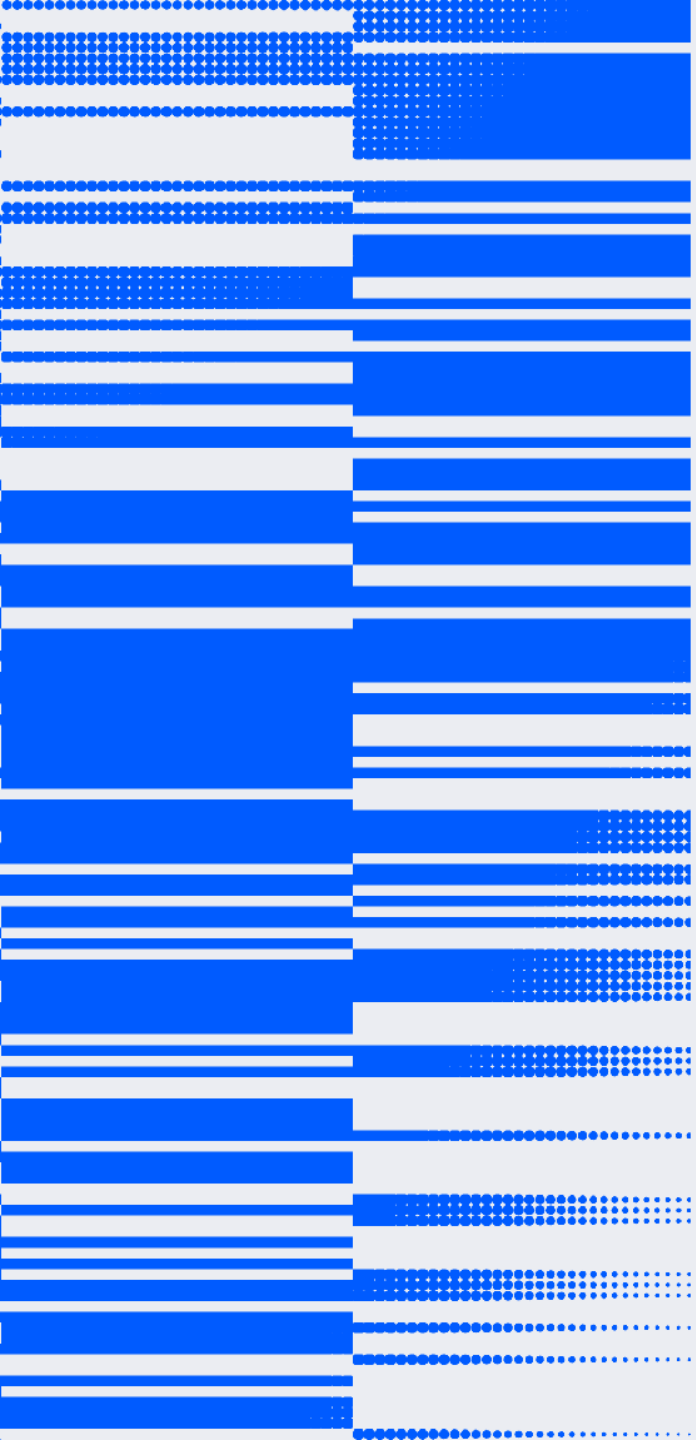


Ozon & Open Source

Зачем мы разработали новый фреймворк тестирования на Go



Вадим Збаращенко
Старший разработчик
информационных систем, Ozon

О себе

- Развиваю QA платформу Ozon
- Разрабатываю TMS систему Bugston
- Делаю тестирование на Go лучше
- Сторонник Open-Source

github.com/metafates

Ключевые цифры Testo

130 000 +

уникальных тестов

860 +

сервисов под тестами

И всё это — на Go,
на фреймворке Testo



О чём поговорим

01 Go и тестирование

02 Чего Go не хватает для больших e2e

03 Testo: как он устроен (на примере)

04 Зачем Open source

01

Go и тестирование

Единый стек, типизация
и параллельность

Тестирование в стандартной библиотеке

bake_test.go

```
func TestBake(t *testing.T) {
    medovik, ok := Bake("медовик")
    if !ok {
        t.Fatal("должны уметь печь медовик")
    }

    if !medovik.Tasty {
        t.Error("должен получиться вкусным")
    }
}
```

