МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Междисциплинарный проектный практикум-3» для студентов направления подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

Содержание

Введение

1. Общая характеристика дисциплины

Дисциплина «Междисциплинарный проектный практикум-3» является ключевой практико-ориентированной дисциплиной в системе подготовки бакалавров по направлению 09.03.04 «Программная инженерия» (профиль «Разработка и сопровождение программного обеспечения»).

Дисциплина реализуется на 3 курсе в 5 и 6 семестрах и представляет собой комплексную проектную деятельность, направленную на формирование у студентов профессиональных компетенций в области сквозной разработки программных продуктов.

2. Актуальность дисциплины

Современная программная инженерия предъявляет высокие требования к выпускникам: они должны не только уметь писать код, но и проектировать архитектуру, управлять
проектами, тестировать, развертывать и сопровождать программные системы.

Данный практикум моделирует реальный промышленный процесс разработки, где студенты проходят полный цикл создания программного продукта:

Анализ → Проектирование → Реализация → Тестирование → Развертывание → Сопровождение

3. Связь с другими дисциплинами

Дисциплина базируется на знаниях и навыках, полученных студентами при изучении следующих дисциплин:

Семестр	Опирающиеся дисциплины	Ключевые темы
1-4 семестры	Введение в	Основы

Семестр	Опирающиеся дисциплины	Ключевые темы
	программирование и алгоритмы	алгоритмизации, базовые конструкции
3-4 семестры	Базы данных	SQL, проектирование БД, нормализация
4 семестр	Информационные технологии и программирование	Основы web-разработки, Git, HTTP
5 семестр	Программная инженерия	ЖЦ ПО, методологии, управление требованиями
5 семестр	Технологии разработки ПО	Java/Spring Boot, JPA/Hibernate, JUnit
5-6 семестры	Проектирование архитектуры ПО	PCMEF, Clean Architecture, DDD
5-6 семестры	Конструирование ПО	TDD, рефакторинг, качество кода, шаблоны GoF
2 курс	Междисциплинарный проектный практикум-2	Основы React, компоненты, состояние, API

4. Сквозной проект

В рамках дисциплины студенты разрабатывают сквозной проект – платформу «Умный дом» (Smart Home).

Контекст проекта:

Компания «Тёплый Дом» начинала как небольшой сервис удаленного управления отоплением с монолитной архитектурой. Система не масштабировалась, не позволяла добавлять новый функционал (освещение, безопасность) и не поддерживала модель самообслуживания.

Цель проекта: Спроектировать, разработать и развернуть прототип новой распределенной платформы «Умный дом» с сервис-ориентированной архитектурой.

Ключевые характеристики целевой системы:

Характеристика	Описание
Распределенный бэкенд	3-4 независимых сервиса (Device, User, Telemetry, Notification)
Асинхронное взаимодействие	RabbitMQ для событий
Мультиклиентность	Web-интерфейс (React) и/или мобильное приложение
Контейнеризация	Docker + Docker Compose
Облачная готовность	Оркестрация для будущего развертывания в Kubernetes
Промышленные практики	API-First, TDD, CI/CD, мониторинг

5. Технологический стек курса

Компонент	Технология	Уровень освоения
Бэкенд	Java 17+, Spring Boot, Spring Data JPA, Spring Security	Продвинутый
Асинхронность	Spring AMQP + RabbitMQ	Продвинутый
БД	PostgreSQL, TimescaleDB, Redis	Продвинутый
Фронтенд (сквозной проект)	React 18+ (углубленный), Vite, React Router, Redux Toolkit, React Query	Углубленный
Фронтенд (самостоятельный проект)	React / Angular / Vue / Svelte (по выбору)	Самостоятельное изучение
Тестирование	JUnit 5, Mockito, TestContainers, Jest, RTL, Cypress	Продвинутый
Контейнеризация	Docker, Docker Compose	Продвинутый
CI/CD	GitHub Actions	Базовый
Мониторинг	Prometheus + Grafana + Loki	Базовый
ML-интеграция	Hugging Face / scikit-	Базовый

Компонент	Технология	Уровень освоения
	learn	
Документирование	OpenAPI/Swagger, PlantUML	Базовый

6. Структура лабораторных работ

Дисциплина включает 27 лабораторных работ, распределенных по двум семестрам:

5 семестр (9 лабораторных работ) – Бэкенд-разработка

№	Название	Основные темы
1	Анализ предметной области и архитектурное видение	Стейкхолдеры, C4 (Context, Container)
2	Детальное проектирование сервисов и API-контрактов	OpenAPI, API First, DDD
3	Асинхронное взаимодействие (RabbitMQ)	EDA, Publisher-Subscriber
4	Контейнеризация (Docker & Docker Compose)	Dockerfile, оркестрация
5	Реализация бэкенда (Spring Boot + JPA)	REST API, ORM, бизнес-логика
6	Аутентификация и авторизация (JWT)	Spring Security, токены
7	Модульное и интеграционное	JUnit, Mockito, TestContainers

№	Название	Основные темы
	тестирование	
8	Документирование API (Swagger/OpenAPI)	springdoc-openapi, E2E тесты
9	Итоговая интеграция и защита (1 часть)	Рефакторинг, техдолг, презентация

6 семестр (18 лабораторных работ) – Фронтенд, DevOps, ML

№	Название	Основные темы
10	Углубленный React (компоненты, хуки, роутинг)	Vite, React Router, оптимизация
11	Redux Toolkit / RTK Query	Управление состоянием, кэширование
12	React Query и CORS	Server-state, мутации
13	React Hook Form + Zod	Формы, валидация
14	Тестирование React (Jest + RTL)	Unit-тесты, MSW
15	Кэширование на бэкенде (Redis)	Cache-Aside, инвалидация
16	WebSockets (Spring + STOMP)	Real-time уведомления

№	Название	Основные темы
17	Нагрузочное тестирование (JMeter)	RPS, latency, профилирование
18	CI/CD (GitHub Actions)	Пайплайн, тесты, деплой
19	Docker-композиция полного стека	Healthcheck, оптимизация
20	Развертывание на VPS + Nginx + SSL	Cloud, reverse proxy
21	Интеграция ML-модели	Transformers, рекомендации
22	Мониторинг (Prometheus + Grafana)	Метрики, дашборды, алерты
23	Рефакторинг: PCMEF → Clean Architecture	DDD, слои, DI
24	Асинхронный инференс ML	RabbitMQ + Worker
25	E2E-тестирование (Cypress)	Сквозные тесты
26	Самостоятельный проект (ТЗ и архитектура)	Свободный выбор стека
27	Самостоятельный проект (реализация и защита)	Демонстрация, защита

7. Формат лабораторных работ

Каждая лабораторная работа имеет единую структуру:

1. Тема

2. Цель работы

3. Задачи

4. Краткое содержание

   ├── 4.1. Теоретическая часть

   └── 4.2. Примеры кода

5. Индивидуальные варианты

6. Контрольные вопросы

Дополнительные элементы:

Индивидуальные варианты – задание адаптируется под конкретную предметную область студента

Контрольные вопросы – для самопроверки и подготовки к защите

Разобранные примеры – готовый код, который студент может адаптировать под свой проект

8. Требования к выполнению лабораторных работ

8.1. Общие требования

Каждая лабораторная работа выполняется в собственном Git-репозитории

Код должен быть задокументирован (комментарии, README)

Для каждой работы создается тег в репозитории (например, lab-5, lab-6)

Отчет оформляется в формате Markdown и размещается в репозитории

8.2. Требования к отчету

Отчет по каждой лабораторной работе должен содержать:

Титульный лист (название работы, ФИО, группа)

Цель работы

Теоретическое обоснование (кратко)

Выполненные задания с указанием номеров заданий

Демонстрация результатов (скриншоты, логи, диаграммы)

Ключевые фрагменты кода

Выводы (что узнали, с какими трудностями столкнулись)

Ссылка на Git-репозиторий с тегом соответствующей работы

8.3. Критерии оценки

Каждая лабораторная работа оценивается по 10-балльной шкале:

Критерий	**8-10 баллов**	**5-7 баллов**	**3-4 балла**	**0-2 балла**
Выполнение требований	Полное соответствие ТЗ	Основные требования выполнены	Часть требований не выполнена	Требования не выполнены
Качество кода	Чистый код, SOLID, тесты	Рабочий код с недочетами	Есть запахи кода, нет тестов	Код не читаем
Документация	Полная (README, отчет)	Базовая	Фрагментарна	Отсутствует
Защита	Уверенно, ответы на вопросы	В целом уверенно	Неуверенно	Не может ответить

9. Самостоятельная работа

В рамках дисциплины студенты выполняют два типа проектов:

9.1. Сквозной проект (обязательный)

Разрабатывается в течение всего курса

Единый стек: Java/Spring Boot + React (углубленный)

Выполняется командой (3-4 человека)

Оценивается по результатам защиты в конце 6 семестра

9.2. Самостоятельный проект (по выбору)

Разрабатывается параллельно со сквозным проектом

Свободный выбор технологического стека:

Python/Django + React

Node.js/Express + React/Angular/Vue

ASP.NET Core + React/Angular

Go + Svelte/Vue

Desktop-приложение (JavaFX / C# WPF)

Выполняется индивидуально или в паре

Предусматривает презентацию ТЗ (ЛР26) и защиту реализации (ЛР27)

10. Формируемые компетенции

В результате освоения дисциплины у студентов формируются следующие профессиональные компетенции:

Код	Компетенция
ПК-1	Способен осуществлять управление программными проектами
ПК-2	Способен выполнять проектирование архитектуры программных систем
ПК-3	Владеет навыками разработки программного обеспечения
ПК-4	Способен проводить тестирование, отладку и оценивать качество ПО

Код	Компетенция
ПК-5	Способен осуществлять сопровождение, настройку, развертывание ПО
ПК-6	Владеет методами работы с данными и информационными системами
ПК-7	Способен применять современные IT-инструменты и DevOps-практики
ПК-8	Способен адаптировать и применять методы ИИ и МО
ПК-9	Умеет оформлять техническую документацию и вести коммуникацию
ПК-10	Готов соблюдать правовые нормы и требования ИБ

11. Как работать с лабораторными работами

Рекомендуемый порядок выполнения:

Прочитайте теоретическую часть – она содержит необходимые концепции

Изучите разобранные примеры кода – они показывают, как применять теорию

Выберите свой индивидуальный вариант (если предусмотрен)

Выполните задания для самостоятельной работы

Зафиксируйте результаты в Git (коммит + тег)

Оформите отчет в формате Markdown

Подготовьтесь к защите, используя контрольные вопросы

Важно: Лабораторные работы имеют сквозной характер – результаты предыдущей работы используются в последующей.

12. Заключение

Дисциплина «Междисциплинарный проектный практикум-3» призвана интегрировать теоретические знания, полученные в различных дисциплинах, и превратить их в практические навыки разработки программного обеспечения.

Успешное освоение дисциплины позволит вам:

Создавать полноценные веб-приложения от идеи до деплоя

Работать в команде с использованием Git и Agile-методологий

Проектировать архитектуру с учетом нефункциональных требований

Писать тесты и обеспечивать качество кода

Развертывать приложения в контейнерах и в облаке

Лабораторная работа №1

Введение в проект. Анализ предметной области и проектирование высокоуровневой архитектуры распределенной системы «Умный дом»

и требований. Контекст и контейнеры.

Цель: научиться проводить анализ предметной области (Domain Analysis) на основе вводных требований, определять границы проектируемой системы, выявлять стейкхолдеров и их потребности, а также создавать первое, контекстное архитектурное видение системы с использованием модели C4 (уровни Контекста и Контейнеров).

Теоретическое обоснование

1.1. Ключевые концепции:

Стейкхолдер (Stakeholder) - любое лицо или группа, которые влияют на систему или находятся под ее влиянием (заказчик, пользователь, инженер поддержки).

Требование (Requirement) - условие или возможность, которой должна соответствовать система (функциональное - «что делает», нефункциональное - «как делает»: надежность, масштабируемость).

Предметная область (Problem Domain): Сфера реального мира, для которой создается программная система (в нашем случае - автоматизация жилых помещений, IoT).

Домен (Domain) в DDD: Определенная область знаний или деятельности, в рамках которой решается бизнес-задача. Граница, внутри которой существует единый язык общения (Ubiquitous Language). [1-3]

2. Модель C4 для визуализации архитектуры программного обеспечения

2.1. Ключевые концепции:

- Абстракция и детализация: Модель C4 использует четыре уровня:

1. Система (System),

2. Контейнеры (Containers),

3. Компоненты (Components),

4. Код (Code).

Каждый следующий уровень раскрывает детали предыдущего.

Диаграмма контекста (Context Diagram, C4 Уровень 1)

Показывает систему в целом, ее взаимодействие с пользователями и другими системами. Ответ на вопрос: Что делает система и в каком окружении она работает?

Диаграмма контейнеров (Container Diagram, C4 Уровень 2)

Раскрывает систему как набор взаимосвязанных «контейнеров» (приложений, хранилищ данных, очередей). Ответ на вопрос: Из каких крупных блоков состоит система и как они взаимодействуют?

Контейнер (Container):

Выполняемая единица/процесс (например, веб-приложение, мобильное приложение, сервис, база данных, файловая система).[4-6]

Например

Уровень 1: Контекст - Система «Умный дом»

Уровень 2: Контейнеры - [Веб-приложение] ←→ [API Gateway] ←→ [Сервис устройств]

Уровень 3: Компоненты - [Контроллер] → [Сервис] → [Репозиторий]

Уровень 4: Код (Code). Например, реализация на Java

Примечание: Уровни Компонентов (Components) и Кода (Code) будут рассмотрены в последующих лабораторных работах при детальном проектировании сервисов.

```java
// DeviceService.java
public class DeviceService {  no usages
    private final DeviceRepository repository;  3 usages
    private final EventPublisher publisher;  2 usages

    // Конструктор, инъекция зависимостей
    public DeviceService(DeviceRepository repo, EventPublisher pub) {  no usages
        this.repository = repo;
        this.publisher = pub;
    }

    // Метод включения устройства
@   public DeviceResponse activateDevice(UUID deviceId, User user) {  no usages
        // 1. Проверка прав доступа
        if (!user.hasPermission(Permission.ACTIVATE_DEVICE)) {
            throw new AccessDeniedException();
        }

        // 2. Получение устройства из репозитория
        Device device = repository.findById(deviceId)
                .orElseThrow(() -> new DeviceNotFoundException());

        // 3. Применение бизнес-правил
        if (device.getStatus() == Status.ACTIVE) {
            throw new DeviceAlreadyActiveException();
        }

        // 4. Изменение состояния
        device.activate();
        repository.save(device);

        // 5. Публикация события
        publisher.publish(new DeviceActivatedEvent(deviceId, user.getId()));

        // 6. Формирование ответа
        return new DeviceResponse(device);
    }
}
```

3. Принципы распределенных систем и сервис-ориентированной архитектуры (SOA)

3.1. Ключевые концепции

Монолитная архитектура (Monolith).

Все компоненты системы тесно связаны, развертываются как единое целое. Достоинства: простота разработки на старте. Недостатки: сложность масштабирования, изменения и внедрения новых технологий.

Распределенная система (Distributed System)

Компоненты системы расположены на разных сетевых узлах и координируют свои действия путем передачи сообщений.

Сервис-ориентированная архитектура (SOA)

Подход к построению ИТ-инфраструктуры в виде набора слабосвязанных, повторно используемых сервисов, каждый из которых инкапсулирует определенную бизнес-функцию.

Слабосвязанность (Loose Coupling)

Целевое свойство архитектуры, при котором изменение одного компонента минимально влияет на другие. Достигается через четкие контракты (API). [7-9]

4. Инструментарий: PlantUML и C4-PlantUML [10-12]

PlantUML.

Язык для описания диаграмм в виде текстового кода.

C4-PlantUML

Библиотека для PlantUML, добавляющая элементы нотации C4.

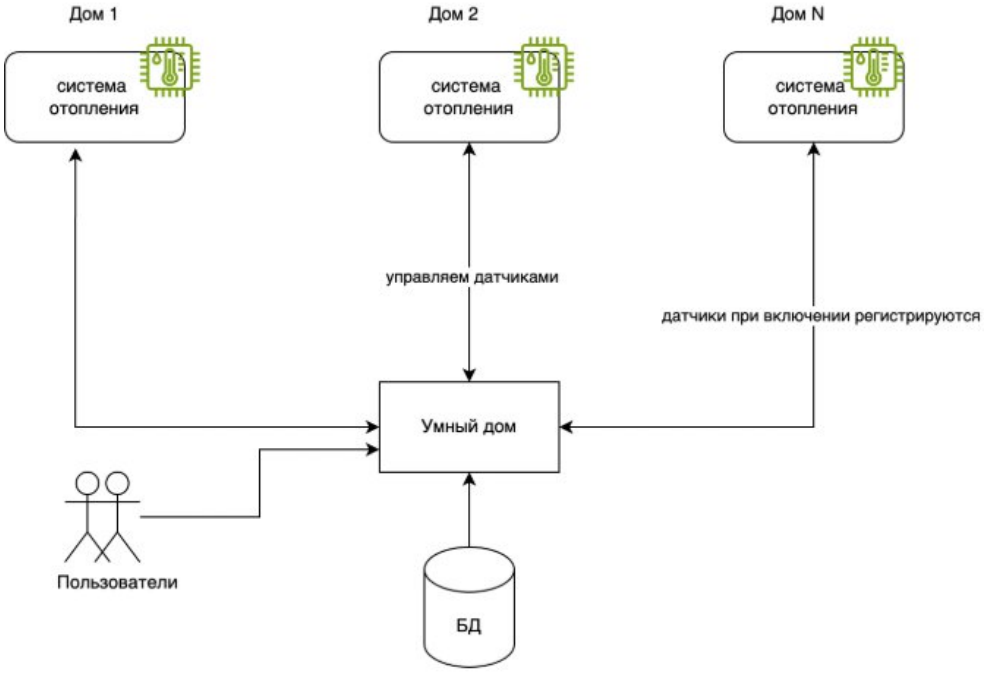
Для подготовки к лабораторной работе использовать источники, приведенные в конце лабораторной работе.

Практическая часть работы

1. Описание проекта разработки

Контекст проекта: Бизнес-задача компании «Тёплый Дом»

Вы присоединяетесь к проекту в роли архитектора и разработчика. Ваш «клиент» - компания «Тёплый Дом», которая начинала как небольшой сервис удаленного управления отоплением.



Текущая ситуация (As-Is):

Продукт: Монолитное приложение на Go, позволяющее 100 пользователям включать/выключать отопление и смотреть температуру в доме.

Архитектура: Один сервер (монолит), одна база данных PostgreSQL. Все взаимодействие синхронное. Для подключения нового дома требуется выезд инженера.

Проблемы: Система не масштабируется, неповоротлива, не позволяет добавлять новый функционал (освещение, безопасность) и не поддерживает модель самообслуживания.

Бизнес-возможность (To-Be):

Компания выиграла крупный тендер на создание экосистемы для «умных посёлков». Теперь им необходимо:

1. Стать платформой для управления любыми умными устройствами (отопление, свет, замки, камеры).
2. Перейти на модель SaaS (Software as a Service) с самообслуживанием: пользователь сам покупает совместимые устройства, подключает их к системе и настраивает.
3. Обеспечить масштабируемость на тысячи пользователей и десятки тысяч устройств.

Технический вызов: Существующий монолит не справится с этими задачами. Компания приняла стратегическое решение о переходе на современную распределенную сервис-ориентированную архитектуру.

Сквозная учебная цель лабораторных работ

В течение лабораторных работ дисциплины «Междисциплинарный практикум» необходимо спроектировать, разработать и развернуть прототип новой платформы «Умный дом».

Ключевые характеристики целевой системы:

1. Распределенный бэкенд: Система будет состоять из 3-4 независимых, слабосвязанных сервисов (например, Сервис управления устройствами,

Сервис телеметрии, Сервис пользователей), общающихся по четким API-контрактам. Вы можете предложить обоснованно и другие сервисы.

2. Мультиклиентность: К системе должен быть возможен доступ через веб-интерфейс и/или нативное мобильное приложение Android.

3. Готовая к облаку: Все компоненты будут контейнеризированы с помощью Docker и оркестрированы через docker-compose, что является основой для будущего развертывания в облаке (Kubernetes).

4. Промышленные практики: В процессе вы примените современные подходы: API-First дизайн, асинхронную коммуникацию через брокер сообщений, централизованный мониторинг и стратегию постепенного замещения монолита.

Ваша роль: Вам необходимо полный цикл программной инженерии - от анализа требований и выбора архитектуры до реализации, интеграции и тестирования рабочего прототипа.

Задание к лабораторной работе

Задание 1. Определение стейкхолдеров (Stakeholders) и их интересов

Составьте таблицу, в которой для каждого типа стейкхолдера укажите:

Роль	**Ключевая цель/потребность**	**Критически важные требования**

Роль (например, Владелец дома, Администратор посёлка, Техническая поддержка, Разработчик нового устройства и т.д.).

Их ключевая цель/потребность в отношении системы «Умный Дом».

Критически важные требования с их точки зрения (например, для владельца: «надёжно управлять замками», для поддержки: «видеть историю ошибок устройства»).

Шаг 2. Анализ функциональных требований и выделение бизнес-доменов

На основе описания целевой экосистемы (To-Be) составьте **список ключевых функциональных возможностей** (например, «Регистрация нового устройства», «Просмотр телеметрии температуры», «Создание сценария «Уйти из дома»).

Примечание: смело предлагайте свои сценарии умного дома

Сгруппируйте эти функции в **логические бизнес-домены**. Каждый домен - это зона ответственности, которая в дальнейшем может стать отдельным сервисом.

Пример группировки: Общий домен **«Управление устройствами»**.

Функции:

- «Добавить устройство»,
- «Получить список устройств»,
- «Отправить команду устройству»,
- и т.д.

Шаг 3. Определение границ системы

Четко сформулируйте, что входит в разрабатываемую систему (например, бэкенд-сервисы, API шлюз, веб-админка).

Четко сформулируйте, что НЕ ВХОДИТ и является внешней системой (например, мобильное приложение *является* клиентом системы, заводской софт устройства *не является* частью системы, облачный хостинг AWS/GCP — это инфраструктура, а не часть кодовой базы).

Шаг 4. Создание архитектурной диаграммы контекста (C4: Context Diagram)

Целью этого шага **будет** показать систему в целом, её взаимодействие с людьми и другими системами.

Инструмент: PlantUML с библиотекой C4-PlantUML.

Требования к диаграмме:

Покажите **Систему «Умный Дом»** как один компонент в центре.

Отобразите всех **ключевых пользователей (акторов)**, определенных на шаге 1 (например, «Владелец дома», «Инженер поддержки»).

Отобразите все внешние системы, с которыми будет взаимодействовать ваша система (например, «Партнёрское умное устройство», «СМС-шлюз», «Платёжный провайдер»).

Подпишите линии взаимодействия ключевыми командами или данными (например, «Управляет устройством», «Отправляет телеметрию»).

Шаг 5. Создание архитектурной диаграммы контейнеров (C4: Container Diagram)

Целью этого шага является раскрыть «Систему «Умный Дом»» из предыдущей диаграммы, показав основные технологические блоки (контейнеры) внутри неё.

Требования к диаграмме:

Покажите **программные контейнеры** (приложения, сервисы): например, «Мобильное приложение», «Веб-интерфейс (SPA)», «API Gateway», «Сервис устройств», «Сервис телеметрии», «Сервис аутентификации».

Покажите контейнеры данных: например, «База данных устройств PostgreSQL», «База данных телеметрии TimescaleDB», «Кэш сессий Redis».

Покажите инфраструктурные контейнеры: например, «Брокер сообщений(RabbitMQ».

Укажите технологии (язык/фреймворк) для ключевых контейнеров (например, Go, React, Kotlin).

Покажите ключевые взаимодействия между контейнерами (например, «Вызывает REST API», «Подписывается на события», «Читает/записывает»).

Литература

1. Карл Айгнс «Современный анализ требований». (Главы о выявлении стейкхолдеров и сборе требований).

2. https://habr.com/ru/companies/oleg-bunin/articles/551270/

3. Эрик Эванс «Предметно-ориентированное проектирование (DDD)» Главы 1-2 о кристаллизации знаний и выделении доменов.

