API-GATEWAY POC

Цели

1. Единая точка входа
2. Стабильность API
3. Retry неудачных сообщений
4. Использование Circuit Breaker для защиты от каскадных отказов
5. Rate Limiting входящих запросов

Стек

* Java 21, Spring Boot 3.5.8, Spring Cloud 2025.0.0
* Spring Cloud Gateway (Webflux)
* Redis
* Resilience4j
* Testcontainers, Wiremock

Описание сервисов (1)

hash-unlocker
Сервис по приему заявок на расшифровку md5 хэшей с одним endpoint’ом
Endpoint идемпотентный, то есть, при повторном запросе будет возвращен id уже существующей заявки

POST /api/local/applications

```
{
   "digest": "2623e0d1f4e1a3093ee71672ec1c771a",
   "algorithm": "MD5"
}
```

```
{
 "externalId": "550e8400-e29b-41d4-a716-446655440000"
}
```

Описание сервисов (2)

api-gateway
Сервис proxy, открытый наружу для клиентских запросов.

Его responsibilities
1.Проксирование запросов к бэкендам с возможностью rewrite path с использованием Spring Cloud Gateway на базе Netty (с возможностью переключения на servlet-контейнер)

2.Rate limiting запросов по API-KEY из заголовка как простая демонстрация security - через фильтр Spring Cloud Gateway с хранением счётчиков в Redis.

3.Retry-фильтры Spring Cloud Gateway для автоматического повтора неуспешных запросов.

4.Circuit Breaker в виде фильтра Spring Cloud Gateway (через реализацию Resilence4j) для отсечения запросов при недоступности бэкенда

Демонстрация build.gradle

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.5.8'
    id 'io.spring.dependency-management' version '1.1.7'
}

group = 'com.sokolov'
version = '0.0.1-SNAPSHOT'
description = 'api-gateway'

java {
    toolchain { JavaToolchainSpec it ->
        languageVersion = JavaLanguageVersion.of(21)
    }
}

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2025.0.0")
    set('wiremockVersion', "3.13.2")
}
```

```
dependencies { Add Starters...
    implementation 'org.springframework.cloud:spring-cloud-starter-gateway-server-webflux'
    implementation("org.springframework.boot:spring-boot-starter-data-redis-reactive")
    implementation "org.springframework.cloud:spring-cloud-starter-circuitbreaker-reactor-resilience4j"

    testImplementation "org.wiremock:wiremock-standalone:${wiremockVersion}"

    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation "org.testcontainers:testcontainers"
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}
```

```
tasks.named('test', Test) { Test it ->
    useJUnitPlatform { JUnitPlatformOptions it ->
        excludeTags 'integration'
    }
}

tasks.register('integrationTest', Test) { Test it ->
    description = 'Runs integration tests'
    group = 'verification'

    useJUnitPlatform { JUnitPlatformOptions it ->
        includeTags 'integration'
    }

    shouldRunAfter(tasks.named('test'))
}

tasks.named("jar") { Task it ->
    enabled = false
}

tasks.named("bootJar") { Task it ->
    archiveFileName = "api-gateway.jar"
}

tasks.named('check') { Task it ->
    dependsOn(tasks.named('integrationTest'))
}
```

Rate Limiter

В качестве реализации rate limiter’а был выбран стандартный фильтр из Spring Cloud Gateway RequestRateLimiter

RequestRateLimiter по умолчанию тянет из контекста бин RateLimiter
Одна из его реализаций RedisRateLimiter
Для корректной работы необходимо добавить бин KeyResolver

```
@Bean
@ConditionalOnMissingBean
public RedisRateLimiter redisRateLimiter(ReactiveStringRedisTemplate redisTemplate,
                                         @Qualifier(RedisRateLimiter.REDIS_SCRIPT_NAME) RedisScript<List<Long>> redisScript,
                                         ConfigurationService configurationService) {
    return new RedisRateLimiter(redisTemplate, redisScript, configurationService);
}
```

В качестве ключа я использую api-key из headers сообщения
Так же я сохраняю sha256 хэш ключа, а не сами ключи для безопасно

```
@RequiredArgsConstructor  2 usages
public class ApiKeyResolver implements KeyResolver {

    private final ApiKeyExtractor apiKeyExtractor;

    private final HashCalculator hashCalculator;

