LOAD TESTING YOUR APP

CONFOO MONTREAL 2020

Ian Littman / @iansltx

follow along at https://ian.im/loadfoo20

SPEED.

SCALABILITY.

STABILITY.

QUESTIONS WE'LL ANSWER

- What types of tests exist, and what sets each type apart?
- When should I build and run performance tests?
- **How can I match my load test with (anticipated) reality?**
- What does a real load test script look like on a small system?
- How do I properly analyze results during and after my test?

QUESTIONS WE WON'T ANSWER

- How do I use $otherPerfTestTool (!== 'k6')?
- How can I set up clustered load testing?
- How can I simulate far-end users?
- How can I test web page performance browser-side?
- How can I do deep application profiling? (Blackfire for PHP)
- What about single-user load testing?

WE'LL BE TESTING WITH K6*

- Write your tests in JS**
- Run via a Go binary***
- HAR import for in-browser recording

* More tools are listed at the end of this presentation.

** Uses goja, not V8 or Node, and doesn't have a global event loop yet.

*** I've used this on a project significantly more real than the example in this presentation, so that's a big reason we're looking at it today.

#IFNDEF

- Smoke Test
- Load Test vs. Stress Test
- Soak Test vs. Spike Test

SMOKE TEST

- An initial test to confirm the system operates properly without a large amount of generated load
- Do this before you load test
- Pick your implementation...
    - Integration tests in your existing test suite
    - Load test script, turned down to one (thorough) iteration and one **Virtual User** (VU)

LOAD TEST

- <= expected peak traffic
- Your system shouldn't break
- If it does, it's a...

STRESS TEST

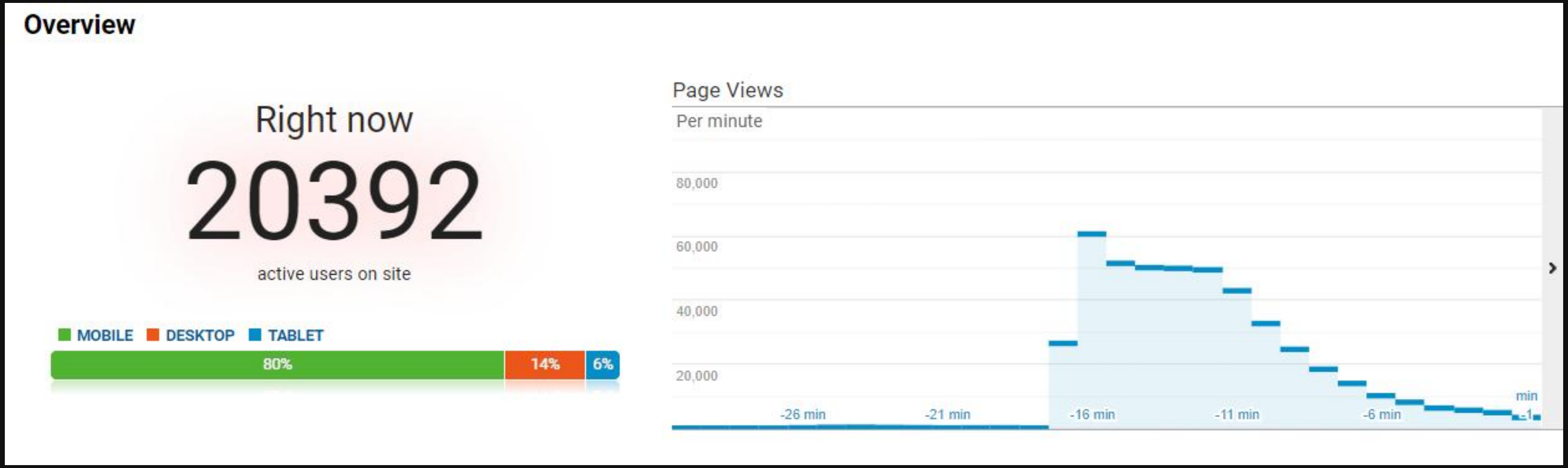
- Increase traffic above peak || decrease available resources
- Try to break your system
- Surface bottlenecks

SOAK TEST

- Extended test duration
- Watch behavior on ramp down as well as ramp up
    - Memory leaks
    - Disk space exhaustion (logs!)
    - Filled caches

SPIKE TEST: A STRESS TEST WITH QUICK RAMP-UP

- Woot.com at midnight
- TV ad "go online"
- System comes back online after downtime
- Everyone hits your API via on-the-hour cron jobs



Source: https://twitter.com/troyhunt/status/1102312963401109504

WHEN SHOULD YOU RUN A LOAD TEST?

