test fast: a practical guide to a livable test suite

A software article by Efron Amber Licht

August 2023

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

1. Introduction: what's wrong with my tests?

Imagine this: it's 2:03 PM on a friday afternoon in the office. Your new code is working great on your local machine. Once the deploy finishes, you'll be able to go home for the weekend.

- 2:04PM: Patch submitted.
- 2:05PM: Unit tests start.
- 2:38PM: Unit tests end: OK.
- 2:39PM: Integration tests start.
- 2:55PM: Integration tests end: known flaky test fails.

OK, not too bad. You slam 're-run' on the test suite and go get a coffee. Surely it'll pass this time.

- 2:56PM: Unit tests start.
- 3:29PM: Unit tests end: OK.
- 3:30PM: Integration tests start.
- 4:32PM: Integration tests end: OK.
- 4:33PM: End-to-end tests start.

- 5:01PM: End-to-end tests end: OK.
- 5:02PM: Deploy to production starts.
- 5:28PM: Deploy to production ends: Turns out you should have spelled `DBUSER` as `DB_USER`.
- 5:29PM: Patch submitted…

Total time: 3 hours, 25 minutes.

Total time spent programming: 1 minute.

Productivity: %0.005.

Your average software project contains hundreds of tests that run before each and every deploy. While this has a number of important benefits, such as

- finding bugs before they hit production
- guarding against potential regressions introduced by new features or other bugfixes
- acting as a "living" set of documentation and examples for the codebase
- giving a nice warm fuzzy feeling to engineers & managers

Tests are not a universal panacea: slow or unreliable tests can cause more damage than they prevent.
A healthy test suite is about far more than coverage: useful tests are *fast*, *reliable*, and *deterministic*. In this article, we'll cover

- Why speed & consistency so crucial to a healthy test suite
- What 'good tests' look like and how to build them.
- Dependency management in tests
- A short-term 'rescue plan' for a test suite that's deeply underwater.

This article is a loose sequel to some of my other posts on infrastructure and speed. I'll link to them where appropriate, but you don't need to read them to understand this article.

2. Why do speed & determinism matter so much?

Programming work varies wildly in scope. Some programming work is the exciting kind where you come up with clever algorithms, pack bits, or implement clever new features. More programming work is 'routine': fixing spelling errors, updating configuration, adding an extra URL parameter to a HTTP request - the kind of task that *should* take thirty seconds. The most productive programming consists of quick iteration on problems of this kind: make a change, test them, rinse, repeat. Only one problem: for the vast majority of professional codebases, the software test suite takes minutes or hours to run. By the time the test suite finishes, the programmer has been called into a meeting, pulled away by another bug, been out to lunch, or simply lost their train of thought. Either way, what should have been a 'quick fix' drags into the next day, or week, or month. ***Slow tests are an enormous productivity killer, a hidden demon sabotaging every attempt to build quality software.***

There is nothing quite so damaging to test infrastructure as a flaky test: a test that *sometimes* passes and sometimes fails. Software Engineers universally develop a defense mechanism against flaky tests: they ignore them. Now, remember, the entire purpose of a test suite is to tell you that your code needs to be fixed; **but since flaky tests are not a reliable indicator, they train developers to treat failures as noise, not signal.** Over time, as the flaky tests build up, developers assume *all* failures are flaky, and they start simply ignoring *all* failures and over-riding the safeguards that prevent broken code from being deployed. This concept, called "alarm fatigue", is well known in a variety of other fields. Disasters from plane crashes to exploding oil rigs to the famous meltdown at the chernobyl nuclear power plant were all, in part, due to alarm fatigue.

It's hard to overstate how demoralizing this can be to a Software Engineer who is bursting w/ good ideas about where and how to fix the problem. Some of the worst nights of programmers' lives involve convincing themselves they'll stick around "just until they fix this simple bug", only to find themselves still at the office at 1am, exhausted, waiting for yet another agoniizingly slow test run to finish. The product managers have to explain to the execs how a seemingly 'trivial' bug took days or weeks to fix. The exhausted programmers are then berated for their lack of productivity, or worse yet, their lack of 'ownership' of the problem, when they've run themselves ragged trying to fix it. This tendency to blame individuals rather than systems for problems is well-known in organizational psychology. But because **infrastructure is all-too-often effictively invisible to leadership, it rarely gets prioritized**, even when it's the root cause of what leadership sees as a 'productivity' problem. Often times, the leadership is too far away to see the problem, and the junior engineers who are most affected by it are too 'close' to the problem to be able to really see it. They know something's wrong - everything takes too

long, everything is too hard - but they don't know *why* - and even if they do, they have no idea how to fix it.

