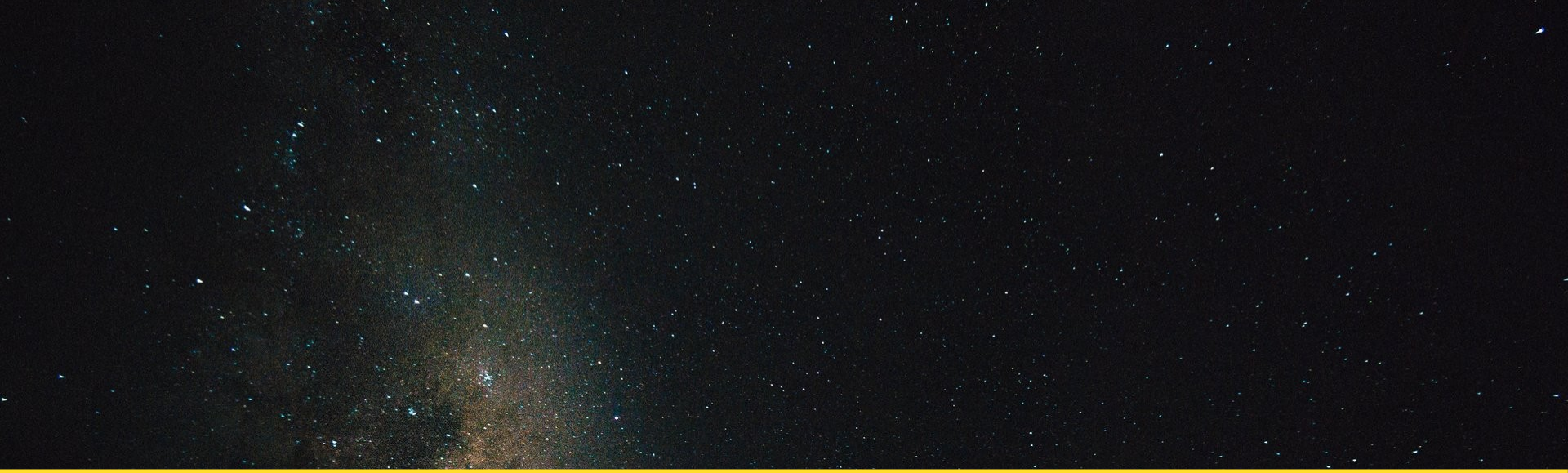
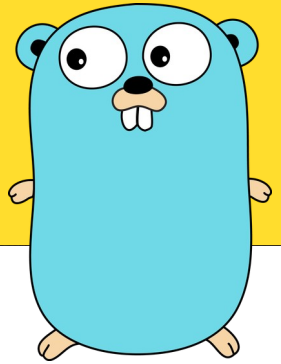


Тестирование



Tinkoff.ru

План занятия

1. Тестирование: что это и зачем
2. Unit-тестирование и тестирование HTTP-сервера
3. Benchmark-тестирование и профилирование
4. CI/CD

Тестирование

Тестирование

Тестирование – это наблюдение за функционированием ПО в специфических условиях с целью определения соответствия ПО требованиям к нему.

Зачем нужно тестирование для разработчика?

