**ЛАБОРАТОРНАЯ РАБОТА №1**

**ИЗУЧЕНИЕ ОСНОВНЫХ ШАБЛОНОВ ПРОЕКТИРОВАНИЯ МИКРОСЕРВИСОВ.**

**Цель:** Изучить основы разработки API Gateway на Node.js с использованием Express.js; получить практический опыт реализации маршрутизации, аутентификации, авторизации и ограничения скорости.Понять, как API Gateway упрощает взаимодействие с микросервисами и обеспечивает безопасность.

**Теоретическая информация.**

Кратко рассмотрим основные шаблоны **взаимодействия между сервисами**:

* API Gateway: *Централизованная точка входа для клиентских приложений*, которая маршрутизирует запросы к соответствующим микросервисам, обеспечивает аутентификацию, авторизацию, ограничение скорости и другие функции.
* Backend for Frontend (BFF): *Создание отдельных API для разных типов клиентских приложений* (например, веб, мобильные), чтобы адаптировать данные и API под нужды конкретного клиента.
* Asynchronous Messaging (*Асинхронный обмен сообщениями*): Использование очереди сообщений (например, RabbitMQ, Kafka) для асинхронного взаимодействия между сервисами, что позволяет им работать независимо и повышает отказоустойчивость.
* Choreography (*Хореография*): Каждый сервис самостоятельно решает, когда и как взаимодействовать с другими сервисами, на основе событий.
* Orchestration (*Оркестрация*): Централизованный сервис (оркестратор) управляет взаимодействием между другими сервисами, определяя последовательность шагов.
* Service Discovery (*Обнаружение сервисов*): Механизм, позволяющий сервисам динамически находить друг друга в распределенной среде.

Наиболее очевидный способ обращения к микросервисам — прямое обращение от клиента к сервису. И его вполне можно применять в небольших проектах. Однако в приложениях корпоративного масштаба с большим числом микросервисов рекомендуется использовать шаблон API Gateway (рисунок 1).

Этот паттерн основан на применении шлюза, который находится между клиентским приложением и микросервисами, обеспечивая единую точку входа для клиента.

В зависимости от конкретной цели использования паттерна иногда выделяют следующие его разновидности:

* Gateway Routing. Шлюз используется как обратный Proxy, перенаправляющий запросы клиента на соответствующий сервис.
* Gateway Aggregation. Шлюз используется для разветвления клиентского запроса на несколько микросервисов и возвращения агрегированных ответов клиенту.
* Gateway Offloading. Шлюз решает сквозные задачи, которые являются общими для сервисов: аутентификация, авторизация, SSL, ведение журналов и так далее.

Применение паттерна сокращает число вызовов, обеспечивает независимость клиента от протоколов, используемых в сервисах: REST, AMQP, gRPC и так далее, обеспечивает централизованное управление сквозной функциональностью. Однако шлюз может стать единой точкой отказа, требует тщательного мониторинга и при отсутствии масштабирования бывает узким местом системы.