And in an attempt to regain the lost speed and regain the trust of leadership & product management, programmers will start cutting *more* corners: ignoring *more* tests, and over-riding *more* safeguards. It's a death spiral through good intentions.

But it's one you can prevent with fast and reliable tests.

3. What do 'good tests' look like?

Good tests are always:

- understandable
- deterministic

The best tests are also:

- fast
- parallelizable.
- dependency free

3.1. Defining our criteria

- #### Understandable

A test is **understandable** if you can read a failure and understand what went wrong, without having to know underlying implementation details. This usually just means being careful with failure messages.
A template like "funcname(arg1, arg2): got <something>, want <somethingelse>" is usually fine. Cryptic failures that just said something like "assertion failed", with (or worse, without) accompanying stack traces are not.

No need to get fancy: this is fine:

```
FAIL: TestAdd/2+3=5 (0.00s)
Add(2, 3) = 6, want 5
```

- #### Deterministic

A test is **deterministic** if when it fails, it always fails.

- #### Fast

A test is **fast** if it takes `<=1ms` to run on the slowest developer machine anyone uses. Parallelizable tests are allowed a little leeway - say `<10ms`. This may seem extreme, but

larger projects may have tens of thousands of tests. If even a hundred of these take `100ms`, you're talking about ten seconds to run.

- #### Parallelizable

A test is **parallelizable** if it can be run in any order or simultaneously with any other test, including multiple copies of itself in other binaries. Weaker forms of concurrency are such as the following are not as good, but still better than nothing:

- "at the same time as other tests *on other machines*"
- "if nothing else is touching the environment variables"
- "not at the same time as another copy of itself"
- "not at the same time as other tests in this package"
- "not at the same time as other tests that touch `postgres`"
- "not at the same time as this one other test"
- "so long as port 8080 is free"
- **Dependency free**

A test is **dependency free** if it doesn't communicate with anything outside the Go runtime. This includes but is not limited to interaction with:

- command line arguments
- environment variables (reading or writing)
- file I/O (including stdin, stdout, stderr)
- foreign function calls of any kind
- mouse, keyboard, or other input devicesz
- network I/O
- random number generation
- syscalls
- system clock (`time.Now()`, `time.Sleep()`, etc)
-

unit tests

Are **understandable**, **deterministic**, **fast**, preferably **parellizable**, and **dependency-free**. We should be able to run these immediately, all the time, and the entire unit test suite should run in `<3s`, preferably `<1s`.

- #### **integration tests**

integration tests are **understandable** and **deterministic** and make best-effort attempts at the other three.

- #### **end-to-end** tests

… aren't the subject of this article.

OK. enough table-setting.

4. Building good tests

Build good tests by:

- making them deterministic
- segregating slow or dependency-heavy tests from unit tests
- running tests in parallel
- optimizing runtime & initialization time
- injecting dependencies to avoid I/O and allocation
- using profilers to understand slow tests by converting them to benchmarks

While all of these techniques apply to any programming language, all examples will be in Go.

4.0.1. Making tests deterministic

Make tests deterministic by removing all sources of nondeterminism. This is basically the list of dependencies above, plus global variables and conccurrency. If a test continues to be flaky, skip it with an error message that explains why as best you can, and try and write a better test (or better code!) as soon as you can.

You don't want to pretend the problem doesn't exist, but it's even worse to have the flaky test clogging up your deployment pipeline while providing no value.

Skipping is deterministic, if unsatisfying.

```
1  func TestFlaky(t *testing.T) {
2
   t.Skipf("SKIP %s: occasionally fails during integration:
   port issue?", t.Name())
```

```
3       if rand.Int() == 666 {
4           t.Fatal("work of the test devil")
5       }
6   }
```

Once an alternative has been discovered, or if more than a couple weeks go by, *delete* the flaky test. At a certain point, it's just going to confuse people.

4.0.2. Running tests in parallel

Tests should be as parallel as possible. The go tool has some level of parallelism by default: when using `go test` in package mode: (i.e., `go test ./...`), go will build a test binary for each package with tests, and run those binaries in parallel.

However, tests *within* a package are run serially by default. Individual tests and their subtests can opt-in to parallelism by calling `t.Parallel()`. Tests will be run serially *up to* the first call to `t.Parallel()`, and then in parallel after that. Execution of a subtest will be serial with regards to its parent test unless both the parent and the subtest call `t.Parallel()`. In practice, this just means **you need to call t.Parallel() once per layer of test**.