4. https://github.com/plantuml-stdlib/C4-PlantUML; https://en.wikipedia.org/wiki/C4_model;

5. https://habr.com/ru/companies/oleg-bunin/articles/525814/

6. https://c4model.com/examples

7. Сэм Ньюмен «Создание микросервисов»

8. https://learn.microsoft.com/ru-ru/azure/architecture/guide/architecture-styles/

9. https://martinfowler.com/bliki/BoundedContext.html

Дополнительные источники (некоторые из них доступны через VPN)

Ресурс	Что содержит
Habr: "Модель C4 для визуализации архитектуры ПО" https://habr.com/ru/companies/oleg-bunin/articles/525814/	Полное описание всех4 уровней C4 Примеры диаграмм Сравнение с другими подходами
Medium: "C4 Model: Визуализация архитектуры ПО" https://medium.com/nuances-of-programming/c4-модель-визуализация-архитектуры-по-8a2f6c73c0b7	Детальное объяснение философии C4 Практические примеры Шаблоны для PlantUML
YouTube: "Модель C4 за 10 минут" https://www.youtube.com/watch?v=G-4Y8eZ85zE	Визуальное объяснение концепции Примеры построения диаграмм Демонстрация инструментов

Ресурс	Что содержит
GitHub: C4-Model Cheat Sheet https://github.com/adi2ish/c4-model-cheatsheet	Краткая шпаргалка по всем элементам C4 в виде схемы.
Structurizr Documentation https://docs.structurizr.com/help/c4	Документация к инструменту от создателя C4. Лучше всего объясняет концепции.
InfoQ: Visualising Software Architecture with the C4 Model https://www.infoq.com/articles/C4-architecture-model/	Статья от Simon Brown (создателя C4).
Wikipedia: C4 Model **https://en.wikipedia.org/wiki/C4_model**	Базовое описание, но достаточно для понимания основ.

Лабораторная работа №2.

Детальное проектирование сервисов и API-контрактов (API First Design)

Цель работы: научиться проектировать слабосвязанные сервисы на основе выделенных предметных доменов и описывать их взаимодействие

через формальные контракты с использованием спецификации OpenAPI (Swagger).

1. Теоретическое обоснование

1.1. API First: Проектирование через спецификацию

API First - это подход к разработке, при котором создание API начинается с проектирования контракта (спецификации) до написания кода. Этот подход обеспечивает:

- Согласованность: Все стейкхолдеры (фронтенд, бэкенд, мобильные разработчики) работают по единому контракту.
- Параллельную разработку: Фронтенд и бэкенд могут разрабатываться независимо, используя моки на основе спецификации.
- Документирование: Спецификация служит актуальной документацией API.
- Тестирование: Контракт можно проверить до реализации.

1.2. OpenAPI / Swagger

OpenAPI (ранее Swagger) - это стандарт описания REST API в формате JSON или YAML. Спецификация включает:

- Информацию о API: название, версия, описание.
- Серверы: базовые URL для доступа к API.
- Пути (Paths): эндпоинты и поддерживаемые HTTP методы.
- Параметры: query, path, header параметры.
- Тела запросов (Request Body): структура данных, отправляемых в API.
- Ответы (Responses): структура данных, возвращаемых API.

- Схемы (Schemas): модели данных (сущности).
- Безопасность: методы аутентификации (JWT, OAuth2, API Key).

Пример базовой структуры OpenAPI:

```
openapi: 3.0.0
info:
  title: Device Service API
  version: 1.0.0
  description: API для управления устройствами умного дома

servers:
  - url: https://api.smart-home.com/v1
    description: Production server

paths:
  /devices:
    get:
      summary: Получить список устройств
      responses:
        '200':
          description: Успешный ответ
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '/components/schemas/Device'

components:
  schemas:
```

```
Device:
  type: object
  properties:
    id:
      type: string
      format: uuid
    name:
      type: string
    type:
      type: string
      enum: [THERMOSTAT, LIGHT, LOCK, CAMERA]
    status:
      type: string
      enum: [ACTIVE, INACTIVE, OFFLINE]
```

1.3. Контрактное тестирование

Контрактное тестирование - это подход к проверке взаимодействия между сервисами, при котором тестируется соответствие реализованного API его спецификации (контракту).

Основные принципы:

- Провайдер (сервис, предоставляющий API) проверяет, что его реализация соответствует контракту.
- Консьюмер (клиент, вызывающий API) проверяет, что он правильно формирует запросы и обрабатывает ответы согласно контракту.
- Инструменты: Spring Cloud Contract, Pact, Postman/Newman с коллекциями из OpenAPI.

1.4. Domain-Driven Design (DDD): Ограниченные контексты

Ограниченный контекст (Bounded Context) - это граница, внутри которой существует единая, непротиворечивая модель предметной области. В разных контекстах одно и то же понятие может иметь разный смысл.

Пример для умного дома:

Контекст	Сущность «Устройство» содержит
Управление устройствами	id, имя, тип, статус, команды (вкл/выкл)
Телеметрия	id устройства, показания, временные метки, история
Пользователи	id устройства не важно, важна принадлежность пользователю

Каждый сервис реализует один ограниченный контекст. Это обеспечивает слабую связанность и независимость развития.

2. Практическая часть

2.1. Исходные данные (из лабораторной работы 1)

На основе анализа предметной области из лабораторной работы №1 выделили следующие домены:

Домен	Ответственность	Примеры функций
Управление устройствами	Жизненный цикл устройств, отправка команд	Добавить устройство, включить/выключить, получить статус
Телеметрия	Сбор и анализ показаний устройств	Получить температуру, историю изменений, агрегаты

Пользователи и доступ	Управление пользователями, ролями, правами	Регистрация, логин, проверка прав
Сценарии и автоматизация	Создание правил автоматизации	«Если температура <18 градусов, то включить отопление»
Уведомление	Оповещения пользователей	Push-уведомление, email, SMS

2.2. Задание 1. Определение эндпоинтов для каждого сервиса

Рассмотрите ниже приведенные примеры и выполните задания для самостоятельного решения.

Сервис управления устройствами (Device Service)

Бизнес-функции:

- Регистрация нового устройства в системе
- Получение списка устройств пользователя
- Получение информации о конкретном устройстве
- Отправка команды устройству (включить, выключить, установить режим)
- Удаление устройства
- Обновление информации об устройстве (имя, комната)

REST Endpoints:

Метод	**Путь**	**Описание**	**Request Body**	**Response**
POST	/api/v1/devices	Регистрация нового устройства	CreateDeviceRequest	Device

GET	/api/v1/devices	Получение списка устройств (с фильтрацией)	Query params: userId, type, room	Device[]
GET	/api/v1/devices/{deviceId}	Получение устройства по ID	-	Device
PATCH	/api/v1/devices/{deviceId}	Обновление информации устройства	UpdateDeviceRequest	Device
DELETE	/api/v1/devices/{deviceId}	Удаление устройства	-	204 No Content
POST	/api/v1/devices/{deviceId}/commands	Отправка команды устройству	DeviceCommand	CommandResponse
GET	/api/v1/devices/{deviceId}/status	Получить текущий статус	-	DeviceStatus

Сервис телеметрии (Telemetry Service)

Бизнес-функции:

- Прием показаний от устройств
- Получение текущего показания
- Получение истории показаний за период
- Агрегация данных (среднее, минимум, максимум)

REST Endpoints:

Метод	**Путь**	**Описание**	**Request Body**	**Response**
POST	/api/v1/telemetry	Отправка показания устройства	Telemetry Data	201 Created
GET	/api/v1/telemetry/devices/{deviceId}/current	Текущее показание устройства	-	TelemetryData
GET	/api/v1/telemetry/devices/{deviceId}/history	История показаний	Query: from, to, limit	TelemetryData[]
GET	/api/v1/telemetry/devices/{deviceId}/statistics	Статистика за период	Query: from, to, aggregation (hour/day)	Statistics

Сервис пользователей (User Service)

Бизнес-функции:

- Регистрация и аутентификация пользователей
- Управление профилем
- Управление ролями и правами доступа
- Привязка устройств к пользователям

REST Endpoints:

Метод	**Путь**	**Описание**	**Request Body**	**Response**
POST	/api/v1/auth/register	Регистрация	RegisterRequest	AuthResponse (с JWT)
POST	/api/v1/auth/login	Вход в систему	LoginRequest	AuthResponse (с JWT)
POST	/api/v1/auth/refresh	Обновление токена	RefreshTokenRequest	AuthResponse
GET	/api/v1/users/me	Получение профиля текущего пользователя	-	UserProfile
PUT	/api/v1/users/me	Обновление профиля	UpdateUserRequest	UserProfile
GET	/api/v1/users/{userId}/devices	Получить устройства	-	UserDevicesResponse

		пользователя		
POST	/api/v1/users/{userId}/devices/{deviceId}	Привязать устройство к пользователю	-	200 OK
DELETE	/api/v1/users/{userId}/devices/{deviceId}	Отвязать устройство	-	204 No Content

Самостоятельное выполнение задания 1

Задание 1.1. Разработать набор эндпоинтов для Telemetry Service

На основе описания домена телеметрии составьте таблицу эндпоинтов по следующему шаблону:

Метод	**Путь**	**Описание**	**Request Body/Params**	**Response**
POST	/api/v1/telemetry	Отправка показания устройства	TelemetryData	201 Created
….	…	…	…	..

Требования к эндпоинтам Telemetry Service:

- Прием показаний от устройств (температура, влажность, освещенность и т.д.)
- Получение текущего показания устройства
- Получение истории показаний за период с пагинацией

- Получение агрегированной статистики (среднее, мин, макс) за день/неделю/месяц

- Получение списка устройств, которые не присылали данные больше N времени (оффлайн)

Задание 1.2. Разработать набор эндпоинтов для Scenario Service

Сервис сценариев позволяет пользователям создавать правила автоматизации типа «Если датчик движения сработал, включить свет».

Разработайте эндпоинты для:

- Создания нового сценария (название, условие, действие)
- Получения списка сценариев пользователя
- Включения/отключения сценария
- Удаления сценария
- Получения истории срабатываний сценариев

Задание 1.3. Проанализировать и дополнить

Проанализируйте свой набор эндпоинтов и ответьте на вопросы:

- Какие эндпоинты требуют аутентификации?
- Какие эндпоинты могут вызываться только от имени администратора?
- Нужны ли webhook-эндпоинты для внешних систем?

Задание 2. Создание OpenAPI спецификации

Пример OpenAPI для Device Service (device-service.yaml)

Создайте файл device-service.yaml со следующим содержимым:

```
openapi: 3.0.0
info:
  title: Device Service API
```

```
  version: 1.0.0
  description: |

    Сервис для управления устройствами умного дома.
    Отвечает за регистрацию, конфигурацию и отправку команд устройствам.
  contact:
    name: Support Team
    email: support@smarthome.com
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html

servers:
  - url: http://localhost:8081/api/v1
    description: Local development server
  - url: https://api.smarthome.com/devices/v1
    description: Production server

tags:
  - name: Devices
    description: Операции с устройствами
  - name: Commands
    description: Отправка команд устройствам

paths:
  /devices:
    post:
      tags:
        - Devices
```

```
      summary: Регистрация нового устройства
      description: Добавляет новое устройство в систему. Устройство привязывается к пользователю из JWT токена.
      operationId: createDevice
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '/components/schemas/CreateDeviceRequest'
      responses:
        '201':
          description: Устройство успешно создано
          content:
            application/json:
              schema:
                $ref: '/components/schemas/Device'
        '400':
          description: Ошибка валидации данных
          content:
            application/json:
              schema:
                $ref: '/components/schemas/ErrorResponse'
        '401':
          description: Не авторизован
          $ref: '/components/responses/UnauthorizedError'

    get:
```

```
    tags:
      - Devices
    summary: Получение списка устройств
    description: Возвращает список устройств текущего пользователя с возможностью фильтрации
    operationId: getDevices
    security:
      - bearerAuth: []
    parameters:
      - name: type
        in: query
        description: Фильтр по типу устройства
        required: false
        schema:
          type: string
          enum: [THERMOSTAT, LIGHT, LOCK, CAMERA, SENSOR]
      - name: room
        in: query
        description: Фильтр по комнате
        required: false
        schema:
          type: string
      - name: status
        in: query
        description: Фильтр по статусу
        required: false
        schema:
          type: string
          enum: [ACTIVE, INACTIVE, OFFLINE]
      - name: limit
```

```
    in: query
    description: Количество записей на странице
    required: false
    schema:
      type: integer
      default: 20
      minimum: 1
      maximum: 100
  - name: offset
    in: query
    description: Смещение для пагинации
    required: false
    schema:
      type: integer
      default: 0
responses:
  '200':
    description: Успешный ответ
    content:
      application/json:
        schema:
          type: object
          properties:
            items:
              type: array
              items:
                $ref: '/components/schemas/Device'
            total:
              type: integer
              description: Общее количество устройств
```

```
            limit:
              type: integer
            offset:
              type: integer
      '401':
        $ref: '/components/responses/UnauthorizedError'

/devices/{deviceId}:
  get:
    tags:
      - Devices
    summary: Получение устройства по ID
    operationId: getDeviceById
    security:
      - bearerAuth: []
    parameters:
      - name: deviceId
        in: path
        required: true
        description: UUID устройства
        schema:
          type: string
          format: uuid
    responses:
      '200':
        description: Информация об устройстве
        content:
          application/json:
            schema:
              $ref: '/components/schemas/Device'
```

```
        '401':
          $ref: '/components/responses/UnauthorizedError'
        '403':
          description: Доступ запрещен (устройство принадлежит другому пользователю)
          content:
            application/json:
              schema:
                $ref: '/components/schemas/ErrorResponse'
        '404':
          description: Устройство не найдено
          content:
            application/json:
              schema:
                $ref: '/components/schemas/ErrorResponse'

    patch:
      tags:
        - Devices
      summary: Обновление информации об устройстве
      operationId: updateDevice
      security:
        - bearerAuth: []
      parameters:
        - name: deviceId
          in: path
          required: true
          schema:
            type: string
            format: uuid
```

```
  requestBody:
    required: true
    content:
      application/json:
        schema:
          $ref: '/components/schemas/UpdateDeviceRequest'
  responses:
    '200':
      description: Обновленное устройство
      content:
        application/json:
          schema:
            $ref: '/components/schemas/Device'
    '400':
      description: Ошибка валидации
    '401':
      $ref: '/components/responses/UnauthorizedError'
    '404':
      description: Устройство не найдено

delete:
  tags:
    - Devices
  summary: Удаление устройства
  operationId: deleteDevice
  security:
    - bearerAuth: []
  parameters:
    - name: deviceId
      in: path
```

```
        required: true
        schema:
          type: string
          format: uuid
      responses:
        '204':
          description: Устройство успешно удалено
        '401':
          $ref: '/components/responses/UnauthorizedError'
        '404':
          description: Устройство не найдено

  /devices/{deviceId}/commands:
    post:
      tags:
        - Commands
      summary: Отправка команды устройству
      description: Отправляет команду управления устройству (включить,
выключить, установить температуру и т.д.)
      operationId: sendCommand
      security:
        - bearerAuth: []
      parameters:
        - name: deviceId
          in: path
          required: true
          schema:
            type: string
            format: uuid
      requestBody:
```

```
      required: true
      content:
        application/json:
          schema:
            $ref: '/components/schemas/DeviceCommand'
    responses:
      '200':
        description: Команда принята к исполнению
        content:
          application/json:
            schema:
              $ref: '/components/schemas/CommandResponse'
      '400':
        description: Некорректная команда для данного типа устройства
      '401':
        $ref: '/components/responses/UnauthorizedError'
      '404':
        description: Устройство не найдено
      '409':
        description: Устройство не в сети или недоступно

  /devices/{deviceId}/status:
    get:
      tags:
        - Devices
      summary: Получение текущего статуса устройства
      operationId: getDeviceStatus
      security:
        - bearerAuth: []
      parameters:
```

```
      - name: deviceId
        in: path
        required: true
        schema:
          type: string
          format: uuid
    responses:
      '200':
        description: Текущий статус устройства
        content:
          application/json:
            schema:
              $ref: '/components/schemas/DeviceStatus'
      '401':
        $ref: '/components/responses/UnauthorizedError'
      '404':
        description: Устройство не найдено

components:
  schemas:
    Device:
      type: object
      description: Модель устройства в системе
      properties:
        id:
          type: string
          format: uuid
          example: "123e4567-e89b-12d3-a456-426614174000"
        name:
          type: string
```

```
      description: Название устройства, заданное пользователем
      example: "Термостат в гостиной"
    type:
      type: string
      enum: [THERMOSTAT, LIGHT, LOCK, CAMERA, SENSOR]
      description: Тип устройства
    model:
      type: string
      description: Модель устройства (производитель)
      example: "Xiaomi Smart Thermostat 2"
    room:
      type: string
      description: Комната, где установлено устройство
      example: "living_room"
    status:
      type: string
      enum: [ACTIVE, INACTIVE, OFFLINE]
      description: Статус подключения
    capabilities:
      type: array
      description: Возможности устройства
      items:
        type: string
        enum: [ON_OFF, TEMPERATURE_SET, BRIGHTNESS, COLOR,
LOCK_UNLOCK, MOTION_SENSOR]
    created_at:
      type: string
      format: date-time
    updated_at:
      type: string
```

```
      format: date-time
    user_id:
      type: string
      format: uuid
      description: ID владельца устройства
  required:
    - id
    - name
    - type
    - status

CreateDeviceRequest:
  type: object
  description: Данные для регистрации нового устройства
  properties:
    name:
      type: string
      description: Название устройства
      example: "Новая лампа"
    type:
      type: string
      enum: [THERMOSTAT, LIGHT, LOCK, CAMERA, SENSOR]
      description: Тип устройства
    model:
      type: string
      description: Модель устройства
      example: "Philips Hue White"
    room:
      type: string
      description: Комната
```

```
        example: "bedroom"
      device_id:
        type: string
        description: Физический ID устройства (MAC адрес или серийный
номер)
        example: "AA:BB:CC:DD:EE:FF"
    required:
      - name
      - type
      - device_id

  UpdateDeviceRequest:
    type: object
    description: Данные для обновления устройства
    properties:
      name:
        type: string
        description: Новое название
      room:
        type: string
        description: Новая комната
    minProperties: 1

  DeviceCommand:
    type: object
    description: Команда для устройства
    properties:
      command:
        type: string
```

```
        enum: [TURN_ON, TURN_OFF, SET_TEMPERATURE, SET_BRIGHTNESS, SET_COLOR, LOCK, UNLOCK]
        description: Тип команды
      parameters:
        type: object
        description: Параметры команды (зависят от типа)
        example:
          temperature: 22.5
      correlation_id:
        type: string
        description: ID для отслеживания команды
        example: "cmd_12345"
    required:
      - command

  CommandResponse:
    type: object
    properties:
      status:
        type: string
        enum: [ACCEPTED, PROCESSING, REJECTED]
      message:
        type: string
      correlation_id:
        type: string

  DeviceStatus:
    type: object
    properties:
      device_id:
```

```
        type: string
        format: uuid
      online:
        type: boolean
      last_seen:
        type: string
        format: date-time
      state:
        type: object
        description: Текущее состояние устройства
        example:
          power: "on"
          temperature: 22.5
          brightness: 80

  ErrorResponse:
    type: object
    properties:
      code:
        type: integer
        example: 400
      message:
        type: string
        example: "Ошибка валидации данных"
      details:
        type: string
        example: "Поле 'name' обязательно для заполнения"

securitySchemes:
  bearerAuth:
```

```
    type: http
    scheme: bearer
    bearerFormat: JWT
    description: JWT токен, полученный при аутентификации

  responses:
    UnauthorizedError:
      description: Отсутствует или недействителен JWT токен
      content:
        application/json:
          schema:
            $ref: '/components/schemas/ErrorResponse'
          example:
            code: 401
            message: "Не авторизован"

security:
  - bearerAuth: []
```

Пример OpenAPI для User Service (user-service.yaml)

Создайте файл user-service.yaml:

```
openapi: 3.0.0
info:
  title: User Service API
  version: 1.0.0
  description: Сервис управления пользователями и аутентификации
```

```
servers:
  - url: http://localhost:8080/api/v1
    description: Local server
  - url: https://api.smarthome.com/users/v1
    description: Production server

tags:
  - name: Auth
    description: Аутентификация и регистрация
  - name: Users
    description: Управление профилями пользователей

paths:
  /auth/register:
    post:
      tags:
        - Auth
      summary: Регистрация нового пользователя
      operationId: register
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '/components/schemas/RegisterRequest'
      responses:
        '201':
          description: Успешная регистрация
          content:
            application/json:
```

```
            schema:
              $ref: '/components/schemas/AuthResponse'
      '400':
        description: Ошибка валидации
      '409':
        description: Пользователь уже существует

  /auth/login:
    post:
      tags:
        - Auth
      summary: Вход в систему
      operationId: login
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '/components/schemas/LoginRequest'
      responses:
        '200':
          description: Успешный вход
          content:
            application/json:
              schema:
                $ref: '/components/schemas/AuthResponse'
        '401':
          description: Неверный email или пароль

  /auth/refresh:
```

```
  post:
    tags:
      - Auth
    summary: Обновление JWT токена
    operationId: refreshToken
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '/components/schemas/RefreshTokenRequest'
    responses:
      '200':
        description: Новый токен
        content:
          application/json:
            schema:
              $ref: '/components/schemas/AuthResponse'
      '401':
        description: Недействительный refresh token

/users/me:
  get:
    tags:
      - Users
    summary: Получение профиля текущего пользователя
    operationId: getCurrentUser
    security:
      - bearerAuth: []
    responses:
```

```
    '200':
      description: Профиль пользователя
      content:
        application/json:
          schema:
            $ref: '/components/schemas/UserProfile'
    '401':
      $ref: '/components/responses/UnauthorizedError'

  put:
    tags:
      - Users
    summary: Обновление профиля
    operationId: updateCurrentUser
    security:
      - bearerAuth: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '/components/schemas/UpdateUserRequest'
    responses:
      '200':
        description: Обновленный профиль
        content:
          application/json:
            schema:
              $ref: '/components/schemas/UserProfile'
      '400':
```

```
        description: Ошибка валидации
      '401':
        $ref: '/components/responses/UnauthorizedError'

/users/{userId}/devices:
  get:
    tags:
      - Users
    summary: Получить список устройств пользователя
    description: Возвращает ID устройств, принадлежащих пользователю
    operationId: getUserDevices
    security:
      - bearerAuth: []
    parameters:
      - name: userId
        in: path
        required: true
        schema:
          type: string
          format: uuid
    responses:
      '200':
        description: Список устройств
        content:
          application/json:
            schema:
              type: object
              properties:
                user_id:
                  type: string
```

```
            format: uuid
          devices:
            type: array
            items:
              type: string
              format: uuid
  '401':
    $ref: '/components/responses/UnauthorizedError'
  '403':
    description: Доступ запрещен

post:
  tags:
    - Users
  summary: Привязать устройство к пользователю
  operationId: addDeviceToUser
  security:
    - bearerAuth: []
  parameters:
    - name: userId
      in: path
      required: true
      schema:
        type: string
        format: uuid
  requestBody:
    required: true
    content:
      application/json:
        schema:
```

```
            type: object
            properties:
              device_id:
                type: string
                format: uuid
            required:
              - device_id
      responses:
        '200':
          description: Устройство привязано
        '400':
          description: Устройство уже привязано к другому пользователю
        '401':
          $ref: '/components/responses/UnauthorizedError'
        '404':
          description: Устройство не найдено


components:
  schemas:
    RegisterRequest:
      type: object
      properties:
        email:
          type: string
          format: email
          example: "user@example.com"
        password:
          type: string
          format: password
          minLength: 8
```

```
      example: "securePass123"
    first_name:
      type: string
      example: "Иван"
    last_name:
      type: string
      example: "Петров"
  required:
    - email
    - password

LoginRequest:
  type: object
  properties:
    email:
      type: string
      format: email
    password:
      type: string
      format: password
  required:
    - email
    - password

RefreshTokenRequest:
  type: object
  properties:
    refresh_token:
      type: string
  required:
```

```
    - refresh_token

AuthResponse:
  type: object
  properties:
    access_token:
      type: string
      description: JWT токен для доступа к API
    refresh_token:
      type: string
      description: Токен для обновления access_token
    expires_in:
      type: integer
      description: Время жизни access_token в секундах
    token_type:
      type: string
      default: Bearer
    user:
      $ref: '/components/schemas/UserProfile'

UserProfile:
  type: object
  properties:
    id:
      type: string
      format: uuid
    email:
      type: string
      format: email
    first_name:
```

```
        type: string
      last_name:
        type: string
      role:
        type: string
        enum: [USER, ADMIN, SUPPORT]
      created_at:
        type: string
        format: date-time
    required:
      - id
      - email

  UpdateUserRequest:
    type: object
    properties:
      first_name:
        type: string
      last_name:
        type: string
      password:
        type: string
        format: password
        minLength: 8
    minProperties: 1

securitySchemes:
  bearerAuth:
    type: http
    scheme: bearer
```

```
    bearerFormat: JWT

  responses:
    UnauthorizedError:
      description: Не авторизован
      content:
        application/json:
          schema:
            type: object
            properties:
              code:
                type: integer
                example: 401
              message:
                type: string
                example: "Не авторизован"
```

На основе выше приведенных примеров выполните самостоятельно следующие задания.

