# Contributing to Dpl

## Table of Contents

\* [Resources](#resources)

\* [Navigating the Codebase](#navigating-the-codebase)

\* [Lifecycle of the Deployment Process](#lifecycle-of-the-deployment-process)

\* [Deployment Tooling](#deployment-tooling)

\* [Runtime Dependencies](#runtime-dependencies)

\* [Unit Tests](#unit-tests)

\* [Runtime Dependency Installation Tests](#runtime-dependency-installation-tests)

\* [Integration Tests](#integration-tests)

\* [Testing Dpl Branches or Forks on Travis CI](#testing-dpl-branches-or-forks-on-travis-ci)

\* [Code Conventions](#code-conventions)

\* [Naming Conventions](#naming-conventions)

\* [Updating the README](#updating-the-readme)

Dpl is a central component in Travis CI, and has been around for a long time.

This library always has been a community effort first. There probably is not a

single person in the world who is very familiar with all deployment providers

supported by Dpl.

\*Thank you all for this!\*

This document is for you if you are looking to contribute to dpl, be it by

adding a new deployment provider, fixing a bug, or adding a new feature.

Dpl has a [code of conduct](CODE\_OF\_CONDUCT.md), please follow it in all

interactions with the project.

Dpl is written in Ruby, and we assume that you familiarize yourself with our

documentation as much as needed.

## Resources

Hopefully helpful resources are:

\* This [document](CONTRIBUTING.md)

\* The [dpl README](README.md)

\* The [dpl API docs](https://www.rubydoc.info/github/travis-ci/dpl) on rubydocs.info

\* The [cl README](https://github.com/svenfuchs/cl/blob/master/README.md)

## Navigating the Codebase

All provider specific classes live in [dpl/providers](lib/dpl/providers).

These represent the CLI commands that are executed when the command line

executable `dpl` is run with a given provider name as the first argument.

Each provider is a subclass of `Dpl::Provider`, which is defined in

[dpl/provider.rb](lib/dpl/provider.rb). The provider base class itself

subclasses from `Cl::Cmd`, so it represents an executable sub command of the

main command `dpl`.

For instance, the command `dpl s3 --bucket bucket` instantiates and runs the

provider class [S3](lib/dpl/providers/s3.rb).

The class `Cl::Cmd` contributes the command line options parser, and its

class level DSL. Please see the [cl README](https://github.com/svenfuchs/cl/blob/master/README.md)

for this DSL, and the [S3 provider](/lib/dpl/provider/s3.rb)

for an example how dpl uses it.

The class `Dpl::Provider` adds, amongst other things, the order of stages

(methods) that make up the deployment process:

\* `init`

\* `install`

\* `login`

\* `setup`

\* `validate`

\* `prepare`

\* `deploy`

\* `finish`

Implementors of concrete provider classes may or may not choose to implement

any of these instance methods according to their needs, and semantics of their

tooling and service providers. Please refer to [Dpl::Provider](/lib/dpl/provider.rb)

for details.

The DSL that is used to declare features, dependencies, environment integration

etc. on the concrete provider classes is defined in the module

`Dpl::Provider::DSL`, in [dpl/provider/dsl](/lib/dpl/provider/dsl.rb).

Also of interest is [Dpl::Ctx::Bash](/lib/dpl/ctx/bash.rb),

the Bash execution context, that runs shell commands, installs dependencies

etc. (while the `Test` context class is used for testing in order to keep your

development machine clean and safe when you run tests locally).

```

lib

??? dpl

??? assets # Stores larger shell scripts

??? ctx

?   ??? bash.rb # Bash execution context

?   ??? test.rb # Test execution context

??? provider.rb # Base class for all providers

??? provider

?   ??? dsl.rb # DSL for defining providers

?   ??? example.rb # Generating example commands for help output

??? providers

   ??? anynines.rb # Concrete providers

   ??? atlas.rb

   ??? azure\_webapps.rb

   ??? bintray.rb

   ??? bitballoon.rb

   ??? ?

```

## Lifecycle of the Deployment Process

When a provider class is instantiated and run it will go through a number

of stages that make up the deployment process.

These are documented in [dpl/provider.rb](/lib/dpl/provider.rb).