`go test` — раннер встроен в язык

- Сабтесты, параллельность и очистка ресурсов — уже есть
- Никаких зависимостей
- Для юнит-тестов этого хватает с головой

Один язык для сервиса и тестов

**Сервис на Go +
тесты на Python**

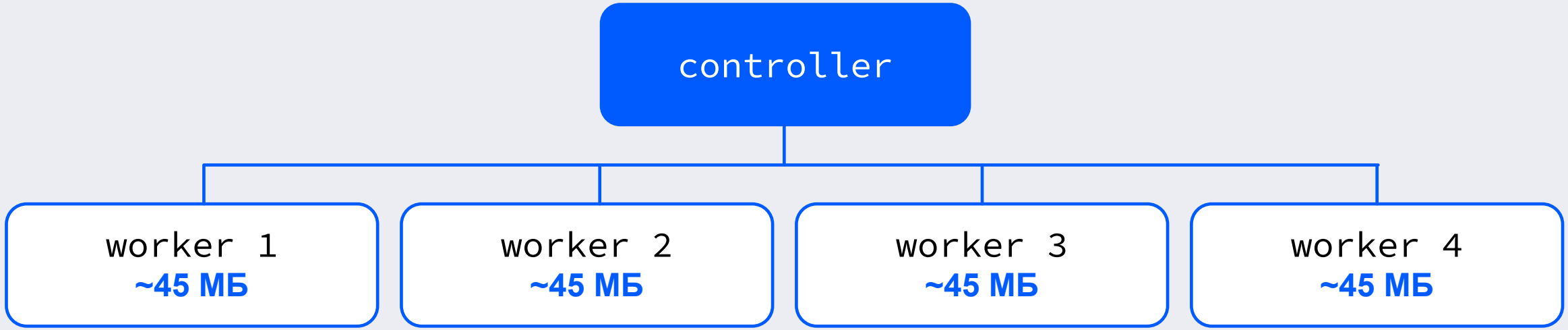
- Дублирование кода
- Сложность поддержки
- Динамическая типизация

Один язык для сервиса и тестов

Сервис на Go + тесты на Python	Сервис на Go + тесты на Go
• Дублирование кода • Сложность поддержки • Динамическая типизация	• Общие protobuf-модели и клиенты • Тест живёт в репозитории сервиса • Ошибки типов ловит компилятор

Тест, который не скомпилировался, сообщает об ошибке сразу — прогон ждать не нужно

Как параллелит pytest



- Единица параллельности — процесс: исторически из-за GIL, плюс изоляция тестов
- Каждый воркер — новый интерпретатор и импорт всего проекта
- Воркеров можно поднять больше числа ядер, но каждый следующий стоит столько же

Чем занят e2e-тест на самом деле



95%

времени e2e-тест просто ждёт сеть.
Воркер всё это время держит память
и ничего не делает

- Масштабировать ожидание с помощью процессов дорого в любом языке, не только в Python
- В Python дёшево ждать умеет Asyncio. В Go это не отдельный режим, а поведение рантайма по-умолчанию