- When your application performance may change
  - Adding or removing features
  - Refactoring
  - Infrastructure changes
- When your load profile may change
  - Initial app launch
  - Feature launch
  - Marketing pushes and promotions

HOW SHOULD I TEST?

HOW SHOULD I TEST?

ACCURATELY.

WHAT SHOULD I TEST?

- Flows (not just single endpoints)
- Frequently used
- Performance intensive
- Business critical

CONCURRENT REQUESTS != CONCURRENT USERS

- **Think Time**
- API client concurrency
- Caching (client-side or otherwise)

HOW NOT TO MODEL THINK TIME

- Ignore it
- Use a static amount
- Use a uniform distribution
  (use a normal distribution instead)
- Assume you have one type of user

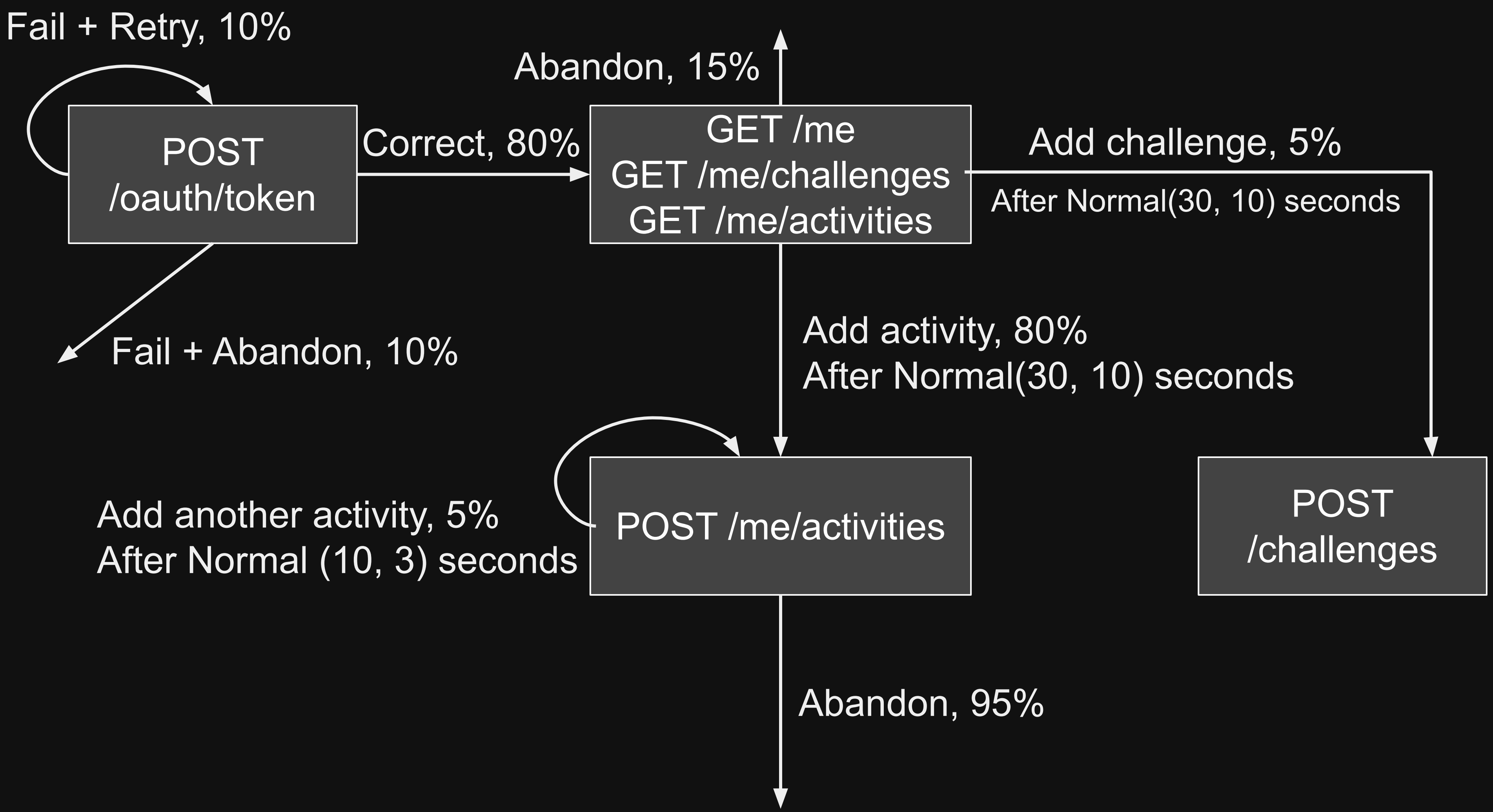
OVERSIMPLIFICATIONS TO AVOID

- No starting data in database
- No parameterization
- No abandonment at each step in the process
- No input errors

VARY YOUR TESTING

- High-load Case: more expensive endpoints get called more often
- Anticipated Case
- Low-load Case: validation failures + think time

SYSTEM UNDER TEST: CHALLENGR



LET'S SEE WHAT THAT LOOKS LIKE WITH K6

Yes, I should've used const instead of let everywhere.

