Geekbrains

Разработка Backend-сервиса для управления задачами с использованием Spring Framework и микросервисной архитектуры.

IT-специалист:
Программист Java. Цифровые профессии
Ямпольский И.В.

Санкт-Петербург
2024

1. Введение

1.1 Постановка проблемы и актуальность темы

Сегодня все большее число компаний и коллективов стремятся предложить свои услуги и продукты огромному числу пользователей в различных регионах нашей планеты. Некоторые из них предлагают свои сервисы в десятках стран мира, которые вовсе не находится рядом друг с другом географически.

Капиталистическое устройство мира прежде всего диктует требование к непрерывному увеличению прибыли, которое, в том числе, достигается за счет привлечения новых пользователей, и расширении имеющихся услуг.

Такое положение вещей вызывает необходимость решения проблем, связанных с обеспечением:

- непрерывности в доставке контента и услуг.
- поддержания должного уровня отказоустойчивости и масштабируемости.
- реализации механизмов по управлению сложностью.
- работы над ускорением разработки и доставки изменений потребителю.
- легкости во внедрении нового функционала и его тестированию.
- выбора оптимальных инструментов для решения поставленных задач.

С появлением мобильных приложений, интернета вещей (IoT), облачных вычислений и других технологических трендов, рынок столкнулся с потребностью решения данных проблем, что способствовало росту популярности микросервисов.

Микросервисная архитектура - подход к разработке программного обеспечения, при котором приложение разбивается на небольшие, автономные сервисы, каждый из которых выполняет определенную функцию. Эти сервисы могут быть развернуты, масштабированы и обновлены независимо друг от друга.

Можно выделить ряд причин, по которым микросервисная архитектура остается актуальной:

- **Гибкость и масштабируемость:** микросервисы позволяют легко масштабировать и изменять отдельные компоненты приложения без необходимости внесения изменений в остальные. Это способствует гибкости и ускоряет процесс развертывания новых функций.

- **Управление сложностью:** разделение приложения на небольшие компоненты упрощает его понимание, разработку и обслуживание.

Каждый микросервис имеет ограниченную область ответственности, что уменьшает сложность отладки и разработки.

- **Технологическая гетерогенность:** микросервисы позволяют использовать различные технологии и языки программирования для каждого сервиса в зависимости от его потребностей. Это позволяет командам разработки выбирать наиболее подходящие инструменты для решения конкретных задач.

- **Легкая заменяемость и масштабируемость:** микросервисы могут быть легко заменены или обновлены без влияния на другие части системы. Это упрощает поддержку и развертывание приложений в условиях высокой нагрузки.

- **Распределенная разработка:** микросервисы позволяют командам разработки работать независимо друг от друга, что способствует распределенной разработке и ускоряет процесс разработки.

- **Улучшенная отказоустойчивость:** при наличии микросервисов отказ одного из них не приведет к полному отказу всей системы. Это повышает отказоустойчивость и надежность системы в целом.

Такие гиганты рынка как **Netflix** и **Amazon** сыграли значительную роль в продвижении микросервисной архитектуры. Их опыт и практики стали примеров для многих компаний в области разработки и обслуживания микросервисов.

1.2 Цели и задачи проекта

Основной целью проекта является закрепление знаний, полученных в процессе обучения на платформе, и успешном их применении на практике в процессе написания кода. Более глубокое знакомство с микросервисной архитектурой и способами обеспечения ее безопасности.

Результатом данного дипломного проекта должен стать разработанный Backend-сервис для управления задачами с использованием Spring Framework и микросервисной архитектуры, который предоставляет следующие возможности:

- Регистрация пользователя
- Аутентификация пользователя
- Авторизация пользователя
- Создание задачи
- Редактирование задачи
- Удаление задачи
- Просмотра списка задач

- Получение информации о задаче

Достижения поставленных целей и результата потребует использования таких инструментов как:

- Java Development Kit (JDK): необходим для компиляции и выполнения Java-кода.

- Spring Boot: позволяет создавать самостоятельные, готовые к запуску приложения с минимальной конфигурацией.

- Spring Framework: предоставляет широкий набор инструментов и функциональности для разработки Java-приложений.

- Spring Data JPA: упрощает доступ к данным в базе данных через Java Persistence API.

- Spring MVC: предоставляет архитектурный шаблон для построения веб-приложений.

- Spring Security: предоставляет мощные средства для обеспечения безопасности вашего приложения.

- Spring AOP: улучшение модульности кода путем выделения повторяющихся аспектов, таких как логированиею.

- Spring Integration: предоставляет интеграционные шаблоны и компоненты для разработки сложных интеграционных приложений. Упрощает интеграцию различных систем, сервисов и приложений через различные протоколы, сообщения и API.

- Maven: Это инструмент для управления зависимостями и сборки проекта.

- СУБД MySQL: хранение данных приложения.

- IntelliJ IDEA: среда разработки с удобной интеграцией Spring и Spring Boot.

- DataGrip: среда разработки для работы с базами данных.

- Docker: инструмент запуска изолированных контейнеров для различных приложений.

- Prometheus: система мониторинга и алертинга.

- Grafana: система визуализации данных и мониторинга.

1.3 Основные этапы разработки проекта.

Планирование и проектирование:

- Определение требований к сервису управления задачами.
- Проектирование базы данных для хранения задач и связанных с ними данных.
- Определение структуры API для взаимодействия с клиентами.

Настройка проекта Spring:

- Создание нового проекта Spring Boot.
- Настройка зависимостей для подключения Spring Web, Spring Data JPA, Spring Security и других необходимых библиотек.

Разработка моделей данных:

- Создание сущностей для представления задач и других объектов в базе данных.
- Определение связей между сущностями.

Разработка API контроллеров:

- Создание контроллеров для обработки HTTP-запросов от клиентов.
- Определение эндпоинтов для создания, чтения, обновления и удаления задач.
- Реализация логики обработки запросов в контроллерах.

Разработка сервисного слоя:

- Создание сервисов для обработки бизнес-логики, связанной с управлением задачами.
- Выделение повторяющейся логики в сервисы для повышения модульности и повторного использования.

Настройка доступа к данным:

- Настройка репозиториев Spring Data JPA для взаимодействия с базой данных.
- Использование репозиториев для выполнения операций CRUD с объектами.

Аутентификация и авторизация:

- Настройка Spring Security для обеспечения безопасности сервиса.

- Реализация механизмов аутентификации и авторизации пользователей.

Тестирование:

- Создание юнит-тестов для проверки отдельных компонентов сервиса.
- Написание интеграционных тестов для проверки взаимодействия различных компонентов.
- Запуск автоматических тестов для обеспечения качества кода.

Документирование:

- Создание документации API с помощью Swagger.
- Написание комментариев к коду и пояснительной документации для разработчиков.

Развертывание и мониторинг:

- Развертывание приложения на сервере или в облачной среде.
- Настройка мониторинга для отслеживания производительности и доступности сервиса.

2. Разработка проекта

2.1 Планирование и проектирование

Планирование и проектирование являются важными этапами в разработке любого продукта или проекта. Основными причинами, по которым его действительно стоит осуществлять, являются:

Определение целей и требований: планирование позволяет определить цели проекта и требования к нему. Это помогает всем участникам понять, что должно быть достигнуто в результате, и выработать стратегию для достижения этих целей.

Предотвращение проблем и ошибок: проектирование позволяет выявить потенциальные проблемы и ошибки на ранних этапах разработки, когда их исправление гораздо проще и дешевле, чем в более поздние фазы проекта. Это помогает снизить риски и улучшить качество конечного продукта.

Оптимизация ресурсов: планирование позволяет оптимизировать использование ресурсов, таких как время, бюджет и человеческие ресурсы. Это помогает избежать излишних затрат и неэффективного использования ресурсов.

Обеспечение согласованности: проектирование позволяет создать структурированный и согласованный план действий для всех участников

проекта. Это помогает избежать путаницы и неопределенности, а также обеспечивает единое понимание целей и задач проекта.

Улучшение коммуникации и сотрудничества: Планирование и проектирование способствуют улучшению коммуникации и сотрудничества между участниками проекта. Это помогает им работать в едином направлении и синхронизировать свои усилия для достижения общих целей.

Повышение эффективности и производительности: хорошо спланированный и продуманный проект обычно более эффективен и производителен, чем тот, который разрабатывается на ходу. Планирование и проектирование помогают предвидеть и учесть различные аспекты проекта, что способствует его успешному завершению в срок и в рамках бюджета.

2.1.1 Требования к сервису

Основные функциональные требования, предъявляемые к проекту:

Аутентификация и авторизация:

- Доступ к функциональности управления задачами должен быть защищен аутентификацией и авторизацией.
- Различным пользователям могут быть предоставлены различные уровни доступа в зависимости от их ролей и прав.

Создание задачи:

- Пользователь должен иметь возможность создавать новые задачи, указывая их название, описание, сроки выполнения и другие связанные данные.

Просмотр списка задач:

- Сервис должен предоставлять возможность просмотра списка всех задач.
- Пользователь должен иметь возможность просматривать задачи

Просмотр деталей задачи:

- Пользователь должен иметь возможность просмотра подробной информации о конкретной задаче, включая название, описание, статус, сроки выполнения и другие связанные данные.

Обновление информации о задаче:

- Пользователь должен иметь возможность обновлять информацию о задаче, такую как название, описание, статус, сроки выполнения и другие атрибуты.

Удаление задачи:

- Пользователь должен иметь возможность удалять задачи из системы, если они больше не нужны или были ошибочно созданы.

Фильтрация и поиск задач:

- Сервис должен предоставлять возможность фильтрации и поиска задач по различным критериям, таким как название, статус, сроки выполнения и другие атрибуты.

Управление статусом задачи:

- Пользователь должен иметь возможность изменять статус задачи, например, помечать задачу как "выполненную", "в процессе выполнения" или "отмененную".

Журналирование и мониторинг:

- Сервис должен вести журнал всех операций, выполняемых с задачами, для обеспечения прозрачности и отслеживания изменений.
- Может быть необходим мониторинг производительности и доступности сервиса для обеспечения надежной работы.

К нефункциональным требованиям стоит отнести:

Производительность:

- Сервис должен обеспечивать быструю и отзывчивую работу, чтобы пользователи могли эффективно управлять своими задачами.
- Максимальное время ответа API не должно превышать, например, 200 миллисекунд при нагрузке до 1000 запросов в секунду.

Масштабируемость:

- Сервис должен быть масштабируемым и способным обрабатывать увеличение числа пользователей и объема данных без существенного снижения производительности.
- Должна быть обеспечена горизонтальная масштабируемость, позволяющая добавлять новые экземпляры сервиса для распределения нагрузки.

Надежность:

- Сервис должен быть надежным и иметь высокую доступность, чтобы минимизировать время простоя и предотвратить потерю данных.
- Время непланового простоя системы должно быть менее 1 часа в год.

Безопасность:[1]

- Сервис должен обеспечивать защиту данных и конфиденциальность информации пользователей.
- Должны применяться современные методы шифрования для защиты данных в покое и в движении.
- Доступ к административным функциям должен быть ограничен и контролируем.

Совместимость:

- Сервис должен быть совместим с различными браузерами и устройствами, чтобы пользователи могли получить к нему доступ с различных платформ.
- API сервиса должно быть совместимо с существующими инструментами и клиентскими приложениями.

Технологические требования:

- Сервис должен быть разработан с использованием современных технологий и стандартов разработки, чтобы обеспечить долгосрочную поддержку и расширяемость.
- Должны использоваться инструменты и практики DevOps для автоматизации развертывания, тестирования и мониторинга сервиса.

Удобство использования:

- Интерфейс пользователя должен быть интуитивно понятным и легко навигируемым, чтобы пользователи могли быстро освоить его и начать работу без дополнительного обучения.

2.1.2 Проектирование баз данных для микросервисов

Сама суть микросервисной архитектуры состоит в том, что каждый отдельный сервис представляет собой приложение, которое выполняет одну единственную задачу, и содержит в себе все необходимое, включая экземпляр базы данных.

Таким образом для сервиса, управляющего пользователями, база данных будет состоять фактически из одной таблицы, которая будет отвечать за хранение пользователей. Набор минимальный набор атрибутов будет следующим:

- id (идентификатор пользователя, первичный ключ)
- username (имя пользователя)
- email (адрес электронной почты пользователя)
- password (пароль пользователя)
- active (состояние ученой записи пользователя)

- roles (роль пользователя в системе)

Следуя данной логике, для микросервиса, управляющего задачами, атрибуты для таблицы, в которой будут храниться задачи примут вид:

- id (идентификатор задачи, первичный ключ)
- title (заголовок задачи)
- description (описание задачи)
- status (статус задачи)
- ownerId (id владельца)
- created (дата создания)
- lastUpdate (дата последнего обновления)
- finished (дата завершения)

В связи с тем, что работа по созданию базы данных и работой с ней будет целиком и полностью возложена на Spring Data JPA, нам потребуется только описаться классы и задать необходимые аннотации в них.

2.1.3 Определение структуры API

Определение структуры API играет ключевую роль в разработке программного обеспечения и взаимодействии между компонентами системы, способствует эффективной интеграции, облегчает разработку клиентских приложений и обеспечивает безопасность и надежность системы. Данный этап позволяет добиться следующего:

Ясного понимания возможностей сервиса:

- Определение структуры API позволяет разработчикам и клиентам четко понимать, какие операции могут быть выполнены с помощью API, какие данные могут быть переданы и какие результаты ожидаются.

Стандартизации взаимодействия:

- Хорошо определенная структура API позволяет стандартизировать взаимодействие между различными компонентами системы. Это упрощает интеграцию и обмен данными между различными сервисами и приложениями.

Облегчения разработки и поддержки клиентских приложений:

- Заранее определенная структура API упрощает процесс разработки клиентских приложений, так как разработчики знают, как обращаться к API и какие данные ожидать в ответе. Это также облегчает поддержку клиентских приложений при изменении API.

Безопасности:

- Определение структуры API позволяет реализовать механизмы аутентификации и авторизации, контролировать доступ к различным ресурсам и обеспечивать безопасность обмена данными между клиентом и сервером.

Облегчения процессов документирования и обучения:

- Хорошо определенная структура API облегчает процесс создания документации для API, что помогает пользователям понять, как использовать сервис, и обучиться работе с ним.

Улучшение масштабируемости:

- Хорошо спроектированная и структурированная API облегчает масштабирование системы, так как позволяет легко добавлять новые функции и изменять существующие без значительных изменений в клиентских приложениях.

Понимание важности данного этапа позволило на этапе проектирования приложения определить основу структуры API для микросервисов. Но не менее значительным было решение о том, какой же ответ отдавать клиентской части во всех возможных случаях, включая ошибки. В идеальном варианте хотелось бы получать информацию, что именно произошло в микросервисе, а не просто сообщать, что пользователь или задача не найдены.

Для ответа клиентской части мной был разработан класс CustomResponse<T>, который представляет собой обобщённый класс, и предназначен для управления ответами, возвращаемыми контроллерами микросервисов.

Компоненты класса:

- **Обобщённый тип T:** Обобщённый тип данных, который представляет собой тип данных ответа. Этот тип может быть любым и будет определён в момент создания объекта CustomResponse.

Поля класса:

- **errorCode:** Целочисленное поле, представляющее код ошибки или статуса ответа.
- **responseData:** Поле обобщённого типа T, которое содержит данные успешного ответа.
- **responseError:** Объект класса ExceptionData, содержащий информацию об ошибке, если она возникла.

Конструкторы:

- **public CustomResponse():** Пустой конструктор, используется для создания объектов CustomResponse без параметров.

- **public CustomResponse(int errorCode, T responseData):** Конструктор, принимающий код ошибки и данные успешного ответа. Используется, когда ответ успешен и не содержит ошибки.
- **public CustomResponse(int errorCode, ExceptionData responseError):** Конструктор, принимающий код ошибки и объект ExceptionData, содержащий информацию об ошибке. Используется, когда в ответе возникла ошибка.

