Using Policyfiles

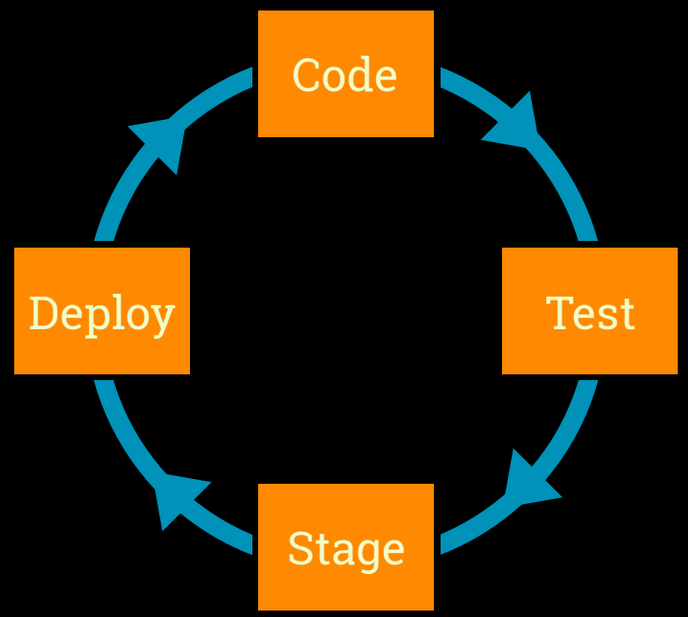
YoloVer as a Workflow

Noah Kantrowitz

tl;dr

10km view of a workflow

Workflow



New Words

- chef Command – Part of ChefDK, like knife
- Policyfile – Source code for a policy
- Policy Name – Replaces role, web/db/etc
- Policy Group – Replaces env, SN1/2/etc
- Compiled Policy – Snapshot of a policy

Policy name corresponds to cluster name, policy group to stage. This is like role and chef environment in the beforetime. A node can be attached to exactly one policy (name) and one policy group.

Smile for the Camera


```
{
  "revision_id": "288ed244f8db8bff3caf58147e840bbe079f76e0",
  "name": "demo_policy",
  "run_list": ["recipe[demo::default]"],
  "cookbook_locks": {
    "demo": {
      "version": "1.0.0",
      "identifier": "f04cc40faf628253fe7d9566d66a1733fb1afbe9",
      "dotted_decimal_identifier": "67630690.23226298.2550585",
      "source": "cookbooks/demo",
      "cache_key": null,
      "scm_info": null,
      "source_options": {"path": "cookbooks/demo"}
    }
  }
}
```

… And Push It Over There

- `chef install`
- Compile and download

- `chef push`
- Upload to Chef Server

Policyfile.rb

Overview of what goes in a Policyfile. Folder layout for multiple policies. Git integration.

Policyfile.rb

```
name "kafka"

default_source :community

run_list "base", "kafka::server"
```

Run Lola Run

```
run_list "foo"

run_list ["foo", "bar"]

# Same as above
run_list "foo", "bar"
```

You can either give multiple arguments or an array.

Marathon Man

```
named_run_list :deploy, "app::deploy"

$ chef-client -n deploy

# Doesn't work anymore
$ chef-client -o "recipe[app::deploy]"
```

Introduce named run lists, like override run lists of yore. We can't use override run lists themselves because we need to use only the cookbooks the policy knows about.

Alice's Restaurant

```
cookbook "monit"

cookbook "monit", "1.0.0"
cookbook "monit", "~> 1.0"

cookbook "monit", path: "../chef-monit"
cookbook "monit", github: "poise/monit"
cookbook "monit", git: "https://…"
```

The Usual Suspects

```
default_source :community

default_source :supermarket, "http://…"

default_source :chef_server, "http://…"

default_source :chef_repo, "../"
```

Rather than have to state specifically where each cookbook comes from, we can provider one or more default sources. The most common is to use https://supermarket.chef.io/, but we can also pull in from a private Supermarket, an organization on a Chef Server, or folder of cookbooks.