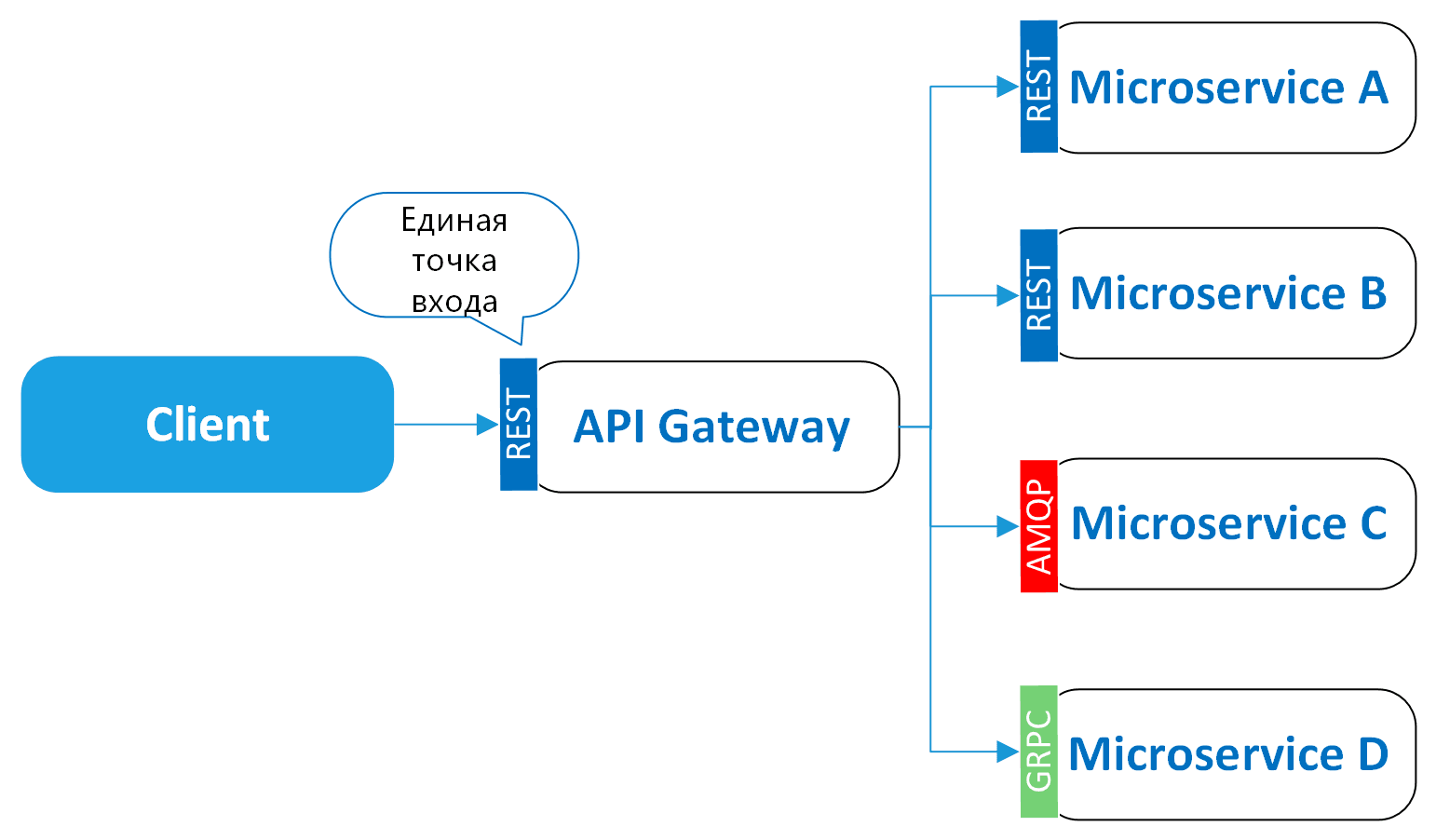


Рисунок 1 - Паттерн API Gateway

**Шаблон «Бэкенды для фронтендов» (Backends for Frontends, BFF)** является вариантом реализации шаблона API Gateway. Он также обеспечивает дополнительный уровень между микросервисами и клиентами, но вместо одной точки входа вводит несколько шлюзов для каждого типа клиента: Web, Mobile, Desktop и так далее (рисунок 2).

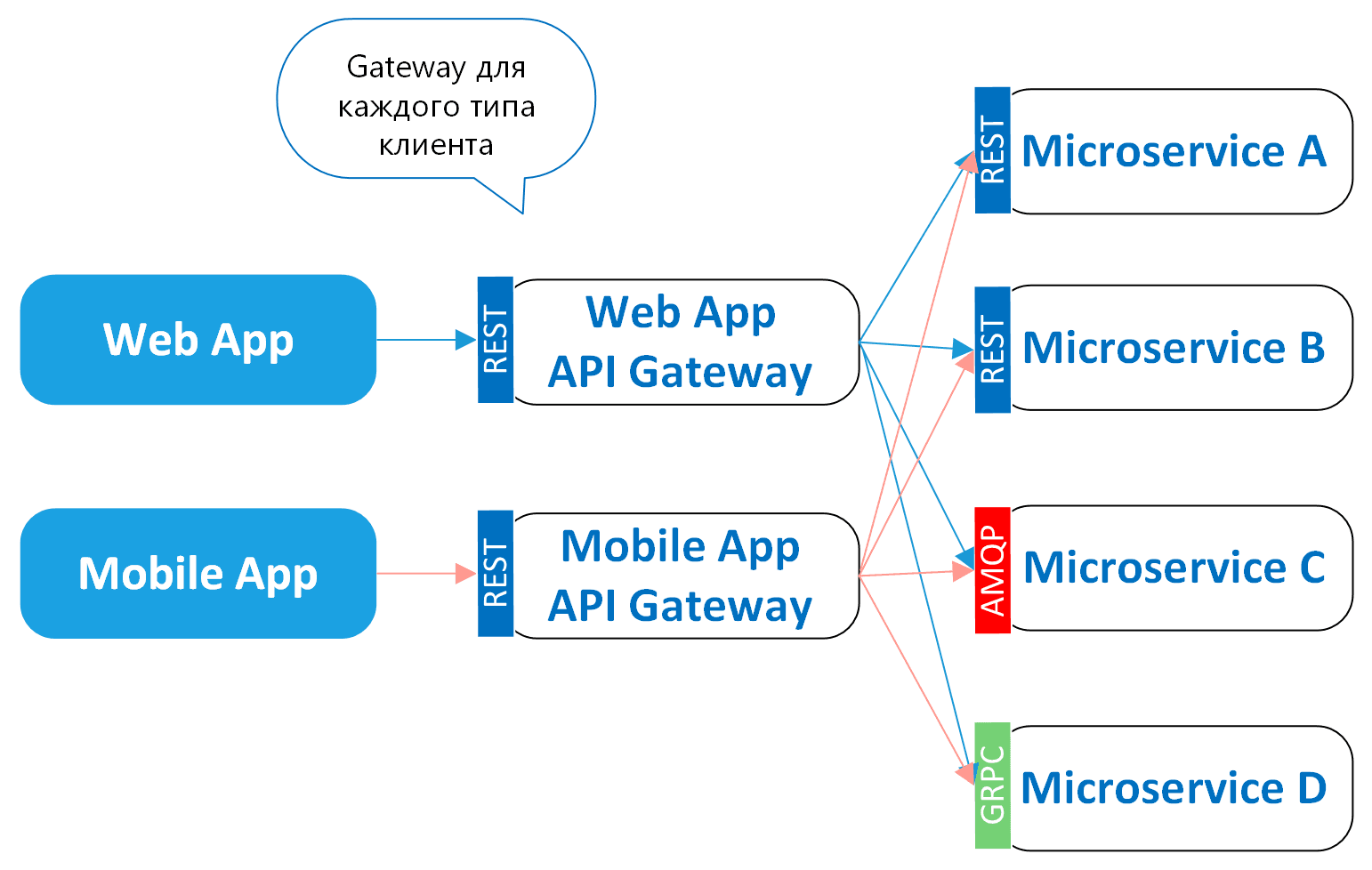


Рисунок 2 - **Шаблон «Бэкенды для фронтендов» (Backends for Frontends, BFF)**

С помощью паттерна можно добавить API, адаптированные к потребностям каждого клиента, избавившись от хранения большого количества ненужных настроек в одном месте. Но шаблон не стоит применять в тех случаях, когда разница в требованиях к API у разных типов клиентов незначительна либо приложение само по себе небольшое: это приведет лишь к дублированию кода и увеличению числа компонентов.