Итого решения использовать такой подход стал тот факт, что ответ по всем эндпоинтам микросервисов будет следующим:

- Статус ответа: 200 OK всегда.
- Тип данных тела ответа: CustomResponse<T>

  Если запрос выполненуспешно CustomResponse содержит:

  - В поле errorCode: 0
  - Поле responseData: содержит запрашиваемый объект.
  - Поле responseError: содержит null.

  Если произошла ошибка:

  - Поле errorCode будет равно 1.
  - Поле responseData будет равно null.
  - Поле responseError будет содержать информацию об ошибке в объекте ExceptionData.

Для микросервиса управления пользователями:

Получение всех пользователей:

- Метод: GET
- Endpoint: /api/users
- Описание: Получение списка всех пользователей.
- Тип данных тела ответа: CustomResponse<List<User>>.

Получение пользователя по его идентификатору:

- Метод: GET
- Endpoint: /api/users/{id}
- Описание: Получение информации о пользователе по его идентификатору.
- Параметры:
  - id: Идентификатор пользователя.
- Тип данных тела ответа: CustomResponse<User>.

Поиск пользователя по его имени пользователя:

- Метод: GET
- Endpoint: /api/users/find/{username}
- Описание: Поиск пользователя по его имени пользователя.
- Параметры:
    - username: Имя пользователя для поиска.
- Тип данных тела ответа: CustomResponse<User>.

Создание нового пользователя:

- Метод: POST
- Endpoint: /api/users
- Описание: Создание нового пользователя.
- Тело запроса: Данные нового пользователя в формате JSON.
- Тип данных тела ответа: CustomResponse<User>.

Обновление данных пользователя:

- Метод: PUT
- Endpoint: /api/users/{id}
- Описание: Обновление данных пользователя по его идентификатору.
- Параметры:
    - id: Идентификатор пользователя.
- Тело запроса: Новые данные пользователя в формате JSON.
- Тип данных тела ответа: CustomResponse<User>.

Удаление пользователя по его идентификатору:

- Метод: DELETE
- Endpoint: /api/users/{id}
- Описание: Удаление пользователя по его идентификатору.
- Параметры:
    - id: Идентификатор пользователя.
- Тип данных тела ответа: CustomResponse<User>.

Для микросервиса управления задачами:

Получение списка задач пользователя:

- Метод: GET
- Endpoint: /api/tasks/user/{id}
- Описание: Получение списка задач пользователя по его идентификатору.
- Параметры:
    - id: Идентификатор пользователя.
- Тип данных тела ответа: CustomResponse<List<Task>>.

Получение списка всех задач:

- Метод: GET
- Endpoint: /api/tasks
- Описание: Получение списка всех задач.
- Тип данных тела ответа: CustomResponse<List<Task>>.

Получение задачи по идентификатору:

- Метод: GET
- Endpoint: /api/tasks/{id}
- Описание: Получение информации о задаче по её идентификатору.
- Параметры:
    - id: Идентификатор задачи.
- Тип данных тела ответа: CustomResponse<Task>.

Создание новой задачи:

- Метод: POST
- Endpoint: /api/tasks
- Описание: Создание новой задачи.
- Тело запроса: Данные новой задачи в формате JSON.
- Тип данных тела ответа: CustomResponse<Task>.

Обновление данных задачи:

- Метод: PUT
- Endpoint: /api/tasks/{id}
- Описание: Обновление данных задачи по её идентификатору.
- Параметры:
    - id: Идентификатор задачи для обновления.

- Тело запроса: Новые данные задачи в формате JSON.
- Тип данных тела ответа: CustomResponse<Task>.

Удаление задачи по идентификатору:

- Метод: DELETE
- Endpoint: /api/tasks/{id}
- Описание: Удаление задачи по её идентификатору.
- Параметры:
  - id: Идентификатор задачи для удаления.
- Тип данных тела ответа: CustomResponse<Task>.

2.2 Создание проекта

Помня о задачах, решение которых преследует мой проект, о том, какой функционал он должен реализовывать, а также основные особенности архитектуры, которую я решил использовать, настройка и создание проекта сводится к конфигурированию нескольких небольших приложений, функциональность которых будет ограничена одним узким направлением. Виду схожести многих моментов, будут описаны отличительные особенности настройки каждого из микросервисов.

К основным этапам можно отнести:

- **Выбор инструмента сборки проекта и среды разработки:** мной было принято решение использовать IntelliJ IDEA и Maven, как наиболее знакомые мне инструменты для сборки проекта и управлению его зависимостями.
- **Структурирование проекта:** функционал разделен на отдельные слои и модули, с целью упрощения разработки проекта и его дальнейшее усовершенствование.
- **Определение необходимых зависимостей:** выбор осуществлял с учетом требуемого функционала, добавляются в pom.xml.
- **Конфигурация проекта:** определена в application.yml или в классах аннотированных @Configuration.
- **Написание классов:** классы реализуют задуманный функционал.
- **Написание тестов:** для тестирования методов классов и взаимодействия основных модулей между собой.

2.2.1 Структура проекта

Выбор микросервисной архитектуры диктует условия, когда каждая функциональная единица архитектуры становится отельным маленьким приложением со своей внутренней архитектурой.

На рисунке 1 приведена структура проекта относительно каталога /project, находящегося в корне репозитория.

рисунок 1

- **ApiGateWay:** Этот микросервис является точкой входа для всех внешних запросов в систему. Он отвечает за маршрутизацию запросов к соответствующим микросервисам . (рисунок 2)

рисунок 2

- **TaskMicroService:** Этот микросервис управляет задачами в системе. Он предоставляет API для создания, чтения, обновления и удаления задач. (рисунок 3)

рисунок 3

- **UserMicroService:** Этот микросервис управляет пользователями в системе. Он предоставляет API для создания, чтения, обновления и удаления пользователей. (рисунок 4)

рисунок 4

- **TaskClient:** Этот микросервис отвечает за взаимодействие с другими сервисами для управления задачами. Он может общаться с другими сервисами для получения информации о задачах, пользователях и их обновления.(рисунок 5)

рисунок 5

- **EurekaServer:** Eureka - это сервер реестра, реализующий паттерн Service Discovery. Он предоставляет механизм для регистрации, обнаружения и выбора микросервисов в системе. (рисунок 6)

рисунок 6

- **configs:** Этот каталог для конфигурационных файлов, в том числе файлов конфигурации для системы мониторинга Prometheus.

- **docker-compose.yml:** Этот файл используется для определения и запуска нескольких контейнеров Docker, каждый из которых соответствует одному из микросервисов и связанных сервисов (например, базы данных, Prometheus, Grafana).
- **tms.env и ums.env:** Эти файлы содержат переменные окружения, которые используются для настройки соответствующих микросервисов (TaskMicroService и UserMicroService).

2.2.2 Подключение зависимостей

Набор зависимостей, используемых в каждом из микросервисов определяется его функционалом. Ниже приведу краткое описание зависимостей, используемых в каждом из микросервисов.

Общие для всех микросервисов:

- **spring-boot-starter-actuator:** предоставляет функциональность мониторинга и управления для приложений Spring Boot. Включает в себя

различные конечные точки, которые можно использовать для мониторинга состояния приложения.

- **micrometer-core и micrometer-registry-prometheus:** предоставляют интеграцию с библиотекой Micrometer, которая предназначена для сбора и экспорта метрик приложения. Micrometer-registry-prometheus обеспечивает экспорт метрик для системы мониторинга Prometheus.
- **lombok:** является необязательной и используется для упрощения разработки с помощью аннотаций. Lombok генерирует геттеры, сеттеры, конструкторы и другие методы автоматически на этапе компиляции.
- **spring-boot-starter-test:** предоставляет стартовые зависимости для написания и запуска тестов в Spring Boot приложениях. Включает в себя различные библиотеки и инструменты для модульного, интеграционного и функционального тестирования.

EurekaServer:

- **spring-cloud-starter-netflix-eureka-server:** Эта зависимость обеспечивает поддержку сервера реестра Eureka в Spring Cloud. С помощью Eureka Server микросервисы могут регистрироваться и находить друг друга в распределенной среде.

Каждый оставшийся микросервис в моем проекте является клиентом EurekaServer.

Для этих клиентов требуется зависимость:

- **spring-cloud-starter-netflix-eureka-client:** позволяет микросервису регистрироваться в сервере реестра Eureka и находить другие зарегистрированные микросервисы.

ApiGateWay:

- **spring-cloud-starter-gateway:** Эта зависимость обеспечивает поддержку шлюза API в Spring Cloud. Она позволяет маршрутизировать запросы HTTP на различные микросервисы и выполнять другие функции, связанные с шлюзом API.

TaskClient:

- **spring-boot-starter-thymeleaf:** предоставляет зависимости для использования Thymeleaf в Spring Boot приложениях. Thymeleaf - это

шаблонизатор, который используется для создания пользовательского интерфейса веб-приложений.

- **spring-boot-starter-web:** предоставляет зависимости для создания веб-приложений с использованием Spring MVC (Model-View-Controller).
- **spring-boot-starter-security:** предоставляет зависимости для включения безопасности в Spring Boot приложения. Она используется для обеспечения аутентификации и авторизации в приложении.
- **spring-boot-starter-integration:** предоставляет интеграцию Spring Integration.
- **spring-boot-starter-validation:** предоставляет зависимости для валидации данных в приложениях Spring Boot.

TaskMicroService и UserMicroService:

- **spring-boot-starter-data-jpa:** предоставляет зависимости для использования Spring Data JPA, что упрощает работу с базой данных.
- **spring-boot-starter-web:** Эта зависимость для реализации функционала RestController.
- **spring-boot-docker-compose:** позволяет легко интегрировать приложение Spring Boot с Docker Compose для развертывания и управления контейнеризированными приложениями.
- **mysql-connector-j:** предоставляет JDBC драйвер для взаимодействия с базой данных MySQL.
- **spring-boot-starter-validation:** предоставляет зависимости для валидации данных в приложениях Spring Boot.

2.3 Разработка моделей данных

Основными сущностями в проекте будут пользователи сервиса и их задачи. Поэтому в разработке моделей я отталкивался именно от этого. В п. 2.1.2 я говорил о том, какие данные будут хранить базы данных соответствующих сервисов.

Поэтому для сервиса пользователей будет две таблицы:

Первая, в которой хранятся непосредственно пользователи, ее атрибуты буду следующими:

- id BIGINT AUTO_INCREMENT PRIMARY KEY
- username VARCHAR(255) NOT NULL
- email VARCHAR(255) NOT NULL
- password VARCHAR(255) NOT NULL

- active BOOLEAN

Вторая, которая будет соотносить пользователей и их роли:

- user_id BIGINT
- role VARCHAR(255)

Для сервиса задач, таблица будет иметь следующие атрибуты:

- id BIGINT AUTO_INCREMENT PRIMARY KEY
- title VARCHAR(255)
- description TEXT
- status VARCHAR(255)
- ownerId BIGINT
- created DATETIME
- lastUpdate DATETIME
- finished DATETIME

2.3.1 Создание сущностей для представления в базе данных

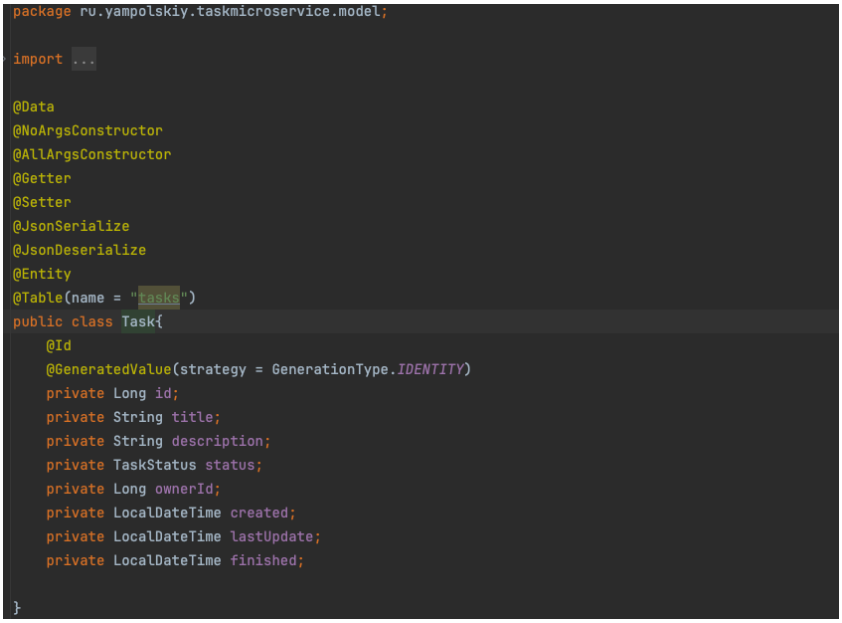
Поэтому классы определяющие сущности для представления в базе данных мной были разработаны следующего вида:

Для сервиса, управляющего пользователями:

```java
package ru.yampolskiy.usermicroservice.model;

import ...

@Data
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@JsonSerialize
@JsonDeserialize
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    private Long id;
    @NotBlank(message = "Логин не может быть пустым")
    @Size(min = 4, max = 15)
    private String username;
    @NotBlank(message = "Адрес электронной почты не может быть пустым")
    @Email
    private String email;
    @NotNull(message = "Пароль не может быть пустым")
    @Size(min = 8)
    private String password;
    private boolean active;
    @ElementCollection(targetClass = Role.class, fetch = FetchType.EAGER)
    @CollectionTable(name ="user_role", joinColumns = @JoinColumn(name = "user_id"))
    @Enumerated(EnumType.STRING)
    private Set<Role> roles;

}
```

рисунок 7

Для сервиса управляющего задачами:



```java
package ru.yampolskiy.taskmicroservice.model;

import ...

@Data
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@JsonSerialize
@JsonDeserialize
@Entity
@Table(name = "tasks")
public class Task{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    private TaskStatus status;
    private Long ownerId;
    private LocalDateTime created;
    private LocalDateTime lastUpdate;
    private LocalDateTime finished;

}
```

рисунок 8

2.4 Разработка API контроллеров

Разработка API контроллеров в приложениях на Spring Boot - это процесс создания точек входа (эндпоинтов) для взаимодействия с приложением через HTTP запросы. К основным шагам этого процесса можно отнести:

- **Определение эндпоинтов:** на этом этапе нужно решить, какие операции стоит выполнять в приложении через API. Это могут быть операции чтения, создания, обновления или удаления данных. Для каждой операции нужно определить соответствующий эндпоинт.
- **Аннотация класса и методов контроллера:** В Spring Boot контроллеры обычно помечаются аннотацией @RestController, а методы обрабатывающие HTTP запросы, помечаются аннотациями, такими как @GetMapping, @PostMapping, @PutMapping, @DeleteMapping в зависимости от типа запроса.
- **Параметры запроса и тело запроса:** стоит заранее определить, какие параметры запроса или данные JSON будут передаваться в ваш метод контроллера. В Spring Boot для этого используются аннотации @RequestParam для параметров запроса и @RequestBody для получения данных из тела запроса в виде объекта.
- **Обработка запросов:** написание логики обработки запросов в методах контроллера. Чаще всего в конечном итоге это вызов сервисного слоя приложения для выполнения бизнес-логики, получение или сохранение данных в базе данных и т. д.
- **Возвращение ответа:** очень важным является определение того, что метод контроллера должен возвращать в ответ на запрос. Это могут быть

объекты, которые автоматически сериализуются в JSON, как в моем случае, а также не упустить момент управления HTTP статусами ответа.