1. Писать более качественный код

2. Понимать специалистов по тестированию

3. Выпускать качественное ПО с минимальным участием тестировщиков

Терминология

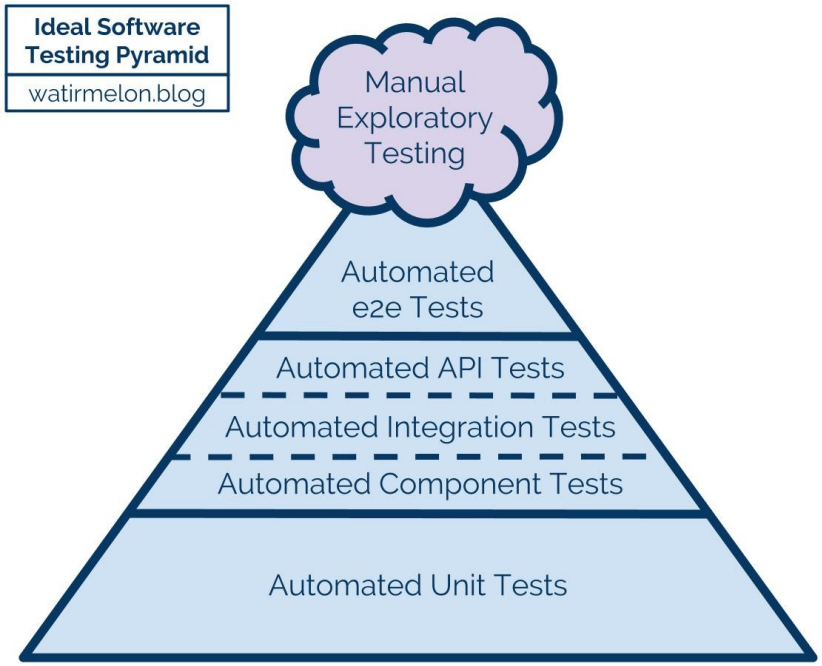
Actual – фактически результат

Expected – ожидаемый результат

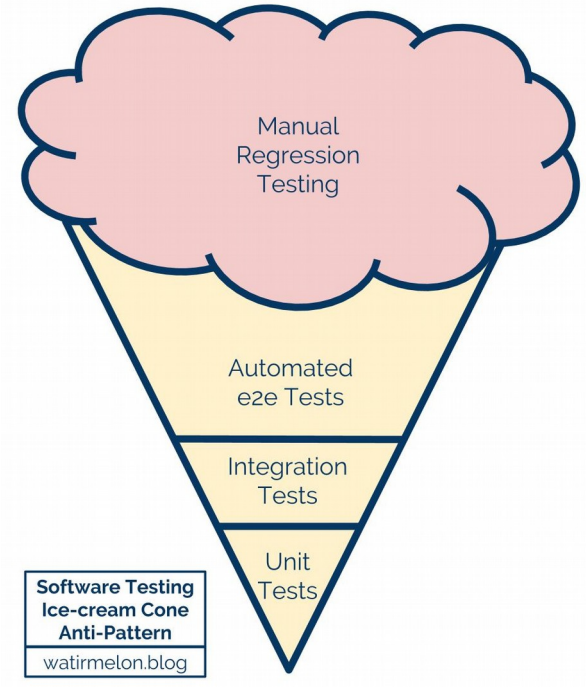
«Тест упал» – фактический результат выполнения не соответствует ожидаемому

QA – quality assurance. Часто так называют специалистов по обеспечению качества (тестировщиков)

Пирамида тестирования



Пирамида тестирования



Рожок тестирования

TDD

TDD (англ. test-driven development) – разработка через тестирование.

Суть: сначала пишется тест, а потом код, удовлетворяющий тестам. На практике очень хорошо подходит для небольших программ. Для более сложных требуется больше усилий в проектировании.

Unit-тестирование в Go

Unit-тестирование в Go

Есть специальный пакет `testing`.

Особенности:

- Тесты и код лежат в одном пакете
- Имя файла имеет вид `*_test.go`
- Имя функции начинается с `Test` и принимает `*testing.T`
- Для запуска используется команда `go test`

Пишем тесты

```go
package primes

import "math"

func IsPrimeNumber(number int) bool {
    if number <= 0 {
        return false
    }

    max := int(math.Sqrt(float64(number)))
    for i := 2; i < max; i++ {
        if number%i == 0 {
            return false
        }
    }

    return true
}
```

Простейший тест

```go
package primes

func TestIsPrime(t *testing.T) {
   if !IsPrimeNumber(7) {
      t.Errorf("7 is prime number")
   }
}

```

```
$ go test -v
=== RUN   TestIsPrime
--- PASS: TestIsPrime (0.00s)
PASS
```

Если тест не пройдёт?

```
=== RUN   TestIsPrime
--- FAIL: TestIsPrime (0.00s)
    primes_test.go:10: 7 is prime number
FAIL
```

Сообщения об ошибках

Общая идея схожа с логированием: по сообщению должно быть понятна причина «падения» теста.

