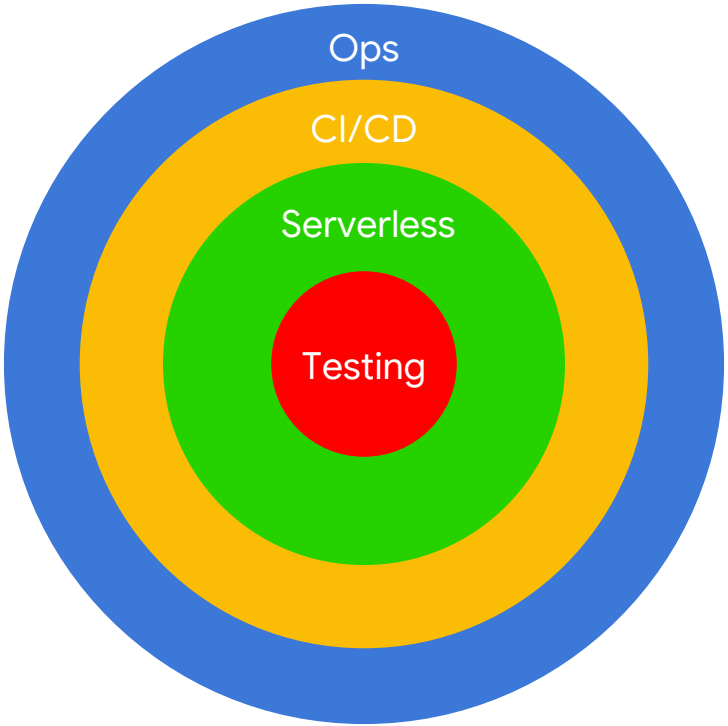
Just keep functioning

Testing, monitoring, and debugging FaaS apps

Ace Nassri
Developer Programs Engineer @ Google
October 4th, 2018

Agenda



1

Testing: a background

An overview of testing strategies

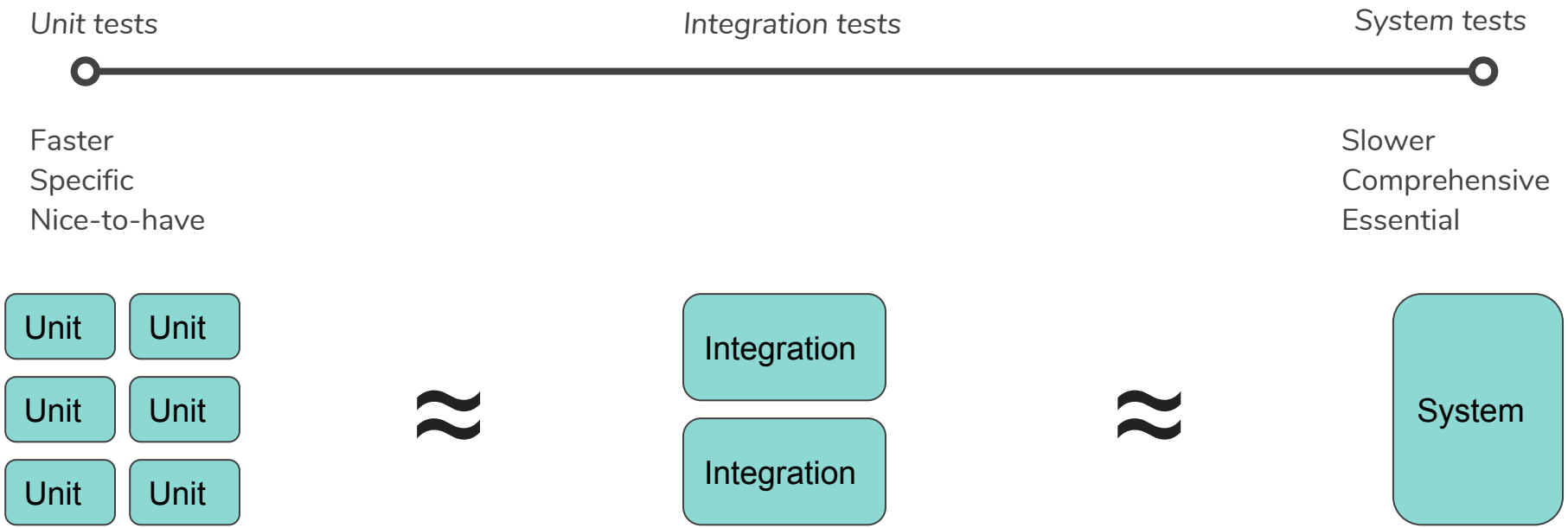
Testing: the what

A high-level definition: *does my code* ***work*** *- and does it* ***work well?***