    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
        return Mono.justOrEmpty(apiKeyExtractor.extract(exchange.getRequest()))
                .map(hashCalculator::sha256Hex);
    }
}
```

Вот что сохраняет Redis у себя
request_rate_limiter.apiKeyHash.tokens = 45
request_rate_limiter.apiKeyHash.timestamp = 1700000000

Tokens - количество текущих токенов по ключу
Timestamp - последний момент обновления

Основные конфиги для rate-limiter’а
replenishRate - кол-во токенов, добавляемых в корзину в секунду
burstCapacity - максимальное количество токенов в корзине
requestedTokens - Сколько токенов списывается за один запрос

```yaml
spring:
  cloud:
    gateway:
      server:
        webflux:
          httpclient:
            connect-timeout: 2000
            response-timeout: 3s
          default-filters:
            - name: RequestRateLimiter
              args:
                key-resolver: "#{@apiKeyResolver}"
                redis-rate-limiter.replenishRate: 1
                redis-rate-limiter.burstCapacity: 60
                redis-rate-limiter.requestedTokens: 10
```

Retry

Retry реализован как один из фильтров Spring Cloud Gateway
Retries - 2 (всего 3 запроса, если не получаем успешный ответ)
Statuses - список HTTP статусов, которые мы ретраем
Backoff:
firstBackoff - 200 ms (при первом фейле след запрос будет отправлен через 200 ms)
factor - 2 (каждый новый ретрай увеличиваем время ожидания перед повторной отправкой в 2 раза)

В данном случае, если бекенд отвечает ошибкой, то мы выполним 1 запрос, 2 запрос после 200 мс и 3 запрос еще после 400 мс

И так как в CircuitBreaker’е указан fallback, то мы получим ответ от fallback’а

```
- name: CircuitBreaker
  args:
    name: applicationsCB
    fallbackUri: forward:/fallback
    statusCodes: BAD_GATEWAY,SERVICE_UNAVAILABLE,GATEWAY_TIMEOUT,INTERNAL_SERVER_ERROR
```

```
@RestController
@RequestMapping("/fallback")
public class FallbackController {

    @PostMapping
    public Mono<ResponseEntity<Map<String, Object>>> applicationsFallback() {
        return Mono.just(
                ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE)
                        .body(Map.of("error", "UPSTREAM_UNAVAILABLE"))
        );
    }

}
```

```
- name: Retry
  args:
    retries: 2
    methods: POST
    statuses: BAD_GATEWAY,SERVICE_UNAVAILABLE,GATEWAY_TIMEOUT,INTERNAL_SERVER_ERROR
    backoff:
      firstBackoff: 200ms
      maxBackoff: 2s
      factor: 2
      basedOnPreviousValue: true
```

Circuit Breaker

Circuit Breaker позволяет нам не пускать запросы дальше, чтобы избегать каскадных ошибок

В данном случае реализован как один из фильтров Spring Cloud Gateway через реализацию Resilence4J

```yaml
resilience4j:
  circuitbreaker:
    instances:
      applicationsCB:
        slidingWindowType: COUNT_BASED
        slidingWindowSize: 5
        minimumNumberOfCalls: 5
        failureRateThreshold: 100
        waitDurationInOpenState: 30s
```

* **slidingWindowType: COUNT_BASED** - оконная модель по количеству Вызовов

* **slidingWindowSize: 5** - размер окна (5 последних вызовов)

```yaml
- name: CircuitBreaker
  args:
    name: applicationsCB
    fallbackUri: forward:/fallback
    statusCodes: BAD_GATEWAY,SERVICE_UNAVAILABLE,GATEWAY_TIMEOUT,INTERNAL_SERVER_ERROR
```

* **minimumNumberOfCalls: 5** - Circuit Breaker не будет принимать решение, пока не выполнится минимум 5 вызовов

* **failureRateThreshold: 100** - Circuit Breaker откроется только если все 5 из 5 вызовов завершились ошибкой

* **waitDurationInOpenState: 30s**
Если CB перешёл в состояние OPEN, он:
30 секунд полностью блокирует вызовы
после этого перейдёт в состояние HALF_OPEN

Рассматриваемые сценарии

Проксирование /api/public/applications -> /api/local/applications
Получение 200ok

API-KEY отсутствует - получаем 403 Forbidden на стороне api-gateway

По api-key из заголовка упираемся в rate-limit и получаем 429 too many requests от api-gateway

Запрос фейлится на стороне бекенда - срабатывает ретрай на стороне api-gateway

Запрос фейлится больше n раз и срабытывает CircuitBreaker (OPEN) - запросы с gateway не уходят на бекенд, сразу получаем fallback