If you are adding a new deployment provider please familiarize yourself with

this lifecycle.

Feel free to pick and interpret these stages according to the needs and

semantics of the service provider you are adding. By no means do all of these

stages have to be filled in or implmented. The `Provider` base class checks for

these methods, and runs them, if present, so that implementors can choose

semantically fitting names for their providers.

## Deployment Tooling

If you are adding a new deployment provider please choose the tooling you are

going to use carefully.

Dpl is a long lived library, and it has outlived many tools that once were

supported, and no longer are. Thus tooling stability is a major concern for

this project.

Ideally use official CLI tooling supported by the company who's service

provider you are about to add. Often, such CLI tools can be installed via

standard package managers, or manually downloaded using `curl` and installed

with a few simple Bash commands.

Such CLI tooling is preferrable over Ruby gem runtime dependencies as they can

be executed in a child process, and won't introduce any dependency resolution

problems later on.

If no such CLI is available, or it does not look well supported, and your

provider implementation needs to talk to an external HTTP API then please consider

using [Net::HTTP](https://ruby-doc.org/stdlib-2.6.3/libdoc/net/http/rdoc/Net/HTTP.html)

from Ruby's standard library.

If you absolutely have to rely on a runtime Ruby gem dependency, such as a

provider client implementation, please only do so if the gem is supported by

the respective company officially. We may choose to reject including runtime

dependencies that do not look stable or widely supported.

## Runtime Dependencies

Runtime dependencies can be declared on the provider class using the

[DSL](lib/dpl/provider/dsl.rb).

In the case of APT, NPM, and Pip dependencies these will be installed via

shell commands at the beginning of the deployment process.

Ruby gem dependencies will be installed using Bundler's [inline API](https://github.com/bundler/bundler/blob/master/lib/bundler/inline.rb),

at the beginning of the deployment process, so they are available in the same

Ruby process from then on.

## Unit Tests

`Dpl` uses [RSpec](https://github.com/rspec) for tests. The specs reside in

`spec`, and each provider class has a corresponding file

`spec/dpl/providers/\*\_spec.rb` to hold tests.

Provider tests should be implemented on an input/output acceptance level, as

much as possible.

They use a [Ctx::Test](blob/masterlib/dpl/ctx/test.rb) execution context in

order to avoid running actual shell commands, or actually installing

dependencies at test time. There are custom [RSpec matchers](spec/support/matchers)

in place that help with making assertions against this execution context.

If your provider has to talk to an external HTTP API then ideally use

[Webmock](https://github.com/bblimke/webmock) to stub external requests. If by

any means possible try to avoid mocking or stubbing Ruby client classes (this

is not always possible, but should be considered).

### Running Unit Tests Locally

You can run the unit test suite locally as follows:

```

bundle install

bundle exec rspec

```

In order to execute tests only for a certain provider you can run:

```

bundle exec rspec spec/dpl/providers/[provider]\_spec.rb

```

In order to execute a single test or group of tests add a line number like so:

```

bundle exec rspec spec/dpl/providers/[provider]\_spec.rb:25

```

These tests can be run safely on any development machine, anywhere.

## Runtime Dependency Installation Tests

We additionally run tests that exercise runtime dependency installation on

Travis CI.

These live in [.travis/test\_install](.travis/test\_install). It is not

advisable to run these tests outside of an ephemeral VM or container that can

be safely discarded, as they are going to leave various artifacts around.

## Integration Tests

In order to ensure proper integration with the service providers supported

we also periodically run a test suite that exercises actual deployments to

these providers.

These tests live in [.travis/providers](/.travis/providers), and the are

triggered using the script [trigger](/.travis/trigger).

An integration test consists of:

\* A setup script that creates an application (or artifact) to deploy (or

upload).

\* A YAML config snippet that configures and triggers the deployment as part of

a build on Travis CI.

\* A test script that tests if the deployment was successful.

For example:

\* [github-pages/prepare](/.travis/providers/github-pages/prepare)

creates a minimal Git repository that serves an `index.html` on GitHub Pages in a temporary directory.

\* [github-pages/travis.yml](/.travis/providers/github-pages/travis.yml)

configures the build to use Dpl 2.0 to deploy this repository to GitHub Pages.