Let's demonstrate with a few examples:

```
1   // TestMul runs serially: no other tests in this package
    will run while it is running,
2   // but it may run in parallel with tests in other packages
3   func TestMul(t *testing.T) {
4       if 5*2 != 10 {
5           t.Fatal("5*2 != 10")
6       }
7   }
8
9   // TestAdd runs in parallel with TestSub, and it's subtests
    will run in parallel both with each other and TestSub.
10  func TestAdd(t *testing.T) {
11      t.Parallel() // Add will run in parallel with other
    tests in this package from this point on
12
13      for _, tt := range []struct{
14          a, b, want int
15      } {
16          {2, 2, 4},
17          {3, 3, 6},
18          {-128, 128, 0},
19      } {
20          tt := tt // capture range variable: see https://
    github.com/golang/go/discussions/56010 for details
21          t.Run(fmt.Sprintf("%d+%d=%d", tt.a, tt.b,
    tt.want), func(t *testing.T) {
22              t.Parallel() // this subtest will run in
    parallel with other subtests of TestAdd
```

```
23                got := tt.a + tt.b
24                if got != tt.want {
25                    t.Errorf("Add(%d, %d) = %d, want %d",
    tt.a, tt.b, got, tt.want)
26                }
27            })
28        }
29    }
30    // TestSub will run in parallel with other tests in this
    package,
31    // but only one of its subtests will run at a time
32    func TestSub(t *testing.T) {
33        t.Parallel()
34
35        for _, tt := range []struct{
36            a, b, want int
37        } {
38            {2, 2, 0},
39            {2, -2, 4},
40        } {
41            t.Run(fmt.Sprintf("%d-%d=%d", tt.a, tt.b,
    tt.want), func(t *testing.T) {
42                got := tt.a - tt.b
43                if got != tt.want {
44                    t.Errorf("Add(%d, %d) = %d, want %d",
    tt.a, tt.b, got, tt.want)
45                }
46            })
47        }
48
49    }
```

We run the tests:

IN

```
1   go test -v ./...
```

```
1
2   OUT:
3   === RUN   TestWriteFile
4   --- PASS: TestWriteFile (0.00s)
5   === RUN   TestMul
6   --- PASS: TestMul (0.00s)
7   === RUN   TestAdd
8   === PAUSE TestAdd
9   === RUN   TestSub
10  === PAUSE TestSub
11  === CONT  TestAdd
12  === RUN   TestAdd/2+2=4
13  === PAUSE TestAdd/2+2=4
14  === RUN   TestAdd/3+3=6
15  === PAUSE TestAdd/3+3=6
16  === CONT  TestSub
```

```
=== RUN   TestAdd/-128+128=0
=== RUN   TestSub/2-2=0
=== PAUSE TestAdd/-128+128=0
=== CONT  TestAdd/2+2=4
=== RUN   TestSub/2--2=4
=== CONT  TestAdd/3+3=6
--- PASS: TestSub (0.00s)
    --- PASS: TestSub/2-2=0 (0.00s)
    --- PASS: TestSub/2--2=4 (0.00s)
=== CONT  TestAdd/-128+128=0
--- PASS: TestAdd (0.00s)
    --- PASS: TestAdd/2+2=4 (0.00s)
    --- PASS: TestAdd/3+3=6 (0.00s)
    --- PASS: TestAdd/-128+128=0 (0.00s)
PASS
ok   gitlab.com/efronlicht/blog/articles/testfast (cached)
```

'PAUSE' is the surefire sign of a parallel test. We can use `grep` to see exactly which tests are running in parallel, and just as we expected, it's `TestAdd` and it's subtests, and `TestSub`, but NOT it's subtests.

```
go test -v ./... | grep PAUSE
```

OUT

```
=== PAUSE TestAdd
=== PAUSE TestSub
=== PAUSE TestAdd/2+2=4
=== PAUSE TestAdd/3+3=6
=== PAUSE TestAdd/-128+128=0
```

I strongly advise you to design all tests to be as parallel as possible by default. In short, they should inject their dependencies rather then relying on synchronization with outside state (like globals), timing (like sleeps), or I/O (network calls, reading and writing files, syscalls). Let's go over how we can avoid those things for fast, parallelizable, and reliable tests.

