Fast, safe, and reliable

The future of configuration

Qui Nguyen
Tech Lead
How can we best empower developers?

Enable them to make changes as quickly as possible, while making it easy to stay safe.
Configuration

Values used internally by applications that you want to change over time.
Decouple the application from resource names like S3 buckets or logs.

Tune the number of retries and timeouts for a request.

Change the number of search results.

Update percentages or whitelists for an experiment.
Constants

config.py
---------
TIMEOUT_SEC = 10

client.py
---------
from config import TIMEOUT_SEC
do_request(TIMEOUT_SEC)

- Reusable
- Need to update code
- Can't vary across environments
Environment variables

```python
client.py

----------
timeout = os.environ.get('TIMEOUT_SEC')
do_request(timeout)
```

$ TIMEOUT_SEC=10 python client.py

+ Reusable
+ Can vary across environments
- Need to restart process
Read from files

client.yaml
-------------
timeout: 10

client.py
---------
load_file_if_changed('client.yaml')
timeout = get_config('timeout')
do_request(timeout)

+ Reusable
+ Can vary across environments
+ Get new value if file changes
$ git push origin i/qui/timeout
...
remote: merged to master
Files from git copied to client hosts every minute, with rsync
Varying files by environment

- configs stored in git for all environments
- Clients materialize a view based on their location ("hiera-merging")
server1

ecosystem: prod
- region: uswest1

common/foo.yaml
  log_name: foo_errors
  max_timeout_secs: 5

ecosystem/prod/foo.yaml
  max_timeout_secs: 1

foo.yaml
  log_name: foo_errors
  max_timeout_secs: 1
srv-configs

I need to decrease that default timeout.

$ git push origin i/qui/timeout
...
remote: merged to master
See changes in 2 minutes.

Push a branch and your service will start using those values as soon as the files update, with no restart required.
Configuration brings down the site

experiments:
    buttons:
      active: True
      cohorts: [...]

cohort = get_cohort('buttons')
# modern_blue, layout_blue
use_blue_buttons = 'blue' in cohort
Configuration brings down the site

```python
experiments:
    buttons:
        active: False
        cohorts: [...]  

cohort = get_cohort('buttons')
# modern_blue, layout_blue
use_blue_buttons = 'blue' in cohort
```

```
TypeError: argument of type 'NoneType' is not iterable
```

---

The dog ate our website.
Ok, ok, ok we know that's not an excuse. We'll have it back up and running for you shortly.
Incidents

- code changes
- srv-configs
srv-configs deploy

$ git push origin i/qui/timeout
...
remote: deploy started
Make it **easy to do the right thing.**

Automatically stage changes, from stage to canary to production.

Roll back in one step.
public deploy of i/qui/demo

(540f76deb3b57cb0d5bc36b0c638eca46e464611) | state: deploying

stage: 100%, dev: 100%, canary: 0%, prod: 0%

Rollback Now

2 replies

Deploying over next 180 seconds

State machine: deploying
Progress: stage: 100%, dev: 100%, canary: 0%, prod: 0%
Last operator action: None

Author: @qui
Files affected: ['common/compute_infra_test_service.yaml']
**Interactive interface**

1. Progress of deploy updated as it goes through stages.
2. Prominent button to encourage rolling back in case of issues.
3. More details for authors in thread.
$ git push origin i/qui/timeout
...
remote: deploy started

--- a/foo.yaml
+++ b/foo.yaml
@@ -1,2 +1,2 @@
  log_name: foo_errors
-max_timeout_secs: 5
+max_timeout_secs: 2

$ git push origin i/qui/timeout
remote: deploy started
Implementing staged deployments

When an integration branch (i/) is pushed, git hooks:
- Generate a deployment version of each affected file
- Update control file with rollout information for that branch
Implementing staged deployments

main/foo.yaml
log_name: foo_errors
max_timeout_secs: 5

deploy/foo.yaml
log_name: foo_errors
max_timeout_secs: 2

branches.json
"i/qui/timeout": {
  "stage": 1.0,
  "canary": 0.0,
  "production": 0.0,
}, ...

Generated by git hooks and synced to clients
New clientlib

Provides an interface between data (files) and logic

Before
load_file('my_service.yaml')
watcher('my_service.yaml').reload_if_changed()

After
load_namespace('my_service')
get_watcher('my_service').reload_if_changed()
Rolling back

When the rollback button is clicked:
- Delete deploy files
- Remove branch from branches.json
Deploying over next 180 seconds

State machine: deploying
Progress: stage: 100%, dev: 100%, canary: 0%, prod: 0%
Last operator action: None

Files affected: ['common/compute_infra_test_service.yaml']
Rolling back later

Tell the user how to rollback after the deploy is done, too.
Automatic rollbacks

Can monitor SLOs or other metrics in deploy process

- Notify users and default to rolling back
This is cool...

but why files?
We should switch to a datastore!

Clients can watch for changes, instead of rsyncing everything

More natural API for automated changes
- Updating deploy progress
- Updating config from other systems
…or not. Files have a lot of features.

Can take advantage of existing workflows for code files
- Editing with text editors
- Searching, reviewing

Version control (git)
- Viewing history
- Reverting changesets

Easy to scale and handle master failures
Time investment vs. Impact on developers

- Files
- Database
Why don’t we have both?
Configs are as powerful as code.
How can we best empower developers?

Dynamically loaded configuration from files allows developers to spend less time updating and deploying code.

Interactive deploy interface provides rich feedback and guides developers on best practices to keep the site safe.