Lone Star

```
default_source :… do |s|
  s.preferred_source_for "monit", "…"
end
```

M*A*S*H

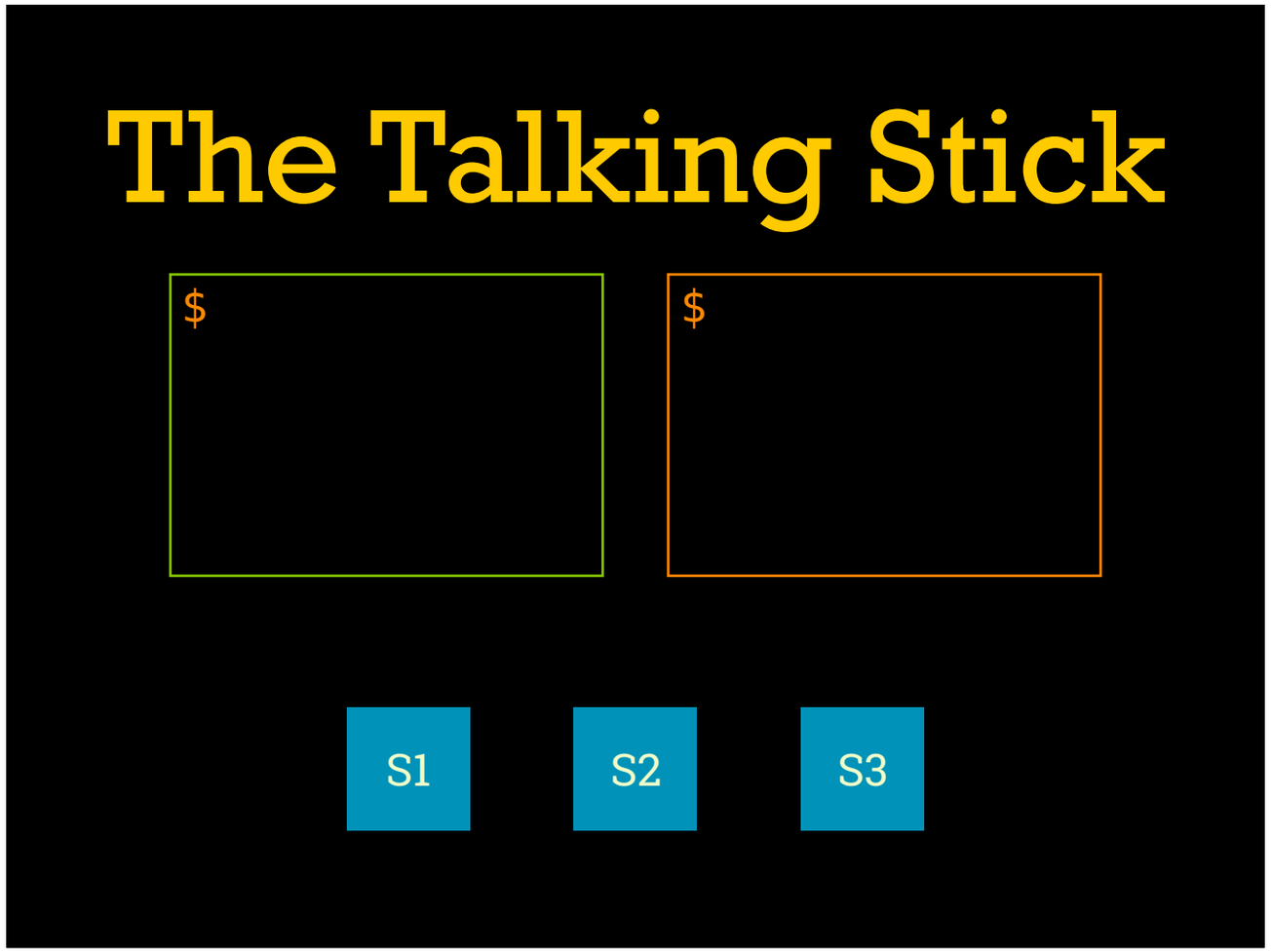
```
default["myapp"]["root"] = "/app"

override["monit"]["port"] = 8080
```

Bleeding Edge

Talk about the downsides of the flow. Single pipeline, machine ownership, etc.

The Talking Stick

- There is only one pipeline
- Multiple releases must be mutex-d



So by way of an example, let's look at a case with two people trying to push an update to a cluster with three stages.

The Talking Stick

```
$ chef install
Policy compiled
$ chef push s1
```

```
$
```

S1 S2 S3

First the green user compiles their policy, and pushes the compiled policy to stage one.

The Talking Stick

```
$ chef install
Policy compiled
$ chef push s1
$ chef push s2
```

```
$
```

S1 S2 S3

They do whatever verifications are needed (not pictured), and then proceed to push the compiled policy to stage two. Then let's say they get bored and go out to lunch.