4.0.3. Segregating slow or dependency-heavy tests from unit tests

You shouldn't need to have access to every dependency to run tests that don't need them. Likewise, if you have *some* fast test, you shouldn't need to wait for the slow ones in order to get feedback. Go provides a built-in mechanism for this: the `testing.Short()` flag. If you run `go test -short`, go will set this flag to true. You can then use this flag to skip slow tests:

```
func TestSlow(t *testing.T) {
    if testing.Short() {
        t.Skipf("SKIP %s: slow", t.Name())
    }
    // ... slow test code here
}
```

```
 7
 8  func TestGetUsers(t *testing.T) {
 9      if testing.Short() {
10          t.Skip("SKIP %s: touches postgres", t.Name())
11      }
12      res, err := postgres.DB().Query("...")
13      // ... database test code here
14  }
```

```
1  go test -short -v ./...
```

OUTPUT

```
1  === RUN   TestSlow
2      testfast_test.go:47: SKIP TestSlow: slow
3  --- SKIP: TestSlow (0.00s)
4  === RUN   TestGetUsers
5      testfast_test.go:54: SKIP TestGetUsers: touches postgres
6  --- SKIP: TestGetUsers (0.00s)
```

You can also use your own criteria: there's nothing special about `testing.Short()`, it's just a wrapper around a standard command-line flag. You can easily add your own or use environment variables:

```
1  var skipPG = flag.Bool("skippg", false, "skip tests that
   touch postgres")
2  func TestDB(t *testing.T) {
3      if testing.Short() || skipPG {
4          t.Skipf("SKIP %s: touches postgres", t.Name())
5      }
6      // ... database test code here
7  }
```

4.0.4. Optimizing runtime & initialization time

Test binaries are just programs, and tests are just functions. To a certain extent, you make tests fast the same way as ordinary programs and functions, by use of appropriate data structures, avoiding I/O and allocation, and so on. However, test binaries differ from conventional programs in one important way: they don't live long. While a server or user program may run for minutes or hours, test binaries run for milliseconds. This means that **initialization time** is a much bigger cost for tests than for ordinary programs.

As mentioned previously, the `go test ./...` command builds and executes a test binary for each package with tests, so initialization must be repeated for each package - so the cost is both larger absolutely and relatively. I covered starting programs quickly in detail in a previous article: start fast, but the short version is:

- measure your initialization time to see if and where you have a problem (`GODEBUG=inittrace=1` still works here)

- use traditional optimization techniques to shave time off hot-spots
- use nonblocking eager initialization or lazy initialization to free the main thread during startup (in tests, the 'main thread' is TestMain() for each package)
- store assets in a way that makes them easy to load quickly, or better yet, omit them entirely

Let's examine that last point in more detail. Many programs require a variety of assets to run (images, sounds, etc), which may either be loaded from disk or embedded in the binary. In either case, loading and processing these assets can take a long time, and this process will have to be repeated for every package that imports the assets package. But your 'short' tests shouldn't need these at all. Consider skipping loading them entirely, or falling back to a default asset.

Nothing prevents us from checking the `testing.Short()` flag in non-test code, so we can use that to skip loading assets in short tests. A quick gotcha: you can't use `testing.Short()` until the `testing` package has been initialized and CLI flags have been parsed. A program like this:

```
1
2  func main() { // https://go.dev/play/p/N0WTz5koKO-
3      testing.Short()
4  }
```

OUT:

```
1  panic: testing: Short called before Init
```

But that's easy enough to solve:

```
1  func main() { // https://go.dev/play/p/OEOS41KZSUM
2      testing.Init()
3      flag.Parse()
4      fmt.Printf("-short: %v", testing.Short())
5  }
```

OUT:

```
1  -short: false
```

Let's use a real-world example from my game, Tactical Tapir, to demonstrate a more practical example:

```
1  // Package static loads static assets like images, sounds,
   fonts, and shaders.
2  // All assets are embedded into the binary.
3  package static
4
5  // Static assets processed during init().
6  var (
7      Audio map[string][]byte
8      Fonts map[string]font.Face
9      Shader map[string]*ebiten.Shader
```

```
10      Img map[string]*ebiten.Image
11  )
12  // populate the static maps in parallel
13  func init() {
14      // if we're in a -short test, don't load the static
    assets to save time
15
16      testing.Init() // add the test flags to the CLI flag
    set
17      flag.Parse()   // parse the flags to set
    testing.Short()
18
19      if testing.Short() {
20          // it's a test AND -short is set
21          log.Printf("static: -short: skipping static init")
22          return
23      }
24      log.Println("loading static assets")
25      start := time.Now()
26      wg := new(sync.WaitGroup)
27      wg.Add(4)
28      go initMap(wg, Audio, "audio", loadAudio)
29      go initMap(wg, Fonts, "font", loadFonts)
30      go initMap(wg, Shader, "shader", loadShader)
31      go initMap(wg, Img, "img", loadImg)
32      wg.Wait()
33  }
```

Now packages that import `static` can run unit tests without loading any assets.