- **Обработка ошибок:** должна быть понятная обработка ошибок, которые могут возникнуть при выполнении запросов. Для этих целей можно использовать аннотацию @ExceptionHandler для обработки исключений и возвращения соответствующих ответов.
- **Тестирование контроллеров:** написание модульных и интеграционных тестов для контроллера, чтобы убедиться, что он работает правильно и отвечает на запросы ожидаемым образом, так же является очень важным, что позволяет на ранних этапах выявить проблемы.

Этот этап стал одним из самых трудоемких в моей работе. Ввиду отсутствия опыта в разработке подобных механизмов, нащупать решение получилось не сразу. Хотя приложение не большое, и в нем всего несколько сервисов, которые участвуют в обработке реальных данных, различие типов этих данных в ответах на запросы заставило потерять много времени не переписывание фактически всей основной логики. Я уже частично касался данного обстоятельства, говоря о том, что мне потребовалось создать класс CustomResponse<T>, представляющий собой единый ответ от всех контроллеров всех сервисов, которые отдают данные клиенту.

2.4.1 Создание контроллеров для обработки HTTP-запросов от клиентов и реализация логики обработки запросов в контроллерах

В моем приложении два сервиса, которые будут осуществлять обработку запросов от клиента, который непосредственно взаимодействует с пользователями. Оба используют java-класс, представляющий собой контроллер REST, и предоставляют конечные точки для получения , создания, обновления и удаления сущностей, для управления которыми создавались. Классы используют аннотации Spring, такие как @RestController, @GetMapping, @PostMapping, @PutMapping и @DeleteMapping для определения сопоставлений HTTP-запросов.

2.4.1.1 Сервис управления пользователями

Как видно на рисунке 9 класс UserController аннотирован как @RestController, с помощью аннотации @RequestMapping предоставляет общий путь запроса для всех конечных точке контроллера.

Содержит в себе экземпляр объекта сервисного слоя, представленного классом UserService, для получения доступа к логике работы с пользователями.

Так же на рисунке виден первый эндпоинт, который предназначен для получения списка всех пользователей, зарегистрированных в системе.

```java
/**
 * Контроллер для работы с пользователями.
 */
@Data
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    /**
     * Получение всех пользователей.
     * @return ответ с кодом статуса и списком пользователей
     */
    @GetMapping
    public ResponseEntity<CustomResponse<List<User>>> getAllUsers() {
        List<User> users = userService.getAllUsers();
        CustomResponse<List<User>> customResponse = new CustomResponse<>( errorCode: 0, users);
        return ResponseEntity.ok(customResponse);
    }
```

рисунок 9

На рисунке 10 представлен ендпоинт, который возвращает пользователя по его id.

```java
    /**
     * Получение пользователя по его идентификатору.
     * @param id идентификатор пользователя
     * @return ответ с кодом статуса и данными пользователя
     */
    @GetMapping("/{id}")
    public ResponseEntity<CustomResponse<User>> getUserById(@PathVariable Long id) {
        User user = userService.getUserById(id);
        CustomResponse<User> customResponse = new CustomResponse<>( errorCode: 0, user);
        return ResponseEntity.ok(customResponse);
    }
```

рисунок 10

При реализации возможности регистрации, аутентификации и авторизации пользователей в моем сервисе возникла необходимость поиска пользователей по их имени. Для этого был написан эндпоинт представленный на рисунке 11.

```java
    /**
     * Поиск пользователя по его имени пользователя.
     * @param username имя пользователя
     * @return ответ с кодом статуса и данными пользователя
     */
    @GetMapping("/find/{username}")
    public ResponseEntity<CustomResponse<User>> findUserByUsername(@PathVariable String username) {
        User user = userService.findUserByUserName(username);
        CustomResponse<User> customResponse = new CustomResponse<>( errorCode: 0, user);
        return ResponseEntity.ok(customResponse);
    }
```

рисунок 11

На рисунке показаны три оставшиеся точки предоставляющие внешнему клиенту в целом стандартный функционал по созданию, обновлению и удалению пользователей в системе.

```
    /**
     * Создание нового пользователя.
     * @param user данные нового пользователя
     * @return ответ с кодом статуса и данными созданного пользователя
     */
    @PostMapping
    public ResponseEntity<CustomResponse<User>> createUser(@Valid @RequestBody User user) {
        User createdUser = userService.createUser(user);
        CustomResponse<User> customResponse = new CustomResponse<>( errorCode: 0, createdUser);
        return ResponseEntity.ok(customResponse);
    }

    /**
     * Обновление данных пользователя.
     * @param id идентификатор пользователя
     * @param user новые данные пользователя
     * @return ответ с кодом статуса и данными обновленного пользователя
     */
    @PutMapping("/{id}")
    public ResponseEntity<CustomResponse<User>> updateUser(@PathVariable Long id, @Valid @RequestBody User user) {
        User updatedUser = userService.updateUser(id, user);
        CustomResponse<User> customResponse = new CustomResponse<>( errorCode: 0, updatedUser);
        return ResponseEntity.ok(customResponse);
    }

    /**
     * Удаление пользователя по его идентификатору.
     * @param id идентификатор пользователя
     * @return ответ с кодом статуса
     */
    @DeleteMapping("/{id}")
    public ResponseEntity<CustomResponse<User>> deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        CustomResponse<User> customResponse = new CustomResponse<>(0, null);
        return ResponseEntity.ok(customResponse);
    }
}
```

рисунок 12

2.4.1.2 Сервис управления задачами

Сервис управления задачами во многом очень похож в плане реализации на сервис управления пользователями в том смысле что они реализуют одинаковую модель, которая позволяет управлять сущностями, за которые они отвечают. Поэтому TaskController имеет много общего с UserController, по этой причине не стану приводить весь код данного класса покажи лишь общую структуру.

Как можно видеть на рисунке 13 TaskController использует те же аннотации и подход, что и контроллер из сервиса, управляющего пользователями, с той лишь разницей, что его задачей является управление задачами.

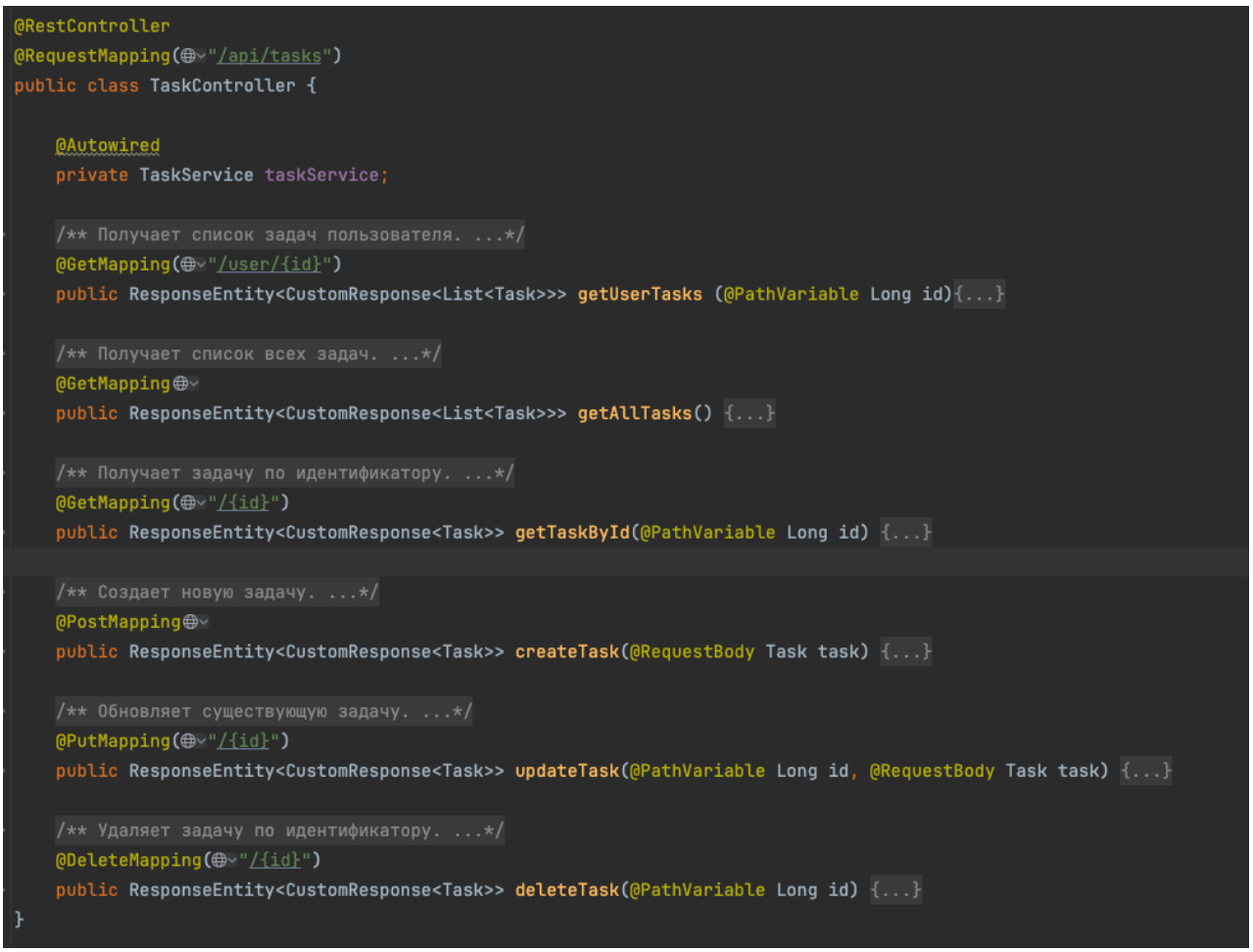


рисунок 13

2.5 Разработка сервисного слоя и настройка доступа к данным.

Прежде чем перейти к описанию мне следует сказать, работа над каждым из сервисов велась часто независимо. И с целью упрощения разработки, базы данных, в которых хранятся данные развертывались с помощью Docker. Поэтому в корне сервисов можно найти docker-compose.yml и .env для автоматизации этого процесса с помощью возможностей среды разработки.

На завершающих этапах в каждый сервис был добавлен Dockerfile, который будет использоваться для создания образов для сборки и тестирования микросервисов, а так же для создания образов, непосредственно, с самим приложением. Что позволит одной единственной командой развернуть контейнеры в Docker.

Второй очень важный момент, который следует отметить это та технология, которую я выбрал для непосредственной работы с базами данных.

Spring Data JPA предоставляет удобный способ работы с базами данных в приложениях, использующих Spring Framework. JPA (Java Persistence API) является стандартом Java EE для управления объектно-реляционным

отображением (ORM), а Spring Data JPA обеспечивает удобный интерфейс и инструменты для работы с JPA в приложениях Spring.

Основные концепции и возможности Spring Data JPA:

- **Репозитории:** Spring Data JPA предоставляет механизм репозиториев, который позволяет создавать репозитории для взаимодействия с базой данных без написания рутины кода. Репозитории могут предоставлять стандартные методы для выполнения CRUD операций (создание, чтение, обновление, удаление), а также пользовательские методы для запросов данных.
- **Генерация запросов**: Spring Data JPA позволяет создавать запросы на основе именованных методов репозиториев, используя соглашения об именах методов. Например, Spring Data JPA автоматически создаст запрос на выборку данных по имени, если метод репозитория будет назван соответственно.
- **Query DSL (Domain Specific Language):** Для более сложных запросов, которые нельзя создать с помощью соглашений об именах методов, Spring Data JPA позволяет создавать пользовательские запросы с использованием Query DSL. Это позволяет писать запросы на языке Java, что обеспечивает типобезопасность и удобство.
- **Поддержка различных баз данных:** Spring Data JPA обеспечивает абстракцию уровня хранилища данных, позволяя легко переключаться между различными базами данных. Он поддерживает большинство популярных реляционных баз данных, таких как MySQL, PostgreSQL, Oracle, и другие.
- **Транзакционная управляемость**: Spring Data JPA интегрируется с механизмом управления транзакциями Spring, что обеспечивает автоматическое управление транзакциями при выполнении операций с базой данных.
- **Внедрение зависимостей:** Spring Data JPA интегрируется с Spring Framework и автоматически создает бины репозиториев, что позволяет внедрять и использовать их в других компонентах приложения.
- **Поддержка пагинации и сортировки:** Spring Data JPA предоставляет удобные методы для выполнения запросов с пагинацией и сортировкой результатов, что позволяет эффективно работать с большими объемами данных.

Все это значительно упростило разработку функционала приложения, связанного с базами данных, позволяя сосредоточиться на бизнес-логике, а не на деталях взаимодействия с базой данных.

2.5.1 Настройка репозиториев Spring Data JPA для взаимодействия с базой данных.

На текущий момент у меня уже описаны классы, которые будут отражать сущности пользователей и их задачи в базе данных. Поэтому создание репозиториев в моем приложении сведется к созданию интерфейсов в сервисах, которые будут генерировать запросы к базе данных.

На рисунке 14 представлено представление репозитория в сервисе, управляющего пользователями:

```java
/**
 * Репозиторий для работы с сущностью пользователя.
 */
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    /**
     * Находит пользователя по имени пользователя.
     * @param username Имя пользователя.
     * @return Найденный пользователь, если существует.
     */
    Optional<User> findUserByUsername(String username);

    /**
     * Удаляет пользователя по идентификатору.
     * @param id Идентификатор пользователя.
     * @return Удаленный пользователь, если существует.
     */
    Optional<User> deleteUserById(Long id);

    /**
     * Проверяет существование пользователя по имени пользователя.
     * @param username Имя пользователя.
     * @return true, если пользователь с таким именем существует, иначе false.
     */
    boolean existsUserByUsername(String username);
}
```

рисунок 14

Как видно в нем дополнительно описаны методы, возвращающие пользователя по его имени, удаляющие пользователя по его id, и проверяющие существование пользователя с заданным именем.

На рисунке 15 показано представление репозитория в сервисе, управляющего задачами:

```java
/**
 * Репозиторий для работы с сущностями задач в базе данных.
 */
public interface TaskRepository extends JpaRepository<Task, Long> {

    /**
     * Находит задачу по идентификатору владельца.
     * @param id Идентификатор владельца задачи.
     * @return Опциональный объект, содержащий найденную задачу или пустое значение.
     */
    Optional<Task> findTaskByOwnerId(long id);

    /**
     * Находит все задачи по идентификатору владельца.
     * @param id Идентификатор владельца задач.
     * @return Опциональный список задач, принадлежащих указанному владельцу, или пустой список.
     */
    Optional<List<Task>> findAllByOwnerId(long id);
}
```

рисунок 15

2.5.2 Создание сервисного слоя для обработки бизнес-логики, связанной с управлением задачами

Структура класса реализующего сервисный слой управления задачами, представлена на рисунке 16.

```java
/**
 * Сервис для управления задачами.
 */
@Service
public class TaskService {

    @Autowired
    private TaskRepository taskRepository;

    /** Получает все задачи. ...*/
    public List<Task> getAllTasks() { return taskRepository.findAll(); }

    /** Получает все задачи пользователя. ...*/
    public List<Task> getAllUserTask(Long id) {...}

    /** Получает задачу по идентификатору. ...*/
    public Task getTaskById(Long id) {...}

    /** Создает новую задачу. ...*/
    public Task createTask(Task task) {...}

    /** Обновляет существующую задачу. ...*/
    public Task updateTask(Long id, Task task) {...}

    /** Удаляет задачу по идентификатору. ...*/
    public void deleteTask(Long id) {...}
}
```

рисунок 16

Класс TaskService аннотирован как @ Service, инкапсулирует бизнес-логику для работы с задачами и обеспечивает простой интерфейс для их управления, скрывая детали взаимодействия с базой данных через TaskRepository. Кроме того, он обрабатывает исключения, связанные с операциями CRUD, чтобы обеспечить надежность и безопасность операций.