Самостоятельное выполнение задания 2.

Задание 2.1. Создать OpenAPI спецификацию для Telemetry Service

Создайте файл telemetry-service.yaml на основе эндпоинтов, разработанных в задании 1.1.

Минимальные требования к спецификации:

- Корректная структура OpenAPI 3.0
- Info блок с названием, версией, описанием
- Servers блок (локальный и продакшн)
- Paths для всех разработанных эндпоинтов
- Компоненты (schemas) для всех моделей данных:
  - TelemetryData

  - TelemetryHistoryResponse
  - StatisticsResponse
  - ErrorResponse
- Security схемы (JWT Bearer)

Задание 2.2. Создать OpenAPI спецификацию для Scenario Service

Создайте файл scenario-service.yaml для сервиса сценариев.

Модели данных для Scenario Service:
- Scenario (id, name, condition, action, enabled, user_id, created_at)
- Condition (type: device_state, time_schedule; parameters)
- Action (type: device_command, notification; parameters)
- ScenarioExecution (id, scenario_id, triggered_at, result)

Задание 2.3: Валидация спецификации

Проверьте созданные YAML файлы с помощью онлайн-инструментов:
- [Swagger Editor](https://editor.swagger.io/)
- [OpenAPI Validator](https://validator.swagger.io/)

Исправьте все синтаксические ошибки.

Задание 3. Описание моделей данных

Ключевые сущности и их атрибуты

Device Service:

- Device: id, name, type, model, room, status, capabilities, created_at, updated_at, user_id

- DeviceStatus: device_id, online, last_seen, state (динамический объект)

- DeviceCommand: command, parameters, correlation_id

Telemetry Service:

- TelemetryData: id, device_id, timestamp, type (temperature, humidity, etc.), value, unit

- Statistics: device_id, period_from, period_to, avg_value, min_value, max_value, count

User Service:

- User: id, email, password_hash, first_name, last_name, role, created_at

- UserDevice: user_id, device_id, added_at, role (owner, guest)

Самостоятельное выполнение задания 3.

Задание 3.1. Создать ER-диаграмму для всех сервисов

Используя PlantUML или любой другой инструмент, создайте диаграмму сущностей и связей для всех проектируемых сервисов.

Требования:

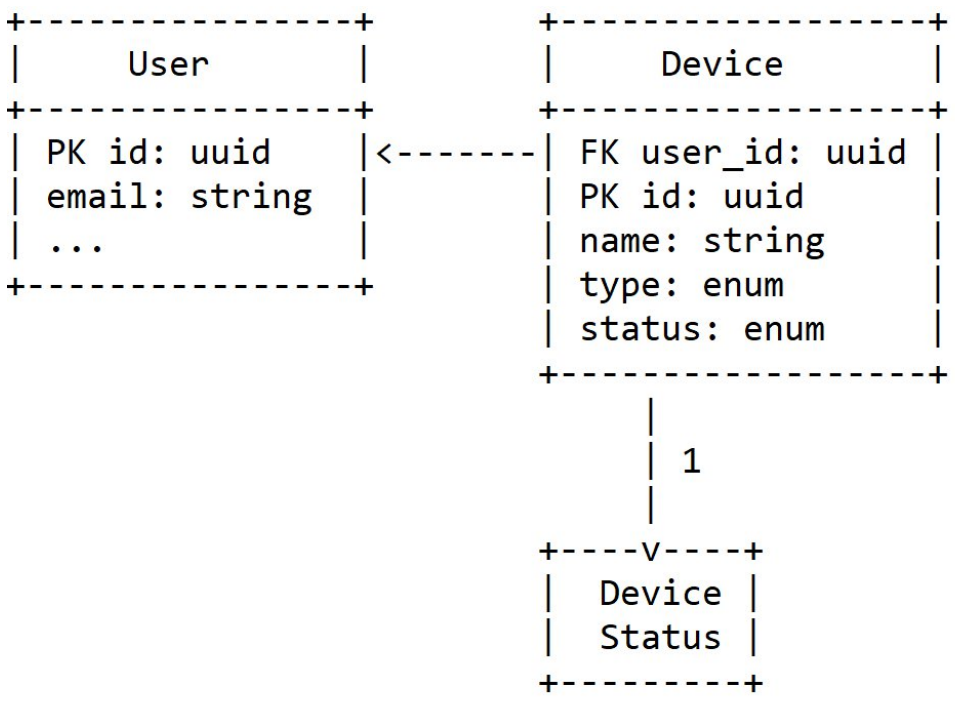
- Показать все сущности с атрибутами и типами данных

- Показать связи между сущностями (один-ко-многим, многие-ко-многим)

- Выделить первичные и внешние ключи

- Указать, в каком сервисе (БД) хранится каждая сущность

Пример для Device Service:



Задание 3.2. Нормализация данных

Проанализируйте модели данных и ответьте:

- Какие сущности дублируются между сервисами? (например, User в Device Service и User Service)

- Какие данные должны быть согласованы?

- Как будет решаться проблема согласованности данных между сервисами?

Задание 3.3. Специфичные типы данных

Опишите, как в OpenAPI спецификации будут представлены:

- Геолокационные данные (latitude/longitude)

- Временные ряды

- JSON-поля с динамической структурой (параметры команд)

- Единицы измерения (температура, влажность и т.д.)

Задание 4. Аутентификация (JWT)

Схема аутентификации

1. Клиент (веб/мобильное приложение) отправляет запрос на /auth/login с email/паролем.

2. User Service проверяет учетные данные и возвращает JWT токен.

3. JWT токен содержит payload с информацией о пользователе:

json

```
{
  "sub": "123e4567-e89b-12d3-a456-426614174000", // user_id
  "email": "user@example.com",
  "role": "USER",
  "exp": 1625097600
}
```

4. Клиент включает токен в заголовок Authorization: Bearer <token> при запросах к другим сервисам.

5. Каждый сервис (Device, Telemetry) проверяет подпись JWT и извлекает информацию о пользователе.

Взаимодействие между сервисами

При вызове одного сервиса другим (межсервисное взаимодействие) используются сервис-сервисные токены или внутренний API с доверенной сетью:

```
GET /api/v1/users/123/devices
X-API-Key: internal-service-key
```

Самостоятельное выполнение задания 4.

Задание 4.1. Разработать структуру JWT токена

Определите, какие поля должны быть в JWT payload для системы умного дома.

Требования:

- Стандартные поля (sub, exp, iat, iss)
- Пользовательские поля (роль, права доступа, id дома/поселка)

Создайте JSON пример JWT payload:

json

```
{
  "sub": "user-123",
  "email": "user@example.com",
  "role": "USER",
  "home_id": "home-456",
  "permissions": ["device:read", "device:write"],
  "exp": 1625097600
}
```

Задание 4.2. Разработать схему межсервисной аутентификации

Опишите, как сервисы будут аутентифицировать друг друга:

- Вариант А: Использование сервис-сервисных JWT
- Вариант Б: Внутренняя сеть Docker с запретом внешнего доступа
- Вариант В: API Key в заголовках

Выберите оптимальный вариант для учебного проекта и обоснуйте выбор.

Задание 4.3. Проверка прав доступа

Для каждого эндпоинта Device Service определите правила проверки прав:

Эндпоинт	Проверка
POST /devices	Только аутентифицированные пользователи
GET /devices/{id}	Пользователь должен быть владельцем
POST /devices/{id}/commands	Пользователь должен иметь права на управление
| DELETE /devices/{id}	Только владелец или администратор

Задание 4.4. Вопросы для анализа

- Как передавать JWT токен от клиента к сервисам?
- Как сервис проверяет подпись JWT?
- Нужно ли валидировать JWT с User Service при каждом запросе или можно проверять локально?
- Как отозвать доступ пользователя (logout)?

Задание 5. Обновление диаграммы контейнеров (C4 Level 2)

Пример кода PlantUML для обновленной диаграммы

Создайте файл container-diagram.puml:

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Container.puml

LAYOUT_WITH_LEGEND()

title Диаграмма контейнеров системы Умный дом (с API контрактами)

Person(user, "Владелец дома", "Пользователь, управляющий умным домом")
Person(support, "Инженер поддержки", "Сотрудник техподдержки")

System_Ext(mobile_app, "Мобильное приложение", "Kotlin/Java", "Клиент для Android/iOS")
System_Ext(web_app, "Веб-приложение", "React SPA", "Клиент для браузера")
System_Ext(iot_device, "Умное устройство", "MQTT/HTTP", "Физическое IoT устройство")

Boundary(backend, "Бэкенд платформы Умный дом") {
Container(api_gateway, "API Gateway", "Go/Java", "Единая точка входа, маршрутизация, проверка JWT", "port:8080")

Container(device_service, "Device Service", "Go/Java", "Управление устройствами, отправка команд", "port:8081\nOpenAPI: device-service.yaml")
Container(telemetry_service, "Telemetry Service", "Go/Java + TimescaleDB", "Сбор и анализ телеметрии", "port:8082\nOpenAPI: telemetry-service.yaml")
Container(user_service, "User Service", "Go/Java", "Управление пользователями и аутентификация", "port:8080\nOpenAPI: user-service.yaml")

ContainerDb(device_db, "Device Database", "PostgreSQL", "Хранит информацию об устройствах")

```
        ContainerDb(telemetry_db, "Telemetry Database", "TimescaleDB",
"Хранит временные ряды телеметрии")
        ContainerDb(user_db, "User Database", "PostgreSQL", "Хранит
пользователей и роли")

        Container(message_broker, "Message Broker", "RabbitMQ",
"Асинхронная коммуникация", "events: device.activated\ntelemetry.received")
    }

    Rel(user, web_app, "Использует", "HTTPS")
    Rel(user, mobile_app, "Использует", "HTTPS")

    Rel(web_app, api_gateway, "API вызовы", "HTTPS/REST")
    Rel(mobile_app, api_gateway, "API вызовы", "HTTPS/REST")
    Rel(iot_device, api_gateway, "Отправка телеметрии", "MQTT/HTTPS")

    Rel(api_gateway, user_service, "Аутентификация, пользователи", "REST
(JWT)")
    Rel(api_gateway, device_service, "Управление устройствами", "REST
(JWT)")
    Rel(api_gateway, telemetry_service, "Телеметрия", "REST (JWT)")

    Rel(user_service, user_db, "Читает/пишет", "SQL")
    Rel(device_service, device_db, "Читает/пишет", "SQL")
    Rel(telemetry_service, telemetry_db, "Читает/пишет", "SQL")

    Rel(device_service, message_broker, "Публикует события", "AMQP")
    Rel(telemetry_service, message_broker, "Подписывается на события",
"AMQP")
```

Rel(device)

Самостоятельное выполнение задания 5

Задание 5.1. Дополнить диаграмму новыми сервисами

На основе разработанных сервисов (Telemetry, Scenario, User) обновите диаграмму контейнеров.

Требования:
- Добавьте все спроектированные сервисы
- Укажите протоколы взаимодействия (REST/gRPC/Async)
- Для каждого сервиса укажите порт и ссылку на OpenAPI файл
- Добавьте внешние системы (если есть)
- Покажите потоки данных для ключевых сценариев:
  - Включение света через мобильное приложение
  - Автоматическое срабатывание сценария
  - Отправка уведомления

Задание 5.2. Создать диаграмму последовательности

Создайте диаграмму последовательности (Sequence Diagram) для сценария:

"Пользователь включает свет через мобильное приложение"

Участники:
- Мобильное приложение
- API Gateway
- User Service (проверка токена)
- Device Service
- MQTT Брокер (если устройство подключается через MQTT)

- Устройство (лампа)

Задание 5.3. Документирование API

Создайте файл API.md с краткой документацией:

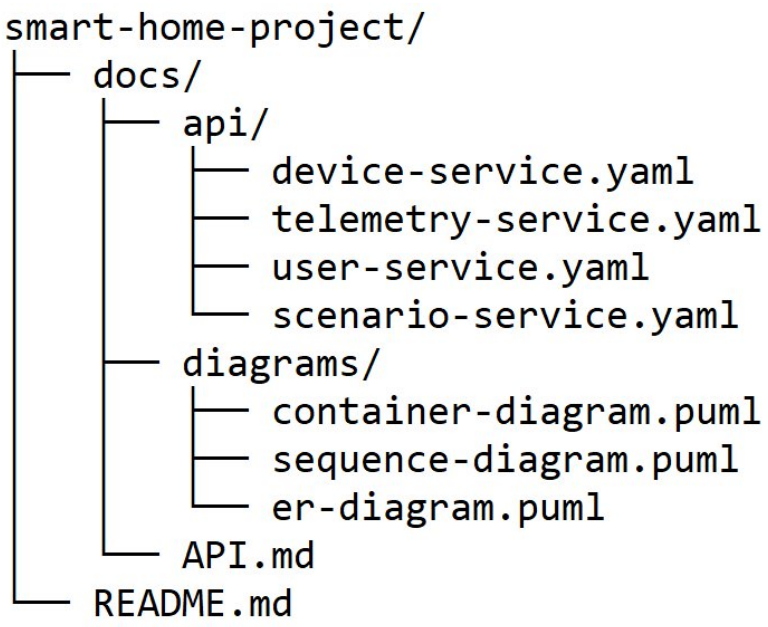
- Базовые URL всех сервисов
- Как получить токен (авторизация)
- Примеры запросов/ответов для ключевых сценариев
- Коды ошибок и их значения

3. Требования к отчету

Отчет по лабораторной работе должен содержать:

1. Титульный лист с названием работы, ФИО, группой
2. Цель работы
3. Задание 1: Таблицы эндпоинтов для Telemetry Service и Scenario Service
4. Задание 2: OpenAPI спецификации (2 файла) — в приложении или ссылки на GitHub
5. Задание 3: ER-диаграмма и описание моделей данных
6. Задание 4: Описание схемы аутентификации и прав доступа
7. Задание 5: Обновленная C4 диаграмма контейнеров и диаграмма последовательности
8. Выводы: Что нового узнали, с какими трудностями столкнулись, как их преодолели

!!В отчете представить ссылку на git. Пример структуры репозитория



Инструменты для выполнения работы:

- Swagger Editor(https://editor.swagger.io/) - онлайн редактор OpenAPI
- Stoplight Studio(https://stoplight.io/studio/) - десктопный редактор API
- Insomnia(https://insomnia.rest/) - тестирование API
- PlantUML Online(https://www.plantuml.com/plantuml/uml/) - создание диаграмм

Лабораторная работа №3.

Асинхронное взаимодействие и событийно-ориентированная архитектура

Цель работы: Научиться реализовывать асинхронную коммуникацию между сервисами через брокер сообщений для обработки доменных событий, формирование понимания принципов событийно-ориентированной архитектуры (EDA) и преимущества слабой связанности сервисов.

Теоретическое обоснование

1. Событийно-ориентированная архитектура (EDA)

Событийно-ориентированная архитектура (Event-Driven Architecture) - это архитектурный паттерн, в котором компоненты системы взаимодействуют путем генерации и обработки событий.

Ключевые понятия:

Термин	Определение
Событие (Event)	Значимое изменение состояния системы (например, "Устройство включено", "Температура превысила порог")
Издатель (Publisher)	Компонент, который генерирует события и публикует их в брокер
Подписчик (Subscriber)	Компонент, который получает уведомления о событиях и реагирует на них
Брокер сообщений (Message Broker)	Промежуточный слой, обеспечивающий доставку событий от издателей к подписчикам

Преимущества EDA:

- Слабая связанность (Loose Coupling): издатели не знают о подписчиках
- Масштабируемость: подписчики могут масштабироваться независимо
- Отказоустойчивость: при падении подписчика события сохраняются в очереди
- Расширяемость: новые подписчики добавляются без изменения существующего кода

Недостатки:

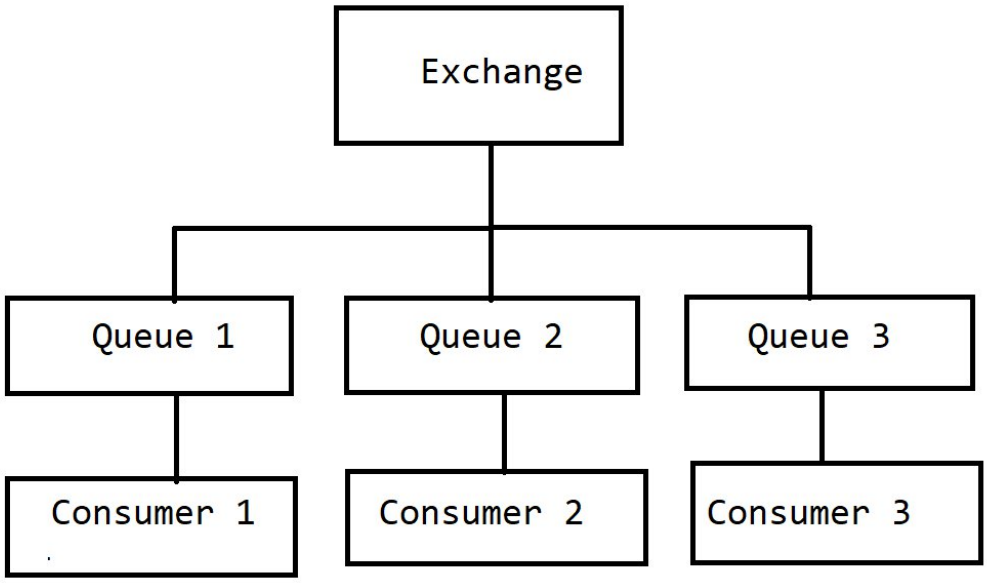
- Сложность отладки и трассировки
- Возможная потеря событий при неправильной конфигурации
- Гарантии доставки (at-least-once, exactly-once) требуют дополнительных усилий

2. Брокеры сообщений

RabbitMQ

RabbitMQ - популярный брокер сообщений, реализующий протокол AMQP (Advanced Message Queuing Protocol).

Ключевые компоненты RabbitMQ:



Компонент	Описание
Producer (Издатель)	Приложение, отправляющее сообщения
Exchange (Обменник)	Получает сообщения от продюсеров и направляет в очереди по правилам (binding)
Queue (Очередь)	Хранит сообщения до их обработки потребителями
Consumer (Потребитель)	Приложение, получающее сообщения из очереди
Binding (Связка)	Правило, связывающее обменник с очередью

Типы обменников (exchange) в RabbitMQ:

Тип	**Описание**	**Пример использования**
Direct	Сообщение доставляется в очередь с точно совпадающим routing key	Команды для конкретного устройства
Topic	Сообщение доставляется в очереди по шаблону routing key (с использованием и)	События по типам устройств (device.thermostat.)
Fanout	Сообщение рассылается во все связанные очереди	Широковещательные уведомления
Headers	Маршрутизация на основе заголовков (не routing key)	Сложная фильтрация

Apache Kafka - распределенная платформа для потоковой обработки данных.

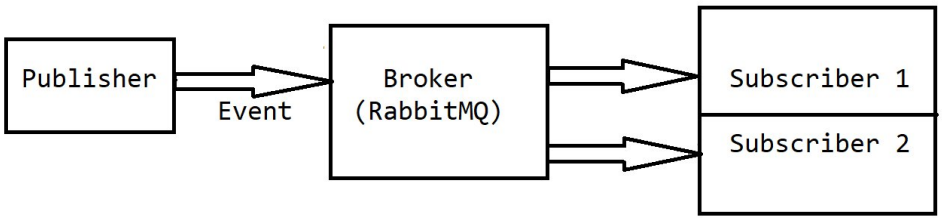
Отличия от RabbitMQ:

- Хранит все сообщения на диске (лог), а не удаляет после доставки
- Поддерживает replay событий (перечитывание с любого момента)
- Лучше подходит для больших объемов данных и аналитики
- Сложнее в настройке, чем RabbitMQ

Для учебного проекта рекомендуется RabbitMQ из-за простоты настройки и понимания.

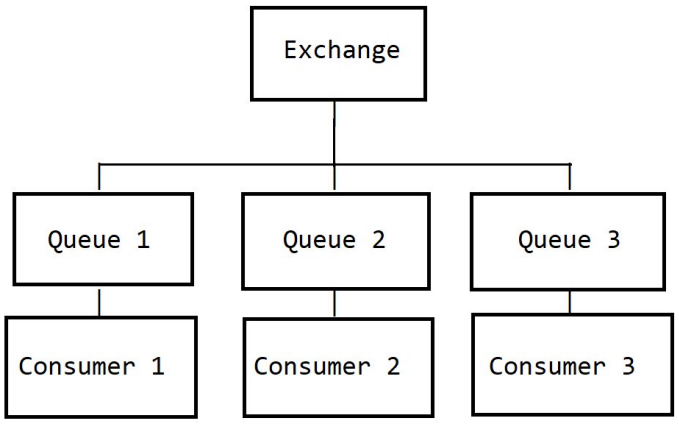
3. Паттерны взаимодействия

Publish-Subscribe (Pub/Sub)



Издатель публикует событие один раз, брокер доставляет его всем подписчикам.

Point-to-Point (Очередь задач)



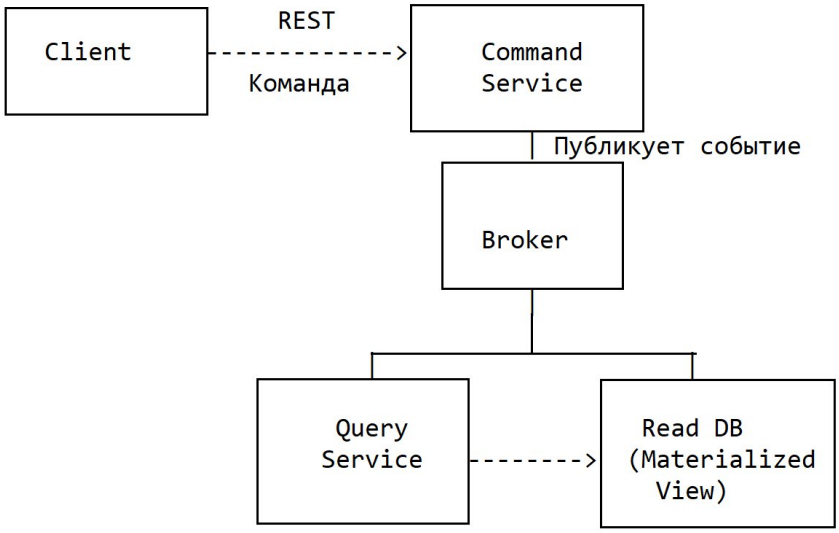
Сообщение доставляется только одному потребителю (конкуренция за задачи).

CQRS (Command Query Responsibility Segregation)

CQRS - разделение моделей чтения и записи. В контексте событийной архитектуры:

- Command (Команда) - изменяет состояние (синхронно, REST)

- Event (Событие) - уведомляет об изменении (асинхронно, через брокер)
- Query (Запрос) - читает состояние (может читать из отдельной БД)



4. Слабая связанность (Loose Coupling)

Слабая связанность - это свойство системы, при котором компоненты минимально зависят друг от друга.

В контексте событий:
- Device Service публикует событие, но не знает, кто его обработает
- Notification Service может быть добавлен без изменения Device Service
- При падении Notification Service, Device Service продолжает работать
- Очередь RabbitMQ накапливает события, пока сервис не восстановится

Как это достигается:
1. Сервисы общаются через посредника (брокер), а не напрямую
2. Используются общие контракты (формат событий), а не API сервисов
3. Издатели не ждут ответа от подписчиков (асинхронность)

Практическая часть

1. Исходные данные (из лабораторных работ 1 и 2)

Из предыдущих лабораторных работ у нас есть:
- Device Service - управляет устройствами, имеет REST API
- Telemetry Service - собирает телеметрию (в разработке)

- User Service - управляет пользователями

В этой работе необходимо:

1. Добавить в Device Service публикацию событий при изменении состояния устройств
2. Создать Notification Service (новый сервис), который подписывается на события
3. Настроить RabbitMQ для асинхронной коммуникации

Задание 1. Настройка RabbitMQ

Пример для разбора

1.1. Установку и настройку Docker смотри в приложении 1 этой лабораторной работы

1.2. Установку и настройку RabbitMQ смотри в приложении 2 этой лабораторной работы

Шаг 1. Запуск RabbitMQ через Docker

Создайте файл docker-compose.yml:

```yaml
version: '3.8'

services:
  rabbitmq:
    image: rabbitmq:3.12-management
    container_name: smart-home-rabbitmq
    ports:
      - "5672:5672"      # AMQP protocol port (для приложений)
      - "15672:15672"    # Management UI port (веб-интерфейс)
    environment:
      - RABBITMQ_DEFAULT_USER=admin
      - RABBITMQ_DEFAULT_PASS=admin123
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    networks:
      - smart-home-net

networks:
  smart-home-net:
    driver: bridge

volumes:
  rabbitmq_data:
```

Шаг 2. Запуск контейнера (bash)

```
docker-compose up -d
```

Шаг 3. Проверка доступа

Откройте браузер и перейдите по адресу: http://localhost:15672

- Логин: admin
- Пароль: admin123

Вы должны увидеть веб-интерфейс управления RabbitMQ.