Readers may be tempted to extend this to mocking out dependencies with some kind of framework. I find this to be a *very bad idea*. Mocks take a lot of space, both within the code and in your head, and they are both fragile to maintain and of questionable value. If you want to see if a database call works, you *need to test the database*. The false sense of confidence mocks give tends to do more harm than good.

5. dependency management

Dependencies kill software by turning structured programs into a loose graph of API calls. Nonetheless, they're unavoidable. Here are some tips for managing them in tests.

5.1. run your dependencies' tests

I don't understand the mindset that tests your OWN code to the brink of insanity, but happily relies on dozens of other people's software packages without even running their tests. Don't do that. Test your dependencies. If they're slow, don't run them every time, but at least run them once in a while to make sure they're reliable. At a bare minimum, run *every* dependency's tests when you update a version of *any* dependency. This means that speed & reliability are important for your dependencies' tests, too.

5.2. Inject dependencies to avoid I/O and allocation

Proper use of interfaces can help you avoid I/O in tests. For instance, many instances of `*os.File` and `net.Conn` can be replaced with another `io.Reader` or `io.Writer`. File systems can be replaced with `io/fs` and outside servers can be replaced using `net/http/httptest`.

Let's demonstrate with an example.

```
1  // Open the file at path and return all lines that contain
   pattern.
2  func findLinesMatchingInFile(path, pattern string)
   ([]string, error) {
3      f, err := os.Open(path)
4      if err != nil {
5          return nil, err
6      }
7
8      var matches []string
9      scanner := bufio.NewScanner(r)
10     for scanner.Scan() {
11         if strings.Contains(scanner.Text(), pattern) {
12             matches = append(matches, scanner.Text())
13         }
14     }
15     if err := scanner.Err(); err != nil {
16         return nil, err
17     }
18     return matches, nil
19 }
```

We'd like to test the behavior of this function. While it's certainly possible to create a lot of files and test it that way, there are some potential issues that hurt it's portability and reliability:

- What if you don't have permissions on this filesystem? Do the names work on windows/linux?
- what about macs and their case-insensitive filesystems?
- how long will it take to create the file?
- what if multiple tests try to create the same file at the same time? (unlikely, I know, but possible.)
- did you remember to close and remove the file when you were done? did you get the order right? what if one or more failed?
- will that poison the filesystem for other tests?

We may run into problems with permissions, etc that might make this test unreliable. And the purpose of testing this function is not to find out whether or not `os.Open` works.

```
1
```

```
func findLinesMatching(r io.Reader, pattern string)
([]string, error) {
    var matches []string
    scanner := bufio.NewScanner(r)
    for scanner.Scan() {
        if strings.Contains(scanner.Text(), pattern) {
            matches = append(matches, scanner.Text())
        }
    }
    if err := scanner.Err(); err != nil {
        return nil, err
    }
    return matches, nil
}
```

Then we can simply pass `strings.Readers` to the function pre-populated with the data we want to test:

```
func TestFindLinesMatching(t *testing.T) {
    t.Parallel()
    for _, tt := range []struct {
        input, pattern string
        want           []string
    } {
        "foo\nbar\nbaz\n", "foo", []string{"foo"},
        "foo\nbar\nbaz\n", "bar", []string{"bar"},
    } {
        tt := tt
        t.Run(tt.input, func(t *testing.T) {
            got, err :=
findLinesMatching(strings.NewReader(tt.input), tt.pattern)
            if err != nil {
                t.Fatalf("findLinesMatching(%q, %q) = %v",
tt.input, tt.pattern, err)
            }
            if len(got) != len(tt.want) {
                t.Fatalf("findLinesMatching(%q, %q) = %v,
want %v", tt.input, tt.pattern, got, tt.want)
            }
            for i := range got {
                if got[i] != tt.want[i] {
                    t.Fatalf("findLinesMatching(%q, %q) =
%v, want %v", tt.input, tt.pattern, got, tt.want)
                }
            }
        })
    }
}
```