**API Gateway: Подробное описание шаблона**

API Gateway (или шлюз API) — это ключевой архитектурный шаблон, используемый в микросервисных архитектурах для управления и контроля доступа к внутренним сервисам. Он служит единой точкой входа для клиентских приложений, скрывая сложность внутренней структуры микросервисов и предоставляя ряд важных преимуществ.

В микросервисной архитектуре клиентским приложениям может потребоваться взаимодействие с множеством различных сервисов для выполнения одной задачи. Без API Gateway клиентам необходимо:

* Знать о существовании каждого сервиса и его местоположении.
* Обрабатывать аутентификацию и авторизацию для каждого сервиса отдельно.
* Преобразовывать запросы и ответы в разные форматы, если сервисы используют разные протоколы.
* Справляться с сетевыми проблемами, такими как задержки и сбои.

Это усложняет клиентский код, увеличивает время разработки и усложняет поддержку системы. Кроме того, это делает систему более уязвимой для атак.

API Gateway действует как "лицо" микросервисной архитектуры. Он находится между клиентскими приложениями и внутренними сервисами, выполняя следующие функции:

* Маршрутизация (Routing): Направляет входящие запросы к соответствующим внутренним сервисам на основе URL-пути, заголовков, содержания запроса или других правил.
* Аутентификация (Authentication): Проверяет подлинность клиента (например, с использованием логина и пароля, JWT, OAuth).
* Авторизация (Authorization): Определяет, имеет ли клиент право доступа к запрашиваемым ресурсам. Может быть основана на ролях, правах доступа или других атрибутах.
* Композиция (Composition) и Агрегирование (Aggregation): Объединяет данные из нескольких внутренних сервисов в один ответ для клиента, уменьшая количество запросов и упрощая клиентский код.
* Преобразование запросов и ответов (Request/Response Transformation): Преобразует запросы клиентов в формат, понятный внутренним сервисам, и преобразует ответы внутренних сервисов в формат, удобный для клиентов. Это позволяет сервисам использовать разные протоколы и форматы данных.
* Ограничение скорости (Rate Limiting): Контролирует количество запросов, которое может отправлять клиент за определенный период времени, для защиты от перегрузок и злоупотреблений.
* Кеширование (Caching): Кэширует ответы от внутренних сервисов, чтобы уменьшить нагрузку и ускорить время отклика.
* Мониторинг и логирование (Monitoring and Logging): Собирает информацию о запросах, ответах и ошибках для мониторинга производительности системы и выявления проблем.
* Защита (Security): Обеспечивает защиту от распространенных веб-атак (например, SQL-инъекций, XSS).
* Управление версиями API (API Versioning): Позволяет поддерживать несколько версий API одновременно, чтобы обеспечить совместимость с существующими клиентами при внесении изменений в API.
* Cross-Origin Resource Sharing (CORS): Обработка CORS для веб-приложений, обращающихся к API из другого домена.

**Преимущества использования API Gateway**:

* Упрощение клиентского кода: Клиенты взаимодействуют только с одним API Gateway, а не с множеством сервисов.
* Централизация управления: Позволяет централизованно управлять аутентификацией, авторизацией, ограничением скорости, мониторингом и другими политиками.
* Улучшение безопасности: Обеспечивает защиту от распространенных веб-атак и позволяет реализовать единую политику безопасности для всех сервисов.
* Гибкость и масштабируемость: Позволяет добавлять, удалять или изменять микросервисы без влияния на клиентские приложения.
* Оптимизация производительности: Может кэшировать ответы, агрегировать данные и преобразовывать запросы/ответы для оптимизации производительности системы.
* Разделение ответственности: Разделяет ответственность между API Gateway и внутренними сервисами, упрощая разработку и поддержку.

Существует **несколько способов реализации API Gateway**:

* Специализированные API Gateway: Коммерческие и opensource продукты, такие как Kong, Tyk, Apigee, Amazon API Gateway, Azure API Management, Mulesoft Anypoint Platform. Они предоставляют готовые функции для маршрутизации, аутентификации, авторизации, ограничения скорости, мониторинга и т.д.
* Фреймворки: Использование фреймворков для создания API Gateway, таких как Spring Cloud Gateway, Zuul (Java), Express.js (Node.js), Ocelot (.NET). Этот подход дает больше гибкости, но требует больше усилий по разработке и поддержке.
* Обратные прокси: Использование обратных прокси, таких как Nginx, HAProxy, Traefik. Этот подход подходит для простых сценариев маршрутизации, но требует дополнительных настроек для реализации более сложных функций.
* Serverless API Gateway: Использование Serverless-платформ (например, AWS API Gateway, Azure API Management, Google Cloud Endpoints) для создания API Gateway без необходимости управления серверами.

**Когда стоит использовать API Gateway**:

* В микросервисной архитектуре, где есть множество сервисов, которыми нужно управлять и предоставлять доступ к ним.
* Когда требуется централизованное управление аутентификацией, авторизацией, ограничением скорости и другими политиками безопасности.
* Когда нужно оптимизировать производительность системы за счет кэширования, агрегирования данных и преобразования запросов/ответов.
* Когда нужно обеспечить гибкость и масштабируемость системы, позволяя добавлять, удалять или изменять микросервисы без влияния на клиентские приложения.

**Когда API Gateway может быть избыточным:**

* В небольших системах с небольшим количеством сервисов, где клиенты могут напрямую взаимодействовать с сервисами без значительных проблем.
* Когда нет особых требований к безопасности, управлению трафиком или оптимизации производительности.

API Gateway — это мощный и гибкий шаблон, который может значительно упростить разработку, развертывание и управление микросервисными приложениями. Он обеспечивает централизованную точку входа, улучшает безопасность, оптимизирует производительность и обеспечивает гибкость и масштабируемость системы.

**Основные команды для работы с Git.**

**Клонирование репозитория:**

git clone <URL>: Копирует удаленный репозиторий по указанному URL к себе на компьютер.