Шаг 4. Создание exchange и очередей (опционально, можно через код)

В веб-интерфейсе:
1. Перейдите вкладку Exchanges
2. Создайте новый exchange:
   - Name: device.events
   - Type: topic
   - Durability: Durable

3. Перейдите вкладку Queues
4. Создайте очередь:
   - Name: notification.service.queue
   - Durability: Durable

5. Создайте binding:
   - Queue: notification.service.queue
   - Exchange: device.events
   - Routing key: device.state.

Самостоятельное задание для студента

Задание 1. Настройка RabbitMQ

Задание 1.1. Запустить RabbitMQ

Выполните шаги 1-4 из разобранного примера. Сделайте скриншот веб-интерфейса RabbitMQ с созданным exchange и очередью.

Задание 1.2. Исследование RabbitMQ

Изучите веб-интерфейс RabbitMQ и ответьте на вопросы:

- Какие еще вкладки есть в интерфейсе? (минимум 3)
- Что показывает вкладка "Connections"?
- Что показывает вкладка "Channels"?
- Для чего нужна вкладка "Admin"?

Задание 1.3. Конфигурация через код

Изучите документацию RabbitMQ и найдите, как создать exchange и очередь не через веб-интерфейс, а через код при старте приложения. Напишите небольшой код на выбранном языке (Java/Python), который при запуске проверяет наличие exchange и создает его, если он отсутствует.

Задание 2. Модификация Device Service (Publisher)

Пример на Java со Spring Boot

Шаг 1. Добавление зависимостей (pom.xml)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Шаг 2. Конфигурация RabbitMQ (RabbitConfig.java)

```
package com.smarthome.device.config;

import org.springframework.amqp.core.;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitConfig {

  public static final String DEVICE_EVENTS_EXCHANGE = "device.events";
  public static final String NOTIFICATION_QUEUE = "notification.service.queue";
  public static final String TELEMETRY_QUEUE = "telemetry.service.queue";

  @Bean
  public TopicExchange deviceEventsExchange() {
    return ExchangeBuilder.topicExchange(DEVICE_EVENTS_EXCHANGE)
        .durable(true)
        .build();
  }

  @Bean
  public Queue notificationQueue() {
    return QueueBuilder.durable(NOTIFICATION_QUEUE).build();
  }

  @Bean
  public Queue telemetryQueue() {
```

```
        return QueueBuilder.durable(TELEMETRY_QUEUE).build();
    }

    @Bean
    public Binding notificationBinding() {
        return BindingBuilder.bind(notificationQueue())
                .to(deviceEventsExchange())
                .with("device.state.");
    }

    @Bean
    public Binding telemetryBinding() {
        return BindingBuilder.bind(telemetryQueue())
                .to(deviceEventsExchange())
                .with("device.telemetry.");
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.setMessageConverter(messageConverter());
        return template;
    }
}
```

Шаг 3. Создание класса события (DeviceEvent.java)

```
package com.smarthome.device.events;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.Instant;
import java.util.UUID;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class DeviceEvent {
    private String eventId;
    private String eventType;
    private UUID deviceId;
```

```
    private UUID userId;
    private String deviceName;
    private String oldState;
    private String newState;
    private Instant timestamp;
    private Object additionalData;
}
```

Шаг 4. Создание фабрики событий (EventFactory.java)

```
package com.smarthome.device.events;

import com.smarthome.device.model.Device;
import org.springframework.stereotype.Component;

import java.time.Instant;
import java.util.UUID;

@Component
public class EventFactory {

    public DeviceEvent createDeviceStateChangedEvent(Device device, String oldState, String
newState, UUID userId) {
        DeviceEvent event = new DeviceEvent();
        event.setEventId(UUID.randomUUID().toString());
        event.setEventType("DEVICE_STATE_CHANGED");
        event.setDeviceId(device.getId());
        event.setUserId(userId);
        event.setDeviceName(device.getName());
        event.setOldState(oldState);
        event.setNewState(newState);
        event.setTimestamp(Instant.now());

        // Дополнительные данные в зависимости от типа устройства
        if ("THERMOSTAT".equals(device.getType()) && newState.equals("ACTIVE")) {
            event.setAdditionalData(Map.of("temperature", 22.5));
        }

        return event;
    }

    public DeviceEvent createDeviceRegisteredEvent(Device device, UUID userId) {
        DeviceEvent event = new DeviceEvent();
        event.setEventId(UUID.randomUUID().toString());
        event.setEventType("DEVICE_REGISTERED");
        event.setDeviceId(device.getId());
        event.setUserId(userId);
```

```
        event.setDeviceName(device.getName());
        event.setNewState(device.getStatus());
        event.setTimestamp(Instant.now());
        event.setAdditionalData(Map.of(
            "deviceType", device.getType(),
            "model", device.getModel()
        ));
        return event;
    }

    public DeviceEvent createDeviceDeletedEvent(UUID deviceId, UUID userId, String
deviceName) {
        DeviceEvent event = new DeviceEvent();
        event.setEventId(UUID.randomUUID().toString());
        event.setEventType("DEVICE_DELETED");
        event.setDeviceId(deviceId);
        event.setUserId(userId);
        event.setDeviceName(deviceName);
        event.setTimestamp(Instant.now());
        return event;
    }
}
```

Шаг 5. Создание сервиса для публикации событий (EventPublisher.java)

```
package com.smarthome.device.service;

import com.smarthome.device.config.RabbitConfig;
import com.smarthome.device.events.DeviceEvent;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Service;

@Slf4j
@Service
@RequiredArgsConstructor
public class EventPublisher {

    private final RabbitTemplate rabbitTemplate;

    public void publishDeviceEvent(DeviceEvent event, String routingKey) {
        try {
            log.info("Publishing event: {} for device: {}", event.getEventType(), event.getDeviceId());
            rabbitTemplate.convertAndSend(
                RabbitConfig.DEVICE_EVENTS_EXCHANGE,
                routingKey,
```

```
            event
        );
        log.debug("Event published successfully");
    } catch (Exception e) {
        log.error("Failed to publish event: {}", e.getMessage(), e);
        // Здесь можно добавить механизм повторных попыток или сохранения в БД
    }
  }

  public void publishDeviceStateChanged(DeviceEvent event) {
    publishDeviceEvent(event, "device.state.changed");
  }

  public void publishDeviceRegistered(DeviceEvent event) {
    publishDeviceEvent(event, "device.state.registered");
  }

  public void publishDeviceDeleted(DeviceEvent event) {
    publishDeviceEvent(event, "device.state.deleted");
  }
}
```

Шаг 6. Модификация сервиса устройств (DeviceService.java)

```
package com.smarthome.device.service;

import com.smarthome.device.events.DeviceEvent;
import com.smarthome.device.events.EventFactory;
import com.smarthome.device.exception.DeviceNotFoundException;
import com.smarthome.device.model.Device;
import com.smarthome.device.repository.DeviceRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Slf4j
@Service
@RequiredArgsConstructor
public class DeviceService {

  private final DeviceRepository deviceRepository;
  private final EventPublisher eventPublisher;
  private final EventFactory eventFactory;
```

```
@Transactional
public Device activateDevice(UUID deviceId, UUID userId) {
    log.info("Activating device: {} for user: {}", deviceId, userId);

    Device device = deviceRepository.findById(deviceId)
            .orElseThrow(() -> new DeviceNotFoundException(deviceId));

    // Проверка прав (упрощенно)
    if (!device.getUserId().equals(userId)) {
        throw new SecurityException("User does not own this device");
    }

    String oldState = device.getStatus();

    // Изменение состояния
    device.setStatus("ACTIVE");
    device.setUpdatedAt(Instant.now());

    Device savedDevice = deviceRepository.save(device);

    // Публикация события
    DeviceEvent event = eventFactory.createDeviceStateChangedEvent(
        savedDevice, oldState, "ACTIVE", userId
    );
    eventPublisher.publishDeviceStateChanged(event);

    log.info("Device activated successfully: {}", deviceId);
    return savedDevice;
}

@Transactional
public Device deactivateDevice(UUID deviceId, UUID userId) {
    log.info("Deactivating device: {} for user: {}", deviceId, userId);

    Device device = deviceRepository.findById(deviceId)
            .orElseThrow(() -> new DeviceNotFoundException(deviceId));

    if (!device.getUserId().equals(userId)) {
        throw new SecurityException("User does not own this device");
    }

    String oldState = device.getStatus();

    device.setStatus("INACTIVE");
    device.setUpdatedAt(Instant.now());

    Device savedDevice = deviceRepository.save(device);

    DeviceEvent event = eventFactory.createDeviceStateChangedEvent(
        savedDevice, oldState, "INACTIVE", userId
```

```
        );
        eventPublisher.publishDeviceStateChanged(event);

        return savedDevice;
    }

    @Transactional
    public Device registerDevice(Device device, UUID userId) {
        log.info("Registering new device for user: {}", userId);

        device.setId(UUID.randomUUID());
        device.setUserId(userId);
        device.setStatus("INACTIVE");
        device.setCreatedAt(Instant.now());
        device.setUpdatedAt(Instant.now());

        Device savedDevice = deviceRepository.save(device);

        DeviceEvent event = eventFactory.createDeviceRegisteredEvent(savedDevice, userId);
        eventPublisher.publishDeviceRegistered(event);

        return savedDevice;
    }

    @Transactional
    public void deleteDevice(UUID deviceId, UUID userId) {
        log.info("Deleting device: {} for user: {}", deviceId, userId);

        Device device = deviceRepository.findById(deviceId)
                .orElseThrow(() -> new DeviceNotFoundException(deviceId));

        if (!device.getUserId().equals(userId)) {
            throw new SecurityException("User does not own this device");
        }

        deviceRepository.delete(device);

        DeviceEvent event = eventFactory.createDeviceDeletedEvent(
            deviceId, userId, device.getName()
        );
        eventPublisher.publishDeviceDeleted(event);
    }
}
```

Шаг 7. Настройка application.yml

```
spring:
```

```
rabbitmq:
  host: localhost
  port: 5672
  username: admin
  password: admin123
  connection-timeout: 5000
```

Остальные настройки (БД, JPA и т.д.) должны быть выполнены в предыдущих работах

Самостоятельное задание к заданию 2 для студента

Задание 2.1. Реализовать Event Publisher в Device Service

На основе разобранного примера, модифицируйте ваш Device Service (из лабораторной работы 2) для публикации событий. Если вы используете другой язык программирования, найдите соответствующую библиотеку для работы с RabbitMQ.

Требования:

- Событие должно публиковаться при любом изменении состояния устройства (активация, деактивация, изменение режима)
- Событие должно содержать всю необходимую информацию: ID устройства, ID пользователя, старое и новое состояние, timestamp
- Используйте JSON формат для сообщений
- Добавьте логирование публикации событий

Задание 2.2. Тестирование публикации

Напишите простой тест (юнит-тест или интеграционный тест), который проверяет, что при вызове метода activateDevice публикуется событие. Используйте моки для RabbitTemplate.

Задание 2.3. Обработка ошибок

Предложите и реализуйте стратегию обработки ошибок при публикации событий. Что делать, если RabbitMQ недоступен? Рассмотрите варианты:

- Retry с exponential backoff
- Сохранение событий в БД и периодическая отправка (Outbox pattern)
- Простое логирование ошибки

Задание 3. Создание Notification Service (Subscriber)

Разобранный пример (на Java со Spring Boot)

Шаг 1. Создание нового проекта

Создайте новый микросервис notification-service со структурой:

```
notification-service/
├── pom.xml
├── src/
│   ├── main/
│   │   ├── java/com/smarthome/notification/
│   │   │   ├── NotificationServiceApplication.java
│   │   │   ├── config/
│   │   │   │   └── RabbitConfig.java
│   │   │   ├── consumer/
│   │   │   │   └── DeviceEventConsumer.java
│   │   │   ├── model/
│   │   │   │   └── DeviceEvent.java
│   │   │   ├── service/
│   │   │   │   └── NotificationService.java
│   │   │   └── repository/
│   │   │       └── NotificationLogRepository.java
│   │   └── resources/
│   │       └── application.yml
│   └── test/
```

Шаг 2. Добавление зависимостей (pom.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.smarthome</groupId>
    <artifactId>notification-service</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.14</version>
    </parent>

    <properties>
```

```
    <java.version>11</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Шаг 3. Конфигурация RabbitMQ (RabbitConfig.java)

```
package com.smarthome.notification.config;

import org.springframework.amqp.core.*;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;
```

```java
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitConfig {

    public static final String DEVICE_EVENTS_EXCHANGE = "device.events";
    public static final String NOTIFICATION_QUEUE = "notification.service.queue";

    @Bean
    public Queue notificationQueue() {
        return QueueBuilder.durable(NOTIFICATION_QUEUE).build();
    }

    @Bean
    public TopicExchange deviceEventsExchange() {
        return ExchangeBuilder.topicExchange(DEVICE_EVENTS_EXCHANGE)
                .durable(true)
                .build();
    }

    @Bean
    public Binding notificationBinding() {
        return BindingBuilder.bind(notificationQueue())
                .to(deviceEventsExchange())
                .with("device.state.");
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public SimpleMessageListenerContainer messageListenerContainer(
            ConnectionFactory connectionFactory,
            MessageListenerAdapter listenerAdapter) {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        container.setQueueNames(NOTIFICATION_QUEUE);
        container.setMessageListener(listenerAdapter);
        return container;
    }

    @Bean
    public MessageListenerAdapter listenerAdapter(DeviceEventConsumer consumer) {
        return new MessageListenerAdapter(consumer, "receiveMessage");
    }
```

```
}
```

Шаг 4. Модель события (DeviceEvent.java)

```
package com.smarthome.notification.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.Instant;
import java.util.UUID;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class DeviceEvent {
    private String eventId;
    private String eventType;
    private UUID deviceId;
    private UUID userId;
    private String deviceName;
    private String oldState;
    private String newState;
    private Instant timestamp;
    private Object additionalData;
}
```

Шаг 5. Сущность для логирования (NotificationLog.java)

```
package com.smarthome.notification.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.;
import java.time.Instant;
import java.util.UUID;

@Entity
@Table(name = "notification_logs")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class NotificationLog {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
```

```
    private UUID id;

    @Column(name = "event_id", nullable = false)
    private String eventId;

    @Column(name = "event_type", nullable = false)
    private String eventType;

    @Column(name = "device_id", nullable = false)
    private UUID deviceId;

    @Column(name = "user_id", nullable = false)
    private UUID userId;

    @Column(name = "device_name")
    private String deviceName;

    @Column(name = "old_state")
    private String oldState;

    @Column(name = "new_state")
    private String newState;

    @Column(name = "received_at", nullable = false)
    private Instant receivedAt;

    @Column(name = "processed", nullable = false)
    private boolean processed;

    @Column(name = "notification_type")
    private String notificationType; // EMAIL, PUSH, SMS

    @Column(name = "sent_at")
    private Instant sentAt;

    @Column(name = "error_message")
    private String errorMessage;
}
```

Шаг 6. Репозиторий (NotificationLogRepository.java)

```
package com.smarthome.notification.repository;

import com.smarthome.notification.model.NotificationLog;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.UUID;
```

```
@Repository
public interface NotificationLogRepository extends JpaRepository<NotificationLog, UUID> {
    List<NotificationLog> findByUserIdOrderByReceivedAtDesc(UUID userId);
    List<NotificationLog> findByProcessedFalse();
}
```

Шаг 7. Consumer событий (DeviceEventConsumer.java)

```
package com.smarthome.notification.consumer;

import com.smarthome.notification.model.DeviceEvent;
import com.smarthome.notification.service.NotificationService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component
@RequiredArgsConstructor
public class DeviceEventConsumer {

    private final NotificationService notificationService;

    public void receiveMessage(DeviceEvent event) {
        log.info("Received event: {} for device: {}", event.getEventType(), event.getDeviceId());

        try {
            switch (event.getEventType()) {
                case "DEVICE_STATE_CHANGED":
                    notificationService.processDeviceStateChanged(event);
                    break;
                case "DEVICE_REGISTERED":
                    notificationService.processDeviceRegistered(event);
                    break;
                case "DEVICE_DELETED":
                    notificationService.processDeviceDeleted(event);
                    break;
                default:
                    log.warn("Unknown event type: {}", event.getEventType());
            }
        } catch (Exception e) {
            log.error("Error processing event: {}", e.getMessage(), e);
            // Здесь можно реализовать механизм повторных попыток
        }
    }
}
```

Шаг 8. Сервис уведомлений (NotificationService.java)

```
package com.smarthome.notification.service;

import com.smarthome.notification.model.DeviceEvent;
import com.smarthome.notification.model.NotificationLog;
import com.smarthome.notification.repository.NotificationLogRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.Instant;

@Slf4j
@Service
@RequiredArgsConstructor
public class NotificationService {

    private final NotificationLogRepository logRepository;

    @Transactional
    public void processDeviceStateChanged(DeviceEvent event) {
        log.info("Processing state change notification for device: {}", event.getDeviceId());

        // Сохраняем в лог
        NotificationLog notificationLog = new NotificationLog();
        notificationLog.setEventId(event.getEventId());
        notificationLog.setEventType(event.getEventType());
        notificationLog.setDeviceId(event.getDeviceId());
        notificationLog.setUserId(event.getUserId());
        notificationLog.setDeviceName(event.getDeviceName());
        notificationLog.setOldState(event.getOldState());
        notificationLog.setNewState(event.getNewState());
        notificationLog.setReceivedAt(Instant.now());

        // Определяем тип уведомления в зависимости от события
        if ("ACTIVE".equals(event.getNewState())) {
            notificationLog.setNotificationType("PUSH");
            // Здесь была бы отправка push-уведомления
            log.info("📱 Sending PUSH notification: Device {} is now ACTIVE",
event.getDeviceName());
        } else if ("INACTIVE".equals(event.getNewState())) {
            notificationLog.setNotificationType("EMAIL");
            // Здесь была бы отправка email
            log.info("📧 Sending EMAIL notification: Device {} is now INACTIVE",
event.getDeviceName());
        }
```

```
        notificationLog.setProcessed(true);
        notificationLog.setSentAt(Instant.now());

        logRepository.save(notificationLog);

        log.info("Notification logged successfully");
    }

    @Transactional
    public void processDeviceRegistered(DeviceEvent event) {
        log.info("Processing device registration notification for device: {}", event.getDeviceId());

        NotificationLog notificationLog = new NotificationLog();
        notificationLog.setEventId(event.getEventId());
        notificationLog.setEventType(event.getEventType());
        notificationLog.setDeviceId(event.getDeviceId());
        notificationLog.setUserId(event.getUserId());
        notificationLog.setDeviceName(event.getDeviceName());
        notificationLog.setNewState(event.getNewState());
        notificationLog.setReceivedAt(Instant.now());
        notificationLog.setNotificationType("WELCOME_EMAIL");

        // Имитация отправки приветственного email
        log.info("📧 Sending welcome email for new device: {}", event.getDeviceName());

        notificationLog.setProcessed(true);
        notificationLog.setSentAt(Instant.now());

        logRepository.save(notificationLog);
    }

    @Transactional
    public void processDeviceDeleted(DeviceEvent event) {
        log.info("Processing device deletion notification for device: {}", event.getDeviceId());

        NotificationLog notificationLog = new NotificationLog();
        notificationLog.setEventId(event.getEventId());
        notificationLog.setEventType(event.getEventType());
        notificationLog.setDeviceId(event.getDeviceId());
        notificationLog.setUserId(event.getUserId());
        notificationLog.setDeviceName(event.getDeviceName());
        notificationLog.setReceivedAt(Instant.now());
        notificationLog.setNotificationType("ALERT");

        // Имитация отправки уведомления об удалении
        log.info("⚠️ Sending alert: Device {} has been removed", event.getDeviceName());

        notificationLog.setProcessed(true);
        notificationLog.setSentAt(Instant.now());
```

```
        logRepository.save(notificationLog);
    }
}
```

Шаг 9. REST Controller для просмотра логов (NotificationController.java)

```
package com.smarthome.notification.controller;

import com.smarthome.notification.model.NotificationLog;
import com.smarthome.notification.repository.NotificationLogRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.;

import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/api/v1/notifications")
@RequiredArgsConstructor
public class NotificationController {

    private final NotificationLogRepository logRepository;

    @GetMapping("/users/{userId}")
    public List<NotificationLog> getUserNotifications(@PathVariable UUID userId) {
        return logRepository.findByUserIdOrderByReceivedAtDesc(userId);
    }

    @GetMapping
    public List<NotificationLog> getAllNotifications() {
        return logRepository.findAll();
    }
}
```

Шаг 10. Настройка application.yml

```
spring:
  application:
    name: notification-service

  rabbitmq:
    host: localhost
    port: 5672
    username: admin
```

```
  password: admin123

 datasource:
  url: jdbc:postgresql://localhost:5433/notification_db
  username: postgres
  password: postgres
  driver-class-name: org.postgresql.Driver

 jpa:
  hibernate:
   ddl-auto: update
  show-sql: true
  properties:
   hibernate:
    dialect: org.hibernate.dialect.PostgreSQLDialect
    format_sql: true

server:
 port: 8083

logging:
 level:
  com.smarthome.notification: DEBUG
  org.springframework.amqp: INFO
```

Самостоятельное задание к заданию 3

Задание 3.1. Реализовать Notification Service

Создайте сервис уведомлений на основе разобранного примера. Если вы используете другой язык программирования, реализуйте аналогичную функциональность.

Требования:

- Сервис должен подписываться на события от Device Service (очередь notification.service.queue)

- При получении события сервис должен логировать его в файл или в БД

- Добавьте простую имитацию отправки уведомлений (push, email, sms) — просто вывод в лог с эмодзи

- Создайте REST эндпоинт для просмотра истории уведомлений

Задание 3.2. Модифицировать Telemetry Service как подписчик

Если у вас уже реализован Telemetry Service из лабораторной работы 2, модифицируйте его, чтобы он также подписывался на события Device Service (например, для записи истории изменений состояний).

Задание 3.3. Демонстрация слабой связанности

Напишите небольшой скрипт или последовательность действий, демонстрирующую, что:

1. Device Service работает и публикует события, даже если Notification Service отключен
2. При запуске Notification Service он получает события, которые были накоплены в очереди (если очередь настроена как durable)

Задание 3.4. Эксперимент с типами exchange

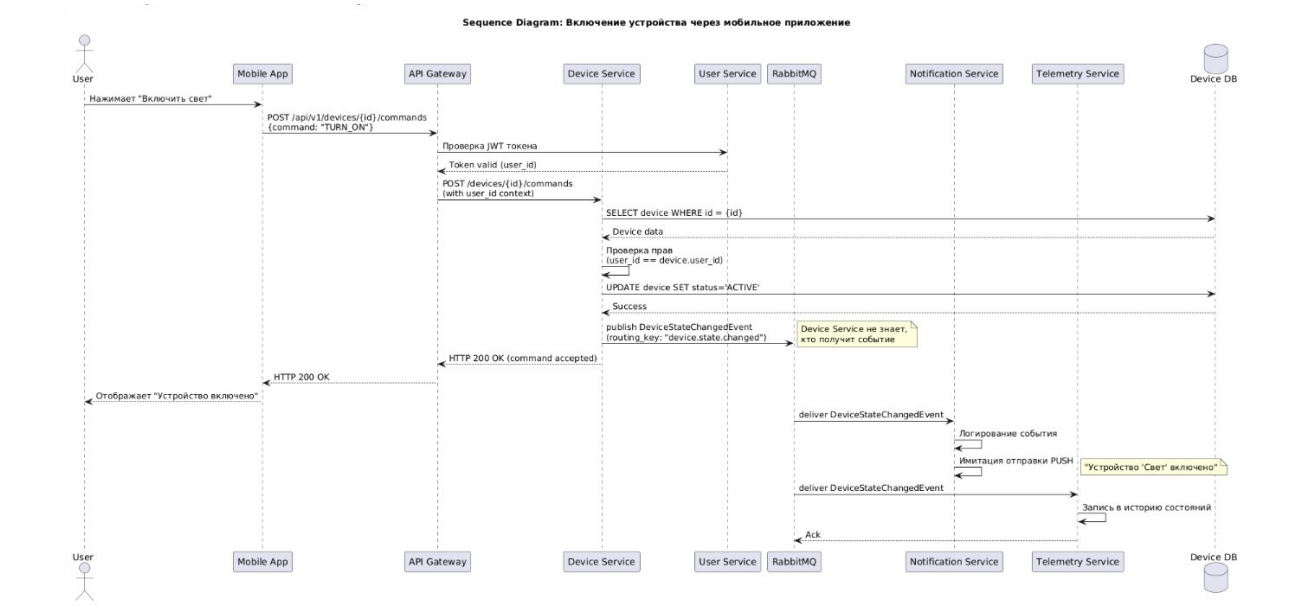
Измените тип exchange с topic на fanout и пронаблюдайте разницу в поведении. Опишите, в каких сценариях какой тип exchange лучше использовать.