Плохо

```
$ go test -v
=== RUN TestIsPrime
--- FAIL: TestIsPrime (0.00s)
prime_test.go:9: Wrong answer
FAIL
```

Сообщения об ошибках

Один из вариантов: записывать в сообщение имя функции, аргументы, фактический и ожидаемый вариант.

```
{func}({args}) = {actual}, want = {expected}
```

```
$ go test -v
=== RUN TestIsPrime
--- FAIL: TestIsPrime (0.00s)
prime_test.go:10: IsPrime(3) = false, want true
FAIL
```

Что если тестов много? – табличные тесты

```
1  func TestIsPrime(t *testing.T) {
2     type testCase struct {
3        Name     string
4        In       int
5        Expected bool
6     }
7
8     testCases := []testCase{
9        {Name: "Simple number", In: 7, Expected: true},
10       {Name: "Zero value", In: 0, Expected: false},
11       {Name: "Negative value", In: -100, Expected: false},
12       {Name: "Non simple number", In: 100, Expected: false},
13    }
14
15
16    // ...
```

Табличные тесты

```
15      // ...
16
17      for _, tc := range testCases {
18          t.Run(tc.Name, func(t *testing.T) {
19              actual := IsPrimeNumber(tc.In)
20              if actual != tc.Expected {
21                  t.Errorf("IsPrime(%d) = %+v, want %+v", tc.In, actual,
    tc.Expected)
22              }
23          })
24      }
25  }
26
```

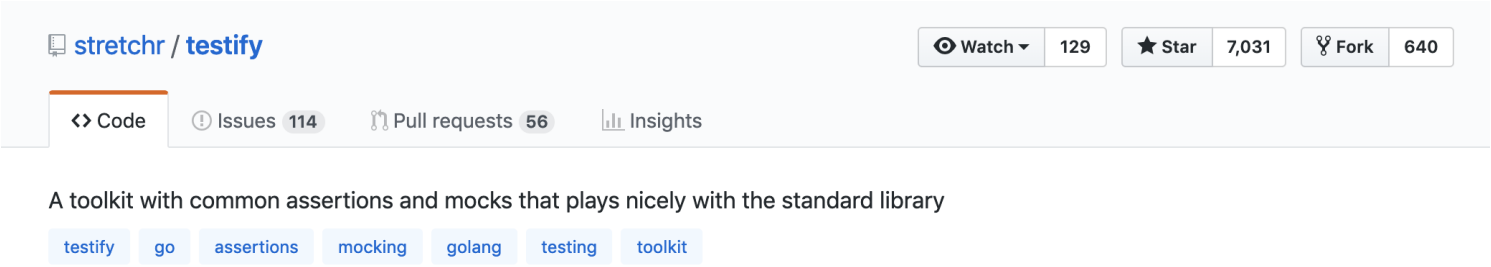
Запускаем

```
$ go test -v
=== RUN   TestIsPrime
=== RUN   TestIsPrime/Simple_number
=== RUN   TestIsPrime/Zero_value
=== RUN   TestIsPrime/Negative_value
=== RUN   TestIsPrime/Non_simple_number
--- PASS: TestIsPrime (0.00s)
    --- PASS: TestIsPrime/Simple_number (0.00s)
    --- PASS: TestIsPrime/Zero_value (0.00s)
    --- PASS: TestIsPrime/Negative_value (0.00s)
    --- PASS: TestIsPrime/Non_simple_number (0.00s)
PASS
```

Интерфейс TB

```
1  // TB is the interface common to T and B.
2  type TB interface {
3     Error(args ...interface{})
4     Errorf(format string, args ...interface{})
5     Fail()
6     FailNow()
7     Failed() bool
8     Fatal(args ...interface{})
9     Fatalf(format string, args ...interface{})
10    Log(args ...interface{})
11    Logf(format string, args ...interface{})
12    Name() string
13    Skip(args ...interface{})
14    SkipNow()
15    Skipf(format string, args ...interface{})
16    Skipped() bool
17    ...
18 }
```

Testify



https://github.com/stretchr/testify библиотека для упрощения тестирования. Имеет методы сравнения (Equal), проверки ошибок, проверки длины, поиска подстроки и т.д.

```
1  r := require.New(t)
2  r.Equal(expected, actual)
3  r.True(IsPrimeNumber(7))
4  err := doSomething()
5  r.NoError(err)
```

Тестирование HTTP-сервера

Реализуется посредством unit-тестов.

Есть вспомогательный пакет httptest, который может:

1. Запустить тестовый сервер (в том числе TLS)
2. Имеет тестовый рекордер (реализует интерфейс `http.ResponseWriter`)
3. Создать тестовый запрос (`http.Request`)

Пример с тестовым сервером

Тестовое покрытие

```
go test -v -coverprofile=coverage.out -
covermode=count
```

Анализ

```
go tool cover -func=coverage.out
```

```
go tool cover -html=coverage.out
```

Пример с тестовым сервером

Benchmark-тестирование и профилирование

Benchmark-тестирование в Go

Пакет `testing`.