The Talking Stick

```
$ chef install
Policy compiled
$ chef push s1
$ chef push s2
```

```
$ chef install
Policy compiled
$ chef push s1
$ chef push s2
$ chef push s3
```

S1 S2 S3

The orange user gets back from lunch early, compiles their own modified policy, and rolls it out to all three stages (again, there would be some verification/burn in between stages but that isn't the point).

The Talking Stick

```
$ chef install
Policy compiled
$ chef push s1
$ chef push s2
$ chef push s3
```

```
$ chef install
Policy compiled
$ chef push s1
$ chef push s2
$ chef push s3
```

Then the green user gets back and finishes their deploy. In the end we have an inconsistent cluster. Stages one and two are running the orange policy but stage three has the green policy.

The Talking Stick

- For now: make sure no one else is deploying
- Future: Deckhand may help lock clusters
- Situational awareness is required

So in short, be aware of when a deploy is happening. If you are deploying on a cluster that you own, this is somewhat easier. If deploying on client clusters, double check with their team(s).

Environment Attributes

- default/override in the policy act like role attires
- No specific support for group-level values

Nesting

```
# Policyfile.rb
default["SN1"]["app"]["dbhost"] = "…"
default["SN2"]["app"]["dbhost"] = "…"

# recipes/default.rb
node[node.policy_group]["app"]["dbhost"]
```

Hoisting

```
# Policyfile.rb
default["SN1"]["app"]["dbhost"] = "…"
default["SN2"]["app"]["dbhost"] = "…"

# attributes/default.rb
default.update(default[node.policy_group])
```

Data Bag

```
# attributes/default.rb
item = DataBagItem.load("env",
                        node.policy_group)
default.update(item.raw_data)
```

Base Role

```
# base.rb
default_source :community
run_list "base"
default["key"] = "value"

# Policyfile.rb
instance_eval(IO.read("base.rb"))
name "web"
run_list << "web"
```

You can't use roles in the run lists of a Policyfile. The biggest case this impacts is having a "base" role applied to all nodes. You can support most of this use case using a shared base Policyfile that is included in all the others. This allows following same snapshot-based workflow while still having some shared data.

Partial Updates

- `chef update` can only regenerate a policy
- Planned for the future
- Use `chef diff` for safety

A downside compared to a Berkshelf-based workflow is that single cookbooks can't be upgraded without fully re-compiling the policy. Support is planned for the future, but for now take care to diff your compiled policy before pushing to ensure you aren't releasing something unexpected.

Danger Zone

- LANA, LANAAAAAAAAAA
- New, fresh, well-tested
- Growing quickly

ChefDK in general and the Policyfile tools in specific are evolving rapidly. Not everything out there has support for the new workflows but it will probably be added soon. If you run into an unsupported corner of the ecosystem, you can always ask me.

Trouble Spots

- Single pipeline
- Group-level attributes
- Shared base configuration
- Partial updates
- Young tooling

So to summarize, some of the major things to look out for are pipeline stomping, data management, and possibly wonky integrations.

Order's Up

Release process or lack thereof. Benefits and downsides of a semver process. Talk about yolover snapshots.

Release Process

- Update cookbook version
- Make a git tag
- Maybe push to (internal?) Supermarket
- Push to Chef Server organization
- Update Chef environments in order
- Repeat for each changed cookbook

A quick version of a traditional Chef cookbook release process. First we updated the cookbook version in metadata.rb, and make sure to follow SemVer. Then we make a git tag, maybe use Stove to push to a Supermarket site, and `berks upload` to push it to a Chef Server. Then we edit the Chef environment for each stage to roll the new version out, making sure chef-client completes successfully on each stage. Importantly this process is cookbook-centric, so we release each cookbook independently.

SemVer FTW!

- Allows looser environment restrictions (~> 1.0)
- Better control for other teams
- More semantic info for Deckhand
- Warm and fuzzy feelings

Usually a release process like this is coupled with SemVer so that we can use that semantic information to structure how new releases flow out to different environments and users. It allows using the pessimistic range operator in environments and dependencies, leaving the version solver in Chef Server and Berkshelf to work out the details.

More Like LameVer

- Mental overhead to establish "compatible"
- Ensure all dependencies are released in order
- Must have linearized x.y.z versions
- No concurrent git branches or pre-releases

But it isn't all positive. Tracking which changes are compatible with which other changes presents some cognitive load during development. Additionally when running a release process you often need to release multiple cookbooks in the right order. On top of that, Chef's version solver is very limited and only supports three component x.y.z version numbers, no extra tags like -pre or -rc1. This makes it difficult to handle releases when different environments are targeting different git branches of the same cookbook.