Задание 4. Диаграмма последовательности

Создайте файл sequence-diagram.puml для процесса включения устройства:

```
@startuml
title Sequence Diagram: Включение устройства через мобильное приложение

actor User as user
participant "Mobile App" as mobile
participant "API Gateway" as gateway
participant "Device Service" as device
participant "User Service" as user_svc
participant "RabbitMQ" as rabbit
participant "Notification Service" as notif
participant "Telemetry Service" as telemetry
database "Device DB" as device_db

user -> mobile: Нажимает "Включить свет"
mobile -> gateway: POST /api/v1/devices/{id}/commands\n{command: "TURN_ON"}
gateway -> user_svc: Проверка JWT токена
user_svc --> gateway: Token valid (user_id)

gateway -> device: POST /devices/{id}/commands\n(with user_id context)
```

```
device -> device_db: SELECT device WHERE id = {id}
device_db --> device: Device data

device -> device: Проверка прав\n(user_id == device.user_id)

device -> device_db: UPDATE device SET status='ACTIVE'
device_db --> device: Success

device -> rabbit: publish DeviceStateChangedEvent\n(routing_key: "device.state.changed")
note right: Device Service не знает,\nкто получит событие

device --> gateway: HTTP 200 OK (command accepted)
gateway --> mobile: HTTP 200 OK
mobile --> user: Отображает "Устройство включено"

rabbit -> notif: deliver DeviceStateChangedEvent
notif -> notif: Логирование события
notif -> notif: Имитация отправки PUSH
note right: "Устройство 'Свет' включено"

rabbit -> telemetry: deliver DeviceStateChangedEvent
telemetry -> telemetry: Запись в историю состояний
telemetry --> rabbit: Ack

@enduml
```



Самостоятельное задание к заданию 4

Задание 4.1. Создать диаграмму последовательности

Создайте диаграмму последовательности для процесса регистрации нового устройства.

Сценарий:

1. Пользователь добавляет новое устройство через мобильное приложение
2. API Gateway проверяет токен
3. Device Service сохраняет устройство в БД
4. Device Service публикует событие DEVICE_REGISTERED
5. Notification Service получает событие и отправляет приветственное уведомление
6. (Опционально) Telemetry Service создает запись для нового устройства

Задание 4.2. Создать диаграмму активностей

Создайте диаграмму активностей (Activity Diagram), показывающую flow обработки события от публикации до доставки всем подписчикам, включая обработку ошибок.

Задание 4.3. Документирование событий

Создайте таблицу со всеми событиями, которые публикуются в системе:

Событие	Описание	Routing Key	Публикует	Подписчики	Пример данных
DEVICE_STATE_CHANG	Изменени	device.state.chang	Device	Notification	{deviceI

ED	e состояния устройства	ed	Service	, Telemetry	d, userId, oldState, newState }
...	...	...	...	...	...

Требования к отчету

Отчет по лабораторной работе должен содержать:

1. Цель работы
2. Задание 1: Скриншоты веб-интерфейса RabbitMQ, ответы на вопросы
3. Задание 2: Код модифицированного Device Service (ключевые фрагменты)
4. Задание 3: Код Notification Service (ключевые фрагменты)
5. Задание 4: Диаграммы последовательности и активностей
6. Демонстрация работы: Скриншоты логов, показывающие:
   - Публикацию события в Device Service
   - Получение события в Notification Service
   - Работу при отключенном подписчике (накопление в очереди)
7. Выводы: Что нового узнали, с какими трудностями столкнулись, как обеспечили слабую связанность

5. Критерии оценки

Критерий	**Отлично (2)**	**Хорошо (1)**	**Удовл. (0.5)**	**Неуд. (0)**
Задание 1	RabbitMQ настроен, созданы exchange/очереди, есть исследование	RabbitMQ запущен, но не все элементы созданы	RabbitMQ запущен, но не настроен	Не выполнено
Задание 2	Device Service публикует все события, обработка ошибок, тесты	События публикуются, но нет обработки ошибок	Только базовая публикация	Не выполнено
Задание 3	Notification	Сервис получает	Только	Не выполнено

	Service полноценный, с БД и REST API	события и логирует	получение без логирования	
Задание 4	Диаграммы полные, соответствуют коду, аккуратно оформлены	Диаграммы есть, но есть несоответствия	Схематичные наброски	Отсутствуют
Демонстрация слабой связанности	Показаны все сценарии (с подписчиком и без)	Показан базовый сценарий	Только логи	Не показано
ИТОГО	10	5-7	3-4	0-2

6. Дополнительные задания повышенного уровня

Задание 5. Реализация Outbox Pattern

Изучите паттерн Transactional Outbox. Реализуйте механизм, при котором события сначала сохраняются в БД в той же транзакции, что и изменение состояния устройства, а затем отдельный процесс отправляет их в RabbitMQ. Это гарантирует, что событие не потеряется даже при падении RabbitMQ.

Задание 6. Dead Letter Queue (DLQ)

Настройте Dead Letter Queue для обработки сообщений, которые не могут быть обработаны (например, после нескольких неудачных попыток). Реализуйте механизм повторных попыток с exponential backoff.

Задание 7. Метрики и мониторинг

Добавьте в сервисы метрики:
- Количество опубликованных событий
- Количество обработанных событий
- Время обработки
- Ошибки

Используйте Micrometer + Prometheus + Grafana.

Задание 8. Сравнение с Kafka

Разверните Kafka (в дополнение к RabbitMQ) и реализуйте того же подписчика на Kafka. Сравните производительность, сложность настройки и удобство использования. Напишите отчет о сравнении.

Приложение 1. Установка и настройка Docker

1. Проверка наличия Docker

Прежде чем устанавливать Docker, проверим, установлен ли он уже:

```
bash
docker --version
docker-compose --version
```

Если команды не найдены или версии устарели, следуйте инструкциям ниже для вашей ОС.

2. Установка Docker на Ubuntu/Debian

2.1. Ubuntu 20.04/22.04/24.04

Шаг 1. Обновление пакетов и установка зависимостей

```
sudo apt-get update
sudo apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common \
    gnupg \
    lsb-release
```

Шаг 2. Добавление официального GPG-ключа Docker

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Шаг 3. Добавление репозитория Docker

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Шаг 4. Установка Docker Engine

```
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Шаг 5. Проверка установки

```
sudo docker run hello-world
```

2.2. Debian 11/12

Шаг 1. Установка зависимостей

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg lsb-release
```

Шаг 2. Добавление GPG-ключа

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Шаг 3. Добавление репозитория

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Шаг 4. Установка Docker

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

3. Установка на Windows

Вариант 1. Docker Desktop with WSL2 (рекомендуется)

1. Включите WSL2 (если не включено):

Запустите PowerShell как администратор

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

Перезагрузите компьютер

2. Установите ядро WSL2:

Скачайте и установите [пакет обновления ядра WSL2](https://learn.microsoft.com/ru-ru/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package)

3. Установите Docker Desktop:

- Скачайте [Docker Desktop for Windows](https://www.docker.com/products/docker-desktop/)
- Запустите установщик
- При установке отметьте "Use WSL 2 instead of Hyper-V"
- После установки перезагрузите компьютер

4. Запустите Docker Desktop и дождитесь появления значка в трее (зеленый статус)

4. Настройка после установки

4.1. Запуск Docker как службы

Включение автозапуска

```
sudo systemctl enable docker
```

Запуск Docker

```
sudo systemctl start docker
```

Проверка статуса

```
sudo systemctl status docker
```

5.2. Добавление пользователя в группу docker (без sudo)

По умолчанию команды Docker требуют прав суперпользователя. Чтобы запускать Docker от имени обычного пользователя:

sudo usermod -aG docker $USER

Важно: После выполнения команды необходимо выйти из системы и зайти снова (или перезагрузить компьютер), чтобы изменения вступили в силу.

Проверка:

После перелогина

docker info

5.3. Проверка установки

Версия Docker

docker --version

Версия Docker Compose

docker compose version

Информация о системе

docker info

Запуск тестового контейнера

docker run hello-world

5.4. Настройка зеркал (registry mirrors)

Для ускорения загрузки образов можно настроить зеркала Docker Hub:

sudo nano /etc/docker/daemon.json

Добавьте:

{

"registry-mirrors": ["https://mirror.gcr.io/",
"https://dockerhub.timeweb.cloud"]

}

Перезапустите Docker:

sudo systemctl daemon-reload

sudo systemctl restart docker

6. Установка Docker Compose

6.1. Docker Compose Plugin (рекомендуется)

При установке Docker через официальные репозитории (как описано выше) плагин `docker-compose-plugin` уже установлен.

Проверка:

docker compose version

6.2. Отдельная установка Docker Compose

Если по какой-то причине плагин не установлен:

Скачивание последней версии

sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Установка прав на выполнение

sudo chmod +x /usr/local/bin/docker-compose

Проверка

docker-compose --version

8. Проверка для лабораторной работы №3

После установки Docker выполните проверочный запуск RabbitMQ, который потребуется в ЛР3:

Запуск RabbitMQ

docker run -d --name rabbitmq-test -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management

Проверка, что контейнер запущен

docker ps

Открытие веб-интерфейса (в браузере)

http://localhost:15672

Логин: guest

Пароль: guest

Остановка и удаление тестового контейнера

docker stop rabbitmq-test

docker rm rabbitmq-test

Если все шаги выполнены успешно, вы готовы к выполнению лабораторной работы №3!

7. Устранение неполадок

Проблема: Permission denied при запуске docker команд

Решение: Добавьте пользователя в группу docker и перелогиньтесь

Проблема: Конфликт сетей с VPN

Решение: Отключите VPN перед запуском Docker, затем подключите снова

Проблема: Не удается подключиться к RabbitMQ

Решение: Проверьте, что контейнер запущен: docker ps. Проверьте порты: netstat -tulpn | grep 5672

Проблема: Docker не запускается на Windows

Решение: Убедитесь, что виртуализация включена в BIOS и WSL2 установлен корректно

Полезные команды Docker

Команда	**Описание**
docker ps	Список запущенных контейнеров
docker ps -a	Список всех контейнеров
docker images	Список образов
docker pull <image>	Скачать образ
docker build -t <name> .	Собрать образ из Dockerfile
docker run <image>	Запустить контейнер
docker stop <container>	Остановить контейнер
docker rm <container>	Удалить контейнер
docker rmi <image>	Удалить образ
docker logs <container>	Просмотр логов контейнера
docker exec -it <container> bash	Подключиться к контейнеру
docker system prune	Очистка неиспользуемых данных

После выполнения всех шагов у вас должна быть рабочая среда Docker, готовая к выполнению лабораторной работы №3. Если возникнут проблемы, обратитесь к разделу "Устранение неполадок" или к официальной документации Docker.

Приложение 2. Установка и настройка RabbitMQ

2. Установка RabbitMQ через Docker

2.1. Предварительные требования

Перед установкой убедитесь, что Docker установлен и работает:

```
docker --version
docker-compose --version
```

Если Docker не установлен, следуйте инструкции из предыдущего приложения.

2.2. Быстрый запуск (для тестирования)

Самая быстрая команда для запуска RabbitMQ:

```
docker run -d \
  --name rabbitmq-test \
  -p 5672:5672 \
  -p 15672:15672 \
  rabbitmq:3.12-management
```

Что происходит:
- -d - запуск в фоновом режиме
- --name rabbitmq-test - имя контейнера
- -p 5672:5672 - порт для AMQP (подключение приложений)
- -p 15672:15672 - порт для веб-интерфейса управления
- rabbitmq:3.12-management - образ с включенным плагином управления

Проверка:

Проверить, что контейнер запущен
```
docker ps
```

Посмотреть логи
```
docker logs rabbitmq-test
```

Остановка и удаление:
```
docker stop rabbitmq-test
docker rm rabbitmq-test
```

2.3. Постоянный запуск через Docker Compose

Создайте файл docker-compose.yml для RabbitMQ:

```
version: '3.8'

services:
  rabbitmq:
    image: rabbitmq:3.12-management
    container_name: smart-home-rabbitmq
    hostname: rabbitmq
    ports:
      - "5672:5672"       AMQP protocol port (для приложений)
      - "15672:15672"     Management UI port (веб-интерфейс)
    environment:
      - RABBITMQ_DEFAULT_USER=admin
      - RABBITMQ_DEFAULT_PASS=admin123
      - RABBITMQ_DEFAULT_VHOST=/
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
      - rabbitmq_log:/var/log/rabbitmq
    networks:
      - smart-home-net
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "ping"]
      interval: 30s
      timeout: 10s
      retries: 5

networks:
  smart-home-net:
    driver: bridge

volumes:
  rabbitmq_data:
  rabbitmq_log:
```

Запуск:
```
docker-compose up -d
```

Проверка:
```
docker-compose ps
```

docker-compose logs

Остановка:
docker-compose down
Для остановки с удалением томов (все данные будут потеряны):
docker-compose down -v

2.4. Настройка прав доступа (если нужно изменить)

Если вы забыли задать пароль при запуске или хотите добавить пользователя:

Зайти в контейнер
docker exec -it smart-home-rabbitmq bash

Создать пользователя (внутри контейнера)
rabbitmqctl add_user newuser newpassword
rabbitmqctl set_user_tags newuser administrator
rabbitmqctl set_permissions -p / newuser "." "." "."

Выйти из контейнера
exit

2.5. Сохранение данных (важно!)

В docker-compose.yml выше используются volumes:
- rabbitmq_data — хранит очереди, сообщения, определения
- rabbitmq_log — хранит логи

Это гарантирует, что при перезапуске контейнера данные не потеряются .

3. Установка RabbitMQ напрямую на Linux (Ubuntu/Debian)

Если Docker недоступен, можно установить RabbitMQ напрямую.

3.1. Установка на Ubuntu 20.04/22.04/24.04

Шаг 1. Установка Erlang (зависимость RabbitMQ)

sudo apt update

sudo apt install -y erlang-nox

Шаг 2: Добавление репозитория RabbitMQ

Добавление GPG-ключа
curl -fsSL https://github.com/rabbitmq/signing-keys/releases/download/2.0/rabbitmq-release-signing-key.asc | sudo apt-key add -

Добавление репозитория
sudo tee /etc/apt/sources.list.d/rabbitmq.list << EOF
deb https://dl.cloudsmith.io/public/rabbitmq/rabbitmq-server/deb/ubuntu $(lsb_release -sc) main
EOF

Шаг 3. Установка RabbitMQ

sudo apt update
sudo apt install -y rabbitmq-server

Шаг 4. Запуск и включение автозапуска

sudo systemctl start rabbitmq-server
sudo systemctl enable rabbitmq-server
sudo systemctl status rabbitmq-server

Шаг 5. Включение плагина управления

sudo rabbitmq-plugins enable rabbitmq_management

Шаг 6. Настройка пользователя

Создание администратора
sudo rabbitmqctl add_user admin admin123
sudo rabbitmqctl set_user_tags admin administrator
sudo rabbitmqctl set_permissions -p / admin "." "." "."

Удаление гостевого пользователя (рекомендуется для безопасности)
sudo rabbitmqctl delete_user guest

Шаг 7. Перезапуск для применения настроек

sudo systemctl restart rabbitmq-server

4. Установка RabbitMQ на Windows (прямая)

4.1. Установка Erlang

1. Перейдите на [официальный сайт Erlang](https://www.erlang.org/downloads)
2. Скачайте установщик для Windows (OTP 25 или 26)
3. Запустите установщик и следуйте инструкциям
4. Важно: Запомните путь установки (обычно C:\Program Files\Erlang OTP)

4.2. Установка RabbitMQ

1. Перейдите на [официальный сайт RabbitMQ](https://www.rabbitmq.com/download.html)
2. Скачайте установщик для Windows (rabbitmq-server-3.12.x.exe)
3. Запустите установщик от имени администратора
4. Следуйте инструкциям установщика

4.3. Включение плагина управления

Шаг 1. Откройте командную строку от имени администратора

Шаг 2. Перейдите в каталог sbin RabbitMQ:

cd "C:\Program Files\RabbitMQ Server\rabbitmq_server-3.12.x\sbin"

Шаг 3. Включите плагин управления:

rabbitmq-plugins.bat enable rabbitmq_management

Шаг 4. Перезапустите службу:
net stop RabbitMQ && net start RabbitMQ

Или через "Службы" (services.msc):
- Найдите службу "RabbitMQ"

- Нажмите "Перезапустить"

5. Проверка установки

5.1. Проверка через веб-интерфейс

1. Откройте браузер и перейдите по адресу: http://localhost:15672
2. Введите учетные данные:
   - Если через Docker с настройками выше: admin / admin123
   - Если через прямую установку без настроек: guest / guest
   - Если создавали своего пользователя: ваш логин/пароль

3. Вы должны увидеть дашборд RabbitMQ Management UI

5.2. Проверка через командную строку

Для Docker:

Проверка статуса контейнера
```
docker ps | grep rabbitmq
```

Проверка логов
```
docker logs smart-home-rabbitmq
```

Проверка здоровья через rabbitmqctl
```
docker exec smart-home-rabbitmq rabbitmqctl status
```

Для Linux (прямая установка):

```
sudo rabbitmqctl status
sudo rabbitmqctl list_users
sudo rabbitmqctl list_queues
```

Для Windows:

```
cd "C:\Program Files\RabbitMQ Server\rabbitmq_server-3.12.x\sbin"
rabbitmqctl.bat status
rabbitmqctl.bat list_users
```

5.3. Проверка подключения из приложения

Создайте простой тестовый скрипт на Python (если есть Python):

```
import pika

 Подключение
credentials = pika.PlainCredentials('admin', 'admin123')
parameters = pika.ConnectionParameters('localhost', 5672, '/', credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()

 Создание очереди
channel.queue_declare(queue='test_queue')

 Публикация сообщения
channel.basic_publish(exchange='', routing_key='test_queue', body='Hello World!')
print(" [x] Sent 'Hello World!'")

 Получение сообщения
method_frame, header_frame, body = channel.basic_get(queue='test_queue')
if method_frame:
    print(f" [x] Received {body.decode()}")
    channel.basic_ack(method_frame.delivery_tag)

connection.close()
```

6. Настройка для лабораторной работы №3

6.1. Создание exchange и очередей

Для лабораторной работы 3 потребуется создать:
- Exchange: device.events (тип topic)
- Очередь: notification.service.queue
- Очередь: telemetry.service.queue

Через веб-интерфейс:
1. Вкладка Exchanges → Add a new exchange
   - Name: device.events
   - Type: topic
   - Durability: Durable
   - → Add exchange

2. Вкладка Queues → Add a new queue
- Name: notification.service.queue
- Durability: Durable
- → Add queue

3. Вкладка Queues → Add a new queue
- Name: telemetry.service.queue
- Durability: Durable
- → Add queue

4. Для каждой очереди создать binding:
- Зайти в очередь → вкладка Bindings
- From exchange: device.events
- Routing key: для notification: device.state., для telemetry: device.telemetry.
- → Bind

Через код (при старте приложения):
В каждом сервисе-подписчике можно добавить код, создающий очередь и binding, если их нет .

6.2. Настройка безопасности

Для учебного проекта достаточно базовой настройки:
- Создать администратора (admin/admin123)
- Удалить гостевого пользователя (guest) или оставить только для localhost

6.3. Проверка персистентности

Проверьте, что сообщения сохраняются при перезапуске:
1. Отправьте сообщение в очередь
2. Перезапустите RabbitMQ: docker restart smart-home-rabbitmq
3. Проверьте, что сообщение осталось в очереди

7. Устранение неполадок

Проблема: Не удается подключиться к RabbitMQ

Решение:

Проверить, слушает ли RabbitMQ порты
netstat -tulpn | grep -E '5672|15672'

Для Docker: проверить логи
docker logs smart-home-rabbitmq

Проверить файрвол
sudo ufw status
Если включен, разрешить порты
sudo ufw allow 5672
sudo ufw allow 15672

Проблема: Ошибка аутентификации

Решение:

Сбросить пароль администратора
docker exec smart-home-rabbitmq rabbitmqctl change_password admin newpassword

Или создать нового пользователя
docker exec smart-home-rabbitmq rabbitmqctl add_user student student123
docker exec smart-home-rabbitmq rabbitmqctl set_user_tags student administrator
docker exec smart-home-rabbitmq rabbitmqctl set_permissions -p / student "." "." "."

Проблема: Контейнер не запускается

Решение:

Проверить, не заняты ли порты
sudo lsof -i :5672
sudo lsof -i :15672

Если заняты, остановить процессы или изменить порты в docker-compose.yml
Например, использовать 5673:5672 и 15673:15672

Проблема: Потеря данных после перезапуска

Решение: Убедитесь, что в docker-compose.yml есть volumes :

volumes:
  - rabbitmq_data:/var/lib/rabbitmq

Проблема: RabbitMQ потребляет слишком много памяти

Решение: Ограничить память :
yaml
environment:
  - RABBITMQ_VM_MEMORY_HIGH_WATERMARK=0.5 50% от доступной памяти

8. Шпаргалка по командам RabbitMQ

Управление через Docker

bash
Запуск
docker-compose up -d

Остановка
docker-compose down

Просмотр логов
docker-compose logs -f

Зайти в контейнер
docker exec -it smart-home-rabbitmq bash

Управление через rabbitmqctl

bash
Статус
rabbitmqctl status

Список пользователей
rabbitmqctl list_users

Добавить пользователя
rabbitmqctl add_user username password
rabbitmqctl set_user_tags username administrator

rabbitmqctl set_permissions -p / username "." "." "."

Список очередей
rabbitmqctl list_queues

Список очередей с подробностями
rabbitmqctl list_queues name messages consumers

Очистить очередь
rabbitmqctl purge_queue queue_name

Удалить очередь
rabbitmqctl delete_queue queue_name

Создать exchange
rabbitmqadmin declare exchange name=device.events type=topic durable=true

Создать очередь и binding
rabbitmqadmin declare queue name=notification.service.queue durable=true
rabbitmqadmin declare binding source=device.events destination=notification.service.queue routing_key="device.state."

Полезные ссылки веб-интерфейса

- Обзор: http://localhost:15672//
- Очереди: http://localhost:15672//queues
- Обменники: http://localhost:15672//exchanges
- Подключения: http://localhost:15672//connections

После выполнения всех шагов у вас должен быть запущен и настроен RabbitMQ, готовый к использованию в лабораторной работе №3.

Чек-лист готовности к ЛР3:

- RabbitMQ запущен (через Docker или напрямую)
- Веб-интерфейс доступен по адресу http://localhost:15672
- Создан пользователь admin/admin123 (или свои учетные данные)
- Создан exchange device.events типа topic
- Созданы очереди notification.service.queue и telemetry.service.queue

- Настроены bindings с routing keys device.state. и device.telemetry.
- Проверена персистентность (сообщения сохраняются после перезапуска)
- Из кода на вашем языке программирования получается подключиться к RabbitMQ

Если возникают проблемы, обратитесь к разделу "Устранение неполадок" или задайте вопрос преподавателю.

Лабораторная работа №4.

Контейнеризация и оркестрация (Docker & Docker Compose)

Цель работы: Научиться упаковывать разработанные сервисы в Docker-образы и оркестрировать всю экосистему (включая базы данных и брокер сообщений) с помощью Docker Compose. Студенты должны освоить принципы контейнеризации и понимать, как настраивать взаимодействие между контейнерами в изолированной сети.

1. Теоретическое обоснование

1.1. Docker: Основные концепции

Docker - это платформа для разработки, доставки и запуска приложений в изолированных контейнерах.