Особенности:

- Тесты и код лежат в одном пакете
- Имя файла имеет вид `*_test.go`
- Имя функции начинается с `Benchmark` и принимает `*testing.B`
- Хитрый запуск через `go test –v –bench .`

СМОТРИМ КОД

Профилирование: утилита pprof

Пакет `pprof` позволяет собирать метрики с приложения (cpu, memory, goroutine, block, mutex).

При профилировании CPU ~100 раз/сек собирается стектрейс горутин. Все данные записываются в специальный файл, который можно прочитать с помощью утилиты **pprof**.

Профилирование CPU приложения

```
1  func main() {
2     var cpuProfile string
3     flag.StringVar(&cpuProfile, "cpuprofile", "", "Write CPU profile to
   `file`")
4     flag.Parse()
5
6     if cpuProfile != "" {
7        f, err := os.Create(cpuProfile)
8        if err != nil {
9           log.Fatal("Can't create CPU profile:", err)
10       }
11       if err := pprof.StartCPUProfile(f); err != nil {
12          log.Fatal("Can't start CPU profile:", err)
13       }
14       defer pprof.StopCPUProfile()
15    }
16    // ...
17 }
```

Теперь память

```
1  func main() {
2     var memProfile string
3     flag.StringVar(&memProfile, "memprofile", "", "Write memory profile to
   `file`")
4     flag.Parse()
5
6     if memProfile != "" {
7        f, err := os.Create(memProfile)
8        if err != nil {
9           log.Fatal("Can't create memory profile:", err)
10       }
11       runtime.GC() // get up-to-date statistics
12       if err := pprof.WriteHeapProfile(f); err != nil {
13          log.Fatal("Can't start CPU profile:", err)
14       }
15    }
16    // ...
17 }
```

Что насчёт HTTP-сервера?

```
1  import _ "net/http/pprof"
```

Под капотом

```
   // $GOROOT/src/net/http/pprof/pprof.go
72 func init() {
73    http.HandleFunc("/debug/pprof/", Index)
74    http.HandleFunc("/debug/pprof/cmdline", Cmdline)
75    http.HandleFunc("/debug/pprof/profile", Profile)
76    http.HandleFunc("/debug/pprof/symbol", Symbol)
77    http.HandleFunc("/debug/pprof/trace", Trace)
78 }
```

Примеры запуска

```
go tool pprof http://localhost:8080/debug/pprof/profile?seconds=30
```

```
go test -v -bench pattern -run=^$ -o test.out -memprofile=mem.out .
```

– запустить только benchmark тесты по регулярному выражению (pattern), собрать бинарный файл в test.out, профилировать память.

Классическая задача: конкатенация строк

```
1  const testStr = "Hello!"
2  func BenchmarkStringsConcat(b *testing.B) {
3     var result string
4     for i := 0; i < b.N; i++ { result += testStr }
5  }
6
7  func BenchmarkStringsBytesBuffer(b *testing.B) {
8     buf := &bytes.Buffer{}
9     for i := 0; i < b.N; i++ { buf.WriteString(testStr) }
10 }
11
12 func BenchmarkStringsBuilder(b *testing.B) {
13    builder := &strings.Builder{}
14    for i := 0; i < b.N; i++ {
15       builder.WriteString(testStr)
16    }
17 }
```

Конкатенация строк

```
$ go test -v -bench Strings -run=^$ -o test.out -memprofile=mem.out .
goos: darwin
goarch: amd64
pkg: gitlab.com/vadimlarionov/fintech-golang/code/testing/primes
BenchmarkStringsConcat-8              500000      172470 ns/op
BenchmarkStringsBytesBuffer-8      100000000          22.5 ns/op
BenchmarkStringsBuilder-8          100000000          13.0 ns/op
PASS
ok   gitlab.com/vadimlarionov/fintech-golang/code/testing/primes90.308s

$ go tool pprof test.out mem.out
$ go tool pprof test.out cpu.out
```

Основные команды для pprof (command line)

`top` – top 10 наиболее «тяжёлых» функций

`top -N` – top N наиболее «тяжёлых» функций

`list {pattern}` – вывести тело функции на экран

`web` – вывести svg-визуализацию в браузере (требуется graphviz)

`help` – список команд

Что такое `flat` и `cum` в результатах pprof?

`flat` – продолжительность (время, %, память), затраченная на выполнение функции.