That's a lot of work.

YoloVer

- Policyfile(s) linked directly to git
- Use a cookbooks/ folder if desired
- `chef install/update` to take a new snapshot
- `chef push` to deploy to stages

So we want a lighter weight solution. Enter YoloVer: a workflow based around snapshots of a whole run list instead of multiple discrete projects with their own release processes. As you might imagine, this workflow is based around Policyfiles. Using all the tools we just learned, we can manage cookbook deployments with a granularity of whole-repository snapshots. This means we don't need the overhead of a per-cookbook release system but still retain control over rollouts.

Example Repo

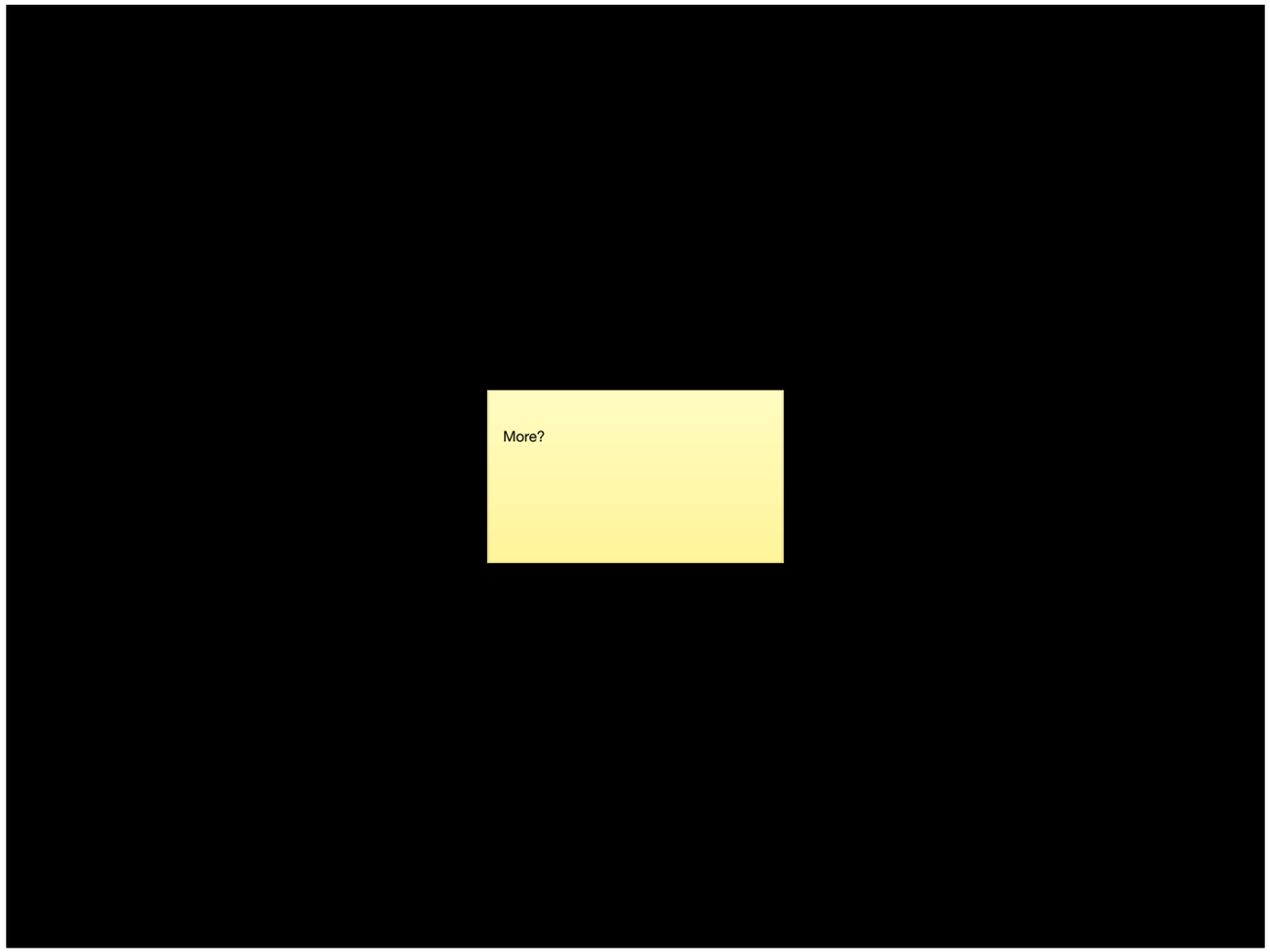
```
$ ls .
cookbooks/ policies/

$ ls cookbooks
bb-kafka/ bb-graphite/ bb-collectd/

$ ls policies/
db.rb frontend.rb
```

