# Getting Involved

## Introduction

First off, high five for coming to visit this page. You are my new hero.

New contributors are always welcome, when it doubt please ask questions. We strive to be an open and welcoming community. Please be nice to one another.

### Coding

\* Pick a task:

\* Offer feedback on open [pull requests](https://github.com/fog/fog/pulls).

\* Review open [issues](https://github.com/fog/fog/issues) for things to help on.

\* [Create an issue](https://github.com/fog/fog/issues/new) to start a discussion on additions or features.

\* Fork the project, add your changes and tests to cover them in a topic branch.

\* Commit your changes and rebase against `fog/fog` to ensure everything is up to date.

\* [Submit a pull request](https://github.com/fog/fog/compare/)

### Non-Coding

\* Offer feedback on open [issues](https://github.com/fog/fog/issues).

\* Write and help edit [documentation](https://github.com/fog/fog.github.com).

\* Translate [documentation](https://github.com/fog/fog.github.com) in to other languages.

\* Organize or volunteer at events.

\* [Donate](https://www.gittip.com/geemus/)

\* Discuss other ideas for contribution with [geemus](mailto:geemus+fog@gmail.com).

### Changes

Before we go into more detail, please be aware that Fog is going through some growing pains right now. A lot of things are moving right now, and things may not be documented as well as we would like.

It might be wise to check out the [Roadmap](https://github.com/fog/fog/wiki/Roadmap) and [Design Document](https://github.com/fog/fog/wiki/fog-design-document) in the Wiki for more information.

For right now, there are three major things in motion:

#### Meta-gem movement

Fog is moving from a monolithic gem structure to a meta-gem structure, where code for specific providers will be housed in `fog-provider` gems. Extracting provider code is an ongoing process, and is also causing some performance issues.

Relevant issues include:

- General: [#1253](https://github.com/fog/fog/issues/1253), [#2602](https://github.com/fog/fog/issues/2602), [#2681](https://github.com/fog/fog/issues/2681), [#2877](https://github.com/fog/fog/issues/2877)

- Performance: [#3430](https://github.com/fog/fog/issues/3430)

#### Testing

In the past, fog has used the [Shindo](https://github.com/geemus/shindo) test framework, but is moving toward using `minitest` instead. Furthermore, the mocking structure fog has used to test in the past is undergoing some scrutiny and changes.

Relevant issues include:

- General: [#1266](https://github.com/fog/fog/issues/1266)

- Shindo -> minitest: [#2630](https://github.com/fog/fog/issues/2630)

- Mocking: [#1252](https://github.com/fog/fog/issues/1252), [#1266](https://github.com/fog/fog/issues/1266) starting about halfway down, [#1418](https://github.com/fog/fog/issues/1418), [#2629](https://github.com/fog/fog/issues/2629)

## Overview

\* Organize your patches by keeping all related changes together in a topic branch.

\* Rebase your branch against master before submitting a pull request (and squish any 'oops' or work in progress commits).

\* We support several rubies! Make sure your changes will work for all the Rubies in `.travis.yml`

\* Submit changes as pull requests describing what the changes should cover and referencing issues (if any).

\* Use 'tags' in your commits to indicate the scope, so things like `[aws|compute] fixed something`.

\* Write and run tests. Tests should follow through usage workflows and ought to pass both with mocking on and off.

## Deep dive

Now then, some of the what makes it tick and why. For simplicity let's pretend you want to implement a new service, from scratch. I will walk through the requisite pieces and important things to keep in mind as you go.

But, before I dive too deep, I'll leave you with an out. Other great ways to contribute are fixing bugs, writing documentation or helping port other projects to use fog. That way everybody wins!

## The Service

First and foremost you'll need to create a service, which should start from something like:

module Fog

class TheService < Fog::Service

requires :necessary\_credential

model\_path 'path/to/models'

collection 'name\_of\_collection'

model 'name\_of\_model'

request\_path 'path/to/requests'

request 'name\_of\_request'

class Mock

include Collections

end

class Real

include Collections

end

end

end

### Highlights:

\* we segregate between real and mock so it is easier to add stuff to one or the other later.

\* this is where any shared stuff will go, like making/signing requests

## Requests

The next thing to bite off are the requests. fog is all about making cloud services easier to use and move between, but requests are not where this happens. Requests should map closely to the actual api requests (you should be able to directly reference the api docs and vice versa). In particular, try to keep the output of any data parsing as close to the actual format as possible. This makes implementation and maintenance much easier and provides a solid foundation for models to build nice things on top of. I generally end up working on stuff to get/list details first and then filling in create/destroy pairs and other requests.

You start with something like this:

<pre>

module Fog

class TheService

class Real

def request(\*args)

end

end

class Mock

def request(\*args)

Fog::Mock.not\_implemented

end

end

end

end

</pre>

### Highlights:

\* You should define the method twice, once for the real implementation and once for mocked (they should take the same arguments).