`cum` – куммулятивная (суммарная) продолжительность с учётом вызовов внутренних функций.

Reddit

Параметры для профилирования памяти

`-inuse_space`	Используемое кол-во памяти
`-inuse_objects`	Используемое количество объектов
`-alloc_space`	Выделенное кол-во памяти
`-alloc_objects`	Выделенное кол-во объектов (в хипе)

```
$ go tool pprof {binary file} {profile file, ex mem.out/cpu.out}
$ go tool pprof –alloc_space {binary file} {mem.out}
```

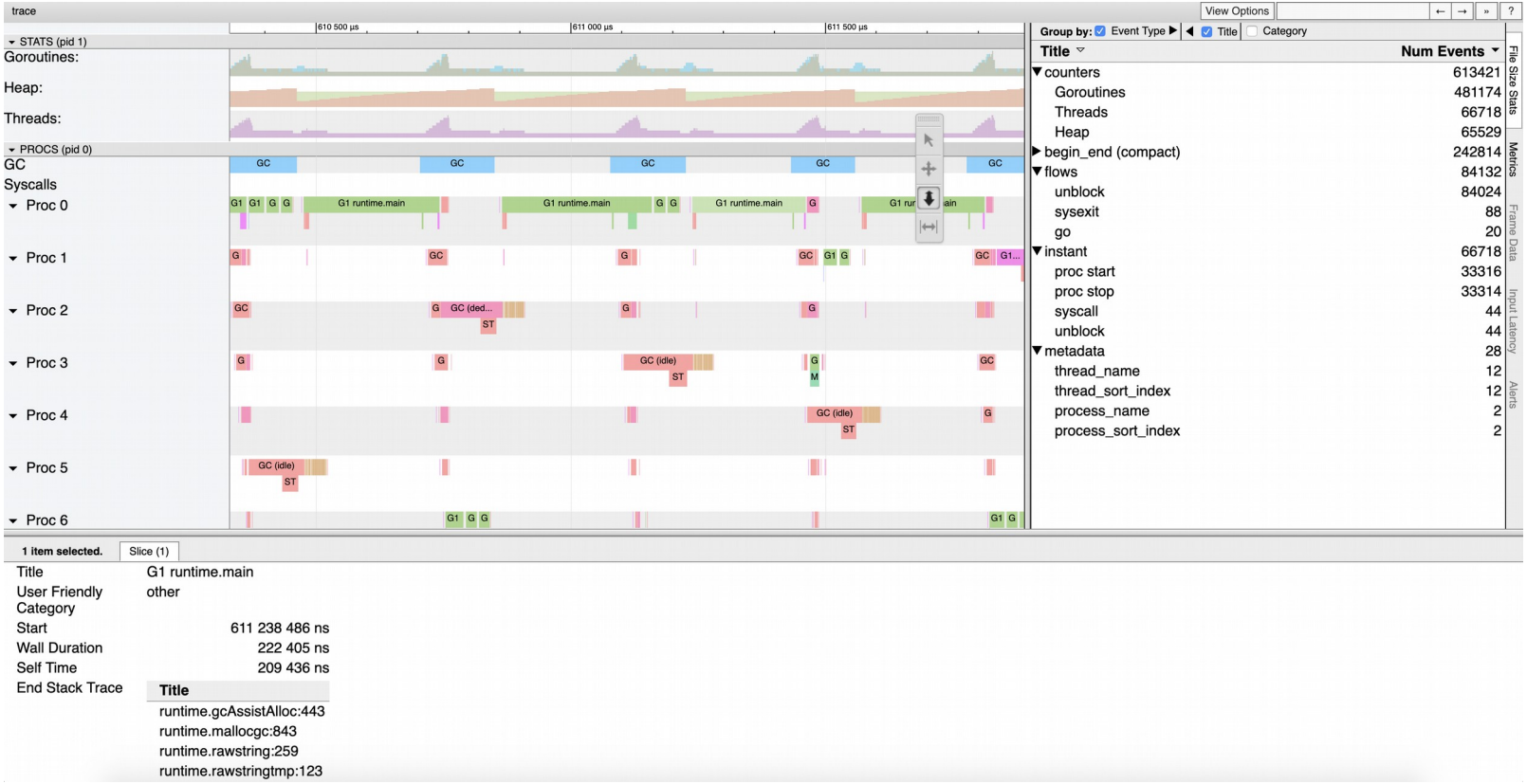
Трассировка приложений

Трассировка позволяет отследить порядок выполнения программы ядрах (горутинах), работу GC, context switch.