Ключевые понятия

Термин	Определение	Аналогия
Образ (Image)	Неизменяемый шаблон с кодом, зависимостями и инструкциями для запуска	Рецепт приготовления блюда
Контейнер (Container)	Запущенный экземпляр образа	Готовое блюдо, приготовленное по рецепту

Dockerfile	Файл с инструкциями по сборке образа	Сам рецепт
Слой (Layer)	Изменяемая часть образа, результат выполнения одной инструкции	Шаг в рецепте
Реестр (Registry)	Хранилище образов (например, Docker Hub)	Поваренная книга

Структура Docker образа (многослойность) (рисунок 1)

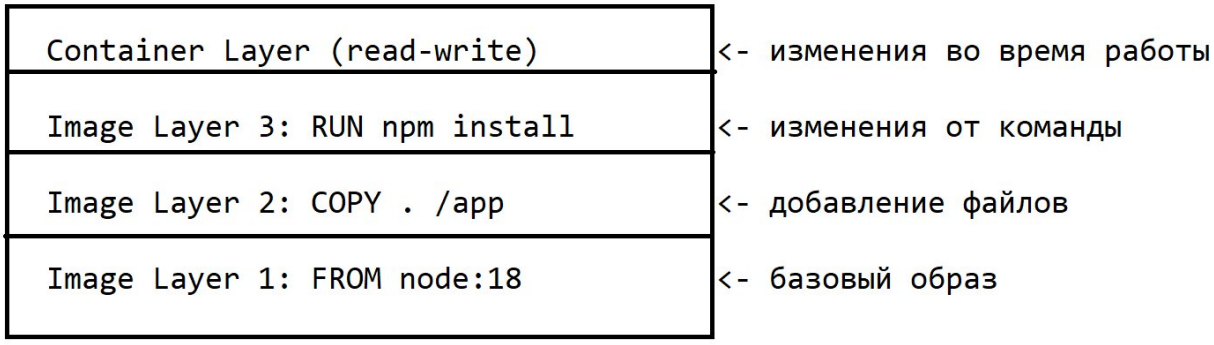


Рисунок 1

Принцип слоев:

- Каждая инструкция в Dockerfile создает новый слой
- Слои кэшируются: если слой не изменился, Docker использует кэш
- Это ускоряет повторные сборки

Dockerfile: Основные инструкции

Инструкция	**Описание**	**Пример**
FROM	Базовый образ	FROM openjdk:17-jdk-slim
WORKDIR	Рабочая директория	WORKDIR /app
COPY	Копирование файлов	COPY target/*.jar app.jar
RUN	Выполнение	RUN apt-get update && apt-get

	команды при сборке	install -y curl
ENV	Установка переменных окружения	ENV SPRING_PROFILES_ACTIVE=prod
EXPOSE	Открытие порта	EXPOSE 8080
CMD	Команда по умолчанию	CMD ["java", "-jar", "app.jar"]
ENTRYPOINT	Основная команда (не переопределяется)	ENTRYPOINT ["java", "-jar"]
HEALTHCHECK	Проверка здоровья контейнера	HEALTHCHECK CMD curl -f http://localhost:8080/actuator/health

1.2. Docker Compose

Docker Compose - инструмент для определения и запуска многоконтейнерных приложений.

Структура docker-compose.yml

```yaml
version: '3.8'  # Версия формата

services:       # Список сервисов (контейнеров)
  service-name:
    build: ./service-dir      # Сборка из Dockerfile
    image: name:tag           # Готовый образ
    ports:                    # Проброс портов "хост:контейнер"
      - "8080:8080"
    environment:              # Переменные окружения
      - DB_HOST=postgres
      - DB_PASSWORD=${DB_PASS} # Из .env файла
    env_file:                 # Файл с переменными
      - .env
    volumes:                  # Монтирование томов
      - data:/var/lib/data    # Именованный том
      - ./config:/app/config  # Bind mount (папка хоста)
    networks:                 # Подключение к сетям
      - app-network
    depends_on:               # Зависимости (порядок запуска)
      - postgres
```

```
    - rabbitmq
    healthcheck:              # Проверка здоровья
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 5

networks:       # Сети (для изоляции и общения)
  app-network:
    driver: bridge

volumes:        # Именованные тома (для данных)
  postgres_data:
  rabbitmq_data:
```

2.3. Healthchecks (Проверки здоровья)

Healthcheck - механизм Docker для определения, работает ли контейнер корректно.

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
  interval: 30s      # Частота проверки
  timeout: 10s       # Таймаут ожидания ответа
  retries: 3         # Количество попыток до признания unhealthy
  start_period: 40s  # Время игнорирования проверок при старте
```

Состояния здоровья:

- starting - начальный период (ждут start_period)
- healthy - контейнер работает нормально
- unhealthy - проверка провалилась, контейнер не готов

Важно: depends_on не ждет готовности БД, только запуска. Для ожидания готовности нужно использовать wait-for-it.sh или healthcheck в сочетании с condition: service_healthy (Compose V3+).

1.4. 12-Factor App: Конфигурация через окружение

12 факторов - методология для создания приложений, которые легко масштабировать и развертывать.

Фактор III: Конфигурация в окружении

1. Конфигурация приложения должна храниться в переменных окружения, а не в коде.

Плохо:

```
String dbUrl = "jdbc:postgresql://localhost:5432/devices";
String apiKey = "my-secret-key";
```

Хорошо:

```
String dbUrl = System.getenv("DB_URL");
String apiKey = System.getenv("API_KEY");
```

docker-compose.yml

```
# docker-compose.yml
environment:
  DB_URL: jdbc:postgresql://postgres-device:5432/device_db
  API_KEY: ${API_KEY}  # берется из .env
```

.env файл:

API_KEY=supersecretkey

DB_PASSWORD=postgres

1.5. Docker Networks

Docker предоставляет несколько типов сетей:

Тип	Описание	Когда использовать
bridge	Изолированная сеть по умолчанию	Для связи контейнеров на одном хосте
host	Контейнер использует сеть хоста	Для максимальной производительности
overlay	Объединение нескольких хостов	Docker Swarm,

		Kubernetes
none	Без сети	Для изолированных контейнеров

Внутри одной bridge-сети контейнеры видят друг друга по именам сервисов.

2. Практическая часть

2.1. Исходные данные (из ЛР1-3)

К началу ЛР4 у нас есть:

- Device Service - Java/Spring Boot приложение (основной стек курса)
- Notification Service - Java/Spring Boot приложение
- Telemetry Service - Java/Spring Boot приложение (опционально)
- User Service - Java/Spring Boot приложение
- RabbitMQ - брокер сообщений
- PostgreSQL - БД для сервисов
- TimescaleDB - БД для телеметрии

Примечание: Все примеры в этой лабораторной работе приведены для Java/Spring Boot - основного технологического стека курса. Если вы по каким-то причинам используете другой язык программирования (Python, Kotlin, C и т.д.), вам потребуется адаптировать примеры под свой стек. Принципы контейнеризации одинаковы для всех языков.

2.2. Задание 1. Создание Dockerfile для сервисов

Разобранный пример (Java/Spring Boot)

Структура проекта (рисунок 2):

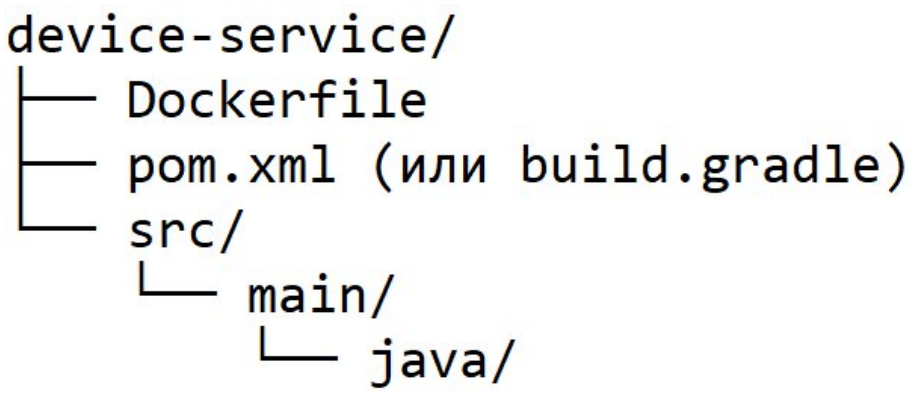


Рисунок 2

Вариант 1. Многостадийная сборка (рекомендуется для продакшн)

Этот подход создает два этапа:

- первый - сборка приложения,

- второй - только запуск.

Это позволяет получить минимальный по размеру итоговый образ.

dockerfile

```
# ============================================================
# СТАДИЯ 1: Сборка (build)
# ============================================================
FROM maven:3.8.4-openjdk-17-slim AS builder

WORKDIR /app

# Копируем файлы зависимостей (для кэширования)
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Копируем исходный код и собираем JAR
COPY src ./src
RUN mvn clean package -DskipTests

# ============================================================
# СТАДИЯ 2: Запуск (runtime)
# ============================================================
FROM openjdk:17-jdk-slim

# Установка curl для healthcheck
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
```

```
WORKDIR /app

# Копируем JAR из стадии сборки
COPY --from=builder /app/target/*.jar app.jar

# Создаем пользователя без прав root (безопасность)
RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser

# Открываем порт
EXPOSE 8080

# Проверка здоровья контейнера
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
  CMD curl -f http://localhost:8080/actuator/health || exit 1

# Запуск приложения
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Вариант 2. Простая сборка (для разработки)

Этот вариант проще, но образ получается больше. Подходит для учебных целей.

dockerfile

```
FROM openjdk:17-jdk-slim

WORKDIR /app

# Установка curl для healthcheck
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*

# Копируем готовый JAR (предварительно собранный локально командой mvn package)
COPY target/*.jar app.jar

# Создаем непривилегированного пользователя
RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
  CMD curl -f http://localhost:8080/actuator/health || exit 1
```

```
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Вариант 3. Для Gradle (если используется вместо Maven)

dockerfile

```
FROM gradle:7.6-jdk17 AS builder

WORKDIR /app
COPY build.gradle settings.gradle ./
COPY gradle gradle
RUN gradle dependencies --no-daemon

COPY src ./src
RUN gradle bootJar --no-daemon -x test

FROM openjdk:17-jdk-slim

RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
WORKDIR /app
COPY --from=builder /app/build/libs/*.jar app.jar

RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
  CMD curl -f http://localhost:8080/actuator/health || exit 1

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Сборка образа:

```
# Для Maven: сначала собираем JAR
cd device-service
mvn clean package

# Затем собираем Docker образ
docker build -t smart-home/device-service:1.0.0 .

# Проверка
docker images | grep device-service
```

Задания для самостоятельного выполнения

Задание 1.1. Создать Dockerfile для Device Service

На основе разобранного примера для Java/Spring Boot создайте Dockerfile для вашего Device Service.

Требования:

- Используйте многостадийную сборку (вариант 1) для получения минимального образа

- Добавьте HEALTHCHECK инструкцию (проверка эндпоинта /actuator/health)

- Создайте непривилегированного пользователя для запуска приложения (appuser)

- Оптимизируйте кэширование слоев (сначала копируйте pom.xml, потом код)

- Протестируйте сборку командой docker build -t device-service:test .

Задание 1.2. Создать Dockerfile для Notification Service

Аналогично создайте Dockerfile для Notification Service. Учтите, что этот сервис:

- Тоже использует Spring Boot

- Должен иметь эндпоинт /actuator/health (добавьте его в код, если еще нет)

Задание 1.3. Создать Dockerfile для User Service (если есть)

Создайте Dockerfile для User Service по тому же шаблону.

Задание 1.4. Создать Dockerfile для Telemetry Service (если есть)

Создайте Dockerfile для Telemetry Service.

Задание 1.5. Анализ размера образов

После сборки всех образов выполните:

```
docker images | grep smart-home
```

Заполните таблицу:

Сервис	Размер образа
device-service	
notification-service	
user-service	
telemetry-service	

Вопросы для размышления:

- Какой сервис получился самым большим? Почему?

- Как можно еще уменьшить размер образа?

Отобразите ответы на эти вопросы в выводах отчета

2.3. Задание 2. Создание docker-compose.yml

Разобранный пример (полная конфигурация)

Создайте файл docker-compose.yml в корне проекта:

```
version: '3.8'

# ============================================================
# СЕТИ
# ============================================================
networks:
  smart-home-net:
    driver: bridge
    name: smart-home-net

# ============================================================
# ТОМА ДЛЯ ДАННЫХ (сохраняются между перезапусками)
# ============================================================
volumes:
  postgres_device_data:
    name: postgres_device_data
  postgres_user_data:
```

```yaml
    name: postgres_user_data
  timescaledb_data:
    name: timescaledb_data
  rabbitmq_data:
    name: rabbitmq_data
  rabbitmq_log:
    name: rabbitmq_log

# ============================================================
# СЕРВИСЫ
# ============================================================
services:
  # ============================================================
  # 1. Базы данных
  # ============================================================

  postgres-device:
    image: postgres:15-alpine
    container_name: postgres-device
    restart: unless-stopped
    environment:
      POSTGRES_DB: device_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: ${DB_PASSWORD:-postgres}
    ports:
      - "5432:5432"
    volumes:
      - postgres_device_data:/var/lib/postgresql/data
    networks:
      - smart-home-net
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d device_db"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s

  postgres-user:
    image: postgres:15-alpine
    container_name: postgres-user
    restart: unless-stopped
    environment:
      POSTGRES_DB: user_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: ${DB_PASSWORD:-postgres}
    ports:
      - "5433:5432"
    volumes:
      - postgres_user_data:/var/lib/postgresql/data
    networks:
```

```
    - smart-home-net
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres -d user_db"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 30s

timescaledb:
  image: timescale/timescaledb:2.11-pg15-alpine
  container_name: timescaledb
  restart: unless-stopped
  environment:
    POSTGRES_DB: telemetry_db
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: ${DB_PASSWORD:-postgres}
  ports:
    - "5434:5432"
  volumes:
    - timescaledb_data:/var/lib/postgresql/data
  networks:
    - smart-home-net
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres -d telemetry_db"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 30s

# ============================================================
# 2. Брокер сообщений
# ============================================================

rabbitmq:
  image: rabbitmq:3.12-management-alpine
  container_name: smart-home-rabbitmq
  restart: unless-stopped
  environment:
    RABBITMQ_DEFAULT_USER: ${RABBITMQ_USER:-admin}
    RABBITMQ_DEFAULT_PASS: ${RABBITMQ_PASSWORD:-admin123}
    RABBITMQ_DEFAULT_VHOST: /
  ports:
    - "5672:5672"   # AMQP порт для приложений
    - "15672:15672" # Management UI порт
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
    - rabbitmq_log:/var/log/rabbitmq
  networks:
    - smart-home-net
  healthcheck:
```

```
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 30s
    timeout: 10s
    retries: 5

# ============================================================
# 3. Сервисы приложения
# ============================================================

user-service:
  build:
    context: ./user-service
    dockerfile: Dockerfile
  container_name: user-service
  restart: unless-stopped
  environment:
    SERVER_PORT: 8080
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres-user:5432/user_db
    SPRING_DATASOURCE_USERNAME: postgres
    SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD:-postgres}
    SPRING_JPA_HIBERNATE_DDL_AUTO: update
    JWT_SECRET: ${JWT_SECRET:-mySecretKeyForJWT}
    JWT_EXPIRATION: 86400000
  ports:
    - "8080:8080"
  networks:
    - smart-home-net
  depends_on:
    postgres-user:
      condition: service_healthy
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8080/actuator/health"]
    interval: 30s
    timeout: 10s
    retries: 5
    start_period: 40s

device-service:
  build:
    context: ./device-service
    dockerfile: Dockerfile
  container_name: device-service
  restart: unless-stopped
  environment:
    SERVER_PORT: 8081
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres-device:5432/device_db
    SPRING_DATASOURCE_USERNAME: postgres
    SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD:-postgres}
    SPRING_JPA_HIBERNATE_DDL_AUTO: update
    SPRING_RABBITMQ_HOST: rabbitmq
```

```
    SPRING_RABBITMQ_PORT: 5672
    SPRING_RABBITMQ_USERNAME: ${RABBITMQ_USER:-admin}
    SPRING_RABBITMQ_PASSWORD: ${RABBITMQ_PASSWORD:-admin123}
    USER_SERVICE_URL: http://user-service:8080
  ports:
    - "8081:8081"
  networks:
    - smart-home-net
  depends_on:
    postgres-device:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
    user-service:
      condition: service_healthy
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8081/actuator/health"]
    interval: 30s
    timeout: 10s
    retries: 5
    start_period: 50s

notification-service:
  build:
    context: ./notification-service
    dockerfile: Dockerfile
  container_name: notification-service
  restart: unless-stopped
  environment:
    SERVER_PORT: 8083
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres-device:5432/device_db
    SPRING_DATASOURCE_USERNAME: postgres
    SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD:-postgres}
    SPRING_JPA_HIBERNATE_DDL_AUTO: update
    SPRING_RABBITMQ_HOST: rabbitmq
    SPRING_RABBITMQ_PORT: 5672
    SPRING_RABBITMQ_USERNAME: ${RABBITMQ_USER:-admin}
    SPRING_RABBITMQ_PASSWORD: ${RABBITMQ_PASSWORD:-admin123}
  ports:
    - "8083:8083"
  networks:
    - smart-home-net
  depends_on:
    postgres-device:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8083/actuator/health"]
    interval: 30s
```

```
    timeout: 10s
    retries: 5
    start_period: 40s

  telemetry-service:
    build:
      context: ./telemetry-service
      dockerfile: Dockerfile
    container_name: telemetry-service
    restart: unless-stopped
    environment:
      SERVER_PORT: 8082
      SPRING_DATASOURCE_URL: jdbc:postgresql://timescaledb:5432/telemetry_db
      SPRING_DATASOURCE_USERNAME: postgres
      SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD:-postgres}
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
      SPRING_RABBITMQ_HOST: rabbitmq
      SPRING_RABBITMQ_PORT: 5672
      SPRING_RABBITMQ_USERNAME: ${RABBITMQ_USER:-admin}
      SPRING_RABBITMQ_PASSWORD: ${RABBITMQ_PASSWORD:-admin123}
    ports:
      - "8082:8082"
    networks:
      - smart-home-net
    depends_on:
      timescaledb:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8082/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 40s

  # ============================================================
  # 4. API Gateway (опционально)
  # ============================================================

  api-gateway:
    build:
      context: ./api-gateway
      dockerfile: Dockerfile
    container_name: api-gateway
    restart: unless-stopped
    environment:
      PORT: 9000
      USER_SERVICE_URL: http://user-service:8080
      DEVICE_SERVICE_URL: http://device-service:8081
```

```
    TELEMETRY_SERVICE_URL: http://telemetry-service:8082
  ports:
    - "9000:9000"
  networks:
    - smart-home-net
  depends_on:
    user-service:
      condition: service_healthy
    device-service:
      condition: service_healthy
    telemetry-service:
      condition: service_healthy
```

Задание 2.1. Создать .env файл

Создайте файл .env в корне проекта для хранения конфиденциальных переменных:

env

Базы данных

DB_PASSWORD=secure_password_123

RabbitMQ

RABBITMQ_USER=admin

RABBITMQ_PASSWORD=secure_rabbit_123

JWT для User Service

JWT_SECRET=your_super_secret_jwt_key_min_32_chars

API Gateway (опционально)

GATEWAY_PORT=9000

Важно: Добавьте .env в .gitignore, чтобы не закоммитить секреты!

Задание 2.2. Добавить healthcheck эндпоинт в сервисы

Убедитесь, что каждый сервис имеет эндпоинт /actuator/health. Для Spring Boot добавьте в application.yml:

```
management:
 endpoints:
  web:
   exposure:
    include: health
 endpoint:
  health:
   show-details: always
```

Задание 2.3. Создать docker-compose.yml

Создайте полный docker-compose.yml файл, включающий:

- Все ваши сервисы (Device, Notification, User, Telemetry)

- Базы данных (PostgreSQL для Device/User, TimescaleDB для Telemetry)

- RabbitMQ

- API Gateway (опционально)

Требования:

- Все сервисы должны использовать depends_on с condition: service_healthy

- Правильно настроенные сети (сервисы видят друг друга по именам)

- Тома для данных БД и RabbitMQ

- Переменные окружения из .env файла

Задание 2.4. Проверка запуска

Запустите систему и проверьте статус:

```
# Запуск всех сервисов
docker-compose up -d
```

```
# Просмотр статуса контейнеров
docker-compose ps

# Просмотр логов всех сервисов
docker-compose logs -f

# Просмотр логов конкретного сервиса
docker-compose logs -f device-service

# Проверка использования ресурсов
docker stats
```

Ожидаемый результат: Все сервисы должны быть в статусе healthy (или running).

Задание 2.5. Тестирование взаимодействия

Проверьте, что сервисы видят друг друга внутри Docker сети:

```
# Зайти в контейнер device-service
docker exec -it device-service sh

# Проверить доступность rabbitmq
ping rabbitmq

# Проверить доступность БД
nc -zv postgres-device 5432

# Проверить доступность user-service
curl http://user-service:8080/actuator/health

# Выйти
exit
```

Задание 2.6. Остановка системы

```
# Остановка всех сервисов
docker-compose down

# Остановка с удалением томов (все данные будут потеряны)
docker-compose down -v
```

Задание 3. Бонус — Веб-клиент (опционально)

Разобранный пример

Если вы разработали веб-клиент (React, Angular, Vue или просто HTML/JS), его можно также запустить в Docker.

Шаг 1. Создание Dockerfile для статического сайта (Nginx)

```
FROM nginx:alpine

# Копируем статические файлы
COPY ./build /usr/share/nginx/html

# Копируем конфигурацию Nginx
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Шаг 2. Конфигурация Nginx (nginx.conf)

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;

    # SPA роутинг (для React/Vue)
    location / {
        try_files $uri $uri/ /index.html;
    }

    # Проксирование API запросов к API Gateway
```

```
    location /api/ {
        proxy_pass http://api-gateway:9000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Шаг 3. Добавление в docker-compose.yml

```yaml
web-client:
  build:
    context: ./web-client
    dockerfile: Dockerfile
  container_name: web-client
  restart: unless-stopped
  ports:
    - "80:80"
  networks:
    - smart-home-net
  depends_on:
    - api-gateway
```

Задание 3.1 (бонус): Создать простой HTML/JS фронтенд

Создайте минимальную HTML страницу с JavaScript, которая:
- Вызывает API Device Service (через API Gateway)
- Отображает список устройств
- Позволяет включить/выключить устройство

Задание 3.2 (бонус): Запустить фронтенд в Docker

Соберите статические файлы и запустите их через Nginx.

3. Требования к отчету

Отчет по лабораторной работе должен содержать:

1. Титульный лист с названием работы, ФИО, группой
2. Цель работы
3. Задание 1:
   - Dockerfile для каждого сервиса (полный код)
   - Команды сборки
   - Скриншот результата docker images
   - Таблица с размерами образов
4. Задание 2:
   - Полный docker-compose.yml
   - Файл .env (без реальных секретов, можно шаблон)
   - Скриншот docker-compose ps (все сервисы healthy)
   - Скриншот логов хотя бы одного сервиса
   - Результаты проверки взаимодействия (ping, curl)
5. Задание 3 (бонус):
   - Скриншот работающего веб-клиента
   - nginx.conf (если делали)
6. Выводы:
   - Что нового узнали
   - С какими трудностями столкнулись
   - Как решили проблему зависимости порядка запуска (healthcheck)
   - Какой объем образов получился и как можно его уменьшить

4. Критерии оценки

Критери	**Отлично (2)**	**Хорошо (1)**	**Удовл.**	**Неуд. (0)**

й			**(0.5)**	
Задание 1	Dockerfile для всех сервисов, многостадийная сборка, healthcheck, непривилегированный пользователь	Dockerfile есть, но без оптимизации	Только базовый Dockerfile	Не выполнено
Задание 2	Полный docker-compose.yml, healthcheck, depends_on с условиями, .env файл	Базовый compose, но есть проблемы с зависимостями	Только запуск отдельных сервисов	Не выполнено
Сети и тома	Корректные сети, тома для данных, сервисы видят друг друга	Сети есть, но тома не настроены	Только базовая сеть	Не настроено
Запуск	docker-compose up -d работает, все сервисы healthy	Работает, но некоторые сервисы unhealthy	Запускается с ошибками	Не запускается
ИТОГО	8	5-6	3-4	0-2

5. Дополнительные задания (продвинутый уровень)

Задание 4. Добавление мониторинга

Добавьте в docker-compose.yml сервисы для мониторинга:

```yaml
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - "9090:9090"

grafana:
  image: grafana/grafana:latest
  container_name: grafana
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=admin
```

Настройте сбор метрик из Spring Boot приложений (Micrometer).

Задание 5. Оптимизация размера образа

Используйте jlink для создания кастомного JRE с минимальным набором модулей:

```dockerfile
FROM eclipse-temurin:17-jdk-alpine AS jre-build
RUN jlink \
    --add-modules java.base,java.sql,java.naming,java.management \
    --strip-debug \
    --no-man-pages \
    --no-header-files \
    --compress=2 \
    --output /custom-jre
```

Задание 6. Docker Compose Profiles

Добавьте профили для разных окружений:

```yaml
services:
  device-service:
    profiles: ["backend", "full"]

  web-client:
```

```
  profiles: ["frontend", "full"]

test-service:
  profiles: ["test"]
```

Запуск только бэкенда: docker-compose --profile backend up

6. Контрольные вопросы

1. Что такое Docker и для чего он нужен?
2. В чем разница между Docker образом и контейнером?
3. Что такое Dockerfile?
4. Назовите основные инструкции Dockerfile (минимум 5).
5. Что такое многостадийная сборка и зачем она нужна?
6. Что делает инструкция HEALTHCHECK?
7. Для чего нужен непривилегированный пользователь в контейнере?
8. В чем разница между CMD и ENTRYPOINT?
9. Зачем в Dockerfile сначала копируется pom.xml, а потом исходный код?
10. Как посмотреть список Docker образов на компьютере?
11. Как посмотреть список запущенных контейнеров?
12. Как зайти внутрь работающего контейнера?
13. Как посмотреть логи контейнера?
14. Как остановить и удалить контейнер?
15. Что такое Docker Compose и для чего он нужен?
16. Что делает ключ depends_on в docker-compose.yml?
17. Почему depends_on недостаточно и что нужно добавлять?
18. Что такое volumes в Docker и зачем они нужны?
19. Чем отличаются именованные тома от bind mounts?
20. Какие типы сетей существуют в Docker?
21. Как контейнеры видят друг друга по имени в Docker Compose?
22. Что делает команда docker-compose up -d?
23. Что делает команда docker-compose down -v?

24. Как посмотреть логи конкретного сервиса в Docker Compose?

25. Как проверить статус всех сервисов в Docker Compose?

26. Что делает ключ restart: unless-stopped?

27. Что такое healthcheck и зачем он нужен?

28. Какие параметры есть у HEALTHCHECK?

29. Какой эндпоинт используется для healthcheck в Spring Boot приложении?

30. Что будет, если не использовать healthcheck?

31. Что гласит III фактор 12-factor app?

32. Почему конфигурацию нельзя хранить в коде?

33. Как передать переменные окружения в Docker контейнер?

34. Что такое файл .env и для чего он нужен?

35. Почему файл .env нужно добавлять в .gitignore?

36. Как проверить, что один контейнер видит другой внутри Docker сети?

37. Как уменьшить размер Docker образа для Java приложения?

38. Что произойдет, если два контейнера используют один порт на хосте?

39. Как пересобрать образ без использования кэша?

40. Как очистить все неиспользуемые Docker образы, контейнеры и сети?

7. Шпаргалка по командам Docker

Команда	Описание
docker build -t name:tag .	Сборка образа из Dockerfile
docker run -d -p 8080:8080 name:tag	Запуск контейнера
docker ps	Список запущенных контейнеров
docker ps -a	Все контейнеры (включая остановленные)
docker stop <id>	Остановка контейнера
docker rm <id>	Удаление контейнера
docker rmi <table>	Удаление образа
docker logs <id>	Просмотр логов

docker exec -it <id> sh	Подключение к контейнеру (shell)
docker-compose up -d	Запуск всех сервисов
docker-compose down	Остановка всех сервисов
docker-compose down -v	Остановка + удаление томов
docker-compose logs -f	Просмотр логов всех сервисов
docker-compose build --no-cache	Пересборка всех образов без кэша
docker system prune -a	Очистка всех неиспользуемых данных

Лабораторная работа №5

Реализация бэкенда (REST API, ORM, бизнес-логика)

Цель работы:

Получить практические навыки разработки бэкенда на Java/Spring Boot: создание сущностей (Entity), репозиториев (Spring Data JPA), сервисного слоя с бизнес-логикой и REST-контроллеров.

Задачи:

Создать JPA-сущности для предметной области (Device, User, Telemetry, Scenario)

1. Настроить связи между сущностями (@OneToMany, @ManyToOne, @ManyToMany)
2. Создать Spring Data JPA репозитории для каждой сущности
3. Реализовать сервисный слой с бизнес-логикой
4. Реализовать REST-контроллеры для CRUD-операций
5. Настроить миграции базы данных (Liquibase или Flyway)
6. Подключить PostgreSQL и настроить connection pool (HikariCP)

Краткое содержание

1. Теоретическая часть

JPA (Java Persistence API) – спецификация для работы с реляционными базами данных в Java.

Основные аннотации JPA:

Аннотация	Назначение
@Entity	Обозначает класс как сущность, отображаемую в таблицу БД
@Table(name = "table_name")	Указывает имя таблицы
@Id	Указывает первичный ключ
@GeneratedValue(strategy = GenerationType.IDENTITY)	Автоматическая генерация ID
@Column(name = "column_name")	Указывает имя колонки
@OneToMany, @ManyToOne, @ ManyToMany, @OneToOne	Связи между сущностями
@JoinColumn	Указывает внешний ключ
@Transient	Поле не сохраняется в БД

Связи между сущностями:

2. Пример кода

Сущность Device:

java

```java
@Entity
@Table(name = "devices")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Device {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private DeviceType type; // THERMOSTAT, LIGHT, LOCK, CAMERA
```

```
    @Column(nullable = false)
    private String model;

    private String room;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private DeviceStatus status; // ACTIVE, INACTIVE, OFFLINE

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
        status = DeviceStatus.INACTIVE;
    }

    @PreUpdate
    protected void onUpdate() {
        updatedAt = LocalDateTime.now();
    }
}
```

Репозиторий:

```
@Repository
public interface DeviceRepository extends JpaRepository<Device, UUID> {

    List<Device> findByUserId(UUID userId);

    List<Device> findByUserIdAndType(UUID userId, DeviceType type);

    Optional<Device> findByIdAndUserId(UUID id, UUID userId);

    @Query("SELECT d FROM Device d WHERE d.user.id = :userId AND
d.status = :status")
    List<Device>    findDevicesByUserAndStatus(@Param("userId")    UUID
userId,
                                               @Param("status") DeviceStatus status);
}
```

Сервис:

```
@Service
@RequiredArgsConstructor
@Slf4j
public class DeviceService {

    private final DeviceRepository deviceRepository;
    private final UserRepository userRepository;

    @Transactional
    public Device createDevice(UUID userId, CreateDeviceRequest request)
{
        User user = userRepository.findById(userId)
                .orElseThrow(() -> new UserNotFoundException(userId));
```

```
        Device device = new Device();
        device.setName(request.getName());
        device.setType(request.getType());
        device.setModel(request.getModel());
        device.setRoom(request.getRoom());
        device.setUser(user);

        Device savedDevice = deviceRepository.save(device);
        log.info("Device created: {} for user: {}", savedDevice.getId(), userId);

        return savedDevice;
    }

    @Transactional(readOnly = true)
    public List<Device> getUserDevices(UUID userId) {
        return deviceRepository.findByUserId(userId);
    }

    @Transactional
    public Device updateDevice(UUID deviceId, UUID userId,
UpdateDeviceRequest request) {
        Device device = deviceRepository.findByIdAndUserId(deviceId, userId)
            .orElseThrow(() -> new DeviceNotFoundException(deviceId));

        if (request.getName() != null) {
            device.setName(request.getName());
        }
        if (request.getRoom() != null) {
            device.setRoom(request.getRoom());
```

```
        }

        return deviceRepository.save(device);
    }

    @Transactional
    public void deleteDevice(UUID deviceId, UUID userId) {
        Device device = deviceRepository.findByIdAndUserId(deviceId, userId)
                .orElseThrow(() -> new DeviceNotFoundException(deviceId));
        deviceRepository.delete(device);
        log.info("Device deleted: {}", deviceId);
    }
}
```

Контроллер:

```
@RestController
@RequestMapping("/api/v1/devices")
@RequiredArgsConstructor
public class DeviceController {

    private final DeviceService deviceService;

    @PostMapping
    public ResponseEntity<Device> createDevice(
            @RequestHeader("X-User-Id") UUID userId,
            @Valid @RequestBody CreateDeviceRequest request) {
        Device device = deviceService.createDevice(userId, request);
        return ResponseEntity.status(HttpStatus.CREATED).body(device);
    }

    @GetMapping
```

```
    public ResponseEntity<List<Device>> getUserDevices(
            @RequestHeader("X-User-Id") UUID userId) {
        return ResponseEntity.ok(deviceService.getUserDevices(userId));
    }

    @GetMapping("/{deviceId}")
    public ResponseEntity<Device> getDevice(
            @RequestHeader("X-User-Id") UUID userId,
            @PathVariable UUID deviceId) {
        Device device = deviceService.getDevice(deviceId, userId);
        return ResponseEntity.ok(device);
    }

    @PatchMapping("/{deviceId}")
    public ResponseEntity<Device> updateDevice(
            @RequestHeader("X-User-Id") UUID userId,
            @PathVariable UUID deviceId,
            @Valid @RequestBody UpdateDeviceRequest request) {
        return ResponseEntity.ok(deviceService.updateDevice(deviceId, userId,
request));
    }

    @DeleteMapping("/{deviceId}")
    public ResponseEntity<Void> deleteDevice(
            @RequestHeader("X-User-Id") UUID userId,
            @PathVariable UUID deviceId) {
        deviceService.deleteDevice(deviceId, userId);
        return ResponseEntity.noContent().build();
    }
}
```

Индивидуальные варианты

Каждый студент получает индивидуальный вариант набора сущностей и функциональности для своего сервиса.

Вариант 1 (Device Service – полный):

Сущности: Device, DeviceType (enum), DeviceStatus (enum)

Функции: CRUD устройств, фильтрация по типу и статусу

Дополнительно: отправка команд устройствам

Вариант 2 (User Service – полный):

Сущности: User, Role (enum), UserSettings

Функции: CRUD пользователей, изменение ролей, блокировка/разблокировка

Дополнительно: аудит действий пользователя

Вариант 3 (Telemetry Service – полный):

Сущности: TelemetryData, Device (связь), AggregatedStats

Функции: запись телеметрии, получение текущих показаний, статистика за период

Дополнительно: агрегация данных (среднее, мин, макс)

Вариант 4 (Scenario Service – полный):

Сущности: Scenario, Condition, Action, ScenarioExecution

Функции: CRUD сценариев, активация/деактивация, просмотр истории выполнения

Дополнительно: проверка условий по расписанию

Вариант 5 (Notification Service – полный):

Сущности: Notification, NotificationLog, DeviceEventLog

Функции: сохранение уведомлений, получение истории по пользователю

Дополнительно: приоритеты уведомлений

Вариант 6–10: Комбинации сущностей для смешанных сервисов.

Контрольные вопросы

1. Что такое JPA и Hibernate? В чем разница?
2. Какие аннотации используются для маппинга сущностей?
3. Какие типы связей существуют между сущностями? Приведите примеры.
4. Что такое @PrePersist и @PreUpdate? Для чего они нужны?
5. В чем разница между FetchType.EAGER и FetchType.LAZY?
6. Что такое Spring Data JPA? Какие методы есть в JpaRepository?
7. Как создать кастомный запрос в репозитории? (@Query)
8. Какие аннотации используются для транзакций? Что делает @Transactional?
9. Какие слои архитектуры присутствуют в Spring Boot приложении?
10. Как настраивается подключение к PostgreSQL в Spring Boot?
11. Что такое Liquibase/Flyway? Для чего нужны миграции?
12. Как обрабатывать ошибки и исключения в REST API?
13. Что такое DTO? Зачем их использовать вместо Entity в контроллерах?
14. Как валидировать входные данные в контроллерах? (@Valid, @NotNull, @Size)
15. Что такое connection pool? Зачем он нужен?

Лабораторная работа №6.

Аутентификация и авторизация (JWT, Spring Security)

Цель работы:

Научиться настраивать Spring Security, реализовывать регистрацию и вход пользователей, генерировать и валидировать JWT-токены, разграничивать доступ к эндпоинтам по ролям.

Задачи:

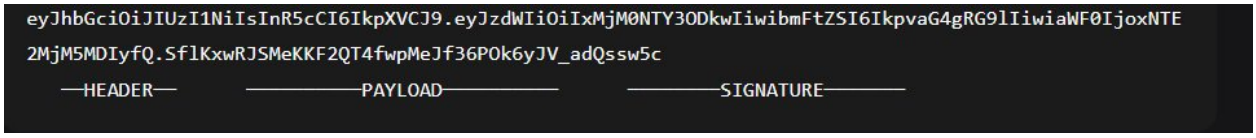
1. Добавить зависимости Spring Security и JJWT

2. Реализовать UserDetailsService для загрузки пользователя из БД
3. Настроить PasswordEncoder (BCrypt)
4. Реализовать генерацию и валидацию JWT (access и refresh токены)
5. Создать JWT-фильтр для аутентификации запросов
6. Реализовать эндпоинты: /api/v1/auth/register, /login, /refresh
7. Настроить защиту эндпоинтов по ролям (USER, ADMIN, SUPPORT)

. Теоретическая часть

JWT (JSON Web Token) – компактный и самодостаточный способ передачи информации между сторонами в виде JSON-объекта.

Структура JWT:



Процесс аутентификации:

1. Client → POST /login {email, password}
2. Server → проверяет credentials → генерирует JWT
3. Client ← {access_token, refresh_token}
4. Client → запрос с Header: Authorization: Bearer <access_token>
5. Server → валидирует JWT → обрабатывает запрос

Роли в системе:

Роль	Права
USER	Чтение своих устройств, управление своими устройствами
SUPPORT	Просмотр устройств всех пользователей (только чтение)

Роль	Права
ADMIN	Полный доступ ко всем ресурсам

2. Пример кода

Модель User:

```
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(unique = true, nullable = false)
    private String email;

    @Column(nullable = false)
    private String password; // хешированный

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Enumerated(EnumType.STRING)
```

```
    @Column(nullable = false)
    private Role role; // USER, ADMIN, SUPPORT

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "is_active")
    private boolean isActive = true;
}
```

UserDetailsService:

```
@Service
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new UsernameNotFoundException("User not found: " + email));

        return org.springframework.security.core.userdetails.User.builder()
            .username(user.getEmail())
            .password(user.getPassword())
            .authorities(user.getRole().name())
            .disabled(!user.isActive())
            .build();
    }
```

```
}
```

JWT Service:

```
@Service
@Slf4j
public class JwtService {

    @Value("${jwt.secret}")
    private String secretKey;

    @Value("${jwt.access-expiration}")
    private long accessExpiration; // 15 минут

    @Value("${jwt.refresh-expiration}")
    private long refreshExpiration; // 7 дней

    public String generateAccessToken(User user) {
        return Jwts.builder()
                .setSubject(user.getId().toString())
                .claim("email", user.getEmail())
                .claim("role", user.getRole().name())
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() +
accessExpiration))
                .signWith(SignatureAlgorithm.HS256, secretKey)
                .compact();
    }

    public String generateRefreshToken(User user) {
        return Jwts.builder()
                .setSubject(user.getId().toString())
```

```
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + refreshExpiration))
                .signWith(SignatureAlgorithm.HS256, secretKey)
                .compact();
        }

        public Claims extractClaims(String token) {
            return Jwts.parser()
                .setSigningKey(secretKey)
                .parseClaimsJws(token)
                .getBody();
        }

        public boolean isTokenValid(String token) {
            try {
                extractClaims(token);
                return true;
            } catch (JwtException | IllegalArgumentException e) {
                log.warn("Invalid JWT: {}", e.getMessage());
                return false;
            }
        }

        public UUID getUserIdFromToken(String token) {
            return UUID.fromString(extractClaims(token).getSubject());
        }
    }
```

JWT-фильтр:

```
    @Component
```

```
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final CustomUserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException,
IOException {

        String authHeader = request.getHeader("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        String token = authHeader.substring(7);

        if (jwtService.isTokenValid(token)) {
            UUID userId = jwtService.getUserIdFromToken(token);
            UserDetails userDetails =
userDetailsService.loadUserByUsername(userId.toString());

            UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                userDetails, null, userDetails.getAuthorities()
            );
```

```
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }

            filterChain.doFilter(request, response);
        }
    }
```

Security Configuration:

```
java
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthFilter;
    private final CustomUserDetailsService userDetailsService;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
{
        return http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/v1/auth/**").permitAll()
                .requestMatchers("/api/v1/devices/**").hasAnyRole("USER",
"ADMIN", "SUPPORT")
                .requestMatchers("/api/v1/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
```

```
                .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .authenticationProvider(authenticationProvider())
                .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class)
                .build();
        }

        @Bean
        public AuthenticationManager
authenticationManager(AuthenticationConfiguration config)
                throws Exception {
            return config.getAuthenticationManager();
        }

        @Bean
        public PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }
    }
```

AuthController:

```
    @RestController
    @RequestMapping("/api/v1/auth")
    @RequiredArgsConstructor
    public class AuthController {

        private final AuthenticationManager authenticationManager;
        private final UserService userService;
        private final JwtService jwtService;
```

```
    @PostMapping("/register")
    public ResponseEntity<AuthResponse> register(@Valid @RequestBody
RegisterRequest request) {
        User user = userService.register(request);
        String accessToken = jwtService.generateAccessToken(user);
        String refreshToken = jwtService.generateRefreshToken(user);
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(new AuthResponse(accessToken, refreshToken, user));
    }

    @PostMapping("/login")
    public ResponseEntity<AuthResponse> login(@Valid @RequestBody
LoginRequest request) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(request.getEmail(),
request.getPassword())
        );

        User user = (User) authentication.getPrincipal();
        String accessToken = jwtService.generateAccessToken(user);
        String refreshToken = jwtService.generateRefreshToken(user);

        return ResponseEntity.ok(new AuthResponse(accessToken,
refreshToken, user));
    }

    @PostMapping("/refresh")
    public ResponseEntity<AuthResponse> refresh(@Valid @RequestBody
RefreshTokenRequest request) {
        if (!jwtService.isTokenValid(request.getRefreshToken())) {
```

```
            throw new InvalidTokenException("Invalid refresh token");
        }

        UUID userId = jwtService.getUserIdFromToken(request.getRefreshToken());
        User user = userService.findById(userId);

        String newAccessToken = jwtService.generateAccessToken(user);
        return ResponseEntity.ok(new AuthResponse(newAccessToken,
request.getRefreshToken(), user));
    }
}
```

Индивидуальные варианты

Вариант 1: Базовая аутентификация (только access token, без refresh)
Вариант 2: Access + refresh токены (как в примере)
Вариант 3: Добавить логирование всех неудачных попыток входа
Вариант 4: Реализовать двухфакторную аутентификацию (email-код)
Вариант 5: Реализовать OAuth2 (вход через Google/GitHub)
Вариант 6: Добавить endpoint для смены пароля
Вариант 7: Добавить endpoint для восстановления пароля
Вариант 8: Реализовать blacklist для отозванных токенов (Redis)

Контрольные вопросы

1. Что такое JWT? Из каких частей состоит JWT?
2. В чем разница между access и refresh токенами?
3. Какие алгоритмы подписи JWT существуют? Что лучше использовать?
4. Как настроить Spring Security для JWT-аутентификации?
5. Что такое UserDetails и UserDetailsService?

6. Зачем нужен PasswordEncoder? Какие реализации существуют?
7. Как разграничить доступ к эндпоинтам по ролям?
8. Где хранить JWT на клиенте? Какие есть риски?
9. Как отозвать JWT до истечения срока действия?
10. Что такое stateless аутентификация? В чем отличие от session-based?
11. Как защитить refresh token от кражи?
12. Что такое CSRF и почему его отключают для JWT?
13. Как реализовать logout при использовании JWT?
14. Какой срок жизни access/refresh токенов рекомендуется?
15. Как добавить дополнительную информацию в JWT (custom claims)?

Лабораторная работа №7.

Модульное и интеграционное тестирование бэкенда

Цель работы:

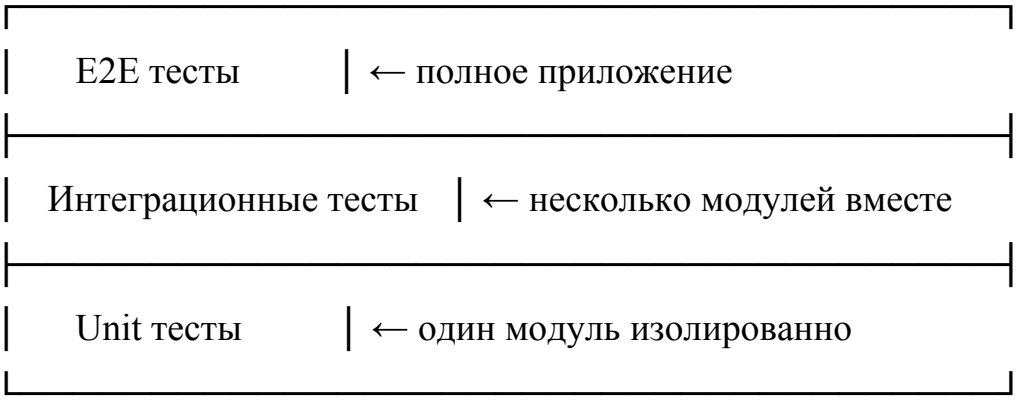
Научиться писать модульные (unit) и интеграционные тесты для бэкенда, использовать мокирование зависимостей и контейнеры для тестирования с реальной БД.

Задачи:

1. Написать unit-тесты для сервисного слоя с Mockito
2. Написать тесты для репозиториев с использованием @DataJpaTest
3. Написать интеграционные тесты для REST API с @SpringBootTest
4. Настроить TestContainers для PostgreSQL
5. Добиться покрытия кода не менее 70%
6. Настроить генерацию отчета о покрытии (JaCoCo)

Теоретическая часть

Виды тестов:



Инструменты:

Инструмент	Назначение
JUnit 5	Фреймворк для написания тестов
Mockito	Создание мок-объектов
AssertJ	Fluent assertions
TestContainers	Запуск Docker-контейнеров для тестов
JaCoCo	Измерение покрытия кода

2. Пример кода

Unit-тест сервиса (Mockito):

```
@ExtendWith(MockitoExtension.class)
class DeviceServiceTest {

    @Mock
    private DeviceRepository deviceRepository;

    @Mock
    private UserRepository userRepository;
```

```
@InjectMocks
private DeviceService deviceService;

private UUID userId;
private UUID deviceId;
private User user;
private Device device;

@BeforeEach
void setUp() {
    userId = UUID.randomUUID();
    deviceId = UUID.randomUUID();

    user = new User();
    user.setId(userId);
    user.setEmail("test@example.com");

    device = new Device();
    device.setId(deviceId);
    device.setName("Test Device");
    device.setType(DeviceType.LIGHT);
    device.setUser(user);
}

@Test
@DisplayName("Должен создать устройство")
void shouldCreateDevice() {
    // given
    CreateDeviceRequest request = new CreateDeviceRequest();
    request.setName("New Light");
```

```
        request.setType(DeviceType.LIGHT);

        when(userRepository.findById(userId)).thenReturn(Optional.of(user));
        when(deviceRepository.save(any(Device.class))).thenAnswer(inv -> {
            Device saved = inv.getArgument(0);
            saved.setId(UUID.randomUUID());
            return saved;
        });

        // when
        Device result = deviceService.createDevice(userId, request);

        // then
        assertThat(result).isNotNull();
        assertThat(result.getId()).isNotNull();
        assertThat(result.getName()).isEqualTo("New Light");
        assertThat(result.getType()).isEqualTo(DeviceType.LIGHT);

        verify(deviceRepository).save(any(Device.class));
    }

    @Test
    @DisplayName("Должен выбросить исключение при создании устройства для несуществующего пользователя")
    void shouldThrowWhenUserNotFound() {
        // given
        CreateDeviceRequest request = new CreateDeviceRequest();
        when(userRepository.findById(userId)).thenReturn(Optional.empty());

        // when/then
```

```
        assertThatThrownBy(() -> deviceService.createDevice(userId, request))
            .isInstanceOf(UserNotFoundException.class);

        verify(deviceRepository, never()).save(any());
    }

    @Test
    @DisplayName("Должен получить все устройства пользователя")
    void shouldGetUserDevices() {
        // given
        List<Device> devices = List.of(device);
        when(deviceRepository.findByUserId(userId)).thenReturn(devices);

        // when
        List<Device> result = deviceService.getUserDevices(userId);

        // then
        assertThat(result).hasSize(1);
        assertThat(result.get(0).getId()).isEqualTo(deviceId);
    }
}
```

Unit-тест контроллера (WebMvcTest):

```
@WebMvcTest(DeviceController.class)
class DeviceControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private DeviceService deviceService;
```

```
    private UUID userId;
    private UUID deviceId;
    private Device device;

    @BeforeEach
    void setUp() {
        userId = UUID.randomUUID();
        deviceId = UUID.randomUUID();

        device = new Device();
        device.setId(deviceId);
        device.setName("Test Device");
    }

    @Test
    @DisplayName("GET /api/v1/devices должен вернуть список устройств")
    void shouldReturnUserDevices() throws Exception {
        // given
        when(deviceService.getUserDevices(userId)).thenReturn(List.of(device));