Реализует в себе следующие методы:

- **getAllTasks():** возвращает список всех задач. Он использует метод findAll() из TaskRepository, чтобы получить все задачи из базы данных.ъ

- **getAllUserTask(Long id):** метод получает список задач для определенного пользователя по его идентификатору. Он вызывает метод findAllByOwnerId(id) из TaskRepository, который ищет все задачи, принадлежащие указанному пользователю.

- **getTaskById(Long id):** возвращает задачу по ее идентификатору. Он использует метод findById(id) из TaskRepository для поиска задачи по идентификатору. Если задача не найдена, он бросает исключение TaskNotFoundException.

- **createTask(Task task):** создает новую задачу. Он проверяет, что идентификатор задачи не установлен (должен быть null), иначе бросает исключение TaskIdNotNullException. Затем он устанавливает дату создания, статус задачи и сохраняет задачу в репозитории.

- **updateTask(Long id, Task task):** обновляет существующую задачу. Он проверяет существование задачи с указанным идентификатором. Если задача не найдена, он бросает исключение TaskNotFoundException. Затем он устанавливает идентификатор задачи и дату последнего обновления, и сохраняет обновленные данные задачи в репозитории.

- **deleteTask(Long id):** удаляет задачу по ее идентификатору. Он проверяет существование задачи с указанным идентификатором. Если задача не найдена, он бросает исключение TaskNotFoundException. В противном случае, он удаляет задачу из репозитория.

2.5.3 Создание сервисного слоя для обработки бизнес-логики, связанной с управлением пользователями

Структура класса реализующего сервисный слой управления пользователями, представлена на рисунке 17.

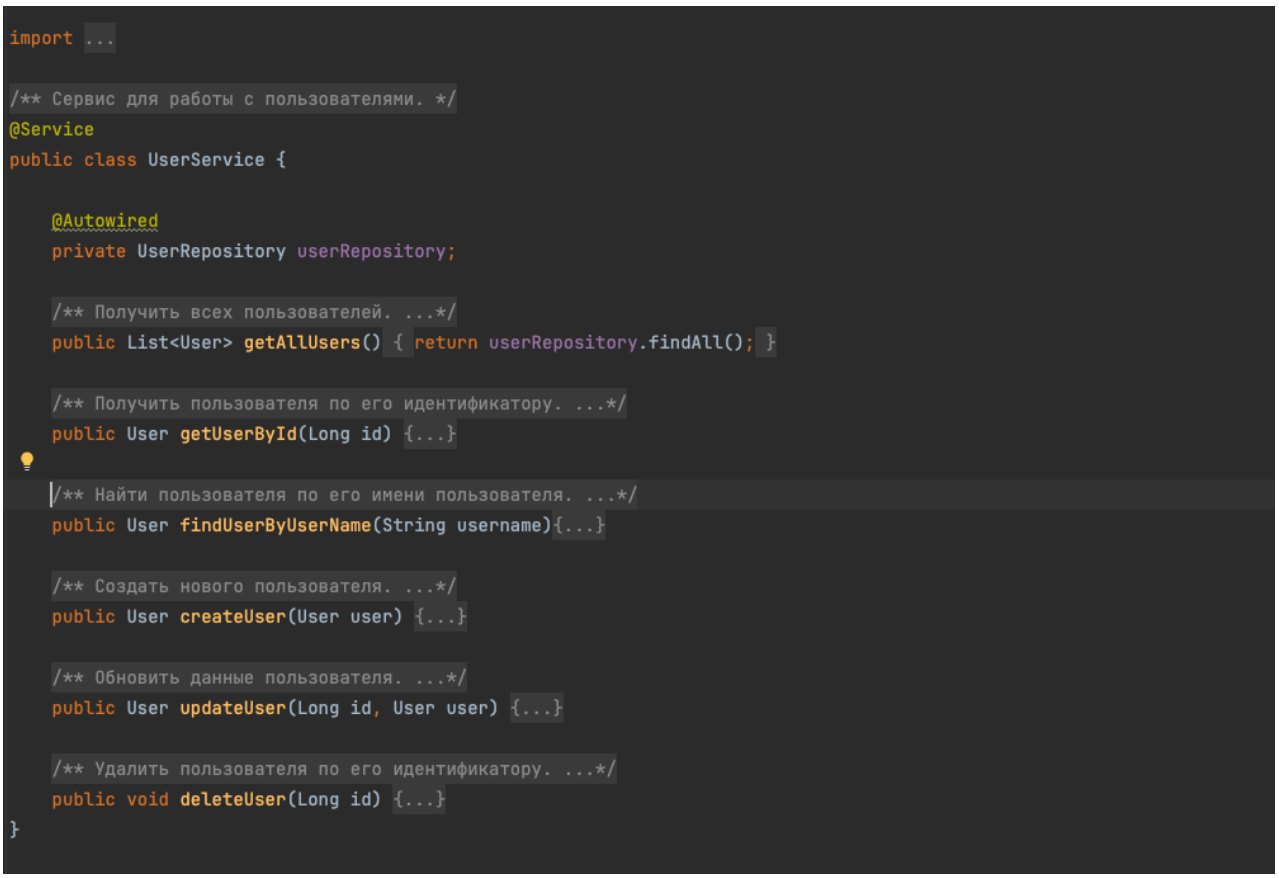


рисунок 17

Класс UserService представляет собой сервисный компонент, служит для управления бизнес-логикой, связанной с операциями CRUD (создание, чтение, обновление, удаление) пользователей, инкапсулирует доступ к данным и обеспечивает контроль целостности данных и безопасность операций.

Методы сервиса:

- **getAllUsers():** получает список всех пользователей, используя метод findAll() из UserRepository.
- **getUserById(Long id):** получает пользователя по его идентификатору. Если пользователь не найден, выбрасывает исключение UserNotFoundException.
- **findUserByUserName(String username):** находит пользователя по его имени. Если пользователь не найден, выбрасывает исключение UserNotFoundException.
- **createUser(User user):** создает нового пользователя. Если пользователь с таким именем уже существует, выбрасывает исключение UserAlreadyExistsException.

- **updateUser(Long id, User user):** обновляет данные пользователя по его идентификатору. Если пользователь с указанным идентификатором не найден, выбрасывает исключение UserNotFoundException.
- **deleteUser(Long id**): удаляет пользователя по его идентификатору. Если пользователь с указанным идентификатором не найден, выбрасывает исключение UserNotFoundException.

2.6 Обработка исключений в сервисах, управляющих данными приложения.

Обработка исключений - важный аспект разработки программного обеспечения, который влияет на надежность, безопасность и удобство использования приложения.

Позволяет добиться:

- **Повышения надежности приложения:** исключения могут возникать из-за различных причин, таких как ошибки ввода-вывода, проблемы с сетью, неверные операции и другие. Обработка исключений позволяет приложению адекватно реагировать на такие ситуации, предотвращая сбои и непредвиденное завершение работы приложения.
- **Улучшения пользовательского опыта:** если приложение корректно обрабатывает исключения, пользователи могут получать информативные сообщения об ошибках, что делает их взаимодействие с приложением более понятным и удобным.
- **Обеспечения безопасности:** обработка исключений также важна для обеспечения безопасности приложения. Ошибки безопасности, такие как доступ к защищенным ресурсам или утечки данных, могут быть обнаружены и обработаны с помощью механизмов обработки исключений.
- **Легкости отладки и поддержки**: обработка исключений делает код более структурированным и управляемым. Когда исключения обрабатываются централизованно, проще отслеживать и исправлять ошибки, а также обеспечивать соответствие стандартам и требованиям проекта.
- **Улучшения производительности:** обработка исключений также может улучшить производительность приложения. Если исключения правильно обрабатываются и не приводят к лишнему использованию ресурсов, это может помочь избежать излишних задержек и нагрузки на систему.

Обработка исключений в микросервисах имеет особое значение из-за архитектурных особенностей и распределенной природы таких приложений. Играет ключевую роль в обеспечении надежности, безопасности и производительности этих систем. Важно разработать соответствующие стратегии обработки исключений, учитывая специфику микросервисной архитектуры и бизнес-требований.

Важными аспектами обработки исключений в микросервисах являются:

- **Распределенная обработка исключений:** в микросервисной архитектуре каждый сервис работает независимо друг от друга, и должен быть способен обрабатывать свои собственные ошибки. При этом важно иметь возможность адекватно реагировать на ошибки каждого отдельно сервиса.

- **Информативные сообщения об ошибках:** поскольку микросервисы взаимодействуют друг с другом через сеть, важно иметь информативные сообщения об ошибках, которые позволяют понять причину сбоя и принять соответствующие меры. Это помогает улучшить пользовательский опыт и упрощает отладку и устранение проблем.

- **Управление транзакциями и согласованностью:** обработка исключений также важна для управления транзакциями и обеспечения согласованности данных в распределенной системе. Например, при возникновении ошибки в одном сервисе может потребоваться выполнить откат транзакции или синхронизировать данные в других сервисах.

- **Отказоустойчивость и восстановление после сбоев:** в микросервисных системах важно иметь механизмы обработки исключений, которые обеспечивают отказоустойчивость и восстановление после сбоев. Это может включать автоматическое перезапускание сервисов, механизмы обнаружения и восстановления сбойных узлов и т. д.

- **Безопасность и защита от уязвимостей:** обработка исключений также играет важную роль в обеспечении безопасности микросервисов. Например, обработка исключений в микросервисе аутентификации может помочь предотвратить атаки перебора паролей или подделки сеанса.

Понимая все это мной было принято решение для организации централизованной обработки исключений на уровне каждого отдельного сервиса.

В Spring Framework позволяет создавать объекты, которые будут единым местом обработки исключений, которые будут в себе содержать глобальные обработчики иксключений. Для того, чтобы фреймворку дать понять, что какой-то класс будет выполнять такую роль, его нужно аннотировать как @ControllerAdvice.

2.6.1 Обработка исключений в микросервисе пользователей

Примером реализации описанного выше подхода в сервисе, управляющего пользователями, является класс UserMicroserviceExceptionControllerAdvice, представленный на рисунке 18.

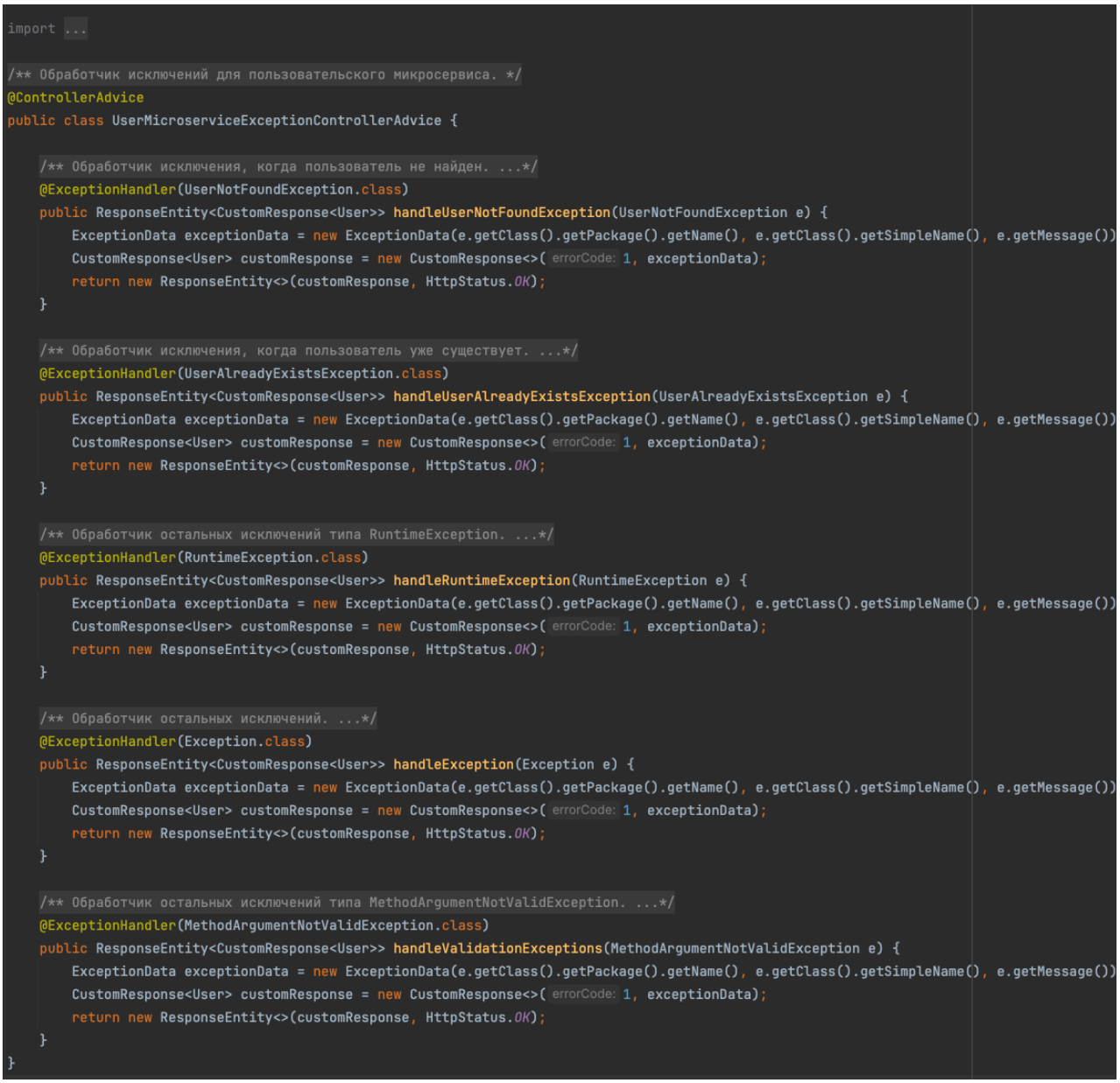


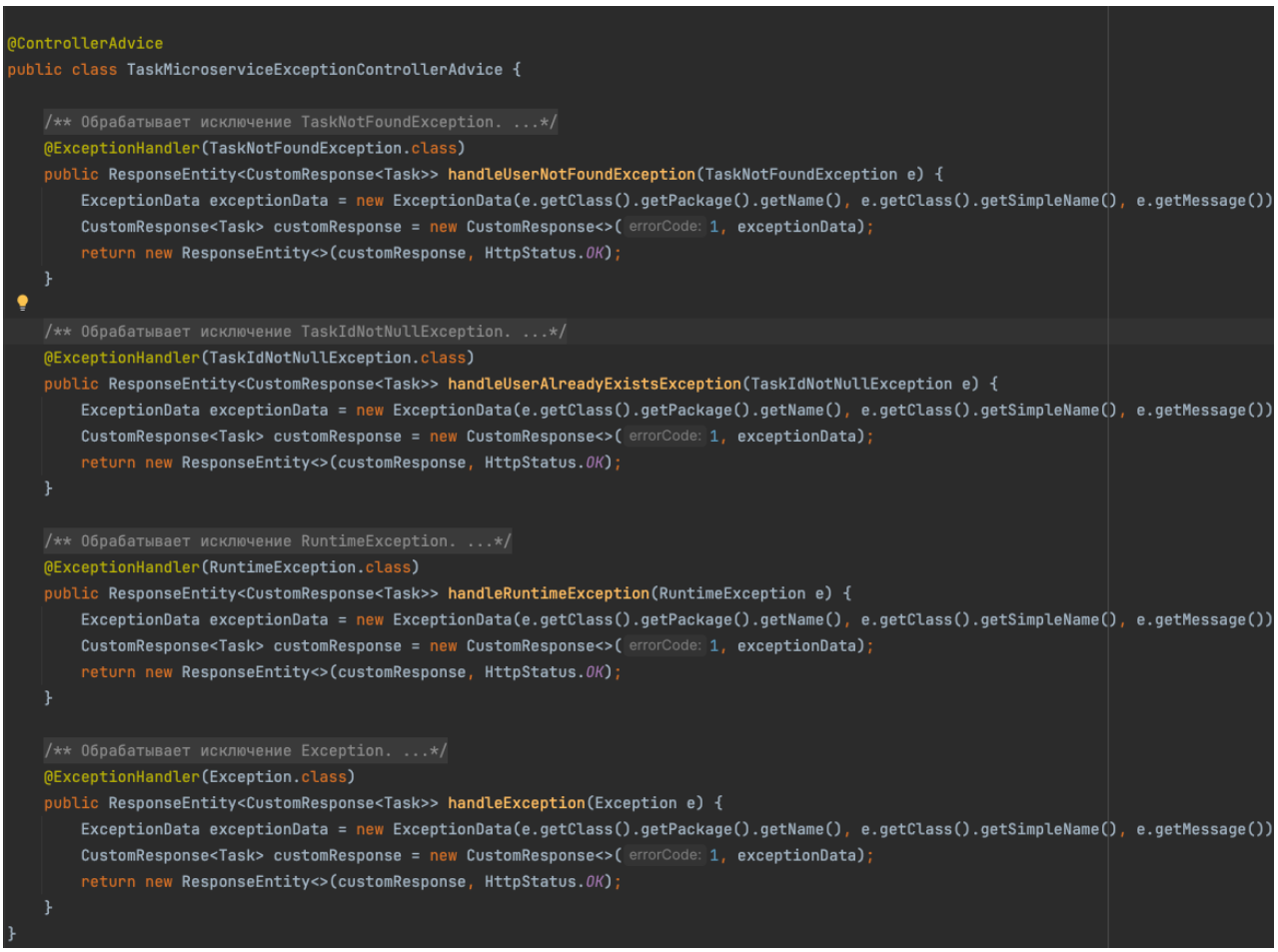
рисунок 18

В каждом методе обработки исключения создается объект ExceptionData, который содержит информацию об исключении (пакет, имя класса и сообщение). Затем создается объект CustomResponse, который содержит этот ExceptionData, и возвращается с помощью объекта ResponseEntity. Такой подход позволяет отправлять информативные ответы об ошибках клиентам.