Воркер и горутина

Воркер xdist

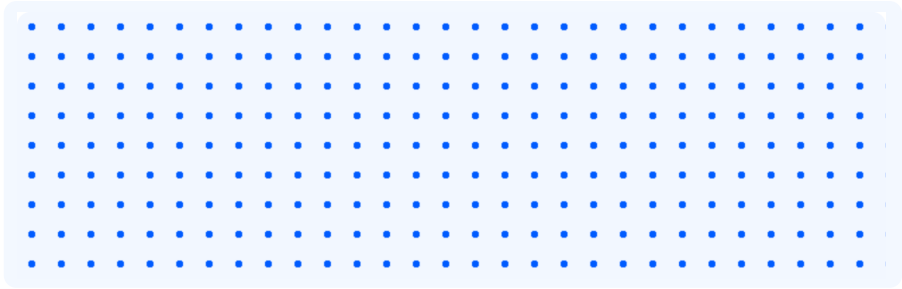
Процесс операционной системы



- Десятки МБ на старте
- Новый интерпретатор и импорт проекта
- Память занята, пока воркер ждёт

Горутина

Планировщик Go, **не ОС**



- ~2 КБ стека
- Микросекунды на создание
- На ожидании паркуется, поток переиспользуется

```
t.Parallel() // и всё — тест параллельный
```

200 I/O-bound тестов на одной машине

Каждый тест ждёт ответа HTTP-сервера 100 мс. Apple M4 Pro, 12 ядер, медианы из 2–3 прогонов

Параллельность	pytest-xdist	go test -parallel
1	21,2 с | 46 МБ	20,8 с | 15 МБ

200 I/O-bound тестов на одной машине

Каждый тест ждёт ответа HTTP-сервера 100 мс. Apple M4 Pro, 12 ядер, медианы из 2–3 прогонов

Параллельность	pytest-xdist	go test -parallel
1	21,2 с | 46 МБ	20,8 с | 15 МБ
8	3,2 с | 427 МБ	3,1 с | 16 МБ

200 I/O-bound тестов на одной машине

Каждый тест ждёт ответа HTTP-сервера 100 мс. Apple M4 Pro, 12 ядер, медианы из 2–3 прогонов

Параллельность	pytest-xdist	go test -parallel
1	21,2 с | 46 МБ	20,8 с | 15 МБ
8	3,2 с | 427 МБ	3,1 с | 16 МБ
32	1,9 с | ~1,5 ГБ	1,2 с | 19 МБ

200 I/O-bound тестов на одной машине

Каждый тест ждёт ответа HTTP-сервера 100 мс. Apple M4 Pro, 12 ядер, медианы из 2–3 прогонов

Параллельность	pytest-xdist	go test -parallel
1	21,2 с | 46 МБ	20,8 с | 15 МБ
8	3,2 с | 427 МБ	3,1 с | 16 МБ
32	1,9 с | ~1,5 ГБ	1,2 с | 19 МБ
200	**7–8 с | 3–5 ГБ | медленнее, чем 8**	**0,65 с | 22 МБ**

Процессы упираются в ресурсы раннера задолго до лимита стенда.
С **горутинами** узкое место — тестируемая система, а не раннер

02

Чего Go не хватает для больших e2e

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Suites

Группировка тестов и общие хуки подготовки окружения

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Suites

Группировка тестов и общие хуки подготовки окружения

Параметризация

Один сценарий — много входных данных, включая комбинации

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Suites

Группировка тестов и общие хуки подготовки окружения

Параметризация

Один сценарий — много входных данных, включая комбинации

Фикстуры

Создание и очистка ресурсов без копипасты в каждом тесте

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Suites

Группировка тестов и общие хуки подготовки окружения

Параметризация

Один сценарий — много входных данных, включая комбинации

Фикстуры

Создание и очистка ресурсов без копипасты в каждом тесте

Расширяемость

У каждой команды свои процессы — фреймворк должен подстраиваться

Чего Go не хватает для больших e2e

Пробел стандартной библиотеки

Отчёты

При сотнях тестов сырые логи читать сложно. Стандарт индустрии — Allure

Suites

Группировка тестов и общие хуки подготовки окружения

Параметризация

Один сценарий — много входных данных, включая комбинации

Фикстуры

Создание и очистка ресурсов без копипасты в каждом тесте

Расширяемость

У каждой команды свои процессы — фреймворк должен подстраиваться

В pytest всё это решается плагинами. А в Go — ?