        // when/then
        mockMvc.perform(get("/api/v1/devices")
                .header("X-User-Id", userId))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$[0].id").value(deviceId.toString()))
            .andExpect(jsonPath("$[0].name").value("Test Device"));
    }
```

```
    @Test
    @DisplayName("POST /api/v1/devices должен создать устройство")
    void shouldCreateDevice() throws Exception {
        // given
        CreateDeviceRequest request = new CreateDeviceRequest();
        request.setName("New Device");
        request.setType(DeviceType.LIGHT);

        when(deviceService.createDevice(eq(userId), any()))
            .thenReturn(device);

        // when/then
        mockMvc.perform(post("/api/v1/devices")
                .header("X-User-Id", userId)
                .contentType(MediaType.APPLICATION_JSON)
                .content(asJsonString(request)))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.id").value(deviceId.toString()));
    }

    private String asJsonString(Object obj) {
        try {
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Интеграционный тест с TestContainers:

```
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@Testcontainers
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
class DeviceIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:15-alpine")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private DeviceRepository deviceRepository;

    private UUID userId;

    @BeforeEach
```

```
    void setUp() {
        deviceRepository.deleteAll();
        userId = UUID.randomUUID();
    }

    @Test
    @DisplayName("Интеграционный тест: создание и получение устройства")
    void shouldCreateAndRetrieveDevice() {
        // given
        CreateDeviceRequest request = new CreateDeviceRequest();
        request.setName("Integration Device");
        request.setType(DeviceType.LIGHT);

        // when - создаем устройство
        ResponseEntity<Device> createResponse = restTemplate.postForEntity(
            "/api/v1/devices?userId=" + userId,
            request,
            Device.class
        );

        // then
        assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);
        Device created = createResponse.getBody();
        assertThat(created).isNotNull();
        assertThat(created.getName()).isEqualTo("Integration Device");

        // when - получаем устройства
        ResponseEntity<Device[]> getResponse = restTemplate.getForEntity(
```

```
            "/api/v1/devices?userId=" + userId,
            Device[].class
        );

        // then
        assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(getResponse.getBody()).hasSize(1);
    }
}
```

Тест репозитория:

```
@DataJpaTest
class DeviceRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private DeviceRepository deviceRepository;

    private User user;
    private Device device;

    @BeforeEach
    void setUp() {
        user = new User();
        user.setEmail("test@example.com");
        user.setPassword("hashed");
        entityManager.persist(user);

        device = new Device();
```

```
        device.setName("Test Device");
        device.setType(DeviceType.LIGHT);
        device.setUser(user);
        entityManager.persist(device);

        entityManager.flush();
    }

    @Test
    void shouldFindDevicesByUserId() {
        List<Device> devices = deviceRepository.findByUserId(user.getId());

        assertThat(devices).hasSize(1);
        assertThat(devices.get(0).getName()).isEqualTo("Test Device");
    }

    @Test
    void shouldFindDeviceByIdAndUserId() {
        Optional<Device> found =
deviceRepository.findByIdAndUserId(device.getId(), user.getId());

        assertThat(found).isPresent();
        assertThat(found.get().getName()).isEqualTo("Test Device");
    }
}
```

Индивидуальные варианты

Вариант 1: Тестирование Device Service (CRUD + команды)

Вариант 2: Тестирование User Service (регистрация + аутентификация)

Вариант 3: Тестирование Telemetry Service (запись + агрегация)

Вариант 4: Тестирование Scenario Service (условия + действия)

Вариант 5: Тестирование всех сервисов с моками внешних вызовов
Вариант 6: Написание параметризованных тестов (@ParameterizedTest)
Вариант 7: Написание тестов для исключений и граничных случаев
Вариант 8: Настройка JaCoCo с порогом покрытия 80%

Контрольные вопросы

1. В чем разница между unit-тестами и интеграционными тестами?
2. Что такое Mockito? Для чего нужны моки?
3. Какие аннотации Mockito вы знаете? (@Mock, @InjectMocks, @Spy)
4. Как проверить, что метод был вызван определенное количество раз?
5. Что такое TestContainers? Зачем они нужны?
6. В чем отличие @DataJpaTest от @SpringBootTest?
7. Как тестировать REST API в Spring Boot?
8. Что такое покрытие кода (code coverage)? Какие метрики существуют?
9. Что такое TDD? Опишите цикл red-green-refactor.
10. Как тестировать методы, возвращающие Optional?
11. Как тестировать методы, выбрасывающие исключения?
12. Что такое параметризованные тесты? Как их создать?
13. Как тестировать асинхронные методы?
14. Как настроить TestContainers для PostgreSQL?
15. Что такое @DynamicPropertySource?

Лабораторная работа №8

Документирование API (Swagger/OpenAPI) и интеграционное тестирование

Цель работы:

Научиться автоматически генерировать документацию API, писать сквозные интеграционные тесты и настраивать автоматическую валидацию соответствия API спецификации.

Задачи;

1. Настроить springdoc-openapi для автоматической генерации OpenAPI документации
2. Настроить Swagger UI для интерактивной документации
3. Добавить аннотации для улучшения документации (@Operation, @ApiResponse, @Schema)
4. Написать сквозные интеграционные тесты для ключевых сценариев
5. Настроить валидацию OpenAPI спецификации
6. Сгенерировать клиентский SDK из OpenAPI спецификации

Теоретическая часть

OpenAPI (ранее Swagger) – спецификация для описания REST API.

springdoc-openapi – библиотека для автоматической генерации OpenAPI документации в Spring Boot.

Структура документации:

OpenAPI Specification
├── info (название, версия, описание)
├── servers (URL серверов)
├── paths (эндпоинты)
│ ├── GET /devices
│ ├── POST /devices
│ └── ...
├── components (схемы, параметры, ответы)
└── security (схемы аутентификации)

2. Пример кода

Зависимости (pom.xml):

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

Настройка application.yml:

```
springdoc:
  api-docs:
    path: /api-docs
    enabled: true
  swagger-ui:
    path: /swagger-ui.html
    enabled: true
    operationsSorter: method
    tagsSorter: alpha
  show-actuator: true
  default-consumes-media-type: application/json
  default-produces-media-type: application/json
```

Аннотирование контроллеров:

```
@RestController
@RequestMapping("/api/v1/devices")
@Tag(name = "Device Management", description = "API для управления устройствами умного дома")
public class DeviceController {

    @Operation(
        summary = "Создание нового устройства",
        description = "Регистрирует новое устройство в системе и привязывает его к пользователю",
        responses = {
```

```
            @ApiResponse(responseCode = "201", description = "Устройство успешно создано"),
            @ApiResponse(responseCode = "400", description = "Некорректные данные запроса"),
            @ApiResponse(responseCode = "401", description = "Не авторизован"),
            @ApiResponse(responseCode = "403", description = "Доступ запрещен")
        }
    )
    @PostMapping
    public ResponseEntity<DeviceDto> createDevice(
        @Parameter(description = "ID пользователя", required = true)
        @RequestHeader("X-User-Id") UUID userId,

        @io.swagger.v3.oas.annotations.parameters.RequestBody(
            description = "Данные для создания устройства",
            required = true,
            content = @Content(schema = @Schema(implementation = CreateDeviceRequest.class))
        )
        @Valid @RequestBody CreateDeviceRequest request) {
        // ...
    }

    @Operation(summary = "Получение всех устройств пользователя")
    @GetMapping
    public ResponseEntity<List<DeviceDto>> getUserDevices(
        @Parameter(description = "ID пользователя", required = true)
        @RequestHeader("X-User-Id") UUID userId) {
```

```
    // ...
}

@Operation(summary = "Включение устройства")
@PostMapping("/{deviceId}/turn-on")
public ResponseEntity<Void> turnOn(
        @Parameter(description = "ID пользователя", required = true)
        @RequestHeader("X-User-Id") UUID userId,

        @Parameter(description = "ID устройства", required = true, example = "123e4567-e89b-12d3-a456-426614174000")
        @PathVariable UUID deviceId) {
    deviceService.turnOn(deviceId, userId);
    return ResponseEntity.ok().build();
}
}
```

Аннотирование DTO:

```
@Schema(description = "Запрос на создание устройства")
@Data
public class CreateDeviceRequest {

    @Schema(description = "Название устройства", example = "Гостиная лампа", required = true)
    @NotBlank(message = "Name is required")
    @Size(min = 3, max = 50)
    private String name;

    @Schema(description = "Тип устройства", example = "LIGHT", required = true,
```

```
            allowableValues = {"THERMOSTAT", "LIGHT", "LOCK",
"CAMERA", "SENSOR"})
        @NotNull
        private DeviceType type;

        @Schema(description = "Модель устройства", example = "Xiaomi Smart
Light 2")
        private String model;

        @Schema(description = "Комната установки", example = "living_room")
        private String room;
    }

    @Schema(description = "DTO устройства")
    @Data
    public class DeviceDto {

        @Schema(description = "Уникальный идентификатор устройства",
example = "123e4567-e89b-12d3-a456-426614174000")
        private UUID id;

        @Schema(description = "Название устройства", example = "Гостиная
лампа")
        private String name;

        @Schema(description = "Тип устройства", example = "LIGHT")
        private DeviceType type;

        @Schema(description = "Текущий статус", example = "ACTIVE")
        private DeviceStatus status;
```

```
    @Schema(description = "Модель устройства", example = "Xiaomi Smart Light 2")
    private String model;

    @Schema(description = "Комната установки", example = "living_room")
    private String room;
}
```

Настройка OpenAPI конфигурации:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .info(new Info()
                .title("Smart Home API")
                .version("1.0.0")
                .description("API для управления умным домом")
                .contact(new Contact()
                    .name("Support Team")
                    .email("support@smarthome.com")
                    .url("https://smarthome.com"))
                .license(new License()
                    .name("Apache 2.0")
                    .url("https://www.apache.org/licenses/LICENSE-2.0.html")))
            .servers(List.of(
                new Server().url("http://localhost:8080/api/v1")
                    .description("Local development server"),
                new Server().url("https://api.smarthome.com/v1")
```

```
                .description("Production server")
        ))
        .components(new Components()
            .addSecuritySchemes("bearerAuth", new SecurityScheme()
                .type(SecurityScheme.Type.HTTP)
                .scheme("bearer")
                .bearerFormat("JWT")
                .description("JWT токен, полученный при аутентификации")))
        .addSecurityItem(new SecurityRequirement().addList("bearerAuth"));
    }
}
```

Сквозной интеграционный тест:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
@Testcontainers
class DeviceE2ETest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:15-alpine");

    @DynamicPropertySource
    static void properties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
```

```
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper objectMapper;

    private String authToken;
    private UUID userId;

    @Test
    @DisplayName("E2E: Регистрация -> Логин -> Создание устройства -> Получение устройств -> Удаление")
    void fullUserJourney() throws Exception {

        // 1. Регистрация
        RegisterRequest registerRequest = new RegisterRequest();
        registerRequest.setEmail("e2e@test.com");
        registerRequest.setPassword("password123");
        registerRequest.setFirstName("E2E");
        registerRequest.setLastName("Tester");

        MvcResult registerResult = mockMvc.perform(post("/api/v1/auth/register")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(registerRequest)))
            .andExpect(status().isCreated())
            .andReturn();

        AuthResponse authResponse = objectMapper.readValue(
            registerResult.getResponse().getContentAsString(),
            AuthResponse.class
```

```
);
authToken = authResponse.getAccessToken();

// 2. Создание устройства
CreateDeviceRequest deviceRequest = new CreateDeviceRequest();
deviceRequest.setName("E2E Light");
deviceRequest.setType(DeviceType.LIGHT);
deviceRequest.setRoom("Living Room");

MvcResult deviceResult = mockMvc.perform(post("/api/v1/devices")
        .header("Authorization", "Bearer " + authToken)
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(deviceRequest)))
    .andExpect(status().isCreated())
    .andReturn();

DeviceDto device = objectMapper.readValue(
    deviceResult.getResponse().getContentAsString(),
    DeviceDto.class
);
assertThat(device.getName()).isEqualTo("E2E Light");

// 3. Получение списка устройств
MvcResult getResult = mockMvc.perform(get("/api/v1/devices")
        .header("Authorization", "Bearer " + authToken))
    .andExpect(status().isOk())
    .andReturn();

DeviceDto[] devices = objectMapper.readValue(
    getResult.getResponse().getContentAsString(),
```

```
                DeviceDto[].class
            );
            assertThat(devices).hasSize(1);

            // 4. Удаление устройства
            mockMvc.perform(delete("/api/v1/devices/{deviceId}", device.getId())
                    .header("Authorization", "Bearer " + authToken))
                .andExpect(status().isNoContent());

            // 5. Проверка, что устройство удалено
            mockMvc.perform(get("/api/v1/devices")
                    .header("Authorization", "Bearer " + authToken))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$").isEmpty());
        }
    }
```

Генерация SDK из OpenAPI (maven plugin):

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>7.1.0</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>

<inputSpec>${project.basedir}/src/main/resources/openapi.yaml</inputSpec>
                <generatorName>java</generatorName>
```

```
                <output>${project.build.directory}/generated-
sources/openapi</output>
                <packageName>com.smarthome.client</packageName>
                <apiPackage>com.smarthome.client.api</apiPackage>
                <modelPackage>com.smarthome.client.model</modelPackage>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Индивидуальные варианты

Вариант 1: Настройка springdoc-openapi для Device Service
Вариант 2: Настройка springdoc-openapi для User Service
Вариант 3: Настройка springdoc-openapi для Telemetry Service
Вариант 4: Генерация клиентского SDK для React
Вариант 5: Написание E2E-тестов для всех сценариев (Device + User)
Вариант 6: Интеграция Swagger UI с Security (только авторизованные пользователи)
Вариант 7: Валидация OpenAPI спецификации в CI/CD пайплайне
Вариант 8: Создание Postman коллекции из OpenAPI спецификации

Контрольные вопросы

1. Что такое OpenAPI? Какие версии существуют?
2. В чем разница между Swagger и OpenAPI?
3. Как настроить springdoc-openapi в Spring Boot?
4. Какие аннотации используются для документирования API?
5. Что такое Swagger UI? Как его открыть?
6. Как документировать параметры запроса (path, query, header)?
7. Как документировать тело запроса и ответа?
8. Как документировать коды ошибок?
9. Как добавить схему аутентификации (JWT) в документацию?

10. Как сгенерировать клиентский SDK из OpenAPI спецификации?
11. Что такое E2E-тестирование? Чем отличается от unit и интеграционных тестов?
12. Какие инструменты для E2E-тестирования REST API существуют?
13. Как тестировать авторизованные запросы в интеграционных тестах?
14. Что такое TestRestTemplate? В чем отличие от MockMvc?
15. Как автоматизировать валидацию соответствия API спецификации?

Лабораторная работа №9

Итоговая интеграция бэкенда, рефакторинг и защита проекта (1 часть)

Цель работы:

Интегрировать все разработанные сервисы, выполнить рефакторинг кода, проанализировать технический долг и подготовить первую часть проекта к защите.

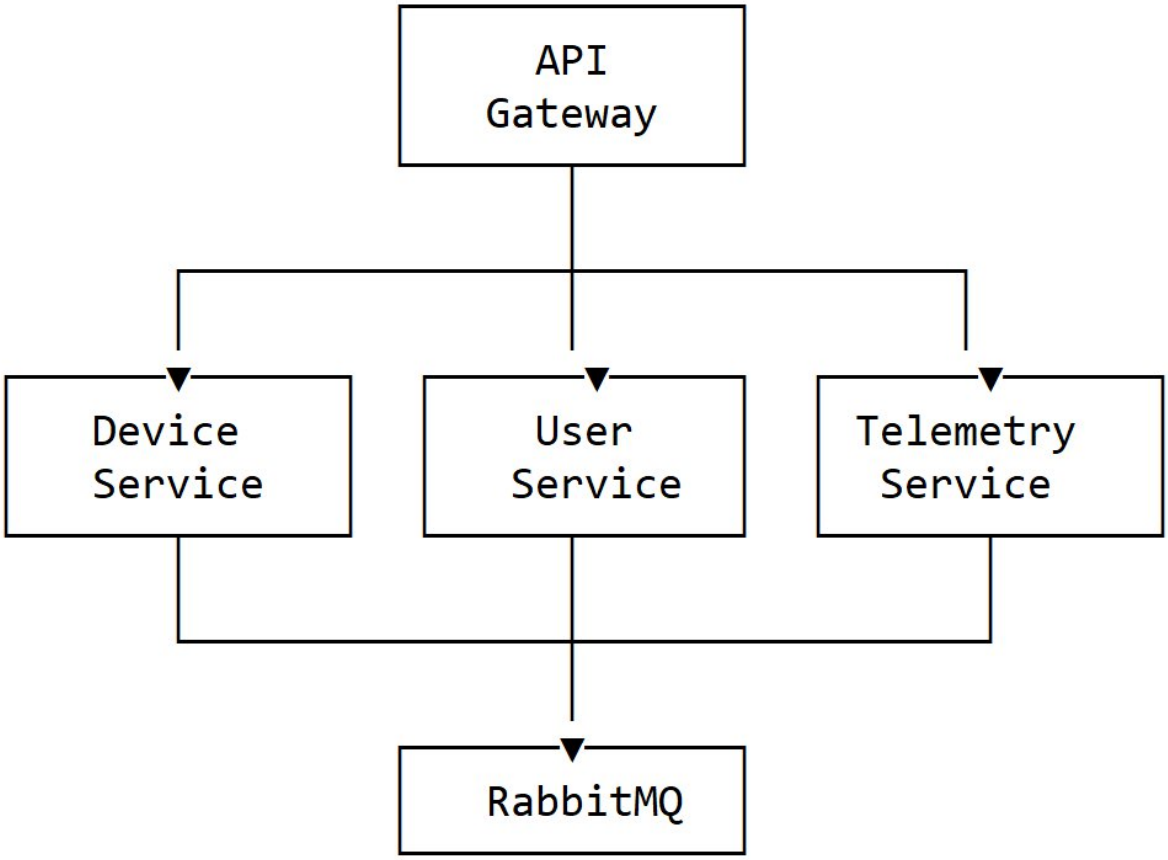
Задачи:

1. Интегрировать все сервисы (Device, User, Notification, Telemetry, Scenario)
2. Настроить взаимодействие между сервисами (синхронное и асинхронное)
3. Выполнить рефакторинг кода (устранение дублирования, длинных методов)
4. Проанализировать технический долг и составить план его устранения
5. Подготовить презентацию проекта (1 часть)

6. Провести ретроспективу работы команды

Теоретическая часть

Интеграция сервисов:



Типы взаимодействия:

Тип	Описание	Пример
Синхронное (REST)	Запрос-ответ, ожидание ответа	Device → User (проверка прав)
Асинхронное (RabbitMQ)	Публикация события, без ожидания	Device → RabbitMQ → Notification

Рефакторинг (каталог Фаулера):

Запах кода	Исправление
Длинный метод	Extract Method
Дублирование кода	Extract Method, Pull Up
Большой класс	Extract Class, Extract Subclass
Длинный список параметров	Introduce Parameter Object
Feature Envy	Move Method
Switch-операторы	Replace Conditional with Polymorphism

Технический долг (Technical Debt):

Тип техдолга	Описание	Пример
Текущий (прямой)	Сознательный компромисс	Отложенное тестирование
Непреднамеренный	Ошибки в проектировании	Плохая архитектура
Устаревание	Технологии устарели	Старые зависимости
Документация	Отсутствие документации	Нет OpenAPI спецификации

2. Примеры рефакторинга

До рефакторинга (дублирование кода):

```
// В DeviceService
public DeviceResponse activateDevice(UUID deviceId, UUID userId) {
```

```
    User user = userService.findById(userId);
    if (!user.getRole().equals(Role.ADMIN)) {
        throw new AccessDeniedException();
    }
    // ... логика активации
}

public DeviceResponse deactivateDevice(UUID deviceId, UUID userId) {
    User user = userService.findById(userId);
    if (!user.getRole().equals(Role.ADMIN)) {
        throw new AccessDeniedException();
    }
    // ... логика деактивации
}
```

После рефакторинга:

```
// Вынесенная проверка прав
private void checkAdminRights(UUID userId) {
    User user = userService.findById(userId);
    if (!user.getRole().equals(Role.ADMIN)) {
        throw new AccessDeniedException();
    }
}

public DeviceResponse activateDevice(UUID deviceId, UUID userId) {
    checkAdminRights(userId);
    // ... логика активации
}

public DeviceResponse deactivateDevice(UUID deviceId, UUID userId) {
    checkAdminRights(userId);
```

```
        // ... логика деактивации
    }
```

Еще пример (длинный метод):

```
    // До рефакторинга
    public void processTelemetry(TelemetryData data) {
        // Валидация
        if (data.getValue() < 0) return;
        if (data.getDeviceId() == null) return;
        if (data.getTimestamp() == null) return;

        // Обогащение
        Device device = deviceRepository.findById(data.getDeviceId()).orElse(null);
        if (device == null) return;
        data.setDeviceName(device.getName());
        data.setRoom(device.getRoom());

        // Сохранение
        telemetryRepository.save(data);

        // Агрегация
        updateAggregatedStats(data);

        // Уведомление
        if (data.getValue() > HIGH_THRESHOLD) {
          notificationService.sendAlert(device.getUserId(), "High value detected");
        }
    }
```

После рефакторинга:

```
    public void processTelemetry(TelemetryData data) {
```

```
        validateTelemetry(data);
        enrichWithDeviceInfo(data);
        saveTelemetry(data);
        updateAggregatedStats(data);
        checkAndNotifyThreshold(data);
    }

    private void validateTelemetry(TelemetryData data) {
        if (data.getValue() < 0) throw new InvalidTelemetryException("Value cannot be negative");
        if (data.getDeviceId() == null) throw new InvalidTelemetryException("Device ID is required");
        if (data.getTimestamp() == null) throw new InvalidTelemetryException("Timestamp is required");
    }

    private void enrichWithDeviceInfo(TelemetryData data) {
        Device device = deviceRepository.findById(data.getDeviceId())
            .orElseThrow(() -> new DeviceNotFoundException(data.getDeviceId()));
        data.setDeviceName(device.getName());
        data.setRoom(device.getRoom());
    }
```

Оценка технического долга:

```
    // Шаблон отчета о техническом долге
    public class TechnicalDebtReport {

        @Data
        public static class DebtItem {
            private String id;
            private String description;      // Описание проблемы
```

```
        private DebtType type;              // CURRENT, UNINTENTIONAL,
OBSOLESCENCE, DOCUMENTATION
        private Severity severity;      // LOW, MEDIUM, HIGH, CRITICAL
        private int estimatedHours;     // Часы на исправление
        private int impactScore;        // Оценка влияния (1-10)
        private String location;        // Файл/класс/метод
    }

    public List<DebtItem> analyzeCode() {
        List<DebtItem> debtItems = new ArrayList<>();

        // Поиск длинных методов (> 30 строк)
        // Поиск дублирования кода
        // Поиск switch/case по типу
        // Поиск TODOs и FIXMEs

        return debtItems;
    }

    public void prioritizeDebtItems(List<DebtItem> items) {
        // Приоритезация: severity + impactScore
        items.sort((a, b) ->
            Integer.compare(b.getImpactScore(), a.getImpactScore())
        );
    }
}
```

Индивидуальные варианты

Вариант 1: Интеграция Device + User Service (синхронная)
Вариант 2: Интеграция Device + Notification (асинхронная, RabbitMQ)
Вариант 3: Интеграция всех трех сервисов (Device, User, Notification)

Вариант 4: Интеграция с добавлением Telemetry Service

Вариант 5: Рефакторинг кода с применением 5+ паттернов из каталога Фаулера

Вариант 6: Анализ технического долга с оценкой времени на исправление

Вариант 7: Написание отчета о ретроспективе команды

Вариант 8: Подготовка полноценной презентации проекта

Контрольные вопросы

1. Какие типы взаимодействия между сервисами существуют?
2. В чем преимущества асинхронного взаимодействия перед синхронным?
3. Что такое рефакторинг? Чем он отличается от переписывания кода?
4. Назовите 5 «запахов» кода по Фаулеру и способы их исправления.
5. Что такое технический долг? Какие виды технического долга существуют?
6. Как оценить приоритет исправления технического долга?
7. Какие инструменты существуют для анализа качества кода?
8. Как настроить взаимодействие между сервисами через REST API?
9. Как обрабатывать ошибки при межсервисном взаимодействии?
10. Что такое retry и circuit breaker? Зачем они нужны?
11. Как организовать логирование в распределенной системе?
12. Что такое ретроспектива в Scrum? Какие вопросы задаются?
13. Как подготовить презентацию проекта? Какова ее структура?
14. Какие метрики качества кода существуют?
15. Как выполнить безопасный рефакторинг без поломки существующей функциональности?