\* [github-pages/test](/.travis/providers/github-pages/test)

tests if the deployment was successful.

The tests can be run on Travis CI individually, or combined, by triggering a

build via our API, using the script [.travis/trigger](/.travis/trigger).

This takes a provider name as an argument, and requires a Travis CI API token.

For example, this triggers a build that executes the GitHub Pages test on

Travis CI:

```

.travis/trigger github-pages --token [token]

```

The token can also be set as an environment variable:

```

export TRAVIS\_API\_TOKEN=[token]

.travis/trigger github-pages

```

The `trigger` script accepts multiple provider names as arguments. If no

arguments are given then tests for all providers will be run.

### Integration Test Configuration

In the build config YAML snippet make sure to use the branch of your fork for the

deployment tooling, and allow the deployment to run on your branch:

```yaml

deploy:

- provider: [name]

edge:

source: [your-login]/dpl

branch: [your-branch]

on:

branch: [your-branch]

```

Ideally use credentials for an isolated account on the service you are deploying to.

This is generally good practice, and way you can hand things off to someone else.

In order to get things working encrypt the credentials against your fork, and

add them to the build config YAML snippet. If you are in the root directory

of your fork then this command should do the trick:

```

travis encrypt password=[password]

```

If you do not have the `travis` CLI installed you can install it using:

```

gem install travis

```

When you add encrypted credentials to the build config YAML snippet also add a comment

that allows others to identify the account used. E.g:

```yaml

deploy:

- provider: pages

github\_token:

# personal access token with repo scope on the account [name]

secure: "[encrypted token]"

```

Open a pull request. In order for us to merge your test, and get it working on

our repository you will need to re-encrypt the credentials against

`travis-ci/dpl`, like so:

```

travis encrypt -r travis-ci/dpl password=[password]

```

Whatever minimal deployment you can get working is be a great contribution.

Even if for some reason it proves hard to test the deployment in an automated

fashion, but you have a successful deployment that can be verified manually,

please still open a pull request, and talk to us. Any test is better than no

test.

## Testing Dpl Branches or Forks on Travis CI

It is possible to test a new deployment provider or new functionality of dpl on

Travis CI. In order to do so, add proper configuration on the `edge` key to

your `.travis.yml` like so:

```yaml

deploy:

provider: [name]

edge:

source: [github-handle]/dpl

branch: [branch]

on:

branch: TEST\_BRANCH # or all\_branches: true

? # rest of your provider's configuration

```

This builds the `dpl` gem on the Travis CI build environment from your

repository, on the given branch. Then it installs the gem built from this code

base, and uses it to run your deployment.

When submitting a pull request, please be sure to run at least one deployment

with the new configuration, and provide a link to the build in your pull

request.

## Code Conventions

Dpl does not follow any strict code styleguide.

Please take a look around other providers, and try to follow a similar code

style to what you find.

Try to use the [DSL](/lib/dpl/provider/dsl.rb) as much

as possible. It keeps the code declarative and readable, so that people not

familiar with Ruby or programming in general can still follow it, and make

sense of it.

If you find yourself trying to achieve something that should be, but is not

supported by the DSL please [open an issue](https://github.com/travis-ci/dpl/issues/new)

about it.

If you are rather unfamiliar with Ruby, and have trouble following our code

style then please submit your pull request anyway, or get in touch, so we can

help.

## Naming Conventions

Dpl uses constant names following Ruby naming conventions. I.e. constant

names use `CamelCase`, and they live in files named in `snake\_case.rb`.

If you pick such names for a new provider please try to follow these

conventions.

Real world service provider or company names do not always translate to such

conventional Ruby names one-to-one. That is ok, they don't have to. These Ruby

constant names are representations of real world service and company names in

Ruby code.

Other Ruby libraries often (not always) follow a similar thinking. E.g.

even though Amazon Web Services brand name is `AWS` the module name

they chose in their [aws-sdk](https://github.com/aws/aws-sdk-ruby) is

`Aws`, not `AWS`.

## Updating the README

The [README](/README.md) is generated from a

[template](/lib/dpl/assets/dpl/README.erb.md).

In order to update the README please edit the template, and run:

```

gem install ffi-icu

bin/readme > README.md

```