Прошлая попытка: Allure-Go

allure-go

- Дал suites и Allure-отчёты
- Отчёты и фреймворк — единый монолит
- Расширить нельзя: архитектура не предполагала плагинов

Всё остальное

- Каждое решение закрывает часть списка требований
- Целиком список не закрывает ни одно

Allure-Go жив и используется. **Новое мы пишем на Testo**, потому что упёрлись в архитектуру без плагинов

Что умеют существующие фреймворки

	Плагины	Параметризация	Параллельность	Native go test
Ginkgo	Нет	Есть	Свой раннер, процессы	Нет
GoConvey	Нет	Нет	Нет	Нет
Godog	Нет	Есть	Есть	BDD-DSL
Testify suite	Нет	Нет	Ограничена в suite	Да
Allure-Go	Нет	Есть	Есть	Да, отчёты только Allure

Полная таблица с источниками — testo/COMPARISON.md

Требования к своему фреймворку

Плагины как в pytest: отчёты, хуки, планирование запуска

Совместимость: запуск через `go test`, работа с библиотеками для `testing`

Suites — и тесты вне их

Параллельность любой вложенности

Тесты выглядят как обычные Go-тесты



Так появился Testo

03

Testo (на примере)

От простого сценария
до полноценного e2e

Исходная точка: обычный Go-тест

```go
func TestBake(t *testing.T) {
    t.Run("Медовик", func(t *testing.T) {
        medovik, ok := Bake("медовик")
        if !ok {
            t.Fatal("должны уметь печь медовик")
        }

        // не выбрасывать же после теста
        t.Cleanup(func() { Eat(medovik) })

        if !medovik.Tasty {
            t.Error("медовик должен получиться вкусным")
        }
    })
}
```

Отчёты | Плагины

Suites | Параметры

Фикстуры | Хуки

Шаги

Подключаем Testo — две строки

```
go get github.com/ozontech/testo
```

```
-t.Run("Медовик", func(t *testing.T) {
+t.Run("Медовик", testo.Test(func(t *testo.T) {
      // ... сценарий не меняется
-})
+}))
```

- `testo.T` — надстройка над `testing.T` с тем же интерфейсом
- `testo.Test` — адаптер между ними
- Запуск — по-прежнему `go test` или кнопка в IDE

У ядра Testo ноль зависимостей — только стандартная библиотека

Плагин Allure: свой тип T

В Pytest: `conftest.py`

```
go get github.com/ozontech/testo-allure
```

```go
type T struct {
    *testo.T
    *allure.PluginAllure
}

func TestBake(t *testing.T) {
    t.Run("Медовик", testo.Test(func(t T) {
        medovik, ok := Bake("медовик")
        // ... сценарий не изменился
    }))
}
```

Плагины — это поля структуры. Обычный Go-эмбеддинг, никакой магии

Этого достаточно: после прогона появится Allure-отчёт

Переписывать тесты только ради отчётов не хочется — а поменять один тип можно за вечер

Плагины добавляют методы

```
t.Run("Медовик", testo.Test(func(t T) {
    medovik, ok := Bake("медовик")
    // ...

    // метод добавлен плагином testo-allure:
    t.Attach("ингредиенты", allure.Bytes(...))

    if !medovik.Tasty {
        t.Error("должен получиться вкусным")
    }
}))
```

В отчёте — вложение со списком ингредиентов

Обычный метод на T. Плагин не подключён — код не соберётся

Интерфейс плагина

```
type Plugin interface {
    Plugin(parent Plugin, options ...Option) Spec
}
```