2.6.2 Обработка исключений в микросервисе задач

Такой же подход был мной использован и в сервисе, отвечающем за работу с задачами. В нем реализован класс TaskMicroserviceExceptionControllerAdvice, который представлен на рисунке 19. Каждый его метод создает объект ExceptionData, содержащий информацию об исключении, и создает объект

CustomResponse, содержащий этот ExceptionData. Подход аналогичен предыдущему.



```
@ControllerAdvice
public class TaskMicroserviceExceptionControllerAdvice {

    /** Обрабатывает исключение TaskNotFoundException. ...*/
    @ExceptionHandler(TaskNotFoundException.class)
    public ResponseEntity<CustomResponse<Task>> handleUserNotFoundException(TaskNotFoundException e) {
        ExceptionData exceptionData = new ExceptionData(e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        CustomResponse<Task> customResponse = new CustomResponse<>( errorCode: 1, exceptionData);
        return new ResponseEntity<>(customResponse, HttpStatus.OK);
    }

    /** Обрабатывает исключение TaskIdNotNullException. ...*/
    @ExceptionHandler(TaskIdNotNullException.class)
    public ResponseEntity<CustomResponse<Task>> handleUserAlreadyExistsException(TaskIdNotNullException e) {
        ExceptionData exceptionData = new ExceptionData(e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        CustomResponse<Task> customResponse = new CustomResponse<>( errorCode: 1, exceptionData);
        return new ResponseEntity<>(customResponse, HttpStatus.OK);
    }

    /** Обрабатывает исключение RuntimeException. ...*/
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<CustomResponse<Task>> handleRuntimeException(RuntimeException e) {
        ExceptionData exceptionData = new ExceptionData(e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        CustomResponse<Task> customResponse = new CustomResponse<>( errorCode: 1, exceptionData);
        return new ResponseEntity<>(customResponse, HttpStatus.OK);
    }

    /** Обрабатывает исключение Exception. ...*/
    @ExceptionHandler(Exception.class)
    public ResponseEntity<CustomResponse<Task>> handleException(Exception e) {
        ExceptionData exceptionData = new ExceptionData(e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        CustomResponse<Task> customResponse = new CustomResponse<>( errorCode: 1, exceptionData);
        return new ResponseEntity<>(customResponse, HttpStatus.OK);
    }
}
```

рисунок 19

2.7 Разработка микросервиса, реализующего клиентскую часть

Данная часть работы была самой интересной, потому как она должна объединять в себе работу со всеми источниками данных в приложении. Именно она диктовала большую часть требований при взаимодействии со всеми микросервисами приложения. С результатами этого этапа разработки будут работать пользователи.

Первое, к чему я приступил на данном этапе – к настройке сервиса, который будет являться единой точкой для обращений к эндпоинтам любых других микросервисов. Его роль была возложена на Spring API Gateway. Вся работа по маршрутизации запросов отведена ему. Пример его конфигурации для локальной разработки приведен на рисунке 20.

На нем хорошо видны адресы, по которым доступны сервисы в сети и путь к их основным эндпоинтам.

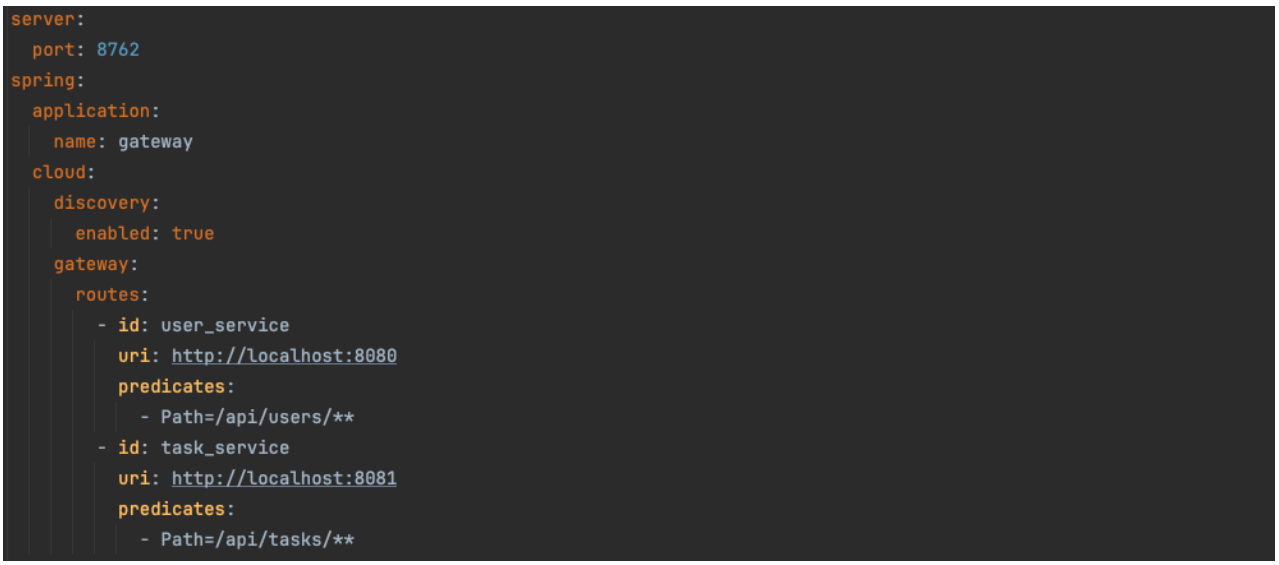


рисунок 20

2.7.1 Клиенты для запросов к api

На этом этапе моей задачей было написать классы, которые бы обращались к Spring API Gateway за доступом к данным определенных сервисов. И после получения ответа правильно десериализовали бы их обратно в java-объекты, с тем чтобы с ними можно было как-то работать.

Для этих целей мне требовался RestClient, который бы решал мои задачи. Поэтому я воспользовался существующей реализацией в Spring Framework. Для этого мне потребовалось определить соответствующие бины в конфигурации приложения клиентского сервиса (рисунок 21).

Я создал класс TaskClientConfiguration, аннотировал его как @Configuration, добавил в него приватное поле, которое должно содержать в себе ссылку базовый URL . К этому полю я добавил аннотацию @Value ("${apigateway.base-url}"), которая позволила внедрить значение свойства из application.yml.

Как можно видеть на рисунке, в классе присутствуют два метода:

- RestClient restClient: определяет бин RestClient, которыйвозвращает сконфигурированный экземпляр RestClient для взаимодействия с удаленным API или микросервисом. Этот бин создается с использованием RestClient.Builder.
- RestClientCustomizer restClientCustomizer: определяет бин для настройки RestClient. В этом методе создается экземпляр RestClientCustomizer, который является функциональным интерфейсом и имеет единственный метод, который возвращает нам RestClient.Builder. Внутри этой функции устанавливается базовый URL для всех запросов с помощью baseUrl(baseUrl).

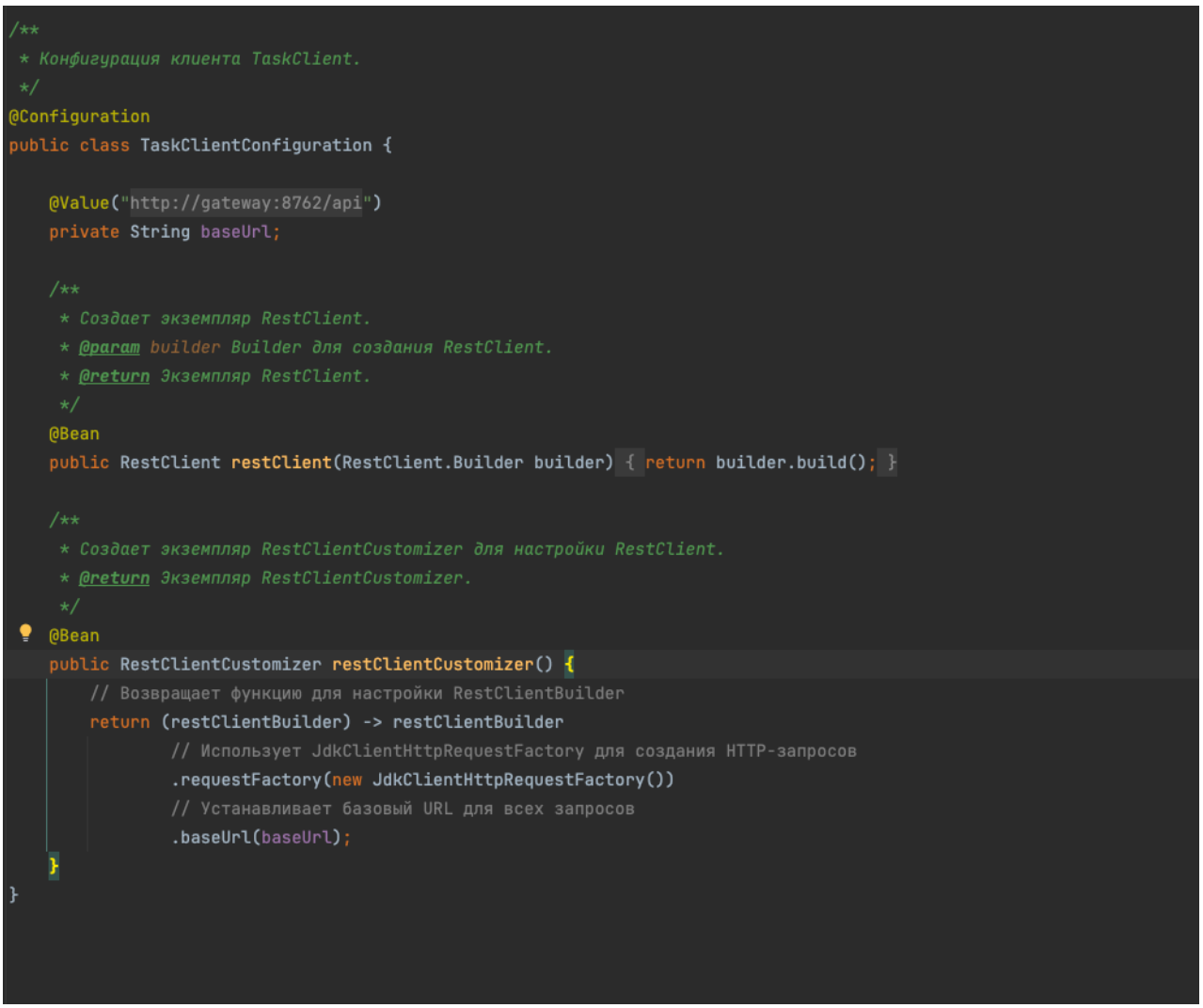


рисунок 21

Для создания запросов о пользователях у меня реализован класс UserClientApi, который аннотирован как @Component, и предоставляет удобный интерфейс для выполнения операций с пользователями через взаимодействие с удаленным API. В нем внедрены зависимости RestClient и ObjectMapper для запросов по HTTP и их десериализации, а методы этого класса выполняют операции взаимодействия с удаленным сервисом и обработки данных. Структура класса хорошо видна на рисунке 20.

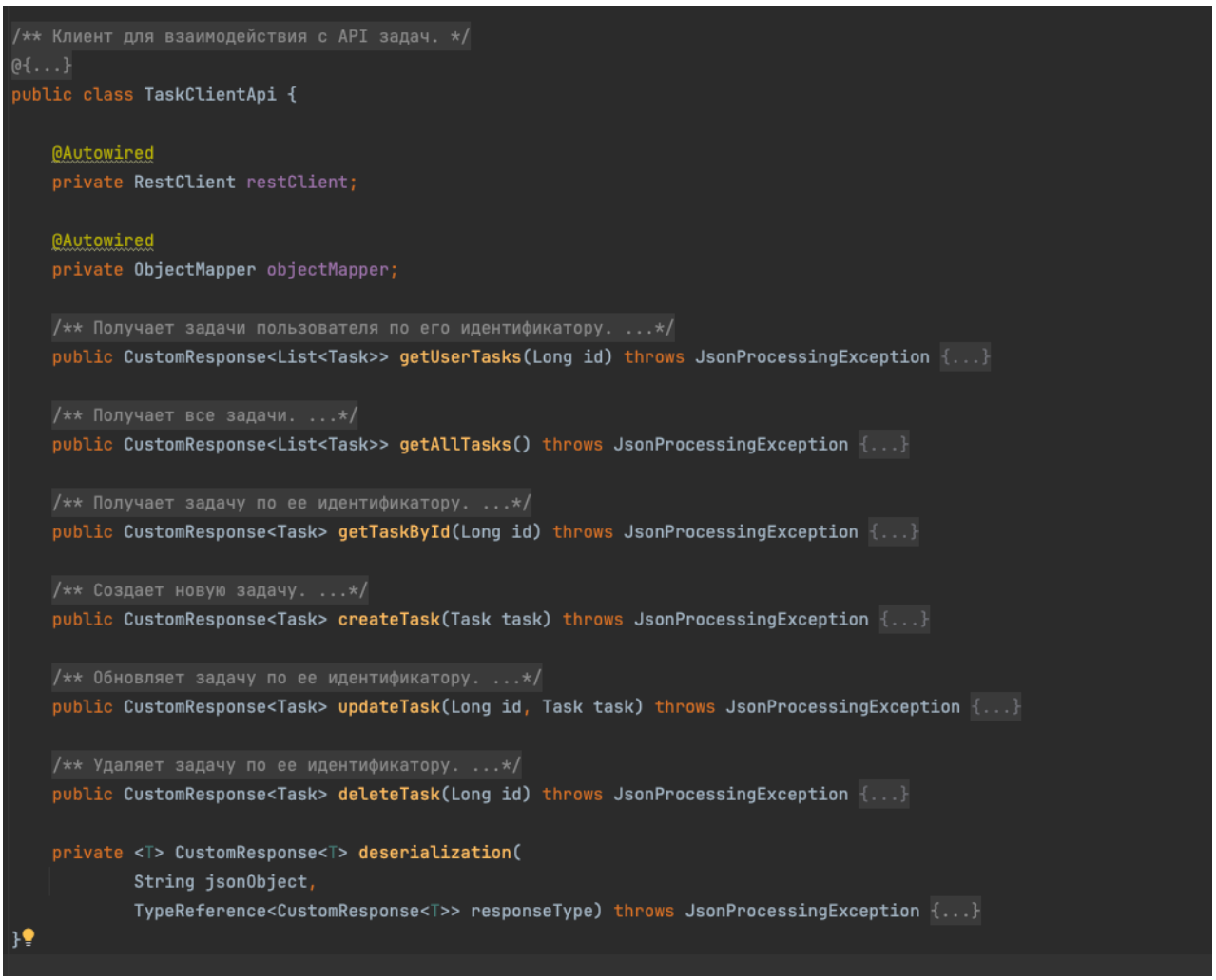
Моменты, которые стоит прояснить:

- **Методы класса:** осуществляют запрос к сервису для выполнения определенных операций, и получают от него строку в виде JSON, которая содержит определенный ответ с результатами этой операции.
- **Преобразования JSON:** метод deserialization используется для преобразования полученной строки в java-объект указанного типа.

```java
/** Клиент API для работы с пользователями. */
@Component
public class UserClientApi {

    @Autowired
    private RestClient restClient;

    @Autowired
    private ObjectMapper objectMapper;

    /** Получает список всех пользователей. ...*/
    public CustomResponse<List<User>> getUsers() throws JsonProcessingException {
        String json = restClient
                .get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "/users") capture of ?
                .retrieve() ResponseSpec
                .body(String.class);
        return deserialization(json, new TypeReference<CustomResponse<List<User>>>() {});
    }

    /** Получает пользователя по его идентификатору. ...*/
    public CustomResponse<User> getUserById(Long id) throws JsonProcessingException {...}

    /** Находит пользователя по его имени пользователя. ...*/
    public CustomResponse<User> findUserByUsername(String username) throws JsonProcessingException {...}

    /** Создает нового пользователя. ...*/
    public CustomResponse<User> createUser(User user) throws JsonProcessingException {
        String json = restClient
                .post() RequestBodyUriSpec
                .uri( uri: "/users") RequestBodySpec
                .body(user)
                .retrieve() ResponseSpec
                .body(String.class);
        return deserialization(json, new TypeReference<CustomResponse<User>>() {});
    }

    /** Обновляет данные пользователя. ...*/
    public CustomResponse<User> updateUser(Long id, User user) throws JsonProcessingException {...}

    /** Удаляет пользователя по его идентификатору. ...*/
    public CustomResponse<User> deleteUser(Long id) throws JsonProcessingException {...}

    /** Десериализует JSON-строку в объект типа CustomResponse. ...*/
    private <T> CustomResponse<T> deserialization(
            String jsonObject,
            TypeReference<CustomResponse<T>> responseType) throws JsonProcessingException {
        JsonNode jsonNode = objectMapper.readTree(jsonObject);
        return objectMapper.convertValue(jsonNode, responseType);
    }
}
```

рисунок 22

Класс TaskClientApi реализует тот же функционал, что и UserClientApi, только применительно к объектам задач пользователей. По этой причине не стану приводить весь код класса, а покажу только общую структуру (рисунок 23).



```java
/** Клиент для взаимодействия с API задач. */
@{...}
public class TaskClientApi {

    @Autowired
    private RestClient restClient;

    @Autowired
    private ObjectMapper objectMapper;

    /** Получает задачи пользователя по его идентификатору. ...*/
    public CustomResponse<List<Task>> getUserTasks(Long id) throws JsonProcessingException {...}

    /** Получает все задачи. ...*/
    public CustomResponse<List<Task>> getAllTasks() throws JsonProcessingException {...}

    /** Получает задачу по ее идентификатору. ...*/
    public CustomResponse<Task> getTaskById(Long id) throws JsonProcessingException {...}

    /** Создает новую задачу. ...*/
    public CustomResponse<Task> createTask(Task task) throws JsonProcessingException {...}

    /** Обновляет задачу по ее идентификатору. ...*/
    public CustomResponse<Task> updateTask(Long id, Task task) throws JsonProcessingException {...}

    /** Удаляет задачу по ее идентификатору. ...*/
    public CustomResponse<Task> deleteTask(Long id) throws JsonProcessingException {...}

    private <T> CustomResponse<T> deserialization(
            String jsonObject,
            TypeReference<CustomResponse<T>> responseType) throws JsonProcessingException {...}
}
```

рисунок 23

2.7.2 Сервисный слой

Классов, реализующих логику сервисного слоя в этом сервисе три. В данном пункте я расскажу о работе над двумя из них. Реализация же третьего останется для раздела, в котором я буду говорить об аутентификации и авторизации в приложении.

Класс UserService представляет собой сервис для работы с пользователями, реализующий некоторый уровень абстрации над клиентским API, с целью скрыть детали взаимодействия с HTTP и обработки JSON. Это позволяет сильно упростить код в остальной части приложения.

В класс инжектирована зависимость UserClientApi, его методы предназначены для выполнения различных операций с пользователями, таких как поиск по имени, регистрация нового пользователя, обновления данных о пользователе, удаление пользователя, получения пользователя по его id.

Структура класса UserService приведена на рисунке 24.

```java
/** Сервис для работы с пользователями. */
@Service
public class UserService {

    @Autowired
    private UserClientApi userClientApi;

    /** Находит пользователя по его имени пользователя. ...*/
    public CustomResponse<User> findUserByUserName(String username) throws JsonProcessingException {
        return userClientApi.findUserByUsername(username);
    }

    /** Регистрирует нового пользователя. ...*/
    public CustomResponse<User> registerNewUser(User user) throws JsonProcessingException {
        return userClientApi.createUser(user);
    }

    /** Удаляет аккаунт пользователя. ...*/
    public void deleteAccount(Long id) throws JsonProcessingException {
        userClientApi.deleteUser(id);
    }

    /** Обновляет данные аккаунта пользователя. ...*/
    public CustomResponse<User> updateAccount(Long id, User user) throws JsonProcessingException {
        return userClientApi.updateUser(id, user);
    }

    /** Получает список всех пользователей. ...*/
    public CustomResponse<List<User>> findAllUsers() throws JsonProcessingException {
        return userClientApi.getUsers();
    }
}
```

рисунок 24

Класс TaskService представляет собой сервис для работы с задачами. В нем так же есть инжектированная зависимость, только в этом случае это TaskClientApi, которая позволяет использовать методы TaskClientApi для взаимодействия с удаленным API задач.

Класс предоставляет методы для выполнения различных операций с задачами, таких как создание новой задачи, удаление задачи, обновление задачи, поиск задачи по идентификатору, поиск всех задач пользователя по его идентификатору и получение всех задач. Эти методы делегируют выполнение соответствующих операций методам TaskClientApi.

TaskService так же предоставляет уровень абстракции над клиентским API, скрывая детали взаимодействия с HTTP и обработки JSON-ответов, что упрощает использование этих операций.

Структура класса TaskService представлена на рисунке 25:

```java
/** Сервис для работы с задачами. */
@Service
public class TaskService {
    @Autowired
    private TaskClientApi taskClientApi;

    /** Создает новую задачу. ...*/
    public CustomResponse<Task> createNewTask(Task task) throws JsonProcessingException {
        return taskClientApi.createTask(task);
    }

    /** Удаляет задачу по ее идентификатору. ...*/
    public CustomResponse<Task>  deleteTask(Long id) throws JsonProcessingException {
        return taskClientApi.deleteTask(id);
    }

    /** Обновляет задачу по ее идентификатору. ...*/
    public CustomResponse<Task> updateTask(Long id, Task task) throws JsonProcessingException {
        return taskClientApi.updateTask(id, task);
    }

    /** Находит задачу по ее идентификатору. ...*/
    public CustomResponse<Task> findTaskById(Long id) throws JsonProcessingException {
        return taskClientApi.getTaskById(id);
    }

    /** Находит все задачи пользователя по его идентификатору. ...*/
    public CustomResponse<List<Task>> findAllUserTasks(Long userId) throws JsonProcessingException {
        return taskClientApi.getUserTasks(userId);
    }

    /** Получает все задачи. ...*/
    public CustomResponse<List<Task>> findAllTask() throws JsonProcessingException {
        return taskClientApi.getAllTasks();
    }

}
```

рисунок 25

2.7.3 Контроллеры

Класс RegistrationController, представленный на рисунке 26, является контроллером и обрабатывает процесс регистрации пользователей. В нем присутствует зависимости UserService и PasswordEncoder.

- Метод showRegistrationForm обрабатывает GET-запросы на URL /register. Он добавляет новый объект User в модель представления и возвращает имя представления "registration".
- Метод registerUser обрабатывает POST-запросы на URL /register. Он принимает данные из формы регистрации в виде объекта User, аннотированного @Valid, чтобы выполнить валидацию полей. Затем он хэширует пароль с помощью PasswordEncoder, регистрирует нового пользователя с помощью UserService и выполняет перенаправление на страницу входа.

```java
@Controller
public class RegistrationController {

    @Autowired
    private UserService userService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    /**
     * Отображает форму регистрации пользователя.
     * @param model Модель представления.
     * @return Шаблон страницы регистрации.
     */
    @GetMapping("/register")
    public String showRegistrationForm(Model model) {
        model.addAttribute( attributeName: "userForm", new User());
        return "registration";
    }

    /**
     * Обрабатывает данные, отправленные из формы регистрации.
     * @param userForm Данные нового пользователя.
     * @return Перенаправление на страницу входа после успешной регистрации.
     * @throws JsonProcessingException Если возникает ошибка при обработке JSON.
     */
    @PostMapping("/register")
    public String registerUser(@Valid User userForm) throws JsonProcessingException {
        // Хэшируем пароль перед сохранением
        userForm.setPassword(passwordEncoder.encode(userForm.getPassword()));
        userService.registerNewUser(userForm);
        return "redirect:/login";
    }
}
```

рисунок 26

На этом этот несложный процесс регистрации нового пользователя заканчивается.

Класс LoginController представляет собой контроллер для страницы входа. Содержит в себе один единственный метод, который обрабатывает GET-запросы на URL /login. В ответ он возвращает представление login, которое является страницей входа.

Структура данного класса представлена на рисунке 27.

```java
/**
 * Контроллер для страницы входа.
 */
@Controller
public class LoginController {

    /**
     * Отображает страницу входа.
     * @return Шаблон страницы входа.
     */
    @GetMapping("/login")
    public String login() { return "login"; }
}
```

рисунок 27

На данном этапе разработки приложения вместо стандартных страниц заглушек, которые выводятся в случае возникновения ошибок различного рода, мной принято решения использовать шаблон, который позволит мне эти ошибки видеть сразу в момент их возникновения. Без необходимости сразу бежать в «логи». Это оказалось особенно полезно, когда проходилось отлаживать приложение развернутое в Docker. Такое решение позволяет сразу понять в каком сервисе возникла ошибка и какая именно. Реализация класса ExceptionDataController приведена на рисунке 29

```java
/**
 * Контроллер для обработки данных об исключениях.
 */
@Controller
public class ExceptionDataController {

    /**
     * Получает данные об исключении из сессии и отображает страницу с информацией об исключении.
     * @param model Модель представления.
     * @param session Сессия HTTP.
     * @return Шаблон страницы с данными об исключении.
     */
    @GetMapping("/exception-data")
    public String getExceptionData(Model model, HttpSession session) {
        // Получаем данные об исключении из сессии
        ExceptionData sessionData = (ExceptionData) session.getAttribute( s: "exception");
        // Передаем данные об исключении в модель представления
        model.addAttribute( attributeName: "exceptionData", sessionData);
        return "exception";
    }
}
```

рисунок 29

Метод getExceptionData Обрабатывает GET-запросы на URL /exception-data. Он принимает два параметра: Model для передачи данных в представление и HttpSession для доступа к данным сеанса. В этом методе данные об исключении

извлекаются из сеанса с помощью метода getAttribute, сохраненные ранее в сессии. Затем эти данные передаются в модель представления с именем "exceptionData". Наконец, метод возвращает представление "exception", для которого существует шаблон.

На рисунке 28 представлен TaskManagerController, он обеспечивает обработку запросов для просмотра, создания, редактирования и удаления задач, а также отображение информации о задачах текущего пользователя.

Методы обработки запросов:

- getAllTasks: получает все задачи текущего пользователя, добавляет их в модель и возвращает шаблон страницы с задачами.

- getTaskById: получает задачу по её идентификатору, добавляет её в модель и возвращает шаблон страницы с информацией о задаче.

- createTaskForm: отображает форму для создания новой задачи.

- createTask: создаёт новую задачу, используя данные из формы, и перенаправляет пользователя на страницу с задачами.

- editTaskForm: отображает форму для редактирования задачи с определённым идентификатором.

- updateTask: обновляет задачу с определённым идентификатором и перенаправляет пользователя на страницу с задачами.

- deleteTask: удаляет задачу с определённым идентификатором и перенаправляет пользователя на страницу с задачами.

- Метод handleResponse: обрабатывает ответ от сервера. Если в ответе содержится ошибка, данные об ошибке добавляются в сессию и происходит перенаправление на страницу с информацией об исключении. В противном случае происходит добавление данных задачи в модель и возвращается соответствующее представление.

- Метод getCurrentUser: получает текущего пользователя из аутентификации и возвращает его.


/** Контроллер для управления задачами пользователя. */
@Controller
public class TaskManagerController {
@Autowired
private TaskService taskService;
@Autowired
private UserService userService;
/** Получает все задачи пользователя. ...*/
@GetMapping("/tasks")
public String getAllTasks(Model model, Authentication authentication) throws JsonProcessingException {...}
/** Получает задачу по ее идентификатору. ...*/
@GetMapping("/tasks/{id}")
public String getTaskById(
@PathVariable Long id,
Model model, HttpSession session) throws JsonProcessingException {...}
/** Возвращает форму для создания новой задачи. ...*/
@GetMapping("/tasks/new")
public String createTaskForm(Model model) {...}
/** Создает новую задачу. ...*/
@PostMapping("/tasks")
public String createTask(
@ModelAttribute Task task,
HttpSession session,
Authentication authentication) throws JsonProcessingException {...}
/** Возвращает форму для редактирования задачи. ...*/
@GetMapping("/tasks/{id}/edit")
public String editTaskForm(
@PathVariable Long id,
Model model, HttpSession session) throws JsonProcessingException {...}
/** Обновляет задачу. ...*/
@PostMapping("/tasks/{id}/edit")
public String updateTask(
@PathVariable Long id,
@ModelAttribute Task task, HttpSession session) throws JsonProcessingException {...}
/** Удаляет задачу. ...*/
@PostMapping("/tasks/{id}/delete")
public String deleteTask(@PathVariable Long id, HttpSession session) throws JsonProcessingException {...}
/** Получает текущего пользователя. ...*/
private User getCurrentUser(Authentication authentication) throws JsonProcessingException {...}
/** Обрабатывает ответ от сервера. ...*/
private String handleResponse(
CustomResponse<?> customResponse, Model model, String viewName, HttpSession session) {...}
/** Обрабатывает ответ от сервера и возвращает перенаправление. ...*/
private String handleResponse(CustomResponse<?> customResponse, HttpSession session, String redirectUrl) {...}
}

рисунок 28

2.7.4 Обработка исключений

Класс TaskManagerExceptionControllerAdvice обеспечивает централизованную обработку исключений в данном микросервисе, позволяя избежать дублирования кода.

```java
/** Контроллер Advice для обработки исключений в приложении Task Manager. */
@ControllerAdvice
public class TaskManagerExceptionControllerAdvice {

    /** Обрабатывает исключение UsernameNotFoundException. ...*/
    @ExceptionHandler(UsernameNotFoundException.class)
    public String handleUsernameNotFoundException(UsernameNotFoundException e, HttpSession session) {
        session.setAttribute( s: "exception", new ExceptionData(
                e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        );
        return "redirect:/exception-data";
    }

    /** Обрабатывает исключение MethodArgumentNotValidException. ...*/
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public String handleValidationExceptions(MethodArgumentNotValidException e, HttpSession session) {
        session.setAttribute( s: "exception", new ExceptionData(
                e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        );
        return "redirect:/exception-data";
    }

    /** Обрабатывает исключение JsonProcessingException. ...*/
    @ExceptionHandler(JsonProcessingException.class)
    public String handleJsonProcessingException(JsonProcessingException e, HttpSession session) {
        session.setAttribute( s: "exception", new ExceptionData(
                e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        );
        return "redirect:/exception-data";
    }

    /** Обрабатывает исключение RuntimeException. ...*/
    @ExceptionHandler(RuntimeException.class)
    public String handleRuntimeException(RuntimeException e, HttpSession session) {
        session.setAttribute( s: "exception", new ExceptionData(
                e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        );
        return "redirect:/exception-data";
    }

    /** Обрабатывает исключение Exception. ...*/
    @ExceptionHandler(Exception.class)
    public String handleException(Exception e, HttpSession session) {
        session.setAttribute( s: "exception", new ExceptionData(
                e.getClass().getPackage().getName(), e.getClass().getSimpleName(), e.getMessage())
        );
        return "redirect:/exception-data";
    }

}
```

рисунок 29

В случае возникновения исключения информация о нем попадет в представление /exception-data. На данном этапе разработки приложения это сильно упрощает отладку. В будущем для любого требуемого исключения можно создать свой обработчик, который будет возвращать HTML-страницу с необходимым сообщением.

