Ай-ТИ Фреш

ТЕХНИЧЕСКИЙ РАЗБОР

Автоматизация тестирования: путь от 0% до 87%

Пирамида автотестов и quality gates: Jest 30, pytest 9, Testcontainers, Playwright, Pact, k6

Июль 2026

itfresh.ru · ИТ-аутсорсинг для юридических лиц

О ЧЁМ ЭТОТ РАЗБОР

Суть проблемы

Продукт растёт, а каждый релиз проверяется вручную: регрессия занимает дни, критические баги всё равно доезжают до продакшна, и команда боится трогать старый код. Мы решаем это внедрением пирамиды автотестов — от unit до контрактных и нагрузочных — с жёсткими quality gates в CI, чтобы релиз проходил путь от коммита до продакшна за минуты без ручной регрессии.

Почему это важно бизнесу

- Неделя ручной регрессии на релиз — это оплаченные фичи, которые не приносят деньги, и фонд оплаты QA на повторяющуюся работу
- Критический баг в проде = возвраты, отток клиентов и штрафы по SLA; исправление в проде на порядок дороже, чем на этапе PR
- Без тестов рефакторинг замораживается: кодовая база стареет, оценки на любые доработки растут с каждым кварталом
- Хотфиксы мимо тестирования ломают смежные модули — один инцидент тянет за собой следующий

ЧТО МЫ ЗАКЛАДЫВАЕМ

Ключевые параметры реализации

Jest 30

раннер unit-тестов JS/TS: Node 18.14+, TS ≥5.4; coverageThreshold валит сборку ниже порога

по докам jestjs.io

pytest 9.0

раннер Python-тестов: async-тест без плагина теперь fail, parametrize для пороговых наборов

docs.pytest.org

p95<500 мс

порог k6-thresholds на http_req_duration; при провале k6 выходит с кодом 99 и валит CI

Grafana k6 2.x

MSI break 50

порог Stryker: сборка падает, если Mutation Score ниже 50% (по умолчанию break: null — гейт выключен)

StrykerJS, stryker-mutator.io

80% → 87%

стартовый гейт coverageThreshold по branches/lines в jest.config.ts; фактическое покрытие доведено до 87%

наш стандарт

≤1%

допустимый flaky-rate; выше — карантин и стоп на новые E2E до разбора причин

наш стандарт

РЕАЛИЗАЦИЯ 1 ИЗ 3

Фундамент пирамиды: unit-тесты и интеграционные на реальных зависимостях

Что настраиваем

Node.js/TypeScript- и Python-сервисы; PostgreSQL 16 и Redis 7 в контейнерах

Как мы это делаем

1. Ставим Jest 30 (Node 18.14+, TS ≥5.4) и pytest 9; в jest.config.ts включаем coverageThreshold {branches: 80, lines: 80}
2. Первыми покрываем деньги и доступ: расчёт тарифов, стек скидок, границы (цена не ниже 0); в pytest — parametrize по пороговым значениям
3. Интеграционные — Testcontainers 12: PostgreSqlContainer('postgres:16') + Redis, миграции прогоняем в beforeAll, реальный SQL вместо моков
4. Изоляция: каждый тест создаёт свои сущности и чистит за собой; shared-фикстуры запрещаем на code review

РЕЗУЛЬТАТ

SQL-запросы, миграции и кэш-инвалидация проверяются на настоящих движках; регрессии в бизнес-логике ловятся до мержа PR, а не в продакшне

КЛЮЧЕВОЙ НЮАНС

Покрытие наращиваем от критичной бизнес-логики, а не от простых утилит; coverageThreshold ставим с первого дня — иначе процент тихо ползёт вниз

РЕАЛИЗАЦИЯ 2 ИЗ 3

Верх пирамиды: E2E, контракты сервисов и нагрузочные прогоны

Что настраиваем

Playwright — ключевые сценарии, Pact — стыки 8 микросервисов, k6 — пиковая нагрузка