**Работа с удаленным репозиторием:**

git push origin <имя\_ветки>: Отправляет локальные изменения в удаленный репозиторий в указанную ветку. origin - обычно имя удаленного репозитория.

git pull origin <имя\_ветки>: Получает последние изменения из удаленного репозитория и объединяет их с вашей локальной веткой.

**Работа с ветками:**

git branch <имя\_ветки>: Создает новую локальную ветку.

git checkout <имя\_ветки>: Переключается на указанную ветку.

git checkout -b <имя\_ветки>: Создает новую ветку и сразу переключается на нее (объединяет git branch и git checkout).

git branch -d <имя\_ветки>: Удаляет локальную ветку.

**Создание коммита:**

git add <имя\_файла>: Добавляет указанный файл в область подготовленных изменений (staging area). git add . добавит все измененные файлы.

git commit -m "Комментарий к коммиту": Создает коммит с подготовленными изменениями и добавляет комментарий. Комментарий должен кратко описывать изменения, внесенные в коммит.

git status: Показывает состояние репозитория, включая измененные, подготовленные и неотслеживаемые файлы.

**Базовый Workflow:**

git pull origin main (или master) - Получить последние изменения из основной ветки.

git checkout -b feature/new-feature - Создать и переключиться на новую ветку для новой фичи.

git add –a - Подготовить все изменения к коммиту.

git commit -m "Added new feature" - Закоммитить изменения с комментарием.

git push origin feature/new-feature - Отправить изменения в удаленный репозиторий (в вашу ветку).

**ШАГ 1: Создание нового репозитория на GitHub**

* Войдите в свой аккаунт на <https://github.com>.
* Нажмите на кнопку **"New"** (или **"Создать"**), чтобы создать новый репозиторий (рисунок 1).

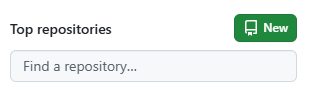


Рисунок 1 – Создание нового репозитория.

* Введите название вашего репозитория (LAB1\_API\_GATEWAY).
* Сделайте репозиторий приватным (рисунок 2).

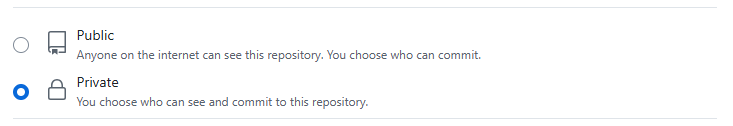


Рисунок 2 – Выбор типа репозитория (публичный или приватный)

* Выберите опцию **"Initialize this repository with a README"** (или **"Инициализировать этот репозиторий с README"**) (рисунок 3).

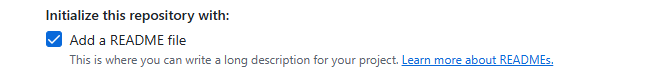


Рисунок 3 – Добавление файла readme

* Выберите опцию добавить файл .gitignore. Шаблон Node (рисунок 4).

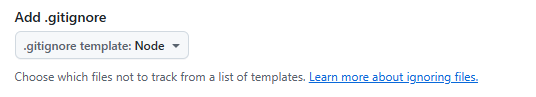


Рисунок 4 – Добавление в проект файла .gitignore

Файл gitignore в Node.js проекте нужен для того, чтобы указать, какие файлы и папки не нужно отслеживать и включать в репозиторий Git.

Это помогает:

* **Избежать коммита лишних файлов**: Например, файлов с конфиденциальной информацией (ключи API), временных файлов, сгенерированных файлов сборки (dist, build), файлов node\_modules (зависимости) и т.д.
* **Оптимизировать размер репозитория**: Исключение больших файлов или папок, которые не нужны для работы проекта, уменьшает размер репозитория и ускоряет клонирование.
* **Улучшить безопасность**: Предотвращает случайное попадание конфиденциальных данных в публичный репозиторий.
* **Улучшить структуру проекта**: Держит репозиторий чистым и фокусируется только на исходном коде и необходимых для развертывания файлах.

Добавьте преподавателя в репозиторий:

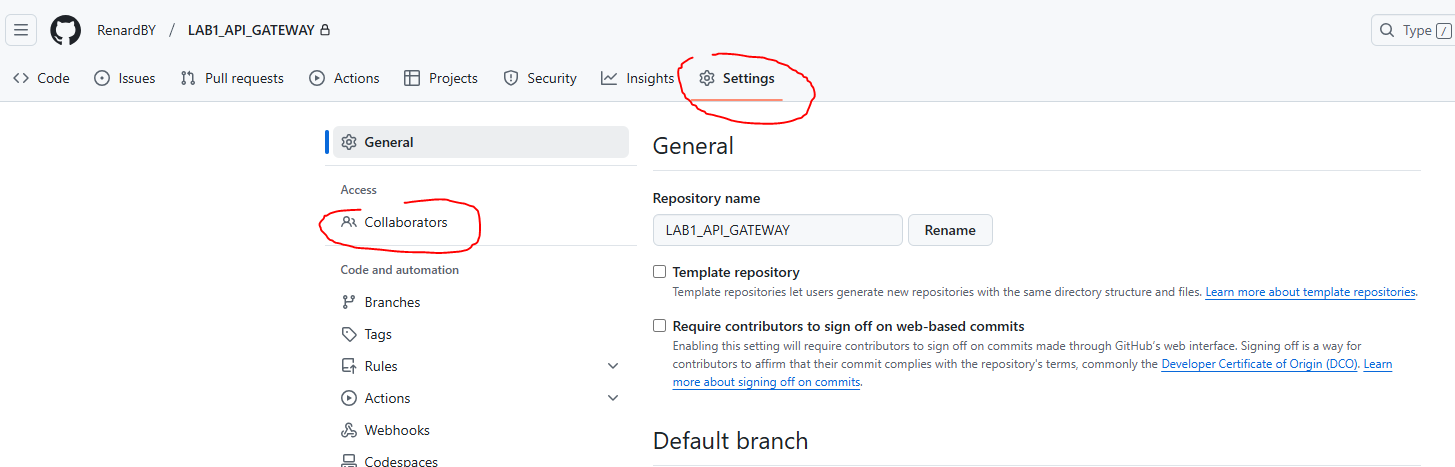


Рисунок 5 – Добавление преподавателя в репозиторий.

