Geekbrains

**Исследование возможности организации микросервисной архитектуры на базе фреймворка Spring**

Программа:

«Программирование»

Заговорчев Владимир Александрович

Москва

2024

**Содержание**

[Введение 3](#_Toc161911976)

[Теоретическая глава 4](#_Toc161911977)

[Практическая глава 9](#_Toc161911978)

[Заключение 10](#_Toc161911979)

[Список используемой литературы 11](#_Toc161911980)

# Введение

**Тема проекта:** Исследование возможности организации микросервисной архитектуры на базе фреймворка Spring.

**Цель:** Изучить особенности организации микросервисной архитектуры на базе фреймворка Spring.

**Какую проблему решает:** Дефицит знаний и навыков по этой тематике.

**Задачи:**

1. Изучить литературу, касающуюся темы исследования.
2. Разработать REST API микросервис по созданию и поиску заметок.
3. РазработатьREST API микросервис по созданию и поиску задач.
4. Подключить для тестирования API Swagger.
5. Настроить Eureka для регистрации и управления микросервисами.
6. Настроить маршрутизацию через Spring ApiGateway.
7. Проверить работу.

**Инструменты:** Java JDK, Intellij IDEA, Spring, Swagger

# Теоретическая глава

**Spring. Spring Boot**

Из-за громоздкой конфигурации зависимостей настройка Spring для корпоративных приложений превратилась в весьма утомительное и подверженное ошибкам занятие. Особенно это относится к приложениям, которые используют также несколько сторонних библиотек

Каждый раз, создавая очередное корпоративное Java-приложение на основе Spring, вам необходимо повторять одни и те же рутинные шаги по его настройке:

* В зависимости от типа создаваемого приложения (Spring MVC, Spring JDBC, Spring ORM и т.д.) импортировать необходимые Spring-модули
* Импортировать библиотеку web-контейнеров (в случае web-приложений)
* Импортировать необходимые сторонние библиотеки (например, Hibernate, Jackson), при этом вы должны искать версии, совместимые с указанной версией Spring
* Конфигурировать компоненты DAO, такие, как: источники данных, управление транзакциями и т.д.
* Конфигурировать компоненты web-слоя, такие, как: диспетчер ресурсов, view resolver
* Определить класс, который загрузит все необходимые конфигурации

Авторы Spring решили предоставить разработчикам некоторые утилиты, которые автоматизируют процедуру настройки и ускоряют процесс создания и развертывания Spring-приложений, под общим названием Spring Boot.

Spring Boot — это полезный проект, целью которого является упрощение создания приложений на основе Spring. Он позволяет наиболее простым способом создать web-приложение, требуя от разработчиков минимум усилий по его настройке и написанию кода.

Spring Boot обладает большим функционалом, но его наиболее значимыми особенностями являются: управление зависимостями, автоматическая конфигурация и встроенные контейнеры сервлетов.

Чтобы ускорить процесс управления зависимостями, Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику посредством так называемых starter-пакетов (spring-boot-starter-web, spring-boot-starter-data-jpa и т.д.).

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволит получить универсальное решение для всех, связанных со Spring технологий, избавляя программиста от лишнего поиска примеров кода и загрузки из них требуемых дескрипторов зависимостей (пример таких дескрипторов и стартовых пакетов будет показан ниже).

Например, если вы хотите начать использовать Spring Data JPA для доступа к базе данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa и все будет готово (вам не придется искать совместимые драйверы баз данных и библиотеки Hibernate).

Если вы хотите создать Spring web-приложение, просто добавьте зависимость spring-boot-starter-web, которая подтянет в проект все библиотеки, необходимые для разработки Spring MVC-приложений, таких как spring-webmvc, jackson-json, validation-api и Tomcat.

Другими словами, Spring Boot собирает все общие зависимости и определяет их в одном месте, что позволяет разработчикам просто использовать их, вместо того, чтобы изобретать колесо каждый раз, когда они создают новое приложение.

Следовательно, при использовании Spring Boot, файл pom.xml содержит намного меньше строк, чем при использовании его в Spring-приложениях.

Второй превосходной возможностью Spring Boot является автоматическая конфигурация приложения.

После выбора подходящего starter-пакета, Spring Boot попытается автоматически настроить Spring-приложение на основе добавленных вами jar-зависимостей.