Plan

Планирование запуска:
фильтровать, дублировать,
переупорядочивать тесты

Hooks

До и после Suites, тестов
и шагов — на любом уровне
вложенности

Overrides

Перехват методов T:
Fail, Log, Skip и остальные

Плагин видит то же состояние теста, что и сам тест: **у них общий T**

Свой плагин: автопараллельность

```go
type PluginParallel struct{ *testo.T }

func (p *PluginParallel) Plugin(
    parent testoplugin.Plugin,
    opts ...testoplugin.Option,
) testoplugin.Spec {
    var spec testoplugin.Spec
    spec.Hooks.BeforeEach.Func = func() {
        p.Parallel()
    }

    return spec
}
```

Осталось подключить — **одной строкой**:

```go
type T struct {
    *testo.T
    *allure.PluginAllure
    *PluginParallel
}
```

Теперь каждый тест в проекте — параллельный

В Pytest: `плагин с hookimpl`

Suites: группировка и общие хуки

```go
type Bakery struct {testo.Suite[T] }

func (Bakery) TestBake(t T) {
    medovik, ok := Bake("медовик")
    // ... тот же сценарий
}

func Test(t *testing.T) {
    testo.RunSuite(t, new(Bakery))
}
```

- [T] — какой набор плагинов используют тесты этого Suite
- Тесты — методы вида `TestXxx`
- Запуск — обычной тестовой функцией: `go test` работает
- Отдельный тест из Suite запускает расширение для VS Code

Suites: группировка и общие хуки

```go
type Bakery struct {testo.Suite[T] }

func (Bakery) TestBake(t T) {
    medovik, ok := Bake("медовик")
    // ... тот же сценарий
}

func Test(t *testing.T) {
    testo.RunSuite(t, new(Bakery))
}
```

- [T] — какой набор плагинов используют тесты этого Suite
- Тесты — методы вида `TestXxx`
- Запуск — обычной тестовой функцией: `go test` работает
- Отдельный тест из Suite запускает расширение для VS Code

Suites: группировка и общие хуки

```go
type Bakery struct {testo.Suite[T] }

func (Bakery) TestBake(t T) {
    medovik, ok := Bake("медовик")
    // ... тот же сценарий
}

func Test(t *testing.T) {
    testo.RunSuite(t, new(Bakery))
}
```

- [T] — какой набор плагинов используют тесты этого Suite
- Тесты — методы вида `TestXxx`
- Запуск — обычной тестовой функцией: `go test` работает
- Отдельный тест из Suite запускает расширение для VS Code

Параметризация: медовик — это параметр

В Pytest: `@pytest.mark.parametrize`

```go
func (Bakery) CasesDessert() []string {
    return []string{"медовик", "тирамису"}
}

func (Bakery) TestBake(t T, p struct{ Dessert string }) {
    baked, ok := Bake(p.Dessert)
    if !ok {
        t.Fatalf("должны уметь печь %s", p.Dessert)
    }

    t.Cleanup(func() { Eat(baked) })

    if !baked.Tasty {
        t.Errorf("%s должен получиться вкусным", p.Dessert)
    }
}
```

Правило: Cases плюс имя параметра

- *Несколько параметров — все комбинации значений, из коробки*
- *В отчёте теперь два теста: медовик и тирамису, у каждого свои параметры*

Фикстуры — это просто методы

```
func (t T) Bake(name string) (pastry Pastry, ok bool) {
    pastry, ok = Bake(name)

    if ok {
        t.Cleanup(func() { Eat(pastry) })
    }

    return pastry, ok
}
```

В тесте — маленький diff:

```
-baked, ok := Bake(p.Dessert)
+baked, ok := t.Bake(p.Dessert)
-t.Cleanup(func() { Eat(baked) })
```

Нужна фикстура на весь Suite — ресурс создаётся в BeforeAll и живёт в поле Suite-структуры.