**Шаг 2: Клонирование репозитория на компьютер**

* Откройте терминал (на Windows можно использовать Git Bash).
* Перейдите в папку, куда вы хотите склонировать репозиторий. Например, cd Путь/к/вашей/папке.
* Склонируйте репозиторий (рисунок 6):

git clone [https://github.com/<username>/lab-work.git](https://github.com/%3cusername%3e/lab-work.git)

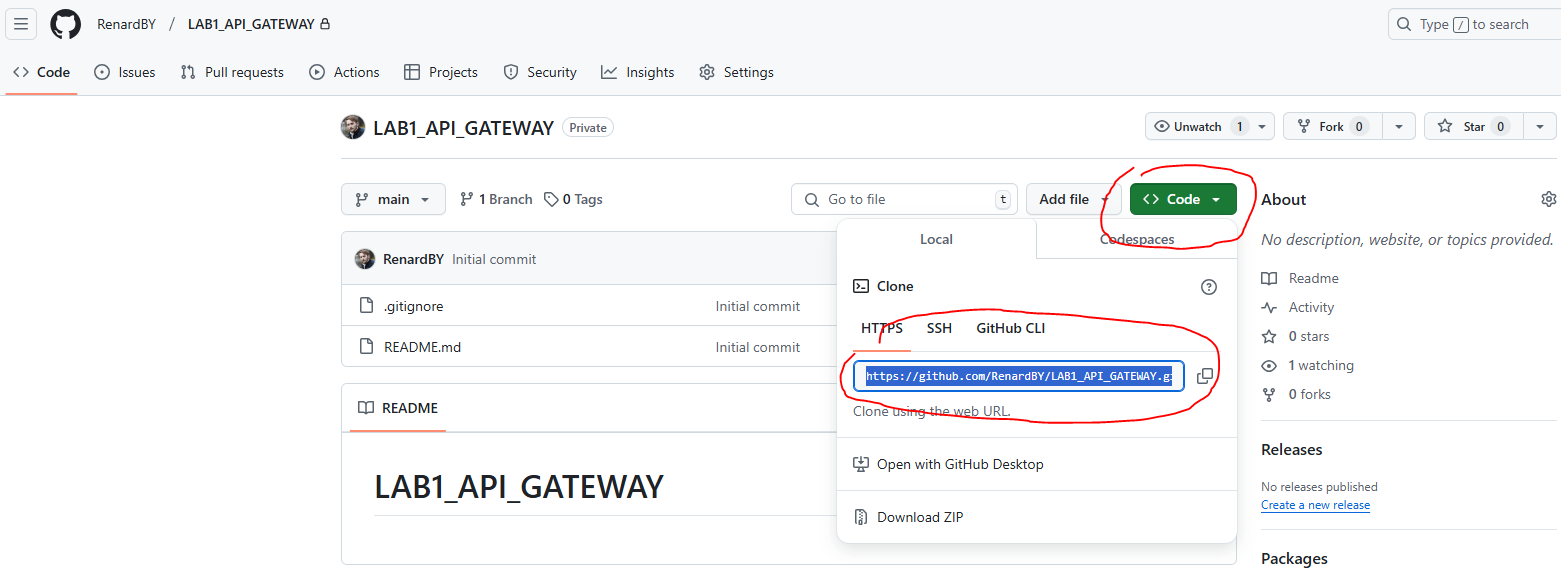


Рисунок 6 – Получение ссылки на репозиторий для последующего клонирования.

**Шаг 3: Создание main ветки**

Перейдите в папку вашего клонированного репозитория:

cd LAB1\_API\_GATEWAY

Убедитесь, что вы находитесь в ветке main. Выполните команду:

git checkout main

**Шаг 4: Создание README файла**

Откройте любой текстовый редактор и создайте файл с именем README.md.

В этом файле укажите свою фамилию и номер группы. Например:

Фамилия: Иванов

Номер группы: ИТ-123456

**Шаг 5: Добавление и коммит изменений**

В терминале выполните команду, чтобы добавить файл:

git add README.md

Закоммитьте изменения:

git commit -m "Добавление README файла"

**Шаг 6: Пуш изменений на удалённый репозиторий**

Отправьте изменения на GitHub:

git push origin main

**Шаг 7: Создание develop ветки в UI GitHub**

Перейдите на страницу вашего репозитория на GitHub.

Нажмите на выпадающее меню веток и введите название новой ветки develop, затем нажмите **"Create branch: develop"** (рисунок 7).

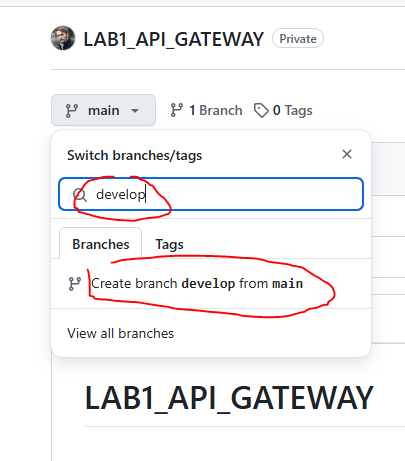


Рисунок 7 – Создание новой ветки

**Шаг 8: Pull изменений**

В терминале выполните команду, чтобы получить изменения с удалённого репозитория:

git checkout develop

git pull origin develop

**Шаг 9: Создание ветки feature/lab1 от develop ветки**

Создайте новую ветку для вашей функции:

git checkout -b feature/lab1

**Почему важно соблюдать GitHub branching strategy?**

Соблюдение стратегии ветвления необходимо для обеспечения упорядоченности и контроля в процессе разработки. Используя разные ветки для различных функций, разработчики могут работать над отдельными задачами, не мешая друг другу. Это помогает избежать конфликтов при слиянии и делает процесс разработки более структурированным. Ветвление позволяет также легче управлять релизами и тестированием, так как каждая новая функция или исправление может разрабатываться независимо, а затем интегрироваться в основную кодовую базу, когда будет готова.

