Программирование в 2026: как находить и устранять каскадные ретраи

23.11.2026 · PDF-версия статьи с сайта cms.dolinais.ru

Почему один сбой превращается в 64 запроса

Представьте: платёжный шлюз отвечает 503. Приложение повторяет запрос, и нагрузка на соседний сервис вырастает в разы. В коде всё выглядит разумно: три слоя, на каждом аннотация @Retryable с привычным значением 3. Проблема обнаруживается после перехода со Spring Retry на Spring Framework 7 (релиз 2026 года), где это число уже означает другое. В результате временный сбой превращается в каскад запросов.

Ретрай (повтор) — это автоматическая повторная отправка запроса при ошибке. **Каскадный ретрай** — ситуация, когда повторы на нескольких уровнях перемножаются, многократно увеличивая число обращений.

В статье разберём механику на примере платёжного клиента, покажем, почему обычные тесты пропускают проблему, и дадим конкретные шаги по исправлению.

Как работает @Retryable в Spring Framework 7

До версии 7 аннотация @Retryable из Spring Retry имела параметр maxAttempts, который задавал общее число попыток, включая первую. Значение 3 означало: одна основная попытка и два повтора.

В Spring Framework 7 (нативный механизм повторных попыток) параметр maxAttempts сохранил название, но семантика изменилась: теперь это число *повторов после первой неудачной попытки*. То есть maxAttempts = 3 даёт одну основную попытку и три повторные — итого 4 вызова.

Если таких слоёв три (например, клиент, сервис и контроллер), и каждый настроен на maxAttempts = 3, общее количество запросов к внешнему сервису составит 4 × 4 × 4 = 64. Именно это и произошло в разобранном примере: одна ошибка шлюза вызвала 64 запроса вместо ожидаемых трёх-четырёх.

> Важно: изменение семантики не является ошибкой фреймворка — это осознанное решение, но оно требует пересмотра конфигураций при миграции.

Почему тесты не замечают проблему

Стандартные юнит-тесты обычно проверяют, что метод возвращает ожидаемый результат при успешном ответе, или что выбрасывается исключение после исчерпания попыток. Они не измеряют *фактическое количество вызовов* внешнего сервиса.

Кроме того, в тестах редко эмулируется каскад: если каждый слой тестируется изолированно с моком, который сразу отвечает успехом, повторы не срабатывают. Проблема проявляется только в интеграционном сценарии, когда все слои включены и внешний сервис стабильно возвращает ошибку.

Чтобы поймать каскад, нужно добавить проверку числа вызовов. Например, с помощью Mockito:

```
verify(paymentClient, times(4)).charge(any());
```

Но и это не гарантирует обнаружение, если тест не воспроизводит все три слоя одновременно.

Как исправить конфигурацию и избежать лавины

Вот конкретные шаги, которые помогут устранить каскад:

1. **Пересмотрите значения maxAttempts на каждом слое.** Если раньше вы использовали 3 в расчёте на общее число попыток, теперь это число повторов. Уменьшите его до 2, чтобы получить три вызова, или настройте каждый слой осознанно.
2. **Оставляйте ретраи только на одном уровне.** Обычно достаточно повторять запрос на уровне клиента внешнего сервиса. Промежуточные слои не должны дублировать эту логику.
3. **Используйте разные политики для разных типов ошибок.** Например, повторять только при 503 и 429, но не при 400 или 404. В Spring Framework 7 это настраивается через @Retryable(retryFor = {ServiceUnavailableException.class}).
4. **Добавьте экспоненциальную задержку (backoff).** Это снизит нагрузку на сервис в момент сбоя. Пример: @Backoff(delay = 100, multiplier = 2).
5. **Включите ограничение общего числа попыток (circuit breaker).** Если сервис недоступен долго, повторять запросы бессмысленно. Библиотека Resilience4j позволяет разомкнуть цепь после нескольких неудач.

После изменения конфигурации обязательно проверьте фактическое количество вызовов в интеграционном тесте с эмуляцией ошибок на всех слоях.

Пример правильной настройки

Допустим, у нас есть платёжный клиент, сервис и контроллер. Раньше на каждом стояло @Retryable(maxAttempts = 3). Теперь мы хотим, чтобы при сбое шлюза выполнялось не более трёх запросов.

Оставляем ретраи только на клиенте и настраиваем maxAttempts = 2 (две повторные попытки, итого три вызова):

```
@Retryable(maxAttempts = 2, backoff = @Backoff(delay = 100, multiplier = 2))
public PaymentResponse charge(PaymentRequest request) { ... }
```

На сервисе и контроллере аннотацию убираем. Если нужно обрабатывать ошибки на этих уровнях, используем @ExceptionHandler без повторов.

Теперь при сбое шлюза будет ровно три запроса вместо 64.

Что ещё изменилось в Spring Framework 7

Помимо семантики maxAttempts, в новой версии появилась поддержка реактивных ретраев (Mono.retryWhen), улучшена интеграция с виртуальными потоками (Project Loom) и добавлены аннотации для настройки на уровне класса. При миграции стоит проверить все места, где используются @Retryable и @Backoff.

Если вы только планируете переход, начните с аудита существующих конфигураций: выпишите все слои, где включены повторы, и пересчитайте ожидаемое число вызовов по формуле (maxAttempts + 1) ^ количество слоёв. Для трёх слоёв с maxAttempts = 2 это 3^3 = 27 запросов — всё ещё много. Поэтому лучше оставить ретраи на одном уровне.

Вывод

Каскадные ретраи — типичная проблема при миграции на Spring Framework 7. Изменение семантики maxAttempts привело к тому, что привычные значения стали давать больше повторов. Решение: пересмотреть конфигурацию, оставить ретраи только на одном уровне, добавить backoff и circuit breaker. И обязательно проверять фактическое количество вызовов в интеграционных тестах — обычные юнит-тесты такую проблему не ловят.

Следите за обновлениями фреймворка и не пренебрегайте аудитом при миграции — это сэкономит ресурсы и нервы.

Источник: https://cms.dolinais.ru/programmirovanie-v-2026-kak-nahodit-i-ustranyat-kaskadnye-retrai