So let's look at an example git repository for a hypothetical monitoring team. We have a folder of local cookbooks specific to the team, and a folder of policies for each type of server we are going to maintain.

policies/db.rb

```
name "db"

default_source :community
default_source :chef_repo, ".."
cookbook "clojure", github: "…/clojure"

run_list "clojure", "git", "bb-kafka"
```

For the "db" policy we have all the things we saw before. We set the name, as always. Then we set two default sources. Remember these have to be non-overlapping, but it means everything in the cookbooks/ folder will be picked up automatically when needed. We also set one cookbook as coming from a specific git repository. When we take the snapshot, we'll capture whatever is in the master branch of that repository and use it until we recompile. Finally we set our run list, using cookbooks from all three sources.


More?

Local development with the policyfile-zero provisioner and Test Kitchen.

replace env cookbook pattern

Mise en Place

Having a production workflow is all well and good, but development starts on a workstation somewhere and generally we want to try things locally first.

Chef Push

Discuss the chef push command and how to use it.

Discuss the chef push command and how to use it.

What baseline are we starting from?

A: Name-level familiarity but teaching from scratch.

Household Staff

Jenkins, CI, maybe ChefSpec and InSpec?

Jenkins, CI, maybe ChefSpec and InSpec?

Testing Tools

- ChefSpec – Unit testing
- Test Kitchen – Integration testing
- InSpec – Integration/acceptance testing

There are four major branches of testing (unit, integration, acceptance, and smoke) but only unit and integration tests generally happen in the context of local testing. With Chef that means ChefSpec and Test Kitchen. We'll see more of InSpec later when we start writing integration tests, but for now just know it is related to Test Kitchen.

Unit Testing

- Test a single unit of logic (recipe, resource)
- Mock/stub at unit boundaries
- Ensure isolation between tests
- Move fast and break things

Before we go over the details of how to use those tools, let's look at the theory behind the types of testing. Unit tests mean testing a single unit of code. With things like web applications a unit is usually a single class or method, but for Chef our units are more often a single recipe or a custom resource. The goal of unit testing is get high-speed tests above almost anything else. This means we focus on mocking out anything outside of the unit especially anything slow or with side effects.

Unit Testing

- Edge cases
- Complex inputs
- Regression checks

This makes unit tests a great choice for testing edge cases, complex inputs, and regression checks. Because unit tests are generally much faster than integration tests, we can have far more of them without our tests taking a frustrating amount of time.

Integration Testing

- Test the integration of multiple units
- Check side effects
- Slower, but closer to real life

Integration tests are the other side of the coin. We throw all the units in a big pile and light it on fire to see what happens. This means tests take a lot longer since we let them do the whole side-effects thing, and rely on higher-level isolation for repeatability.

Integration Testing

- Real world use cases
- Performance tests (sometimes)

So this means generally we use integration tests to look at bigger slices of real-world use cases. Sometimes they can also be used for performance analysis, but beware of how much the test environment impacts the numbers, you probably can't compare directly to production.

ChefSpec

- RSpec 4lyfe
- Really runs Chef
- Provider stubs, `step_into` specific providers
- Stub helpers for data_bags, search, commands

ChefSpec is the primary unit testing tool for Chef. It is a set of RSpec helpers and extensions for running a Chef converge without actually modifying the system. By default all providers are stubbed to be a no-op, but you can re-enable specific ones for testing.

Test Kitchen

- Create a fresh virtual machine
- Install and run Chef
- Run verification tests via InSpec
- Uses Policyfile via policyfile_zero plugin

Test Kitchen is a suite of tools to create a blank VM using any of a number of drivers (Vagrant, AWS, OpenStack, Docker), install Chef and copy our cookbook(s) to the VM, run a normal Chef converge, and then run some tests to confirm the cookbook did what we think it was supposed to.

RSpec

Before we dive into the specifics of ChefSpec, let's talk about writing RSpec tests in general. ChefSpec builds on top of RSpec so you need to walk before you can run (or converge, ::cymbal crash::).