**Шаг 10: Инициализация npm в новой ветке feature/lab1**

В терминале выполните команду для инициализации npm:

npm init

Следуйте инструкциям, чтобы создать файл package.json.

Команда npm init - это команда, которая инициализирует новый Node.js проект, создавая файл package.json.

Зачем это нужно:

* **Описывает проект**: package.json содержит метаданные о вашем проекте: имя, версия, описание, автор, лицензия, зависимости и скрипты.
* **Управление зависимостями:** Позволяет указать, какие пакеты (библиотеки) нужны вашему проекту. npm будет использовать package.json для установки и управления этими зависимостями.
* **Скрипты**: Позволяет определить скрипты для автоматизации задач, таких как запуск тестов, сборка проекта и т.д. (например, npm run build).
* **Публикация пакета**: package.json необходим для публикации вашего проекта как npm пакета.
* **Удобство**: Структурирует проект и предоставляет стандартный способ управления им.

Таким образом, npm init создает основу для вашего Node.js проекта, позволяя управлять его зависимостями, автоматизировать задачи и предоставлять информацию о проекте другим разработчикам. Это обязательный шаг для любого серьезного Node.js проекта.

* Установите Express и Axios: Выполните следующую команду в терминале, чтобы установить необходимые библиотеки на основе вашего проекта:

**npm install express axios**

**Express:** Это минималистичный и гибкий веб-фреймворк для Node.js. Он предоставляет набор функций для создания веб-приложений и API. По сути, это библиотека, которая упрощает обработку HTTP-запросов, маршрутизацию и шаблонизацию, позволяя быстро создавать веб-серверы.

**Axios:** Это HTTP-клиент на основе промисов для Node.js и браузеров. Он позволяет делать HTTP-запросы (GET, POST, PUT, DELETE и т.д.) к удаленным серверам. Axios упрощает процесс отправки запросов и обработки ответов, обеспечивая удобный API и такие функции, как автоматическое преобразование данных JSON, перехват запросов и ответов, защита от XSRF и т.д. Он используется для взаимодействия с API и получения данных из внешних источников.

**Шаг 11: Реализация паттерна API Gateway на основе Node.js** (**Express)**

* Создайте файл для вашего API Gateway: В директории вашего проекта создайте файл с именем server.js
* Реализуйте следующий код:

const express = require('express');

const axios = require('axios');

const app = express();

const PORT = 3000;

app.use(express.json());

const services = {

userService: 'http://localhost:4001',

orderService: 'http://localhost:4002',

productService: 'http://localhost:4003'

};

// Прокси для пользовательского сервиса

app.get('/users/:id', async (req, res) => {

try {

const response = await axios.get(`${services.userService}/users/${req.params.id}`);

res.json(response.data);

} catch (error) {

res.status(error.response.status).send(error.message);

}

});

// Прокси для сервисов заказов

app.get('/orders/:id', async (req, res) => {

try {

const response = await axios.get(`${services.orderService}/orders/${req.params.id}`);

res.json(response.data);

} catch (error) {

res.status(error.response.status).send(error.message);

}

});

// Прокси для сервисов продуктов

app.get('/products/:id', async (req, res) => {

try {

const response = await axios.get(`${services.productService}/products/${req.params.id}`);

res.json(response.data);

} catch (error) {

res.status(error.response.status).send(error.message);

}

});

// Объединения запросов

app.get('/dashboard', async (req, res) => {

try {

const [users, orders, products] = await Promise.all([

axios.get(`${services.userService}/users`),

axios.get(`${services.orderService}/orders`),

axios.get(`${services.productService}/products`)

]);

res.json({ users: users.data, orders: orders.data, products: products.data });

} catch (error) {

res.status(500).send('Ошибка при соединении с сервисами');

}

});

app.listen(PORT, () => {

console.log(`API Gateway запущен на порту ${PORT}`);

});

Рассмотрим, как работает этот код. Этот код представляет собой базовую структуру для микросервисной архитектуры, использующей Node.js, Express и Axios. Разберем его построчно:

const express = require('express');

Импортирует библиотеку Express.js в переменную express. express - это функция, которая создает приложение Express.

const axios = require('axios');

Импортирует библиотеку Axios в переменную axios. axios используется для выполнения HTTP-запросов.

const app = express();

Создает экземпляр приложения Express и присваивает его переменной app. app - это основной объект, с которым вы будете работать для определения маршрутов и обработки запросов.

const PORT = 3000;

Определяет константу PORT со значением 3000. Это номер порта, на котором будет запущен ваш Express-сервер (микросервис).

app.use(express.json());

Это middleware, который добавляет поддержку обработки JSON в теле запросов. Когда клиент отправляет данные в формате JSON, этот middleware автоматически преобразует их в объект JavaScript, доступный через req.body. Без этого middleware, req.body будет undefined для JSON-запросов.

const services = { ... };

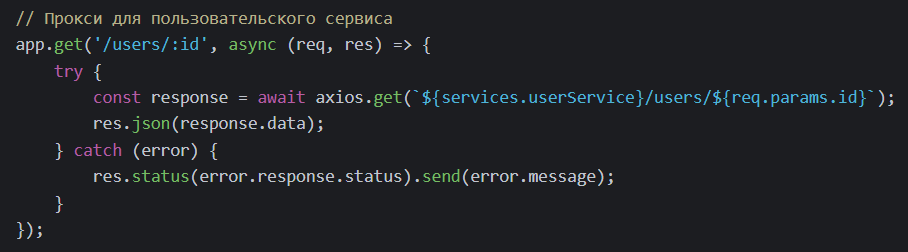
Создает объект services, который содержит URL-адреса других микросервисов.