2.8 Аутентификация и авторизация

Аутентификация и авторизация пользователей в сервисе является важной составляющей приложения, в котором требуется ограничения пользователей в праве доступа к определенным его частя.

В Spring Boot приложениях для этих целей есть мощный инструмент – Spring Security. Он предоставляет гибкие и настраиваемые возможности для аутентификации на основе различных факторов, таких как базы данных пользователей, LDAP или OAuth2.

На данном этапе моей работы я решил, что для доступа к ресурсам моего приложения достаточно предварительной регистрации пользователя, и дальнейшей авторизации и аутентификации по логину и паролю.

Понимая, что для взаимодействия с системой пользователю будет доступен только сервис с web-клиентом, функцию регистрации, авторизации и аутентификации пользователей решил реализовывать тоже в нем.

2.8.1 Настройка Spring Security для обеспечения безопасности сервиса.

Используемые зависимости описаны в пункте 2.2.2. После подключения последних необходимо добавить аннотацию @EnableWebSecurity, а также создать конфигурацию, в которой будут произведены настройки безопасности. Структура класса SecurityConfig, представлена на рисунке 30.

Основные элементы этой конфигурации:

- **Метод filterChain:** создает и настраивает цепочку фильтров безопасности с помощью объекта HttpSecurity, определяет правила доступа к различным URL-адресам. Например, все запросы к /register и /exception-data разрешены для всех, в то время как для остальных запросов требуется аутентификация. Так же настраивает базовую аутентификацию и форму входа, а также обработку выхода из системы.

- **Метод authenticationManager:** создает AuthenticationManager, который используется для аутентификации пользователей. Внутри метода создается экземпляр DaoAuthenticationProvider, который использует реализацию UserDetailsService и PasswordEncoder для проверки учетных данных пользователя.

- **Метод passwordEncoder:** создает экземпляр PasswordEncoder, который используется для шифрования паролей пользователей. В данном случае используется BCryptPasswordEncoder.

```java
/** Конфигурация безопасности приложения. */
@Configuration
public class SecurityConfig {

    /**
     * Настройка цепочки фильтров безопасности.
     * @param http Объект HttpSecurity для настройки безопасности.
     * @return Цепочка фильтров безопасности.
     * @throws Exception Если происходит ошибка в процессе настройки.
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {...}

    /**
     * Создает AuthenticationManager для аутентификации пользователей.
     * @param userDetailsService Реализация интерфейса UserDetailsService.
     * @param passwordEncoder Экземпляр PasswordEncoder для шифрования паролей.
     * @return AuthenticationManager для аутентификации пользователей.
     */
    @Bean
    public AuthenticationManager authenticationManager(
            UserDetailsServiceImpl userDetailsService,
            PasswordEncoder passwordEncoder) {...}

    /**
     * Создает экземпляр PasswordEncoder для шифрования паролей.
     * @return PasswordEncoder для шифрования паролей.
     */
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }

}
```

рисунок 30

2.8.2 Реализация механизмов аутентификации и авторизации пользователей.

Процесс аутентификации и авторизации пользователей в сервисе использует классы SecurityConfig, UserPrincipal и UserDetailsServiceImpl, и может быть описан следующим образом:

Аутентификация:

Когда пользователь пытается войти в систему, он отправляет запрос на вход, предоставляя свое имя пользователя и пароль.

Запрос на аутентификацию обрабатывается SecurityConfig, который настраивает механизмы аутентификации, такие как форма входа и базовая аутентификация.

При попытке входа пользователь отправляет запрос на URL /login. При получении запроса на /login Spring Security вызывает метод loadUserByUsername в UserDetailsServiceImpl, который загружает информацию о пользователе из базы данных по его имени пользователя (рисунок 31).

```java
/**
 * Реализация интерфейса UserDetailsService для аутентификации пользователей.
 */
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserService userService;

    /**
     * Загружает пользователя по его имени пользователя для аутентификации.
     * @param username Имя пользователя.
     * @return UserDetails, представляющий пользователя для аутентификации.
     * @throws UsernameNotFoundException Если пользователь с указанным именем не найден.
     */
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        try {
            // Получаем пользователя по его имени пользователя
            CustomResponse<User> customResponse = userService.findUserByUserName(username);
            if(customResponse.getErrorCode() == 1 || customResponse.getResponseData() == null)
                    throw new UsernameNotFoundException("Пользователь с именем " + username + " не найден");
            // Возвращаем UserDetails, используя объект UserPrincipal
            return new UserPrincipal(customResponse.getResponseData());
        } catch (UsernameNotFoundException | JsonProcessingException e) {
            // Если пользователь не найден или возникает ошибка при обработке JSON, выбрасываем исключение
            throw new UsernameNotFoundException("Пользователь с именем " + username + " не найден");
        }
    }
}
```

рисунок 31

Если пользователь найден, его данные передаются в объект UserPrincipal (рисунок 32), который представляет пользователя в системе безопасности Spring. Этот объект возвращается как UserDetails.

Spring Security использует предоставленные учетные данные для проверки пароля пользователя. Если пароль соответствует хранимому паролю, пользователь аутентифицирован успешно.

Авторизация:

После успешной аутентификации пользователь получает доступ к защищенным ресурсам и функциональности системы.

Права доступа к различным частям приложения определяются с помощью аннотаций в контроллерах или настройкой доступа в SecurityConfig.

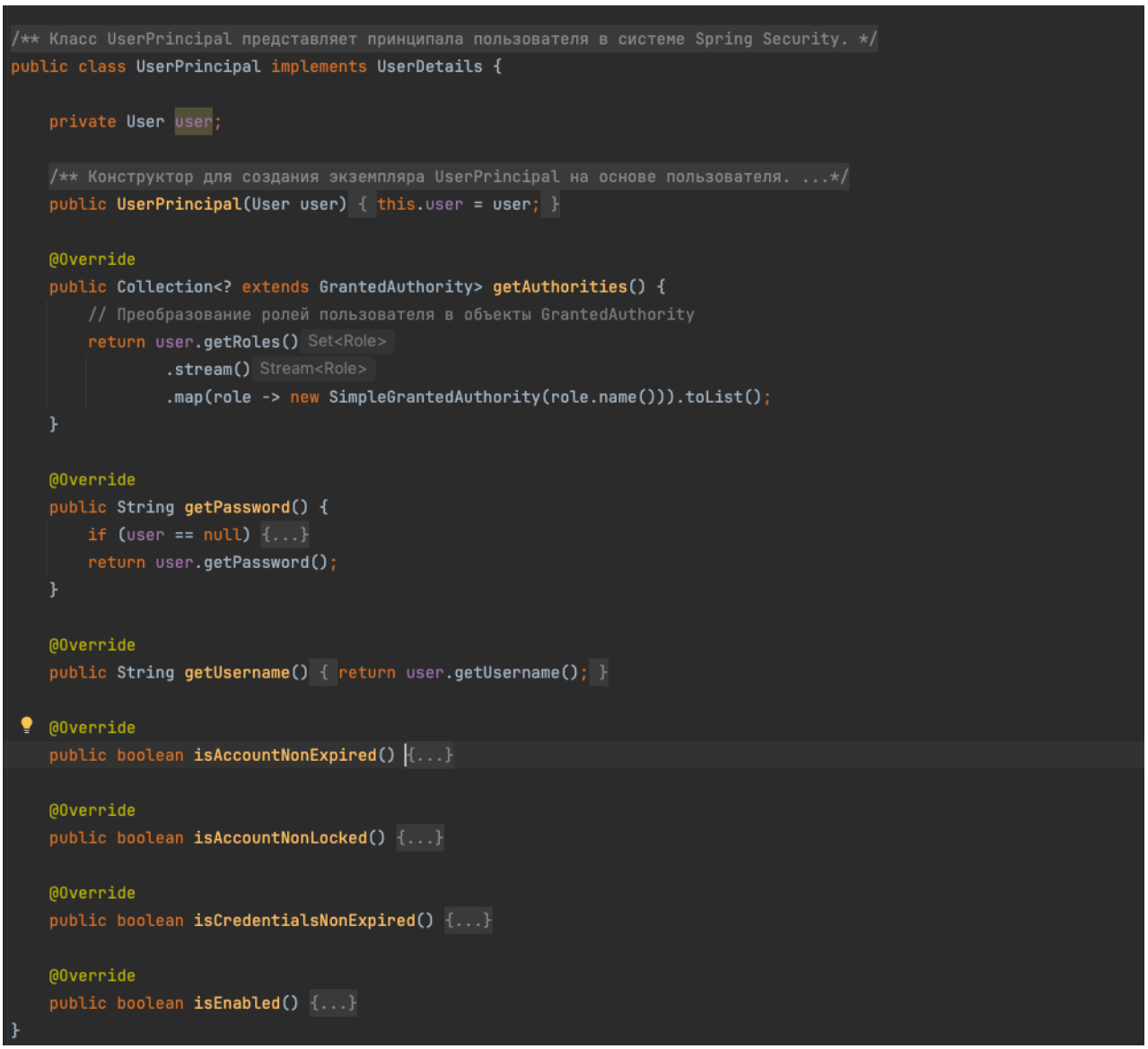


рисунок 32

Например, в SecurityConfig определены правила доступа к URL-адресам. В данном случае, доступ к странице регистрации (/register) и странице с данными об исключении (/exception-data) разрешен для всех пользователей, в то время как доступ к остальным URL-адресам требует аутентификации.

Если пользователь пытается получить доступ к защищенному ресурсу без аутентификации или с недостаточными правами, Spring Security перенаправляет его на страницу входа (/login) или страницу с сообщением об ошибке.

Таким образом, благодаря классам SecurityConfig, UserPrincipal и UserDetailsServiceImpl, Spring Security обеспечивает безопасную аутентификацию и авторизацию пользователей в сервисе.

2.9 Тестирование

Тестирование является важной частью разработки приложения. Реализация тестов позволяет:

- **Обеспечить качества кода:** Тесты позволяют проверить корректность реализации функциональности приложения и убедиться, что код выполняет заданные требования.
- **Выявить ошибки:** Тестирование позволяет выявить ошибки в коде на ранних стадиях разработки, что позволяет их исправить до того, как они приведут к серьезным проблемам в продукте.
- **Поддерживать надёжности приложения:** Запуск тестов после каждого изменения кода помогает предотвратить появление дефектов и обеспечивает стабильную работу приложения.
- **Упрощать рефакторинг:** Наличие хорошего набора тестов позволяет проводить рефакторинг кода с уверенностью, что изменения не нарушат работу приложения.

2.9.1 Создание юнит-тестов для проверки отдельных компонентов сервиса.

В сервисе, управляющем пользователями у меня описаны следующие классы тестов:

- **UserServiceTest:** класс содержит модульные тесты для сервиса пользователей (UserService). Внутри каждого теста создаются моки (поддельные объекты), которые имитируют поведение зависимостей, таких как UserRepository. Это позволяет изолировать тестируемый класс от его зависимостей и проверить его поведение независимо. В каждом тесте вызываются методы сервиса и проверяются ожидаемые результаты с помощью утверждений JUnit.
- **UserControllerTest:** класс содержит модульные тесты для контроллера пользователей (UserController). Как и в UserServiceTest, здесь также создаются моки для сервиса пользователей. Однако, в этом случае тестируется взаимодействие контроллера с клиентом (в данном случае, предполагается, что это HTTP клиент). Тесты проверяют, что контроллер правильно обрабатывает запросы от клиента и возвращает ожидаемые ответы.
- **UserMicroserviceExceptionControllerAdviceTest:** класс содержит модульные тесты для UserMicroserviceExceptionControllerAdvice, который является обработчиком исключений в микросервисе пользователей. Тесты здесь проверяют, что обработчик исключений корректно обрабатывает различные типы исключений (например, UserNotFoundException, UserAlreadyExistsException, RuntimeException, Exception) и возвращает

соответствующие ответы клиенту. Такие тесты важны для убеждения в том, что обработчик исключений возвращает правильные данные и корректные HTTP статусы в случае возникновения ошибок.

Для сервиса, управляющего задачами, реализованы следующие классы тестов:

- **TaskMicroserviceExceptionControllerAdviceTest:** класс тестирует обработчики исключений (ControllerAdvice) приложения. Он убеждается, что обработчики исключений корректно реагируют на различные виды исключений, которые могут возникнуть при выполнении запросов к API. Тесты проверяют, что обработчики формируют правильные ответы с соответствующими кодами состояния HTTP и сообщениями об ошибке.
- **TaskControllerTest:** класс тестирует контроллер задач приложения. Он проверяет, что методы контроллера правильно обрабатывают запросы и возвращают ожидаемые результаты. Каждый тест проверяет разные методы контроллера и убеждается, что они корректно работают в различных сценариях использования.
- **TaskServiceTest:** класс тестирует сервис задач приложения. Он проверяет, что методы сервиса корректно выполняют операции с данными, возвращают правильные результаты и обрабатывают возможные исключения. Каждый тест фокусируется на конкретном методе сервиса и проверяет его функциональность в различных сценариях.

Для сервиса с клиентской частью приложения:

- **UserServiceTest:** класс содержит юнит-тесты для проверки функциональности класса UserService, который предоставляет методы для управления пользователями в приложении. В тестах проверяются различные сценарии, такие как успешный поиск пользователя по имени, регистрация нового пользователя, обновление и удаление учетной записи пользователя, а также успешный поиск всех пользователей.
- **TaskServiceTest:** класс содержит юнит-тесты для проверки функциональности класса TaskService, который предоставляет методы для управления задачами в приложении. В тестах проверяются различные сценарии, такие как создание, обновление, удаление и поиск задач, как для отдельного пользователя, так и для всех задач в системе.
- **UserDetailsServiceImplTest:** класс содержит юнит-тесты для проверки функциональности класса UserDetailsServiceImpl, который реализует интерфейс UserDetailsService из Spring Security. В тестах проверяется, что сервис правильно загружает данные пользователя для аутентификации и авторизации, в том числе в ситуациях, когда пользователь существует и когда его нет.