По-умолчанию сбор – 1с. Можно трассировать как приложения, так и http-сервер.

```
$ go tool trace trace.out # Трассировка по файлу trace.out
$ wget http://localhost:8080/debug/pprof/trace?seconds=1 # Трассировка
http-сервера
```

Трассировка приложений



CI/CD

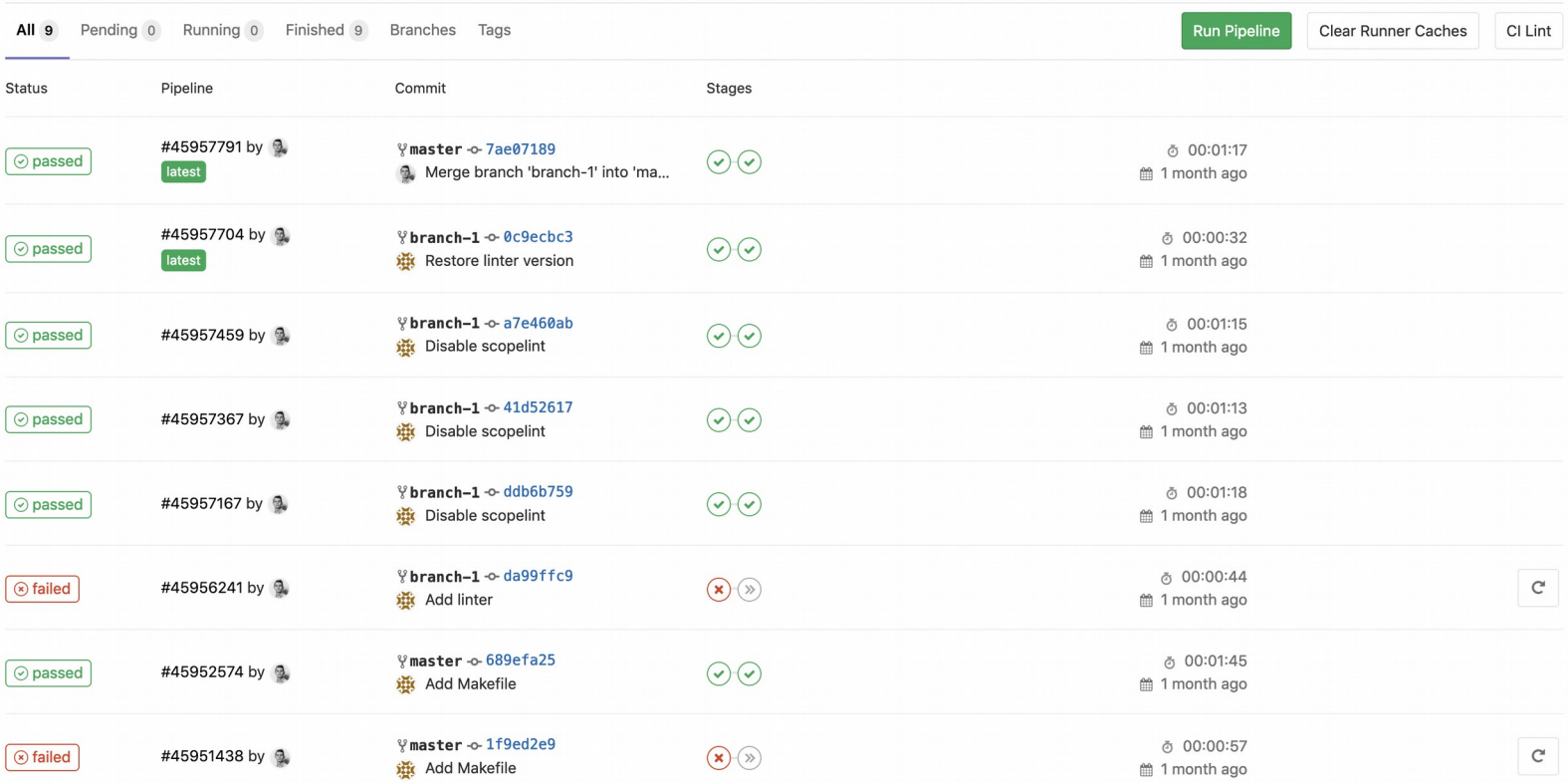
CI

Continuous Integration (CI) – практика разработки ПО, заключающаяся в постоянном слиянии рабочих веток в основную и выполнение частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

Обычно включает шаги: получение исходного кода, прогон линтеров, сборка, тестирование, развёртывание в dev-среду, формирование отчётов.