• userService: 'http://localhost:4001' - Указывает URL для сервиса пользователей.

• orderService: 'http://localhost:4002' - Указывает URL для сервиса заказов.

• productService: 'http://localhost:4003' - Указывает URL для сервиса продуктов.

Рассмотрим, как работает этот код:



Этот код определяет маршрут (/users/:id) на сервере Express, который действует как прокси для другого микросервиса (userService). Разберем его по частям:

app.get('/users/:id', async (req, res) => { ... });

Определяет обработчик для GET-запросов по адресу /users/:id. :id - это параметр маршрута, который будет содержать идентификатор пользователя.

async (req, res) => { ... } - Это асинхронная функция обработчик запроса. req - объект запроса (содержит информацию о запросе, например, параметры маршрута, заголовки). res - объект ответа (используется для отправки ответа клиенту). async позволяет использовать await внутри функции.

try { ... } catch (error) { ... }

Блок try...catch используется для обработки ошибок. Если во время выполнения кода в блоке try возникает ошибка, управление передается в блок catch.

const response = await axios.get(\${services.userService}/users/${req.params.id}\);

Отправляет GET-запрос к userService с использованием axios.

services.userService - Получает URL сервиса пользователей (например, 'http://localhost:4001') из объекта services, который мы определили ранее.

/users/${req.params.id} - Добавляет к URL сервиса путь /users/ и идентификатор пользователя (req.params.id). req.params.id содержит значение параметра id из URL-адреса запроса (например, если запрос был /users/123, то req.params.id будет равен 123). ${...} используется для интерполяции строк (вставки значений переменных в строку).

await - Ожидает завершения HTTP-запроса, прежде чем продолжить выполнение кода. Это необходимо, потому что HTTP-запросы являются асинхронными.

response - Содержит объект ответа, возвращенный userService. response.data обычно содержит данные, возвращенные сервисом (например, информацию о пользователе).

res.json(response.data);

Отправляет данные, полученные от userService, обратно клиенту в формате JSON. res.json() автоматически устанавливает заголовок Content-Type: application/json.

catch (error) { ... }

Обрабатывает ошибки, возникшие при выполнении запроса к userService.

res.status(error.response.status).send(error.message);

error.response.status - Получает HTTP-статус ответа от userService (например, 404, 500).

res.status(error.response.status) - Устанавливает HTTP-статус ответа для клиента.

error.message - Получает сообщение об ошибке из объекта error.

res.send(error.message) - Отправляет сообщение об ошибке клиенту.

В целом, эта часть кода выполняет следующее:

1. Получает GET-запрос на адрес /users/:id.

2. Перенаправляет этот запрос к микросервису userService (например, http://localhost:4001/users/123).

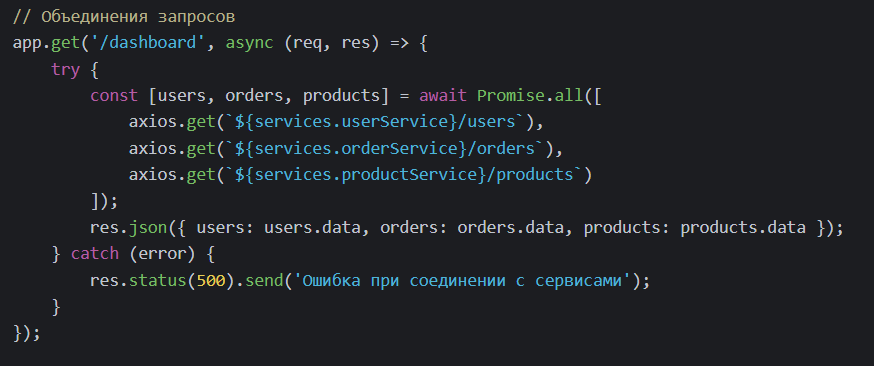
3. Получает ответ от userService.

4. Отправляет полученные данные (в формате JSON) обратно клиенту.

5. В случае ошибки, возвращает клиенту HTTP-статус ошибки и сообщение об ошибке.

Этот паттерн часто используется в микросервисной архитектуре для создания API-шлюза, который упрощает взаимодействие с несколькими микросервисами и обрабатывает ошибки.

Разберем как работает эта часть кода:



app.get('/dashboard', async (req, res) => { ... });

Определяет обработчик для GET-запросов по адресу /dashboard. Это означает, что когда клиент отправляет GET-запрос на этот адрес, будет выполнена функция внутри скобок.

async (req, res) => { ... } - Асинхронная функция обработчик запроса. req - объект запроса, res - объект ответа. async позволяет использовать await.

Блок try...catch используется для обработки ошибок. Если в try происходит ошибка, выполнение переходит в catch.

const [users, orders, products] = await Promise.all([ ... ]);

Promise.all([...]): Это ключевой момент. Promise.all принимает массив промисов и возвращает новый промис, который разрешается, когда все промисы в массиве разрешаются. Если хотя бы один промис отклоняется, Promise.all тоже отклоняется.

axios.get(${services.userService}/users): Отправляет GET-запрос к userService для получения списка пользователей.

axios.get(${services.orderService}/orders): Отправляет GET-запрос к orderService для получения списка заказов.

axios.get(${services.productService}/products): Отправляет GET-запрос к productService для получения списка продуктов.

await: Ожидает разрешения всех промисов, возвращенных Promise.all. Это значит, что скрипт ждет, пока все три HTTP-запроса завершатся.