IMPORTS

```
import http from "k6/http";
import {check, fail, sleep} from "k6";
import {Trend} from "k6/metrics";
import {Normal} from [some gist URL];
// Browserified AndreasMadsen/distributions
```

AUTH + FIXTURE DATA

```
// baseURL e.g. http://my-load-test-system.local/
const [baseURL, clientId, clientSecret] =
        open('./config.txt').split("\n"),
    // start on the second line of the document, one per line
    emails = open("./emails.csv").split("\n").slice(1),
```

PROBABILITIES AND INPUT SPECS

```
pCorrectCredentials = 0.8,
pRetryAfterFailedCreds = 0.5,
pAbandonAfterHomeLoad = 0.15,
pAddChallenge = 0.05,
pAddAnotherActivity = 0.05,
pIncludeChallengeDuration = 0.5,
pIncludeChallengeMileage = 0.5,
// start with larger units for more accurate approximation
// of what challenges look like
challengeMinHalfHours = 1,
challengeMaxHalfHours = 80,
challengeMinTenMiles = 1,
challengeMaxTenMiles = 20,
activitySpeed = new Normal(15, 3),
activityMinSeconds = 180,
activityMaxSeconds = 10800,
```

THINK TIME DISTRIBUTIONS

```
challengeThinkTime = new Normal(30, 10),
activityThinkTime = new Normal(30, 10),
secondActivityThinkTime = new Normal(10, 3),
```

TRENDS: HOW LONG DID IT TAKE?

```
challengeListResponseTime
    = new Trend("challenge_list_response_time"),
activityListResponseTime
    = new Trend("activity_list_response_time"),
userProfileResponseTime
    = new Trend("user_profile_response_time");
```

NOW THAT OUR SETUP IS DONE...

```
export default function () {
    let isIncorrectLogin = Math.random() > pCorrectCredentials,
        email = emails[getRandomInt(0, emails.length)];
```

LET'S MAKE AN HTTP CALL

```
let resLogin = http.post(baseURL + "oauth/token", {
    "client_id": clientId,
    "client_secret": clientSecret,
    "grant_type": "password",
    "username": email,
    "password": isIncorrectLogin ? "seekrit" : "secret",
}, {
    headers: {
        "Content-Type": "application/x-www-form-urlencoded"
    }
})
```

MAKE SURE WE FAIL SUCCESSFULLY

```
if (isIncorrectLogin) {
    check(resLogin, {
        "invalid login caught": (res) => res.status === 401
    }) || fail("no 401 on invalid login");

    if (Math.random() > pRetryAfterFailedCreds) {
        return; // abandon on incorrect login
    }

    // log in the correct way this time
    resLogin = http.post(baseURL + "oauth/token", {
        "client_id": clientId,
    // ...snip...
}
```

MAKE SURE WE SUCCEED SUCCESSFULLY

```
check(resLogin, {
    "login succeeded": (res) => res.status === 200
        && typeof res.json().access_token !== "undefined",
}) || fail("failed to log in");
```

MAKING SIMULTANEOUS REQUESTS

```
let params = {
    headers: {
        "Content-Type": "application/json",
        "Accept": "application/json",
        "Authorization": "Bearer " + resLogin.json().access_token
    }
}, makeGet = function (path) {
    return {method: "GET", url: baseURL + path, params: params};
};

let homeScreenResponses = http.batch({
    "me": makeGet("api/me"),
    "challenges": makeGet("api/me/challenges"),
    "activities": makeGet("api/me/activities")
});
```

CHECKING SIMULTANEOUS RESPONSES

```
check(homeScreenResponses["me"],
    {"User profile loaded": (res) => res.json().email === email})
    || fail("user profile email did not match");
check(homeScreenResponses["challenges"],
    {"Challenges list loaded": (res) => res.status === 200})
    || fail("challenges list GET failed");
check(homeScreenResponses["activities"],
    {"Activities list loaded": (res) => res.status === 200})
    || fail("activities list GET failed");
```

TIMING SIMULTANEOUS RESPONSES

```
activityListResponseTime
    .add(homeScreenResponses["activities"].timings.duration);
challengeListResponseTime
    .add(homeScreenResponses["challenges"].timings.duration);
userProfileResponseTime
    .add(homeScreenResponses["me"].timings.duration);
```