Например, если вы добавите Spring-boot-starter-web, Spring Boot автоматически сконфигурирует такие зарегистрированные бины, как DispatcherServlet, ResourceHandlers, MessageSource.

Если вы используете spring-boot-starter-jdbc, Spring Boot автоматически регистрирует бины DataSource, EntityManagerFactory, TransactionManager и считывает информацию для подключения к базе данных из файла application.properties.

Если вы не собираетесь использовать базу данных, и не предоставляете никаких подробных сведений о подключении в ручном режиме, Spring Boot автоматически настроит базу в памяти, без какой-либо дополнительной конфигурации с вашей стороны (при наличии H2 или HSQL библиотек).

Автоматическая конфигурация может быть полностью переопределена в любой момент с помощью пользовательских настроек.

Каждое Spring Boot web-приложение включает встроенный web-сервер. Посмотрите на список контейнеров сервлетов, которые поддерживаются "из коробки".

Разработчикам теперь не надо беспокоиться о настройке контейнера сервлетов и развертывании приложения на нем. Теперь приложение может запускаться само, как исполняемый jar-файл с использованием встроенного сервера.

Если вам нужно использовать отдельный HTTP-сервер, для этого достаточно исключить зависимости по умолчанию. Spring Boot предоставляет отдельные starter-пакеты для разных HTTP-серверов.

Создание автономных web-приложений со встроенными серверами не только удобно для разработки, но и является допустимым решением для приложений корпоративного уровня и становится все более полезно в мире микросервисов. Возможность быстро упаковать весь сервис (например, аутентификацию пользователя) в автономном и полностью развертываемом артефакте, который также предоставляет API — делает установку и развертывание приложения значительно проще.

**Микросервисная архитектура**

Микросервисная архитектура — вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps.

Если в традиционных вариантах сервис-ориентированной архитектуры модули могут быть сами по себе достаточно сложными программными системами, а взаимодействие между ними зачастую полагается на стандартизованные тяжеловесные протоколы (такие, как SOAP, XML-RPC), в микросервисной архитектуре системы выстраиваются из компонентов, выполняющих относительно элементарные функции, и взаимодействующие с использованием экономичных сетевых коммуникационных протоколов (в стиле REST с использованием, например, JSON, Protocol Buffers, Thrift). За счёт повышения гранулярности модулей архитектура нацелена на уменьшение степени зацепления и увеличение связности, что позволяет проще добавлять и изменять функции в системе в любое время.

Свойства, характерные для микросервисной архитектуры:

модули можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;

* модули организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;
* модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных архитектурах: приоритет отдаётся в пользу наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения;
* архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Философия микросервисов фактически копирует философию Unix, согласно которой каждая программа должна «делать что-то одно, и делать это хорошо» и взаимодействовать с другими программами простыми средствами: микросервисы минимальны и предназначаются для единственной функции. Основные изменения в связи с этим налагаются на организационную культуру, которая должна включать автоматизацию разработки и тестирования, а также культуру проектирования, от которой требуется предусматривать обход прежних ошибок, исключение по возможности унаследованного кода (микросервисы часто заменяют целиком, поскольку их функции элементарны).

Наиболее популярная среда для выполнения микросервисов — системы управления контейнеризованными приложениями (такие как Kubernetes и её надстройки OpenShift и CloudFoundry[en], Docker Swarm, Apache Mesos[en]), в этом случае каждый из микросервисов как правило изолируется в отдельный контейнер или небольшую группу контейнеров, доступную по сети другим микросервисам и внешним потребителям, и управляется средой оркестрации, обеспечивающей отказоустойчивость и балансировку нагрузки. Типовой практикой является включение в контур среды выполнения системы непрерывной интеграции, обеспечивающее автоматизацию обновления и развёртывания микросервисов.

**Spring Cloud**

Spring Cloud упрощает подключение к сервисам и получение возможностей окружения в облачных платформах, таких как Cloud Foundry и Heroku. Особая поддержка Spring-приложений через Java и XML конфигурации делает подключение к облачным сервисам тривиальной задачей. Вы можете использовать существующие облачные коннекторы(Cloud Foundry и Heroku) или написать собственный для вашей облачной платформы. Пока что "из коробки" поддерживаются наиболее популярные сервисы(реляционные СУБД, MongoDB, Redis, Rabbit), но также возможно расширение для ваших сервисов. Ни один из сервисов не требует изменения самого Spring Cloud, достаточно просто добавить необходимую вам jar-библиотеку в область видимости classpath.