RSpec

```
describe "a thing" do
  it "is a thing" do
    expect(1).to eq 1
  end
end
```

The simplest possible RSpec test. We have one example group, created using "describe" and labeled "a thing". Inside that is one example ("it") labeled "is a thing". Group and example labels can be anything, and we'll cover them in more detail in a moment, but overall the standard is to make the RSpec code read like prose. Inside the example we have one expectation ("expect()"), in this case just comparing 1==1.

Describe

```
describe MyClass do
  # ...
end

describe "label" do
  # ...
end
```

Context

```
describe "a thing" do
  context "with A" do
    # …
  end
  context "with B" do
    # …
  end
end
```

It

```
it "works" do
  expect(val).to …
end

it { expect(val).to … }
```

All Together

```
describe "addition" do
  context "with 1" do
    it { expect(1+1).to eq 2 }
  end
  context "with 2" do
    it { expect(2+2).to eq 4 }
  end
end
```

Point out that I have no "it" labels here

Running

- Put that in spec/thing_spec.rb
- $ chef exec rspec

Expectations

```
expect(value).to matcher
```

eq Matcher

Point out function-y nature just this once.

```
expect(value).to eq(other)

expect(1).to eq 1

expect("a").to eq "a"
```

to_not Mode

Quick diversion, not a matcher but a matcher mode

```
expect(1).to_not eq 2

expect("a").to_not eq "b"
```

be Matcher

```
expect(1).to be > 0

expect(-1).to be < 0

expect(0).to be == 0
```

Same as eq matcher

Boolean Matchers

```
expect(nil).to be_nil

expect(true).to be true
expect(false).to be false

expect(1).to be_truthy
expect(nil).to be_falsey
```

String Matchers

```
expect("abc").to include "a"

expect("abc").to match /a.c/

expect("abc").to start_with "a"

expect("abc").to end_with "c"
```

Class Matchers

```
expect("a").to be_a String

expect(1).to be_an Integer
```

Error Matchers

Point out the block here

```
expect { myfunc() }.to raise_error

…to raise_error ArgumentError

…to raise_error /message/
```

Subject

```
describe "a thing" do
  subject { 1 }
  it do
    expect(subject).to eq 1
  end
end
```

Is Expected

```
describe "a thing" do
  subject { 1 }
  it do
    is_expected.to eq 1
  end
end
```

Should (Okay)

```
describe "a thing" do
  subject { 1 }
  it do
    should eq 1
  end
end
```

Should (Not Okay)

```
describe "a thing" do
  subject { 1 }
  it do
    subject.should eq 1
  end
end
```

Let

```
describe "a thing" do
  let(:myval) { 1 }
  it do
    expect(1+myval).to eq 2
  end
end
```

Complex Let

```
describe "a thing" do
  subject { val + 1 }
  context "with 1" do
    let(:val) { 1 }
    it { is_expected.to eq 2 }
  end
  context "with 2" do
    let(:val) { 2 }
    it { is_expected.to eq 3 }
  end
end
```

Before

```
describe "a thing" do
  before do
    puts "BEFORE!"
  end
  it { … }
end
```