- **TaskManagerControllerTest:** класс содержит юнит-тесты для проверки функциональности контроллера TaskManagerController, который отвечает за управление задачами в приложении. В тестах проверяются различные методы контроллера в различных сценариях использования, такие как получение всех задач, получение задачи по идентификатору, создание, обновление и удаление задачи.
- **RegistrationControllerTest:** класс содержит юнит-тесты для проверки функциональности контроллера RegistrationController, который, вероятно, отвечает за регистрацию новых пользователей в приложении. В тестах проверяются методы отображения формы регистрации и регистрации пользователя.

2.10 Логирование

В моей работе так же реализовано логирование работы некоторых компонентов. Сделано это в первую очередь с целью упрощения поиска решения проблем в случае выявления последних.

Как и в случае централизованной обработки исключения, идея логировать работу каждого отдельного сервиса с помощью отдельного компонента, показалась мне наиболее правильной.

Поэтому мной было принято решение для этих целей использовать возможности Spring AOP.

К примеру, в сервисе, управляющем пользователями, логирование контроллера осуществляется с помощью класса, структура которого приведена на рисунке 33.

Приведенный код представляет собой аспект аспектно-ориентированного программирования (AOP) в Spring Framework для логирования вызовов контроллера.

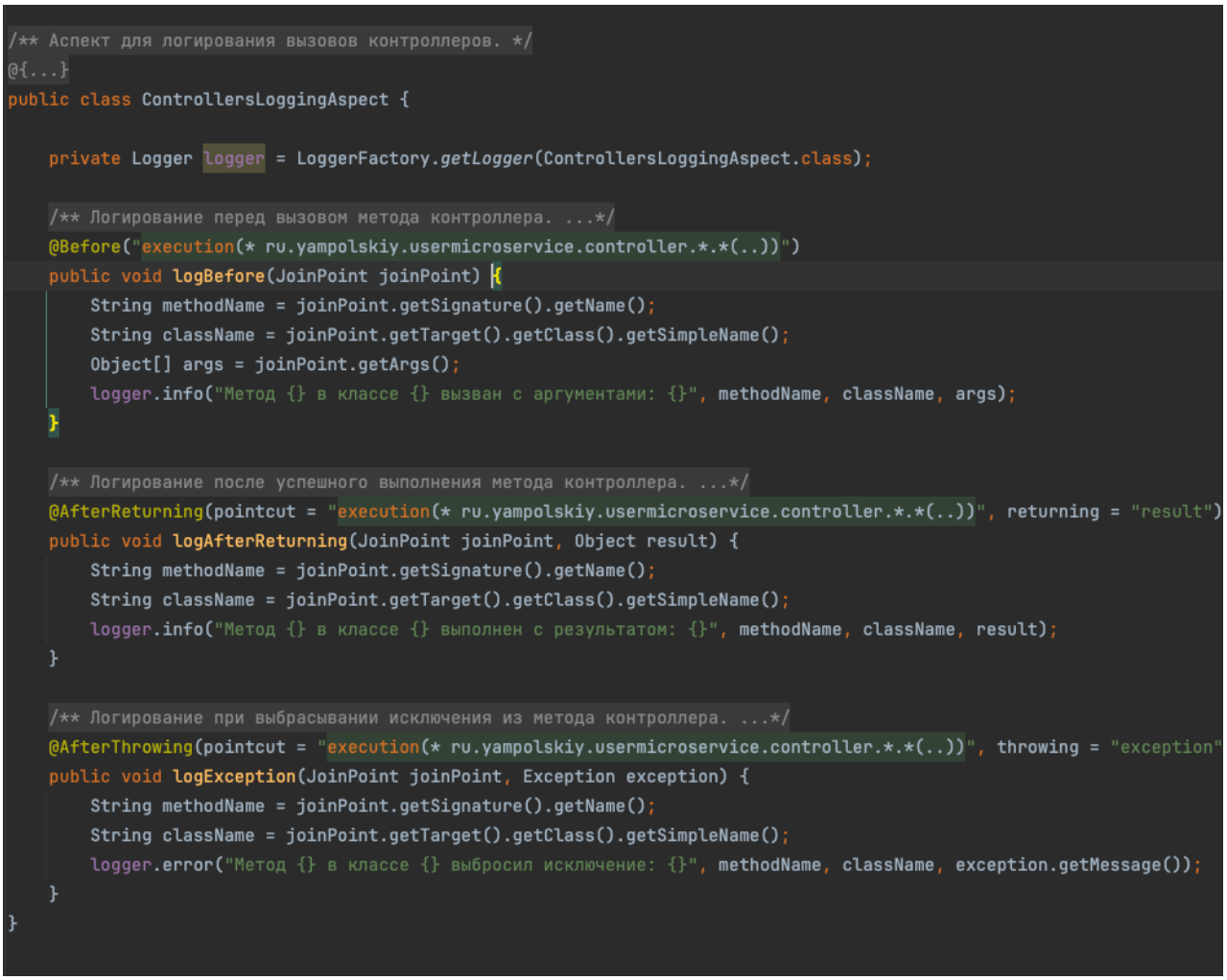
Точки среза (Pointcuts) определены аннотациями @Before, @AfterReturning, и @AfterThrowing. Они указывают на места в коде, в которых должен быть применен соответствующий совет.

Советы (Advices):

- logBefore(JoinPoint joinPoint): выполняется перед вызовом метода контроллера. Он использует JoinPoint для получения информации о методе, вызывающем точку соединения, и записывает информацию о вызове в лог.
- logAfterReturning(JoinPoint joinPoint, Object result): выполняется после успешного выполнения метода контроллера и возвращает результат. Он также использует JoinPoint для получения информации о методе и результате выполнения, и записывает информацию в лог.

- logException(JoinPoint joinPoint, Exception exception): выполняется при выбрасывании исключения из метода контроллера. Он также использует JoinPoint для получения информации о методе и выброшенном исключении, и записывает информацию об исключении в лог с уровнем error.

Логгеры: Используется LoggerFactory.getLogger() для создания экземпляра логгера, который будет записывать информацию в лог. Логгер инициализируется с классом ControllersLoggingAspect, который отвечает за логирование.



```
/** Аспект для логирования вызовов контроллеров. */
@{...}
public class ControllersLoggingAspect {

    private Logger logger = LoggerFactory.getLogger(ControllersLoggingAspect.class);

    /** Логирование перед вызовом метода контроллера. ...*/
    @Before("execution(* ru.yampolskiy.usermicroservice.controller.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();
        Object[] args = joinPoint.getArgs();
        logger.info("Метод {} в классе {} вызван с аргументами: {}", methodName, className, args);
    }

    /** Логирование после успешного выполнения метода контроллера. ...*/
    @AfterReturning(pointcut = "execution(* ru.yampolskiy.usermicroservice.controller.*.*(..))", returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();
        logger.info("Метод {} в классе {} выполнен с результатом: {}", methodName, className, result);
    }

    /** Логирование при выбрасывании исключения из метода контроллера. ...*/
    @AfterThrowing(pointcut = "execution(* ru.yampolskiy.usermicroservice.controller.*.*(..))", throwing = "exception"
    public void logException(JoinPoint joinPoint, Exception exception) {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();
        logger.error("Метод {} в классе {} выбросил исключение: {}", methodName, className, exception.getMessage());
    }
}
```

рисунок 33

Аналогичный подход был применен и в других сервисах приложения, что позволило мне решить такие проблемы как:

- Отслеживание действий и событий: запись событий, таких как ошибки, помогло понять, что происходит во время сбоя.
- Диагностика и устранение проблем: логи были использованы для выявления и анализа проблем, возникающих в приложении.

2.11 Сборка и развертывание

Сборку и его развертывание я решил автоматизировать с помощью Docker. Для этого мне потребовалось для каждого моего сервиса написать Dockerfile, который разделен на два этапа.

Этап сборки (build):

- Используется образ Maven maven:3.9.6-amazoncorretto-21 в качестве базового образа для сборки приложения.
- Устанавливается рабочая директория /app.
- Копируется файл pom.xml в текущую директорию.
- Выполняется команда mvn dependency:go-offline для загрузки зависимостей.
- Копируются исходные файлы из каталога src в текущую директорию.
- Выполняется сборка приложения с помощью команды mvn clean package, чтобы создать JAR-файл.

Этап создания образа:

- Используется образ OpenJDK openjdk:23-oracle в качестве базового образа для запуска приложения.
- Устанавливается рабочая директория /app.
- Из предыдущего этапа сборки копируется собранный JAR-файл app.jar в текущую директорию.
- Задается аргумент SPRING_PROFILE, который определяет профиль Spring (по умолчанию - prod).
- Команда CMD определяет способ запуска приложения при старте контейнера, используя команду java -jar app.jar.

Такой подход позволяет уменьшить размер итогового образа за счет отделения среды сборки Maven от среды выполнения Java.

Следующим шагом потребовалось написать docker-compose.yml, для того, чтобы по инструкциям из Dockerfile для каждого сервиса собрать образ, а потом из них запустить контейнеры с определенными сервисами.

Кратко перечислю сервисы, которые описаны в docker-compose.yml.

- **UsersDB и TasksDB:** контейнеры базы данных MySQL для пользователей и задач соответственно. Они используют образ mysql:latest, монтируют тома для сохранения данных и загружают переменные среды из файлов ums.env и tms.env.

- **Eureka:** Это служба реестра Eureka, построенная из Docker-образа, собранного по инструкции Dockerfile, находящегося в каталоге ./EurekaServer.
- **ApiGateWay:** данный контейнер представляет собой API-шлюз, построенный из Docker-образа, собранного по инструкции Dockerfile, находящегося в каталоге ./ApiGateWay. Он зависит от службы Eureka.
- **UserService (ums) и TaskService (tms):** службы представляют микросервисы пользователей и задач соответственно. Они построены из Docker-образов, собранных по инструкции Dockerfile, находящихся в каталогах ./UserMicroService и ./TaskMicroService соответственно. Зависят от служб Eureka и API-шлюза.
- **TaskClient:** Это клиентское приложение задач, построенное из Docker-образа, собранного по инструкции Dockerfile, находящегося в каталоге ./TaskClient. Оно также зависит от служб Eureka, API-шлюза, микросервисов пользователей и задач.
- **Prometheus:** Это служба мониторинга Prometheus, она использует образ prom/prometheus и настраивается с помощью файла конфигурации prometheus.yml. Она также зависит от службы TaskClient.
- **Grafana:** Это служба визуализации мониторинга Grafana, она использует образ grafana/grafana и зависит от службы Prometheus.

Результатом этой части работы стало полностью контейнеризированное приложение развернутое в Docker всего с помощью одной команды “*docker-compose up -d*”.

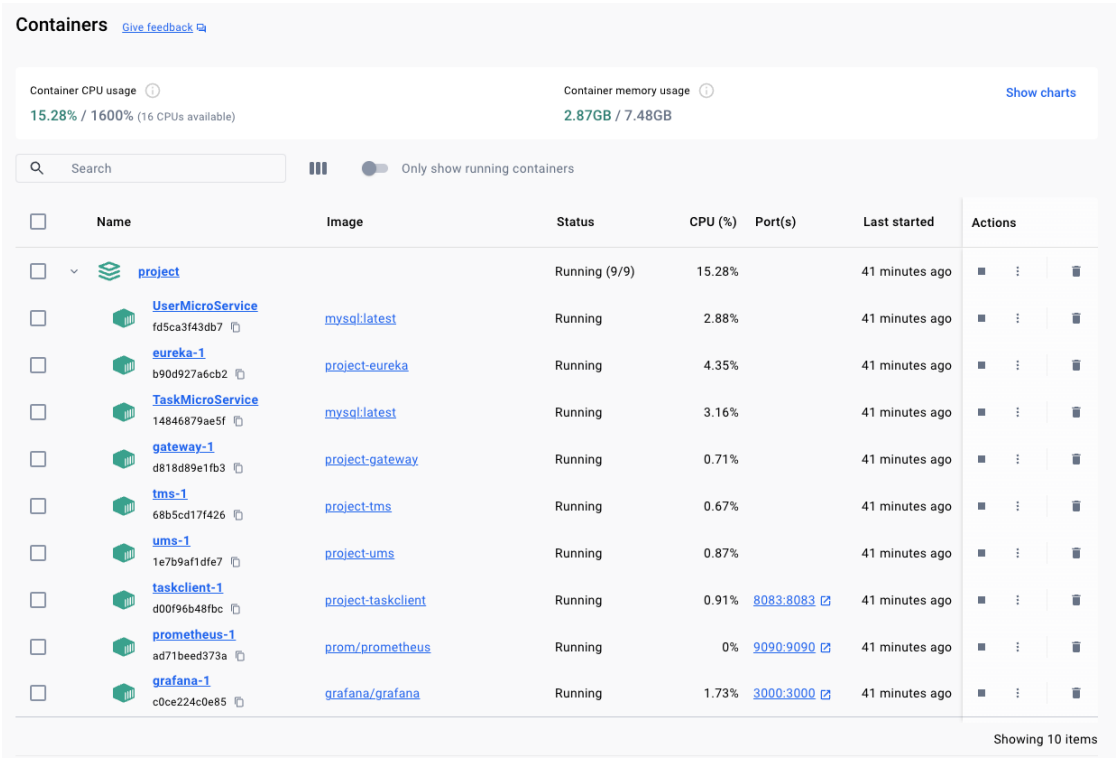


рисунок 34

3. Заключение

Подводя итоги данной работы хочется сказать, что это было увлекательное путешествие в мир разработки на Java. Более детальное погружение в особенности работы Spring Framework и архитектуру микросервисов обнажило тот огромный пласт информации, который еще предстоит изучить. Возможности, которые разработчики заложили в данный фреймворк просто потрясают, в этот момент становится понятно, почему он так популярен сейчас, и что позволит ему оставаться на вершине потом.

Реализация этого небольшого проекта позволила:

- Лучше понять внутренне устройство Spring Framework
- Ближе познакомится с микросервисной архитектурой
- Понять почему так важно планировать и проектировать
- Какую роль в этом играет тестирование
- Насколько сильно современные технологии могут помочь в работе над ежедневными задачами

Результатом стало:

- Приложение, написанное с использованием современного фреймворка с оглядкой на популярную архитектуру
- Применение на практике знаний принципов ООП и SOLID
- Работа с REST и Spring MVC
- Опыт в работе над обеспечением безопасности приложения
- Работа с базами данных
- Работа с Docker
- Практика в написании кода, его отладке, тестировании и документировании

Отдельно хочется сказать, что, наверное, самым сложным моментом во всей работе было заставить работать все микросервисы как одно целое. Настроить трансфер всех данных, включая ошибки, от сервисов к клиенту. Именно в тот момент мне стало понятно, насколько важно на этапе планирования и проектирования заложить «контракты», по которым независимые части будут общаться друг с другом.

Что можно было бы сделать лучше в этой работе? Да почти все! Это только начало пути…

4. Список используемой литературы

Книги:

- "Spring Microservices in Action" by John Carnell
- "Java Persistence with Hibernate" by Christian Bauer and Gavin King
- "Spring Security in Action" by Laurentiu Spilca
- "Docker Deep Dive" by Nigel Poulton
- "Monitoring with Prometheus" by Brian Brazil
- "Effective Java" by Joshua Bloch
- "Head First Java" by Kathy Sierra and Bert Bates
- "Java Concurrency in Practice" by Brian Goetz
- "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
- "Java: The Complete Reference" by Herbert Schildt
- "Thinking in Java" by Bruce Eckel

Интернет ресурсы:

- Spring Boot Documentation
- Spring Cloud Netflix Documentation
- Официальная документация Java SE
- Официальная документация MySQL
- Официальная документация Thymeleaf
- Baeldung
- StackOwerflow
- Habr