Lower level questions we might ask:

- Does my code work *according to language and style guidelines*?
- Does my code work *with the latest dependencies*?
- Does my code work *when subjected to extreme loads*?
- Does my code work *when provided edge cases, malicious data, or other invalid input*?

Testing: does my code work?



(We'll explain what each test consists of later.)

Testing: does my code work well?

Static tests - test your code for various things (such as style) without running it

Load tests - system test that tests your code against (un)expected loads

Fuzz tests / Vulnerability Scanners - test your code's handling of potentially malicious input

Testing: the why

Testing is an *insurance policy* against potential bugs and their associated business impact

Goal #1: *reduce the likelihood* of software errors happening

Goal #2: verify code correctness *on an ongoing basis*

Stretch goal: *quickly* identify errors caught by tests

Impact of testing

- Google's OSS security testing discovered >250 potential security issues in ~5 months[1]
- Inadequate software testing costs the US up to ~1% of annual GDP[2]

1: https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html

2: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf

2

Hello Serverless!

Testing strategies for FaaS

Unit tests

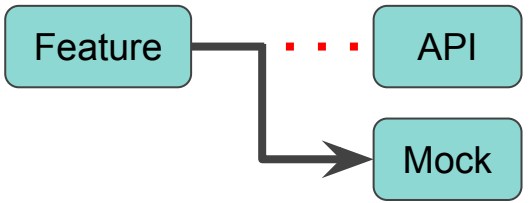
Verify that *the minutiae* of your code work as expected (such as edge case checking)

No reliance (or verification!) whatsoever on external dependencies (such as libraries or APIs)

Use mocking frameworks (such as `Sinon/Proxyquire` in Node.js) to fake external dependencies

Good at isolating causes of known problems, but bad at detecting unanticipated ones

Very quick to run, and rarely require billed resources



Sample HTTP function

```
1: const storage = require(`@google-cloud/storage`)();
2:
3: exports.createBucket = (req, res) => {
4:  return storage.createBucket(req.body.name)
5:      .then(res.send('Success!'))
6:      .catch(err => res.send('Error:', err));
7: }
```

Sample function unit test

```
1: // {proxyquire, sinon, ava} are libraries - imports omitted
2: const storageMock = { createBucket: sinon.stub().resolves() };
3: const program = proxyquire('..', // Mock Cloud Storage library
4:   { '@google-cloud/storage': sinon.stub().returns(storageMock) });
5:
6: ava('should attempt to create a bucket', t => {
7:   const req = { body: { name: 'name' } }; // Fake 'req' object
8:   const res = { send: sinon.stub() }; // Fake 'res' object
9:
10:   program.createBucket(req, res); // Call tested function
11:   t.deepEqual(storageMock.createBucket.firstCall.args, ['name']);
12: });
```

Integration tests

Verify that parts of your code *fit together* as expected

Run locally on a development machine, using the Node.js emulator or a shim

Can either use mocking or rely on external dependencies directly

Required dependencies are usually quick (< 1 second) to use, so tests are moderately quick to run

May require small amounts of billed resources

Balances detection and isolation of problems; can detect unanticipated problems in some cases

Sample function integration test

```
1: // {uuid, ava, request-promise-native} are libraries - imports omitted
2: const storage = require('@google-cloud/storage')();
3:
4: ava('should create a real bucket', async t => {
5:   const name = `serverless-test-${uuid.v4()}`; // Make test runs unique
6:   const url = `${process.env.BASE_URL}/createBucket`;
7:   await requestPromiseNative.post(url, { name: name }); // Eventually consistent
8:   await setTimeout(() => { return Promise.resolve() }, 1000); // Ideal: exponential backoff
9:   const exists = await storage.bucket(name).exists();
10:  t.is(exists, true);
11: }
```