Продукты: Travis, Jenkins, Gitlab.

Gitlab CI


All 9
Pending 0
Running 0
Finished 9
Branches
Tags
Run Pipeline
Clear Runner Caches
CI Lint
Status
Pipeline
Commit
Stages
passed
#45957791 by
latest
master 7ae07189
Merge branch 'branch-1' into 'ma...
00:01:17
1 month ago
passed
#45957704 by
latest
branch-1 0c9ecbc3
Restore linter version
00:00:32
1 month ago
passed
#45957459 by
branch-1 a7e460ab
Disable scopelint
00:01:15
1 month ago
passed
#45957367 by
branch-1 41d52617
Disable scopelint
00:01:13
1 month ago
passed
#45957167 by
branch-1 ddb6b759
Disable scopelint
00:01:18
1 month ago
failed
#45956241 by
branch-1 da99ffc9
Add linter
00:00:44
1 month ago
passed
#45952574 by
master 689efa25
Add Makefile
00:01:45
1 month ago
failed
#45951438 by
master 1f9ed2e9
Add Makefile
00:00:57
1 month ago

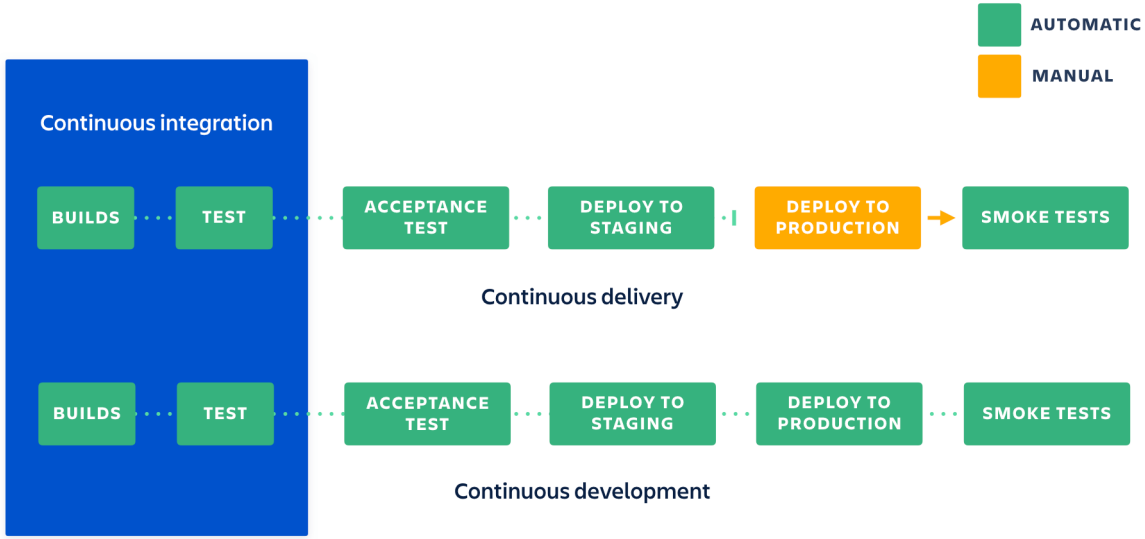
Gitlab CI

В репозитории создаём файл .gitlab-ci.yml

```yaml
image: golangci/golangci-lint:v1.15.0
stages:
  - lint-and-test
  - print-success
test:
  stage: lint-and-test
  script:
    - make test
lint:
  stage: lint-and-test
  script:
    - make lint
print-success:
  stage: print-success
  script:
    - echo "Success"
```

CD

Continuous Delivery (CD) – CI с доставкой до production по кнопке.



Процесс разработки

1. Пишем код по задаче из таск-трекера в отдельной ветке, с именем `{номер задачи}-{краткое описание}`
2. Создаём MR на основную ветку (master)
3. В MR запускается CI (линтеры, тесты)
4. Ревью кода другими разработчиками
5. Исправление замечаний, повторное ревью, …
6. `git merge --squash ...` – объединит все коммиты в 1 и выполнит merge
7. Раскатка на dev/test/stage

Обратная связь

Спасибо за внимание

Tinkoff.ru