transmutator Documentation

Release latest

May 28, 2015

Contents

Transmutator is a general purpose migration framework. It focuses on automating actions you perform to upgrade (or downgrade) a product.

> **Warning:** This project is experimental. At this stage, it just describes concepts. Perhaps the concepts are implemented by some existing tools.

A typical migration for a web service could be:

- ask admin for confirmation
- enable maintenance page
- stop frontends
- backup data
- update configuration
- provision machines (upgrade software)
- migrate databases
- restart frontends
- run smoketests
- disable maintenance page.

CHAPTER 1

Upgrades are migrations

Provisioning is not enough to manage upgrades. Database migrations are just a part of the upgrade procedure. We need migration scripts and workflows.

CHAPTER 2

Migration scripts

- Recipes are libraries that provide classes with forward() and backward() methods.
- Dispatchers manage lists of actions to run and watch/notify stop conditions.
- Migration scripts load, configure and run recipes.
- Collecter grabs the list of unapplied migration scripts.

Migration scripts are small shell scripts that accept standardized arguments: they "forward" by default, they optionally implement "–backward". The language in which scripts are written does not matter.

There is two kind of migration scripts:

- atomic mutations, migrations that focus on one thing;
- orchestration, migrations that focus on running smaller scripts.

An upgrade from a release to another is typically encapsulated in an orchestration-type script, which itself groups atomic-type scripts.

CHAPTER 3

Migration workflows

Automating everything is hard, sometimes impossible, sometimes unwanted. A migration procedure is a workflow: it passes from one state to another via transitions. Most transitions can be automated, but some may require human interaction.

Example of human interactions:

- setup SSH keys
- update configuration where defaults are not suitable
- review and confirm some actions
- perform actions that have not been automated yet

CHAPTER 4

Iterative deployment development

When you start a project, you do not want to spend days to get the perfect deployment workflow. In fact, you usually cannot even get a suitable deployment workflow at first. Partly because you do not know how to deploy things you have not developed yet. Partly because you want to focus on proof of concepts, where automated deployment is not top priority.

Transmutator allows you to setup interactive workflows, where you can tell the user to perform actions you have not automated yet.

CHAPTER 5

Remote-control multiple machines

On distributed architectures, you have to orchestrate migrations on multiple machines. Transmutator runs high-level migration scripts that use your favorite remote-control tools, such as fabric or salt.

CHAPTER 6

From DEV to PROD

Migrations are part of the development process. Several developers can contribute to migrations, concurrently. Transmutator is made to reproduce migrations over every environments, from DEV to PROD. The differences between DEV (tends to be monolithic) and PROD (tends to be distributed) are managed via configuration. Transmutator supposes you manage architecture as configuration.

CHAPTER 7

Features / workflows / demo

At last, here is implementation... *transmutator* tries to implement the concepts above. It is a proof of concept. If you feel something is going wrong, please tell us ;)

Note: This is kind of a plan for some tests.

Note: All features are not be implemented yet.

- *transmutator* provides `transmute` command.
- `transmute` without arguments runs mutations "forward".
- `transmute` reads mutations in *mutations directory*: `mutations` folder in current directory (pwd).
- here is a sample "mutations" folder tree:

```
mutations
-- 0001_hello_world.py
-- 0040_1234.sh
-- 1.2
|   -- 0093_print_version.sh
-- 1.3
|   -- 0060_print_version.sh
-- development
|   -- 0077_refactoring.py
-- recurrent
    -- 0050_syncdb.sh
```

- A mutation file must be executable. Else, it is ignored.
- All mutation scripts/binaries implement the *mutation interface*:
  - no arguments means "forward"
  - accept `--backward` argument to run "backward" instead of "forward"
  - that's all for now. Later, additional options such as `help` may be added.
- Mutations can be grouped by "release/version". In the example above:
  - `0001_hello_world.py` and `0040_1234.sh` have "no release".
  - `1.2/0093_print_version.sh` has release "1.2"
  - `1.3/0060_print_version.sh` has release "1.3"
  - mutations in `development/` have not been released yet, their content may change during developement.

  - mutations in `recurrent/` are special kind of mutations, they are to be executed for every release.
- Mutations are executed in order:
  - first ordering criteria is "release/version" groups:
    * `1.2/0093_print_version.sh` is executed before `1.3/0060_print_version.sh`
    * mutations in `development/` are executed at the end. "development" is a special release, the latest.
    * mutations in `recurrent/` are considered part of every release, so they are run for each release.
  - then, in a release, mutations are sorted by filename:
    * `0001_hello_world.py` is executed before `0040_1234.sh`
    * `recurrent/0050_syncdb.sh` is executed before `1.3/0060_print_version.sh`
- Once mutations have been executed, they are not executed again. Except recurrent and in-development mutations:
  - recurrent mutations are executed (forward) for each release
  - in-development mutations are always executed. But they are run "backward" then "forward" (undo/redo).