\* The mock versions should just start out by raising a not implemented error, you can come back and fill this in later.

\* The real version should make a request, probably by a method defined on the real class in the service you defined earlier.

\* Each request should either return an Excon::Response (with a parsed body where appropriate) or raise an error.

## Tests

Now would be a good time to write some tests to make sure what you have written works (and will continue to). I've tried a couple variations on testing in the past, but have settled on consolidated lifetime testing. These vary enough that its hard to give a single simple example, but you can see many examples in [tests/aws/requests/compute](https://github.com/fog/fog/tree/master/tests/aws/requests/compute).

### Highlights:

\* Reuse the same objects and take them through their whole life cycle (this is much faster, and most of the time if one portion fails the others would anyway).

\* Test the format of the output to ensure parsers match expectations (check the provider's api docs) and that mocks return matching data.

\* Test common failure cases and their behavior, you'll need to know how the service acts in these cases to make better mocks.

## Models

You could also skip to the mocks here if you wanted, but I usually find the more time I spend working with the service the easier it is to build mocks. The models are the real pay dirt, you have slogged through low level requests that map to the provider api and now you want a nice interface. This is where models and collections come in. Collections provide access to lists of data on the provider and for creating new objects. Models represent the individual objects.

If you know which object you'd like to represent you should start with the collection. When naming, please refer to the names that have been chosen for other services. I haven't standardized all nouns yet, but a few are already shared (Flavor, Image, Server)

An example servers collection:

require 'fog/collection'

require 'fog/theservice/models/server'

module Fog

class TheService

class Servers < Fog::Collection

model Fog::TheService::Server

def all

# get list of servers

load(data) # data is an array of attribute hashes

end

def get(identity)

# get server matching id

new(data) # data is an attribute hash

rescue Excon::Errors::NotFound

nil

end

end

end

end

### Highlights

\* First make an accessor in the Collections model so it will be included in Real and Mock.

\* `#model` will take a reference to the class that will be instantiated to represent individual objects.

\* `#all` should get a list of servers from the provider and pass an array of attribute hashes, one per server, to load.

\* `#get` should take an identity reference and instantiate a new model object with an attribute hash returned from the remote server, or return nil of no such object exists.

Models handle remapping attributes into friendlier names and providing the rest of the interface.

An example model:

require 'fog/model'

module Fog

module TheService

class Server << Fog::Model

identity :id

attribute :state, 'StatusValue'

def destroy

requires :identity

connection.destroy\_server(identity)

true

end

def ready?

state == 'running'

end

def save

requires ...

connection.create\_server(options)

true

end

end

end

end

### Highlights

\* `#identity` captures the id/name that the objects are identified by and takes the same arguments as attribute.

\* `#attribute` takes the name to make a variable available as and one or more aliases that parsers/requests will return this value as.

\* `#destroy` will require the identity of the model and should destroy it and return true.

\* `#ready?` should return whether the object has finished being initialized (where appropriate).

\* `#save` should take any required objects and instantiate the object on the provider's service.

\* These models just rely on underlying collections and requests, so it should not be necessary at this level to distinguish between Real and Mock methods.

## Mocks

Mocks provide a powerful tool for users of fog to experiment with their implementations much more quickly and without incurring costs. I usually save these for last, as implementing the requests and models provide some necessary context to finally put the mocks together. Your services mock class should have a data method that will return mocked data like so:

module Fog

module TheService

class Mock

def self.data

@data ||= Hash.new do |hash, key|

hash[key] = {}

end

end

end

end

end

The keys in this hash should represent a unique identifier of the user accessing the data and the value assigned should contain any default data that a new user might have. Any implemented mock requests should then return data retrieved from here or raise an error.

For instance:

module Fog

module TheService

class Mock

def destroy\_server(server\_identity)

if data = self.data[:servers].delete(server\_identity)

response = Excon::Response.new

response.status = 202

response.body = data

response

else

raise Fog::TheService::NotFound

end

end

end

end

end

### Highlights

\* Mock requests should return the same type of data as an already parsed real response or should return the same error as a real problem.

\* By mocking at this low level, higher level functions are automatically mocked out for you.

\* The extra rigorous tests related to output formatting and error messages should help keep you honest, and each should pass in both mocked and unmocked modes.

## Summary

That provides a lot more detail than you will probably need right away, but hopefully you can refer back to different sections as you need them. If you have any questions send me a github message or email me (address is on my profile). You should always start development by creating your own fork. When you feel confident about your fork, send me a pull request. Be forewarned that I may edit some things before it gets to master, but I'll do my best to take care of this in a timely manner.

Thanks again for your interest and let me know if there is anything else I can do to help.