Spring Cloud использует следующие основные концепции:

* Cloud Connector: Интерфейс, через который можно реализовать облачный провайдер, чтобы дать возможность библиотекам работать с облачной платформой
* Service Connector: Объект, подобный javax.sql.DataSource, представляющий собой соединение как сервис
* Service information: Информация о сервисе - хост, порт, параметры доступа
* Application information: Информация о приложении и сведения о подключенных библиотеках

Spring Cloud ориентирован на предоставлении необходимого функционала "из коробки" в большинстве случаев, а также механизм раширений для других. Существуют два пути расширения:

* Java и XML конфигурация для Spring-приложений: Простые пути создания бинов для сервисов в приложении
* Расширение облачной платформы: Используя Cloud Connector, возможно расширение Spring Cloud для других облачных платформ
* Service Information и расширение Connector: Spring Cloud позволяет соединяться с различными типами сервисов настолько долго, пока есть возможность получать информацию о соединении из приложений операционной системы и, при неоходимостти, преоразовать в service connector

**Swagger**

Swagger — это инструмент, который помогает разработчикам создавать, документировать и проверять API. API — это набор правил и и протоколов, которые позволяют различным системам обмениваться информацией между собой.

API — это общий термин, который применяется в контексте совершенно разных систем. Есть более узкий термин, REST API — это конкретный API, который используется для обмена данными между клиентами и серверами в интернете.

Часть работы с REST API — это создание описаний работы API: информации о ресурсах, параметрах запросов, возвращаемых данных, конечных точках и других важных вещах. Чтобы автоматизировать это описание, сделать его структурированным и прозрачным, разработчики используют Swagger. А системные аналитики, в свою очередь, с его помощью формируют требования к IT-системам.

Документация содержит следующую информацию:

* Заголовок с названием и версией API и описание
* Описание ресурса с описанием операции
* Ответ, который будет возвращаться при успешном выполнении операции, с указанием ожидаемого формата (application/json) и модели данных (UsersList)
* Компоненты, где определена модель данных "UsersList", которая представляет собой массив объектов с полями "id" и "name"

# Практическая глава

**Микросервис «Список задач»**

Необходимо создать REST API приложение для управления списком задач с использованием Spring Boot и Spring Data JPA. Требуется реализовать следующие функции:

* Добавление задачи.
* Просмотр всех задач.
* Просмотр задач по статусу (например, "завершена", "в процессе", "не начата").
* Изменение статуса задачи.
* Удаление задачи.

Структура задачи (класс Task):

* ID (автоинкрементное) (тип Long)
* Описание (не может быть пустым) (тип String)
* Статус (одно из значений: "не начата", "в процессе", "завершена") (Тип TaskStatus )
* Дата создания (автоматически устанавливается при создании задачи) (Тип LocalDateTime)

Включить в приложение Swagger для тестирования API.

**Микросервис «Список заметок»**

Необходимо разработать веб-приложение на Spring Boot, которое будет представлять из себя сервис для учета личных заметок. Приложение должно поддерживать следующие функции:

Все методы контроллера возвращают ResponseEntity

* Добавление заметки.
* Просмотр всех заметок.
* Получение заметки по id.
* Редактирование заметки.
* Удаление заметки.

Структура заметки:

* ID (автоинкрементное) (тип - Long)
* Заголовок (не может быть пустым) (тип - String)
* Содержимое (не может быть пустым) (тип - String)
* Дата создания (автоматически устанавливается при создании заметки) (тип - LocalDateTime)

Включить в приложение Swagger для тестирования API.

**Настройка микросервисов**

Подключить менеджер сервисов Eureka и добавить туда ранее созданные микросервисы. Для корректной маршрутизации запросов настроить API Gateway.

# Заключение

1. В результате проделанной работы можно выделить следующее:
2. Создан микросервис «Список задач»
3. Создан микросервис «Список заметок»
4. Настроен менеджер сервисов на базе сервиса Eureka.
5. Настроена маршрутизация через API Gateway.
6. Добавлен Swagger для тестирования REST API.

Разработанный программный код находится в репозитории GitHub: <https://github.com/Pl0tter/java_spring_cloud>

# Список используемой литературы

* + <https://habr.com>
  + <https://javarush.com>
  + <https://spring.io/>
  + <https://gb.ru>