Before Timing

```
before(:each) { … }

before(:all) { … }
```

Other Hooks

```
before { … }

after { … }

around {|ex| …; ex.run; … }
```

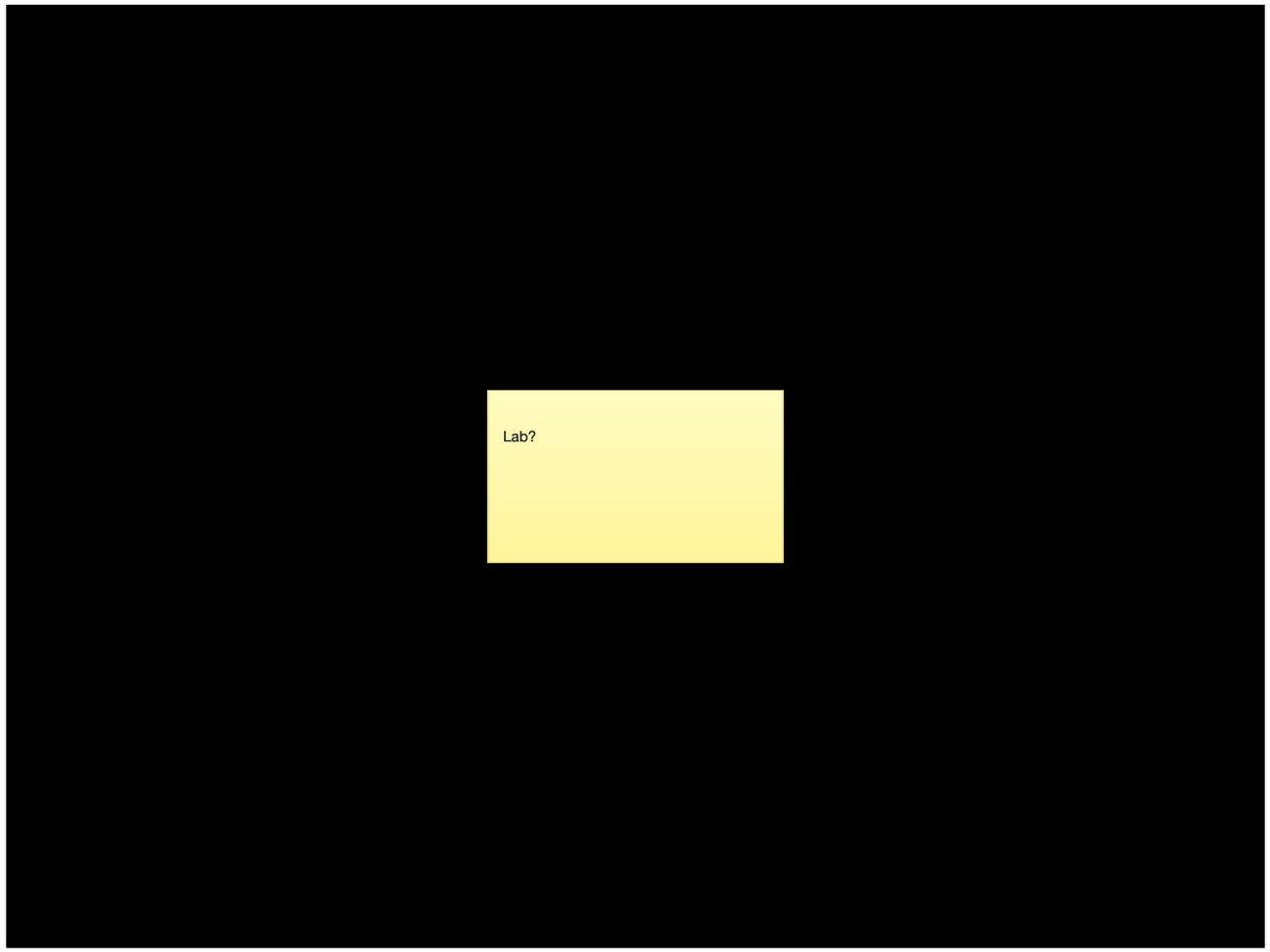
Mention timing works on all of them

Spec Helper

```
# spec/spec_helper.rb
require "…"

RSpec.configure do |config|
  # …
end
```

Spec Helper

```
# spec/thing_spec.rb
require "spec_helper"

describe …
```


Lab?

Mocks

- Helpers for faking out methods
- Avoid "dangerous" call (IO.write, shell_out)
- Call without depending on internals (unit isolation)

Mocks

```
allow(IO).to receive(:read)

expect(IO).to receive(:read)
```

Argument Matchers

```
… receive(:read).with("/foo")

… receive(:read).with(match /foo.*/)
```

Return Value

```
… receive(:read).and_return("lorem")

… receive(:read) {|path| "lorem" }
```

Doubles

```
double()

double(method: "1")

double("label", method: "1")
```

Doubles

```
double(x: 1).x == 1

fake = double
expect(fake).to receive(:x) { 1 }
```

Default mode is allow

Example

```
describe MyLib do
  let(:node) do
    double(name: "test.example",
           chef_environment: "prod")
  end
  subject { MyLib.myfunc(node) }
  it { is_expected.to eq … }
end
```

Example

```
describe "myfunc" do
  subject { myfunc("foo", "abc") }
  it do
    expect(IO).to receive(:write) \
                  .with("/foo", "abc")
    subject
  end
end
```

Explain expect mock before subject

Example

```
describe "myotherfunc" do
  subject { myotherfunc("bar") }
  before do
    allow(IO).to receive(:read) \
    .with("/bar").and_return("abc")
  end
  it { is_expected.to eq "abc" }
end
```

ChefSpec

ChefSpec

```
# spec/spec_helper.rb
require "chefspec"
require "chefspec/policyfile"

# Policyfile.rb
name "cookbookname"
run_list name
default_source :community
cookbook name, path: "."
```