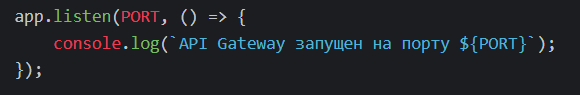
const [users, orders, products] = ...: Использует деструктурирующее присваивание (destructuring assignment), чтобы присвоить результаты запросов соответствующим переменным. users получит результат запроса к userService, orders - результат запроса к orderService, и products - результат запроса к productService. Обратите внимание, что порядок важен. Результаты присваиваются переменным в том же порядке, в котором промисы были переданы в Promise.all.

res.json({ users: users.data, orders: orders.data, products: products.data }); Отправляет ответ клиенту в формате JSON.

users.data, orders.data, products.data: Извлекает данные из объектов ответов axios. Axios обычно оборачивает данные в объект с ключом data.

{ users: ..., orders: ..., products: ... }: Создает JSON-объект, который содержит результаты всех трех запросов. Это позволяет агрегировать данные из разных микросервисов и отправить их клиенту в одном ответе.

res.status(500).send('Ошибка при соединении с сервисами'); Отправляет клиенту HTTP-статус 500 (Internal Server Error) и сообщение об ошибке. Статус 500 указывает на то, что произошла ошибка на стороне сервера.



app.listen(PORT, () => { ... }); Запускает Express-сервер на порту, указанном в переменной PORT (в нашем случае, 3000).

console.log(API Gateway запущен на порту ${PORT}); - Выводит сообщение в консоль, подтверждающее, что сервер запущен.

В целом, этот код реализует endpoint /dashboard, который агрегирует данные из трех разных микросервисов (userService, orderService, productService) и возвращает их клиенту в одном JSON-ответе. Он использует Promise.all для параллельного выполнения запросов к микросервисам, что повышает производительность. Он также включает обработку ошибок, чтобы клиент получал информативный ответ в случае возникновения проблем.

Этот код показывает распространенный паттерн в микросервисной архитектуре, когда API Gateway (в данном случае, Express-сервер) объединяет данные из нескольких сервисов, чтобы предоставить клиенту единую точку доступа к информации.

**Шаг 12 – Добавление авторизации**

Авторизацию можно реализовать следующим образом:

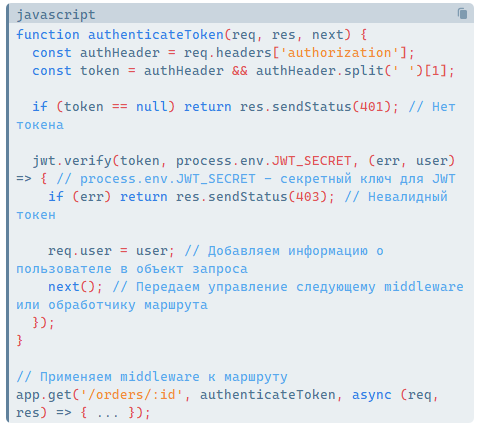


Рисунок 8 – Код функции авторизации

Для этого нужно следующее.

Установите пакет jsonwebtoken:

npm install jsonwebtoken

Импортируйте его в ваш файл:

const jwt = require('jsonwebtoken');

Установите пакет dotenv:

npm install dotenv

В самом начале вашего основного файла (server.js), добавьте следующие строки:

require('dotenv').config(); // Загружает переменные окружения из .env

Это необходимо сделать до того, как вы используете process.env.JWT\_SECRET где-либо в вашем коде.

Создайте файл с именем .env (без расширения) в корневой директории вашего Node.js проекта. Важно, чтобы он находился в той же папке, где находится ваш package.json файл и ваш основной файл (server.js).

Внутри файла .env добавьте строку, определяющую JWT\_SECRET:

JWT\_SECRET=your\_super\_secret\_key

Замените your\_super\_secret\_key на действительно случайную и сложную строку. Используйте генератор случайных строк для создания надежного секрета. Чем сложнее секрет, тем безопаснее ваши токены.

**Важно: Убедитесь, что файл .env добавлен в ваш .gitignore, чтобы случайно не закоммитить его в репозиторий Git. Он содержит конфиденциальную информацию.**

Реализуйте код который представлен на рисунке 8.

Шаг 13 Реализация Rate Limiting в Node.js

Есть несколько популярных middleware для rate limiting в Node.js. Один из самых простых в использовании - express-rate-limit. Он использует память сервера для хранения информации о лимитах (не подходит для масштабируемых приложений).

* Установите express-rate-limit в проект.
* В вашем основном файле приложения (server.js), импортируйте express-rate-limit.
* Настройте middleware express-rate-limit.

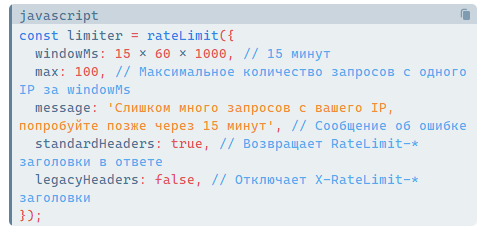


Рисунок 9 - Реализация Rate Limiting

Рассмотрим какие настройки используются в коде представленном выше.

windowMs: Окно времени в миллисекундах, в течение которого применяется ограничение. В этом примере – 15 минут.

max: Максимальное количество запросов, разрешенных с одного IP-адреса в течение windowMs. В этом примере – 100 запросов.

message: Сообщение, которое будет отправлено клиенту, если он превысит лимит.

standardHeaders: Включает отправку стандартных заголовков RateLimit-\* в ответе.

legacyHeaders: Отключает отправку устаревших заголовков X-RateLimit-\*.

* Примените rate limiter глобально ко всему приложению.

app.use(limiter); // Применяем rate limiter ко всем маршрутам

**Контрольные вопросы:**

* Что такое API Gateway и какова его основная роль в микросервисной архитектуре?
* Какие преимущества предоставляет использование API Gateway в системе микросервисов?
* Какие сторонние библиотеки и фреймворки можно использовать для реализации API Gateway на базе Node.js?
* В чем суть стратегии GitHub Flow? Какие ветки она использует и для чего они предназначены?
* Что такое "pull request" (запрос на слияние)? Для чего он используется?