5.2.1. timing

Another kind of subtle dependency is timing. We often sleep in functions when we're waiting for some condition to be met. We may wait 1s for a database to be available, for instance, a pattern that looks like this:

```
 1  func TestMain(m *testing.M) {
 2      // EXAMPLE ONLY: don't do this
 3      go func() {
 4          db, err = setupPostgres()
 5          if err != nil {
 6              log.Fatalf("postgres: %v", err)
 7          }
 8      }
 9      go func() {
10          redis, err = setupRedis()
11          if err != nil {
12              log.Fatalf("redis: %v", err)
13          }
14      }
15      time.Sleep(1 *
    time.Second) // wait for the database to be available
16      m.Run()
17  }
```

This sets an artificial floor on the latency of your tests (that is, you make them **at least that slow**, when they could potentially be a dozen times faster).

In general, try to avoid sleeping entirely by using a channel, mutex, waitgroup, or condition variable to signal when the condition is met, roughly in that order of preference. That is, **push** synchronization events from your dependencies to the test setup code.

Sometimes the service you're depending on isn't nice enough to signal *you* when it's ready and you need to poll it instead. If you absolutely must, set up repeated polls at 1-2ms and then push updates from there. Go's channels can be a good way to do this: send an error or nil for each dependency on a channel, and just drain one for each dependency you're waiting on.

The following example shows **one way** to convert a poll into a push:

```
1
2   package main_test
3
4   var redis *redis.Client
5   var db *sql.DB
6
7   func TestMain(m *testing.M) {
8      // setup dependencies. when each dependency finishes or
    times out, send a message on the channel
9      res := make(chan error, 2) // postgres & redis
```

```
10      ctx, cancel :=
    context.WithTimeout(context.Background(), 1 * time.Second)
11      defer cancel()
12      go func() {
13         var err error
14         db, err = setupPostgres()
15         if err != nil {
16            res <- fmt.Errorf("postgres: %w", err)
17            return
18         }
19         for {
20           switch err := db.
21         }
22
23
24      }()
25      go func() {
26         var err error
27         redis, err = setupRedis()
28           if err != nil {
29               log.Fatalf(" to redis: %v", err)
30           }
31           for { // keep pinging until we get a response or
    time out
32               switch err := redis.PingContext(ctx);  {
33                   case err == nil:
34                       res <- nil
35                       return
36                   case errors.Is(err,
    context.DeadlineExceeded:)
37                       res <- fmt.Errorf("redis: %w", err)
38                       return
39                   default:
40                       time.Sleep(2 * time.Millisecond)
41               }
42           }
43      }
44       // wait for all dependencies to be ready
45       for i := 0; i < 2; i++ {
46           if err := <-res; err != nil {
47               log.Fatalf("failed to connect to %s: %v", err)
48           }
49       }
50      os.Exit(m.Run())
51  }
```

6. Test Rescue Plan

The above advice is handy for writing *new* tests, but what if you're already in the weeds? Here's a technique I've used with some success to rescue a test suite that's already in

trouble. This won't fix the problem, but it can help you get back to a place where you can start fixing it.

- Run your test suite 30 times or so.
  - Mark any test which takes over `10ms` as slow and skip it using the `-short` flag.
  - Mark any package which takes `>20ms` to initialize as slow.
  - Mark any test which fails *some* of the time as flaky and skip it unconditionally.
- Unplug the internet and run your test suite again under `-short` using `GODEBUG=inittrace=1` to find packages which are slow to initialize.

```
1    GODEBUG=inittrace=1 go test -short  ./...
```

- Mark any **function** which fails as dependency-heavy and skip it using the `-short` flag.
- Mark any **package** which takes `>20ms` to initialize as slow. If possible, refactor the package to lazily load dependencies or skip loading during `-short` until it doesn't.
- If a **package** fails entirely, refactor the package to lazily load dependencies or skip loading during `-short` until it doesn't.

Keep repeating this process until you've hit all the low-hanging fruit and have a *unit test* step which runs in a reasonable amount of time. In general, you can get this done in a day or so, and this will free up your team to start fixing the underlying problems.

7. Conclusion

Keeping your tests fast and reliable is fundamental to having them work for you instead of against you. Strive to keep the *performance* of your tests in mind, not just coverage, or you'll strangle a codebase you thought you were nurturing.

Like this article? Need help making great software, or just want to save a couple hundred thousand dollars on your cloud bill? Hire me, or bring me in to consult. Professional enquiries at
efron.dev@gmail.com or linkedin

MORE ARTICLES