Runner

```
subject do
  ChefSpec::SoloRunner.converge("name")
end
```

Basics

```
describe "myrecipe" do
  subject do
    ChefSpec::SoloRunner.converge("myrecipe")
  end

  it { is_expected.to … }
end
```

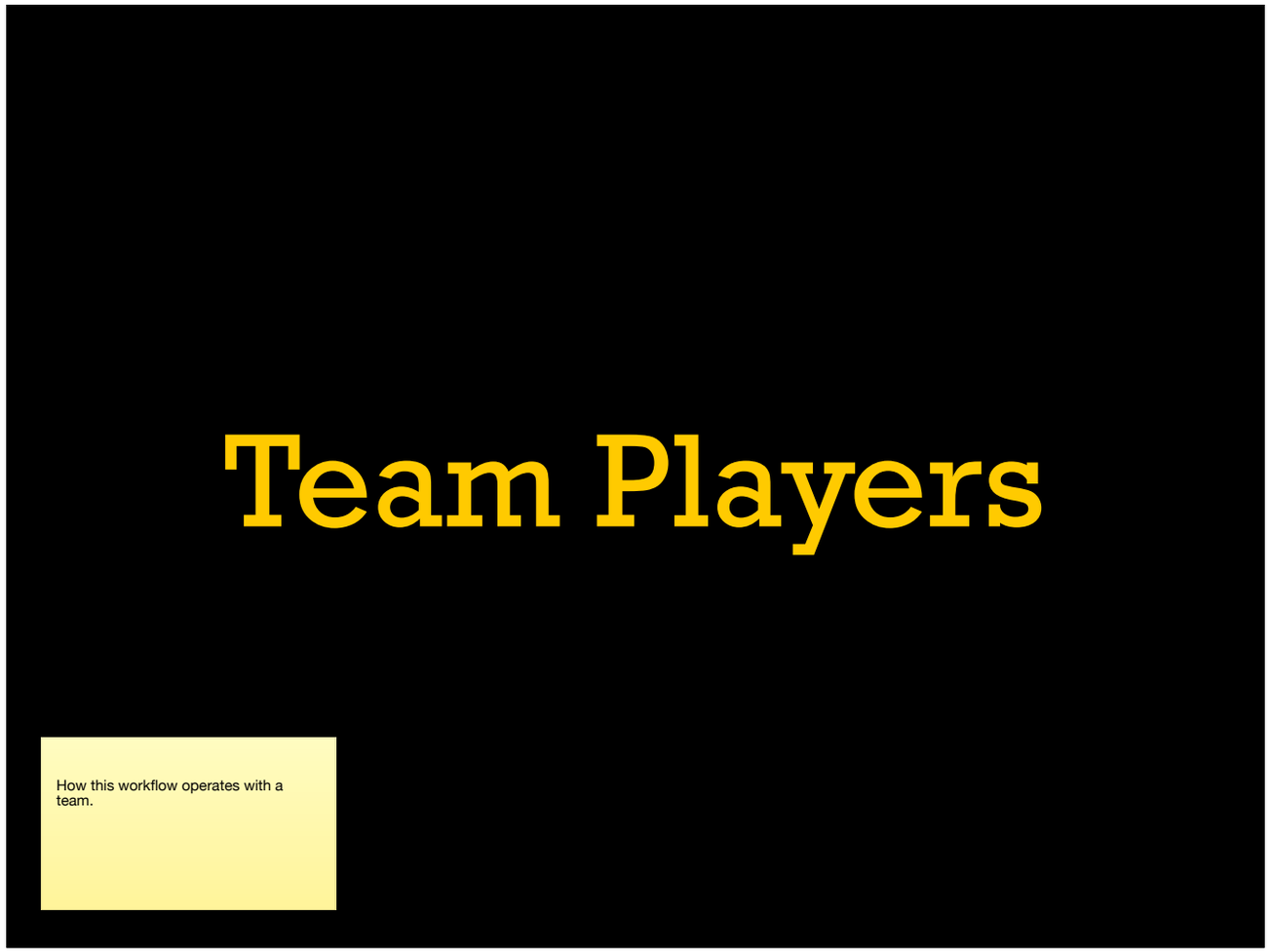
Matchers

```
…to ACTION_RESOURCE(NAME)

…to install_package("nginx")
…to create_user("myapp")
```

With

```
.with(prop: val, prop: val)
```

```
install_package("nginx").with(version: "1.2")
create_user("myapp").with(group: "nogroup")
```



How this workflow operates with a team.