Как мы это делаем

1. Playwright 1.61: локаторы только по data-testid, retries: 2 в CI, trace: 'on-first-retry'; отказ CDN имитируем page.route() с ответом 503
2. Контракты: @pact-foundation/pact 16 (PactV3, MatchersV3), публикация в Pact Broker; провайдер верифицирует контракт в своём CI, токен — PACT_BROKER_TOKEN
3. Перед выкаткой — pact-broker can-i-deploy: релиз блокируется, пока контракт не верифицирован в целевой среде
4. k6 2.x еженедельно и перед релизом: stages с ramp-up до пиковых VU, thresholds http_req_duration ['p(95)<500','p(99)<1000'], errors rate<0.01

РЕЗУЛЬТАТ

Слом контракта валит сборку провайдера, а не потребителя — стыки чинятся за минуты; деградация latency видна до релиза, а не в час пик

КЛЮЧЕВОЙ НЮАНС

E2E держим тонким слоем: всё, что проверяется контрактом или интеграционным тестом, спускаем вниз по пирамиде — иначе CI разбухает и флейкует

РЕАЛИЗАЦИЯ 3 ИЗ 3

Quality gates в CI и карантин flaky-тестов

Что настраиваем

GitHub Actions: матрица сервисов, стадии unit → integration → E2E → quality gate

Как мы это делаем

1. actions/checkout@v5, unit-джобы матрицей по сервисам, postgres/redis как service-контейнеры, покрытие — codecov-action@v5 с flags по сервисам
2. Gate: PR без тестов не мержится — branch protection + required checks; покрытие <80% или MSI ниже break — сборка красная
3. Stryker в incremental-режиме (файл stryker-incremental.json в кэше CI): мутационный прогон PR за минуты, полный — ночью
4. Flaky: retry ≤2 с логом; прошёл со 2-й попытки — метка flaky и тикет, после 2 нестабильных прогонов — карантин без блокировки CI
5. Убираем недетерминизм: инжектируемые часы вместо Date.now(), фиксированный seed, ожидание состояния (expect.poll), а не sleep

РЕЗУЛЬТАТ

Готовность релиза видна по одному статусу gate; flaky не девальвируют красный CI — команда доверяет пайплайну и катит релизы ежедневно

КЛЮЧЕВОЙ НЮАНС

Мутационное тестирование — честный ответ, ловят ли тесты баги: высокое покрытие при выживших мутантах — иллюзия; стартуем с break: 50 и поднимаем порог поэтапно

ЧТО МЫ ВЫУЧИЛИ НА ПРАКТИКЕ

Подводные камни

- ✕ **Гонка за процентом покрытия**
  покрытие без ассертов ничего не ловит; качество тестов проверяем Stryker-ом: выжившие мутанты = дыры, MSI-порог держим в CI
- ✕ **Перевёрнутая пирамида**
  сотни E2E вместо unit: прогон на часы, флейки, разбор падений дольше написания кода; в E2E оставляем только ключевые пользовательские пути
- ✕ **Моки вместо реальных зависимостей**
  замоканный SQL не ловит ошибки миграций и индексов; интеграционные тесты гоняем на Testcontainers с боевыми образами postgres/redis
- ✕ **Shared-фикстуры и общий state**
  результат зависит от порядка прогона; каждый тест создаёт свои данные и чистит за собой — иначе параллельный запуск невозможен
- ✕ **Ретраи как лекарство от flaky**
  retry прячет гонки в коде; тест, прошедший со 2-й попытки, — flaky, а не зелёный: тикет и карантин после 2 нестабильных прогонов
- ✕ **Date.now() и sleep() в тестах**
  реальное время = недетерминизм; инжектируем часы, ждём состояния через expect.poll/waitFor, фиксируем seed генераторов данных
- ✕ **Gate без защиты ветки**
  quality gate обходится прямым пушем; включаем branch protection и required checks — иначе правило «PR без тестов не мержится» не работает
- ✕ **Нагрузка без порогов**
  k6 без thresholds лишь рисует графики; задаём p(95)/p(99) и rate ошибок — при провале exit code 99 останавливает деплой

