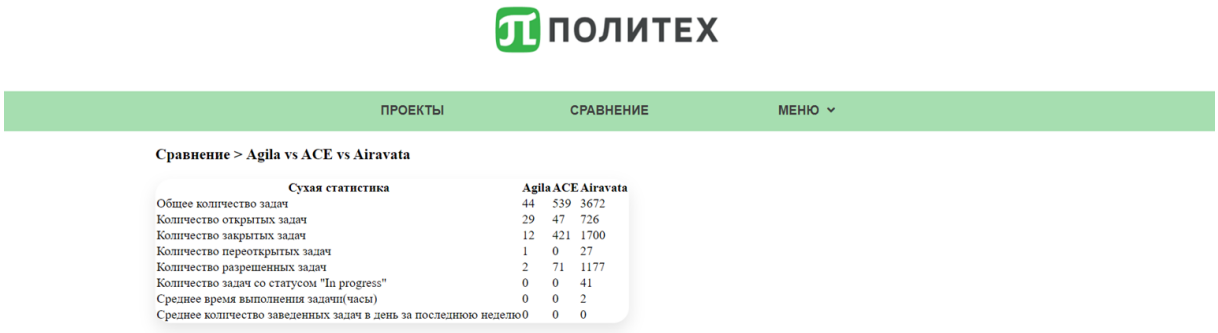

Рисунок 11 - Страница с сравнением выбранных проектов

Диаграмма, демонстрирующая распределение времени по состоянию "Resolve"

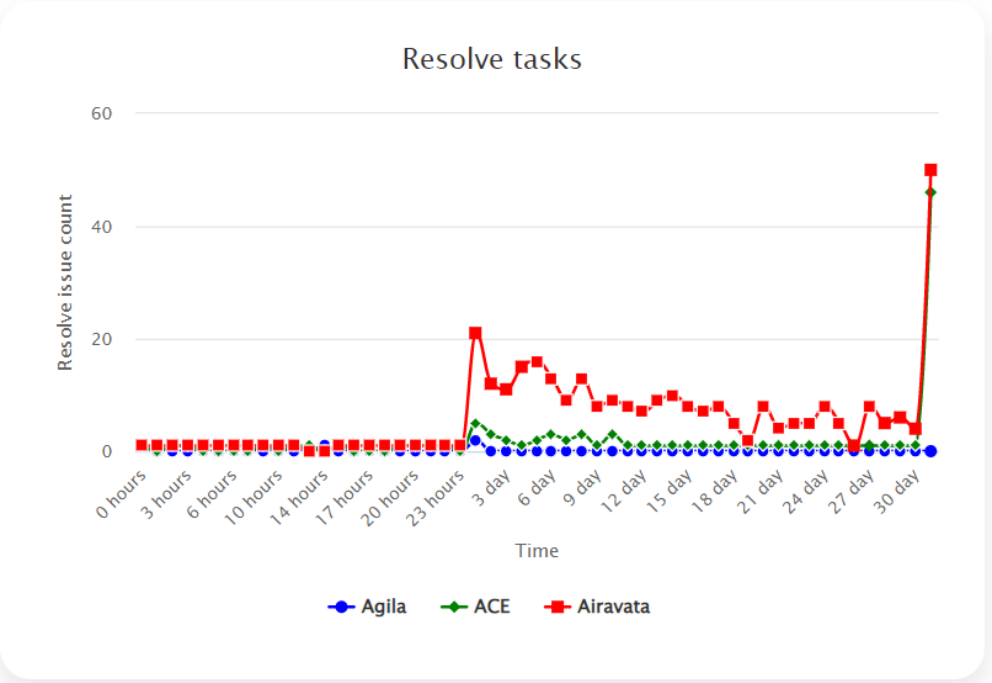

Рисунок 12 - Пример диаграммы, демонстрирующей распределение времени по состоянию “Решенные”

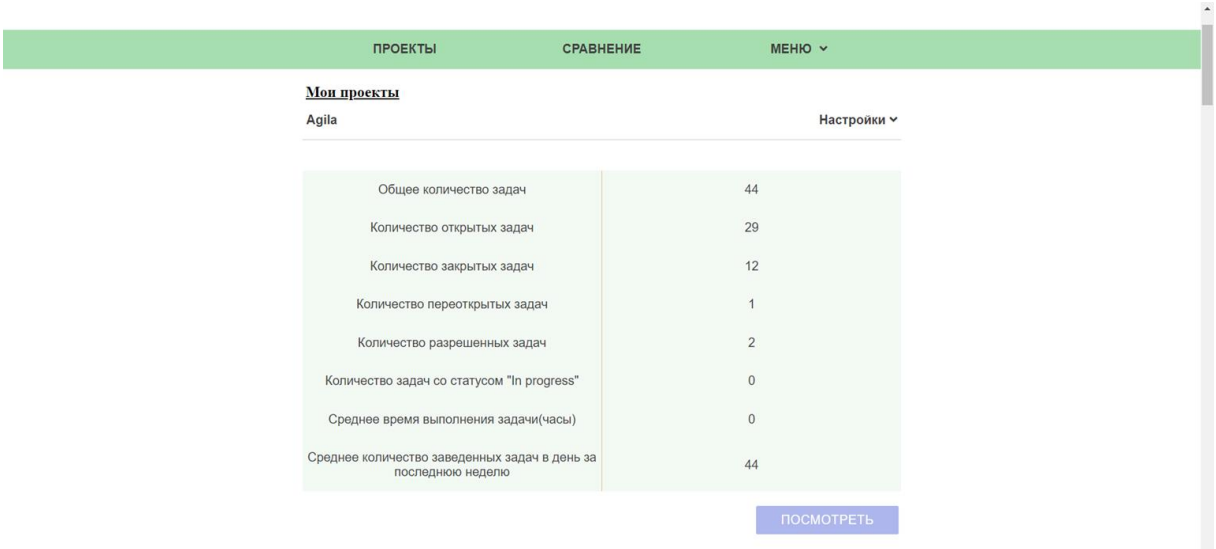

Рисунок 13 - Страница с добавленными проектами в Базу Данных