Программирование: как разбирать нестабильные тесты — флаки в CI

26.10.2026 · PDF-версия статьи с сайта cms.dolinais.ru

Ситуация знакома почти каждой команде: интеграционный тест стабильно проходит локально, но в CI падает примерно раз в 30 прогонов. Перезапуск сборки возвращает зелёный статус, причина остаётся неизвестной, а доверие к тестам падает. Такой тест называют **флаки** (flaky test) — нестабильный тест, результат которого зависит не от кода, а от условий запуска. В 2026 году это одна из самых дорогих категорий технического долга: она маскирует реальные дефекты и приучает команду игнорировать красные сборки.

Почему локально зелено, а в CI красно

Локальная среда и CI различаются по множеству параметров: количество ядер, скорость диска, версия JDK, набор переменных окружения, доступность сети и, главное, порядок выполнения тестов. CI часто запускает тесты параллельно и в случайном порядке, чтобы быстрее находить связанность. Именно эта оптимизация и вскрывает скрытые зависимости.

Типичные причины флаки-тестов:

- **Общее состояние.** Тесты пишут в одну и ту же базу, таблицу или файл и не убирают за собой. Один тест оставляет строку, другой на неё натыкается.
- **Случайный порядок запуска.** JUnit 5 по умолчанию использует детерминированный, но неочевидный порядок; при включённом случайном порядке или параллельном выполнении зависимости проявляются.
- **Асинхронность.** Тест ждёт фиксированный Thread.sleep вместо события, и на медленном раннере не успевает.
- **Время и часовые пояса.** Логика зависит от текущей даты, а раннер живёт в UTC.
- **Сеть и внешние сервисы.** Тест ходит в реальный API или очередь, которая в CI недоступна или отвечает медленнее.
- **Порядок инициализации контекста Spring.** Кэш контекстов переиспользуется между классами, и настройки одного теста протекают в другой.

Шаг 1. Зафиксируйте факт и соберите данные

Прежде чем что-то менять, нужно получить воспроизводимость. Перезапуск сборки — не диагностика, а способ скрыть проблему. Полезно:

1. Включить сохранение отчётов и логов упавших прогонов как артефактов CI.
2. Записать в лог имя теста, порядок запуска, идентификатор потока и состояние базы на момент старта.
3. Запустить набор локально в том же режиме, что и CI: параллельно и в случайном порядке.

В JUnit 5 порядок задаётся через @TestMethodOrder и реализации MethodOrderer, а параллельность — параметром junit.jupiter.execution.parallel.enabled в junit-platform.properties. Временно включите случайный порядок (MethodOrderer.Random) и повторите прогон десятки раз — частота падений вырастет настолько, что причина станет видимой.

Шаг 2. Проверьте изоляцию данных

Самый частый источник — общая база. Проверьте, что каждый тест создаёт свои данные с уникальными идентификаторами и удаляет их в @AfterEach. Если используется @Transactional с откатом, убедитесь, что тест не проверяет поведение, требующее реального коммита: откат не сработает для кода, который открывает собственную транзакцию.

> Правило: тест не должен зависеть от строк, которые он не создал сам. Если для проверки нужны справочники — наполняйте их в самом тесте или в изолированной схеме.

Для Spring Boot удобно поднимать отдельную схему на каждый запуск (например, через Testcontainers с PostgreSQL) и очищать её перед классом. Это медленнее, но убирает целый класс флаки-падений.

Шаг 3. Замените ожидания на события

Фиксированные паузы — прямой путь к нестабильности. Вместо Thread.sleep(2000) используйте Awaitility или аналогичные библиотеки: они опрашивают условие с таймаутом и возвращают управление, как только состояние достигнуто. Это делает тест быстрым на быстрой машине и терпеливым на медленной.

Для асинхронных операций проверяйте не «прошло ли время», а «наступило ли событие»: сообщение в очереди, запись в базе, вызов мока. Тогда тест перестаёт зависеть от производительности раннера.

Шаг 4. Изолируйте контекст Spring

Spring кэширует ApplicationContext между тестовыми классами. Если один класс меняет свойства через @TestPropertySource или @DynamicPropertySource, а другой ожидает исходные, поведение становится зависимым от порядка. Проверьте, что конфигурации не пересекаются, а изменяемые бины помечены @DirtiesContext там, где это действительно нужно.

Полезно вывести в лог, сколько контекстов поднялось за прогон: неожиданно большое число — сигнал, что тесты не переиспользуют окружение и тратят время, а неожиданно малое при разных настройках — сигнал о протекании конфигурации.

Шаг 5. Отделите внешние зависимости

Если тест ходит в реальный сервис, он нестабилен по определению. Замените внешние вызовы на локальные эмуляторы: WireMock для HTTP, встроенную очередь или Testcontainers для брокера. Тогда сеть перестаёт быть переменной, а падение снова начинает означать дефект в коде.

Что делать, если причину найти не удалось

Иногда разбор затягивается. В этом случае помогает временная мера: пометить тест как нестабильный, вынести его в отдельный набор и запускать по расписанию с подробным логированием, не блокируя основную сборку. Главное — не удалять тест молча и не добавлять автоматический перезапуск без задачи на разбор: перезапуск лечит симптом и прячет причину.

Хорошая практика — вести журнал флаки-тестов: дата, имя, частота, гипотеза, статус. Через несколько недель видно, какие классы дают больше всего шума, и разбор идёт по приоритету, а не по настроению.

Профилактика

- Запускайте тесты в CI в случайном порядке — это дешёвый способ ловить связанность заранее.
- Держите тесты независимыми: каждый создаёт и убирает свои данные.
- Не используйте фиксированные паузы; ждите событий с таймаутом.
- Изолируйте внешние сервисы эмуляторами.
- Считайте флаки-тесты дефектом процесса, а не «особенностью CI».

Нестабильный тест — это не мелочь, а сигнал о скрытой связанности в коде и окружении. Разбор, построенный на данных и воспроизводимости, превращает «падает раз в 30 прогонов» в конкретное исправление, после которого сборка снова становится надёжным индикатором качества.

Источник: https://task.dolinais.ru/programmirovanie-kak-razbirat-nestabilnye-testy-flaki-v-ci