System tests

Verify that your code *works as a system*

Heavy reliance on external dependencies, which can slow down the test itself

Require moderate amounts of billed resources

Great for detecting problems, including unanticipated ones and those residing outside your codebase

Bad at isolating problems and their root causes

Caveat: state matters when testing cloud-based dependencies

State dependence may introduce *eventual consistency* and/or *shared resource* issues

Sample function system test

```
1: // {uuid, ava, request-promise-native} are libraries - imports omitted
2: const storage = require('@google-cloud/storage')();
3:
4: ava('should create a real bucket', async t => {
5:   const name = `serverless-test-${uuid.v4()}`; // Make test runs unique
6:   const url = `${process.env.BASE_URL}/createBucket`;
7:   await requestPromiseNative.post(url, { name: name }); // Eventually consistent
8:   await setTimeout(() => { return Promise.resolve() }, 1000); // Ideal: exponential backoff
9:   const exists = await storage.bucket(name).exists();
10:  t.is(exists, true);
11: }
```

Static tests

Verify your code follows *language/style conventions* and *dependency management best-practices*

Many are free to install and quick to use

Linters detect (and may fix) style issues in your code (e.g. `prettier` for Node.js, `pylint` for Python)

Dependency management tools check for issues with dependencies (e.g. `Snyk` for Node.js)

Major con: narrow focus

```
1 let a, b = null;
2
3 console.log(a, b); // undefined, null
4 console.log(a == b); // true
```

Before linting

```
1 let a;
2 let b = null;
3
4 console.log(a, b); // undefined, null
5 console.log(a === b); // false
```

After fixing linter errors

Load tests

Verify that your entire system (including non-autoscaled components) *keeps up with user demand*

Heavy reliance on external dependencies

Required dependencies may be slow to access, which can slow down the test itself

Tools themselves are typically free, but test runs can require large amounts of billed resources

Helps protect against a “hug of death” or other sudden (and possibly beneficial) spike in traffic

Examples: Apache Bench (`ab` on most Mac/Linux systems), Apache JMeter

DEMO: Apache Bench

Security testing

Verify your code (and dependencies) work with *potentially malicious input*

Motivation: US businesses lost $67 billion due to cyberattacks (in 2005)[1]

Can be part of unit, system, integration, and/or static tests

Tools are often free to install, but may cost money if billed resources are involved

Examples: Zed Attack Proxy, Snyk.io, the Big List of Naughty Strings, Google's `oss-fuzz`

Caveat #1: no automated tool is perfect - *if security matters, do your research and find a consultant*

Caveat #2: may damage the target application environment

1: https://www.gillibrand.senate.gov/imo/media/doc/Sen.%20Gillibrand%20CyberSecurity%20Report.pdf

3 Continuous Deployment

Syncing your codebase and the Cloud

Keeping deployments up-to-date

For many developers, git branches are the source of truth

Many Serverless providers require manual CLI invocation to test + re-deploy code

Continuous Integration + Deployment (CI/CD) systems automate this process

Many different CI/CD platforms available

Cloud Build is Google Cloud Platform's built-in CI/CD platform

Several other Cloud-agnostic hosted options out there

Basic CI/CD systems for Serverless deployments are fairly easy to set up

Example Cloud Build config (Node.js)

```
steps:
- name: 'gcr.io/cloud-builders/npm'     # Install dependencies
  args: ['install']
- name: 'gcr.io/cloud-builders/npm'     # Run tests
  args: ['test']
- name: 'gcr.io/cloud-builders/gcloud'  # Deploy if previous steps pass
  args: ['functions', 'deploy', '[FUNCTION_NAME]', '[TRIGGER]']
```

DEMO: Cloud Build

4

(O)ops!

Keeping deployments healthy

Serverless Operationalization 101

Tests can only detect problems **out of** production

What about problems **in** production?