УРОВНИ РЕАЛИЗАЦИИ

Как правильно

МИНИМУМ

- Unit-тесты на деньги и доступ: Jest 30 / pytest 9, coverageThreshold от 60%
- CI на каждый PR: линтер + unit; мерж только с зелёной обязательной проверкой
- Изоляция данных: тест сам создаёт и чистит свои сущности

НОРМАЛЬНО

- Интеграционные тесты на Testcontainers 12 с боевыми образами БД и миграциями
- E2E на Playwright: data-testid, retries 2, trace on-first-retry
- Порог покрытия 80%, карантин flaky, flaky-rate держим ниже 1%

ХОРОШО

- Pact 16 + Broker: can-i-deploy как обязательное условие деплоя
- k6 2.x по расписанию и перед релизом с порогами p(95)/p(99)
- Stryker incremental на PR с break-порогом MSI, полный прогон ночью
- Дашборд качества: покрытие, MSI, flaky-rate, время CI-прогона

ПРОВЕРЬТЕ У СЕБЯ

Чек-лист самопроверки

- [ ] Покрыта ли unit-тестами логика денег и прав доступа (тарифы, скидки, промокоды)?
- [ ] Падает ли сборка при покрытии ниже порога coverageThreshold?
- [ ] Запрещён ли мерж PR без зелёных обязательных проверок (branch protection)?
- [ ] Гоняются ли интеграционные тесты на реальных СУБД и кэше, а не на моках?
- [ ] Есть ли карантин flaky-тестов с тикетами и известен ли текущий flaky-rate?
- [ ] Проверяются ли контракты между сервисами до деплоя (pact-broker can-i-deploy)?
- [ ] Запускается ли нагрузочный тест с порогами p(95)/p(99) перед каждым релизом?
- [ ] Известен ли Mutation Score, а не только процент покрытия строк?
- [ ] Укладывается ли полный CI-прогон в 15 минут от коммита?

Если хотя бы на два вопроса ответ «нет» или «не знаю» — тема требует внимания.

ЧТО МЫ ДЕЛАЕМ

Как поможет ITFresh

ITFresh — ИТ-аутсорсинг для юридических лиц до 50 рабочих мест в Москве и области. 15+ лет практики, собственная инфраструктура в дата-центре МТС (8 серверов Dell Xeon Platinum).

- Аудит QA-процесса: считаем стоимость ручной регрессии и багов в проде против плана автоматизации
- Внедряем пирамиду под ключ: Jest/pytest, Testcontainers, Playwright, Pact, k6 — с обучением команды
- Собираем CI с quality gates: пороги покрытия и MSI, карантин flaky, контроль времени прогона
- Сопровождаем: дашборд метрик качества, разбор flaky, поэтапное поднятие порогов

15+
лет в ИТ-поддержке

50
рабочих мест — наш профиль

МТС
дата-центр, Москва

КОНТАКТЫ

Обсудить вашу задачу

Сайт	**itfresh.ru**
Телефон	**+7 903 729-62-41**
Telegram	**@ITfresh_Boss**

Бесплатно посмотрим вашу инфраструктуру по этому чек-листу и скажем, где тонко — без обязательств.

itfresh.ru

ДОКУМЕНТАЦИЯ И НАШ СТЕК

Техническая база

01 Jest: Configuring Jest — coverageThreshold, testEnvironment (jestjs.io — v30)

02 pytest: how-to guides, parametrize, changelog (docs.pytest.org — v9.0)

03 Playwright Test: configuration, retries, trace viewer (playwright.dev — v1.61)

04 Testcontainers for NodeJS: модуль PostgreSQL (node.testcontainers.org — v12)

05 Pact JS: Consumer Tests (PactV3), Pact Broker CLI (docs.pact.io — v16)

06 Grafana k6: Thresholds, Options reference (grafana.com — k6 2.x)

07 StrykerJS: Configuration, Incremental mode (stryker-mutator.io — 2026)

08 GitHub Actions: workflow syntax, service containers (docs.github.com — 2026)

09 Наш шаблон CI test.yml: unit → integration → e2e → gate (itfresh.ru — 2026)

Основано на официальной документации продуктов и нашей практике внедрения.