В Pytest: `@pytest.fixture`

Хуки: кто-то выключил печь

Тест стабильно работал, пока однажды не упало вообще всё.
Правильный тест сам готовит окружение

```
func (Bakery) BeforeAll(t T) {
    if err := Oven(true); err != nil {
        t.Fatalf("Не удалось включить печь: %v", err)
    }
}

func (Bakery) AfterAll(t T) {
    if err := Oven(false); err != nil {
        t.Errorf("Не удалось выключить печь: %v", err)
    }
}
```

BeforeAll

BeforeEach

AfterEach

AfterAll

В Pytest: `class-scoped fixture`

Шаги и проверки: что тест делал внутри

Оборачиваем фикстуру в подтест — она становится шагом

```
func (t T) Bake(name string) (p Pastry, ok bool) {
    testo.Run(t, "bake: "+name, func(t T) {
        p, ok = Bake(name)
    })

    if ok {
        t.Cleanup(func() { Eat(p) })
    }
    return p, ok
}
```

Проверки из testify — каждая уже обёрнута в шаг

```
func (Bakery) TestBake(t T, p ...) {
    baked, ok := t.Bake(p.Dessert)
    t.Require().True(ok, "должны уметь печь")
    t.Attach("ингредиенты", ...)

    t.Assert().True(baked.Tasty, "вкусный")
}
```

Require — упало и остановились.
Assert — зафиксировали ошибку и едем дальше

Итог

Отчёты | Плагины | Suites | Параметры | Фикстуры | Хуки | Шаги

Было: testing

```
func TestBake(t *testing.T) {
    m, ok := Bake("медовик")
    if !ok {
        t.Fatal("...")
    }
    t.Cleanup(func() { Eat(m) })
    if !m.Tasty {
        t.Error("...")
    }
}
```