Stackdriver - tools for monitoring + debugging code on Google Cloud Platform/Amazon Web Services

- Azure's equivalent - App Insights

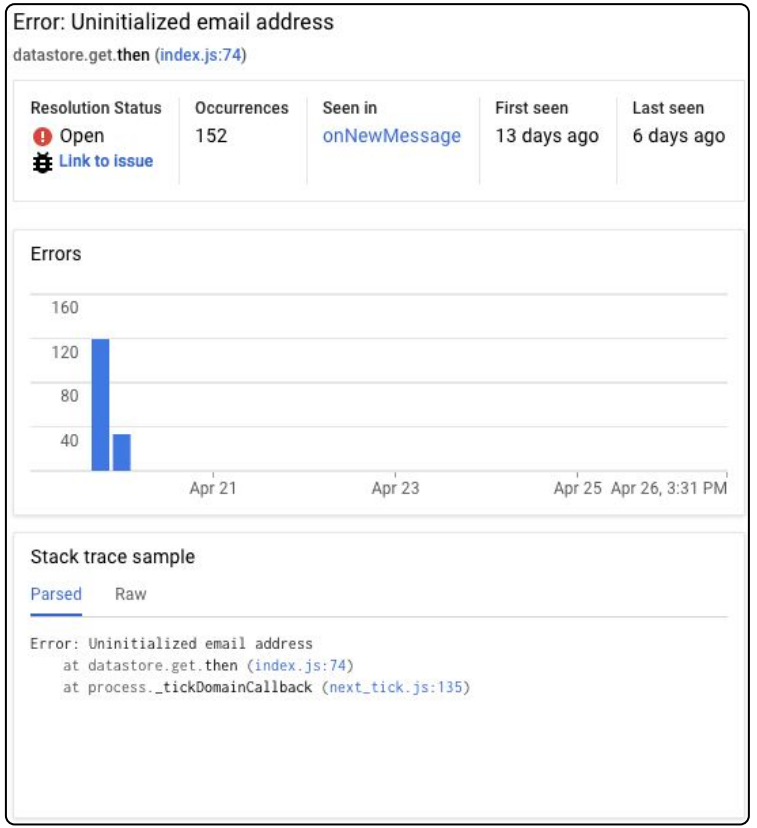
The most useful tools for Serverless developers

- Monitoring + Metrics - *Is* my code broken?
- Logging + Error Reporting - *Where* is my code broken?

Logging + Error reporting

Logging automatically stores + indexes logs from functions

Error Reporting aggregates logs into meaningful error reports



Monitoring + Metrics: a crash course

Monitoring + metrics tools help detect unhealthy code in production quickly

Goal: catch as many errors as we can *with the least setup possible*

Simple config: alert if log entries of a certain severity (e.g. error) occur too frequently (>1% of invocations)

More complex config: alert if execution {time, count} or memory substantially (>2-3x) exceed normal limits

Pros: simple to configure, determines *if* production code is broken, covers any function and/or app

Cons: requires good logging statement coverage, not always helpful when locating broken code

Additional checks: HTTP[S] uptime checks can be used with HTTP[S]-triggered functions

DEMO: Stackdriver Monitoring

Recap

Recap

Background

An overview of testing strategies

Hello Serverless!
Testing Strategies for Cloud FaaS

Continuous Deployment
Syncing your codebase and the Cloud

[O]ops!
Keeping functions functioning

Key Points:

Testing helps detect bugs (and other issues) on an ongoing basis

- An *insurance policy* against potential bugs and their associated business impact

Many testing options for function/app deployments, with different {accuracy, breadth, speed, cost} tradeoffs

- Mission-critical, ongoing-use, or rarely maintained code: write lots of tests with various scopes
- Unimportant, temporary, or rapidly-maintained code: write fewer, higher-level tests

If your tests miss a bug, logging + monitoring tools increase chances of detecting it in production quickly

- Broad “is something broken?” configuration: check for abnormal error rates/execution times

Tests must address Cloud-specific challenges (such as eventual consistency and shared resources)

- Eventual consistency: address using timeouts or exponential backoff
- Shared resources: address by using temporary test resources/environments

Thank you

faast@google.com