DECIDE WHAT TO DO NEXT

```
let pNextAction = Math.random();
if (pNextAction > (1 - pAbandonAfterHomeLoad)) {
    return; // abandon here
} else if (pNextAction >
        (1 - pAbandonAfterHomeLoad - pAddChallenge)) {
    // think time before creating challenge
    sleep(fromDist(challengeThinkTime));
```

LET'S CREATE A CHALLENGE

```
let startMonth = getRandomInt(1, 3), endMonth = startMonth + getRandomInt(1, 2),
    challengeRes = http.post(baseURL + "api/challenges", JSON.stringify({
        "name": "Test Challenge",
        "starts_at": "2020-0" + startMonth + "-01 00:00:00",
        "ends_at": "2020-" + (endMonth >= 10
            ? endMonth : ("0" + endMonth)) + "-01 00:00:00",
        "duration": Math.random() > pIncludeChallengeDuration ? null
            : secondsToTime(
                getRandomInt(challengeMinHalfHours, challengeMaxHalfHours) * 1800),
        "distance_miles": Math.random() > pIncludeChallengeMileage ? null
            : getRandomInt(challengeMinTenMiles, challengeMaxTenMiles) * 10
    }), params);
```

CHALLENGE ACCEPTED?

```
check(challengeRes, {"challenge was created":
                        (res) => res.status === 201 && res.json().id
}) || fail("challenge create failed");

let challengeListRes = http.get(baseURL + "api/me/challenges", params);
check(challengeListRes, {
    "challenge is in user challenge list": (res) => {
        let json = res.json();
        for (let i = 0; i < json.created.length; i++)
            if (json.created[i].id === challengeRes.json().id)
                return true;
        return false;
    }
}) || fail("challenge was not in user challenge list");
```

…OR WE CREATE ACTIVITIES THE SAME WAY

UNDERSTAND YOUR LOAD TEST TOOL

FOR EXAMPLE, ARRIVAL RATE VS. LOOPING

k6 is working on it...slowly...

AGGREGATE YOUR METRICS REPSONSIBLY

- ~~Average~~
- Median (~50th percentile)
- 90th, 95th, 99th percentile
- Standard Deviation
- Distribution of results
- Explain (don't discard) your outliers

KEEP IT REAL

- Use logs and analytics to determine your usage patterns
- Run your APM (e.g. New Relic) on your system under test
  - Better profiling info
  - Same performance drop as instrumenting production
- Is your infrastructure code? (e.g. Terraform, CloudFormation)
  - Easier to copy environments
  - Cheaper to set up an environment for an hour to run a load test
- Decide whether testing from near your env is accurate enough
- Test autoscaling and/or load-shedding facilities

LET'S RUN ANOTHER TEST!

WARNING: TRICKY BOTTLENECKS AHEAD

- Just because a request is expensive
  doesn't mean it's the biggest source of load

- As a system reaches capacity
  you'll see nonlinear performance degradation

BOTTLENECKS: WEB SERVER + DATABASE

- Web workers (e.g. FPM)/Apache processes
- DB Connections
- CPU + RAM utilization
- Network utilization
- Disk utilization (I/O or space)

BOTTLENECKS: LOAD BALANCER

- Network utilization/warmup
- Connection count

BOTTLENECKS: EXTERNAL SERVICES

- Rate limits (natural or artificial)
- Latency
- Network egress

BOTTLENECKS: QUEUES

- Per-job spin-up latency
- Worker count
- CPU + RAM utilization
  - Workers
  - Broker
- Queue depth

BOTTLENECKS: CACHES

- Thundering herd
- Churning due to cache evictions

LET’S FIX SOME BOTTLENECKS…

BONUS: MORE TOOLS

- Artillery.io (Node)
  - Simple stuff in Yaml
  - Can switch to JS (with npm)
- Molotov (by Mozilla, in Python)
  - Uses async IO via coroutines
- Locust (Python)
  - Can be run clustered
- Siege
  - GitLab Large Staging Collider
- Apache JMeter (Java)
- Gatling (Java)
  - Tests in Scala...
  - ...or use the recorder
- ab
- httperf
- Wrk2 (C)
  - Scriptable via Lua

WHAT WE LEARNED

- What types of tests exist, and when you should use them
- **How to match load tests with (anticipated) reality**
- What a real performance test script looks like in K6
- How to analyze results during and after your test

FURTHER READING

- Performance Testing Guidance for Web Applications (from Microsoft)
- Blazemeter Blog - solid info on load testing topics
- ian.im/loadarch - an article version of this talk (php[architect] sub req'd)
- test-api.loadimpact.com - an API to load test against from the k6 folks

THANKS! QUESTIONS?

- ian.im/loadfoo20 - these slides
- github.com/iansltx/challengr - this code
- twitter.com/iansltx - me
- github.com/iansltx - my code
- Please leave feedback; thanks :)