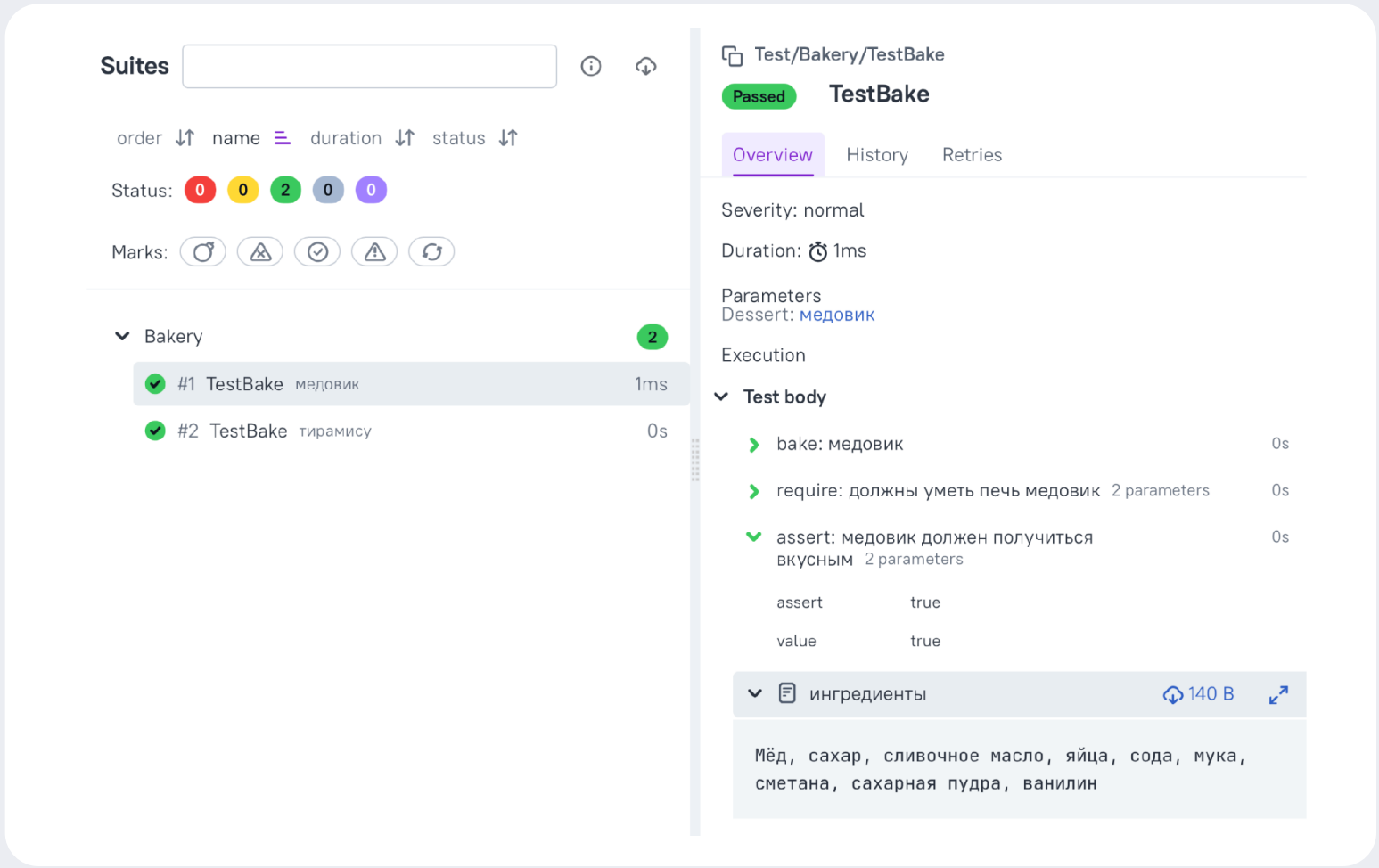
Лог в консоли

Стало: Testo

```
func (Bakery) TestBake(
    t T,
    p struct{ Dessert string },
) {
    b, ok := t.Bake(p.Dessert)
    t.Require().True(ok, "...")
    t.Assert().True(b.Tasty, "...")
}
```

Allure-отчёт, шаги, параметры,
параллельность, хуки, фикстуры

Готовый отчёт: шаги, параметры, структура



Сценарий теста стал наглядным и обрёл детали

- Параметры проверок видны прямо в шаге, ингредиенты — во вложении
- **Любой разбор падения начинается с этого экрана**

Фреймворк, отчёты и плагины

Вместе с фреймворком мы выложили плагины, которыми пользуемся сами

testo

Фреймворк: ~4 000 строк,
ноль зависимостей.
Плагинов внутри нет

testo-allure

Полный функционал Allure:
шаги, вложения, параметры,
проверки

testo-toppings

`Retry` — перезапуск
только упавших

`xfail` — ожидаемо падающие
тесты

`parallel` — все тесты
параллельны

`async` — ожидание
нескольких проверок

Плюс **расширение для VS Code**:
запуск и отладка отдельного теста из Suite кнопкой

В Pytest: `--last-failed` · `xfail`

04

Зачем open source

Документация, совместимость и работа с сообществом

Что получает компания

Чистая архитектура

Open source не оставляет выбора:
всё внутреннее обязано быть плагином,
ядро остаётся чистым

Обратная связь

Публичен с мая 2026-го: внешние
контрибьюторы приносят сценарии,
с которыми внутри не сталкиваемся

Что получает сообщество

Фреймворк, проверенный на 130 000 тестах и 860 сервисах

4 000
строк в ядре

0
зависимостей

5+
готовых плагинов

3
последних релиза Go

Плагинная модель:
дорабатывать под себя можно, не форкая проект

Лицензия Apache-2.0

Что стоит забрать с собой

01 Go в QA — это всерьёз: типизация, единый стек и почти бесплатная параллельность на горутинах

02 Из коробки Go не хватает инфраструктуры для больших e2e — мы закрыли это Testo

03 Ядро плюс плагины: расширяемость как в pytest, ощущение — как от go test

04 Open source помогает проекту становиться лучше

Спасибо!
Вопросы?



Вадим Збаращенко
QA-платформа, Ozon

https://ozon.tech/testo