API/Provider Writing Practices

This section covers practices that are encouraged or discouraged. Reading this section will help guide development in a way that keeps the tenets of jclouds in-tact, and review comments to a minimum.

- **Start** In many cases, we are already practicing things mentioned as Start, just not consistently. Start is mentioned here to keep people on the same page.
- **Stop** Stop means stop propagating, ex. things we should change, but not necessarily backport. For example, we should not employ practice named Stop in any labs api or anything promoted out of labs. However, if something mentioned as Stop exists in a current non-labs api or provider, we should more carefully plan, and possibly allow `Stop` practice to persist for sake of consistency with other code in that api
- **Undo** Undo are unconditionally something we should work on, in any maintained version of jclouds, as these things limit our ability to progress the entire codebase.

General Practice

(Stop) silo'ing fixes!

If you encounter a small problem you need fixed for a downstream api or provider, find some scope to fix uniformly. For example, other classes in the same package, or other methods in the same Api.

Fixing one and leaving the other cleanup for others to address doesn't respect the time of reviewers, or those who merge code at release time. It takes the same time to review a tiny update vs 4 tiny updates. Moreover, a user (or developer) would be unpleasently surprised to find only 1 of 3 edge cases fixed.

When doing bug fixes, or tech-debt removal, can you make it worth it? Could you fix all similar broken windows, or at least more than one? If you can afford time to fix a bug, would spending an extra hour to fix similar ones be tolerable?

Bottom-line, we are all in this together.

(Stop) using Set for everything!

List is the preferred default for a collection, as it maintains order, provides random access, and is a cheap data structure. We've in the past used Set for all collections. Only use Set where it is a very-specific domain concern. Otherwise, use List.

(Undo) putting `@Singleton` on everything!

@Singleton indicates you are holding shared state not given to you in the constructor. Putting @Singleton on everything waters down the concept and slows the runtime of jclouds.

(Stop) Using guava for user-visible types, especially simple ones.

Guava limits our compatibility with users, and limits a future of platforms such as Android. Guava conflicts are very often brought up as reasons why users cannot use latest jclouds. Instead of representing things like `Content-MD5` as a `HashCode`, use the form actually returned by the server (usually a String). Not only does this help with compatibility, but it also prevents us from spending more time re-serializing String->type->String, as well the class of errors that are possible attempting to do so. Stop using `Optional` for example. Just use nullable as that doesn't conflict with Java 8 Optional, Scala Option, etc. Internal use of guava is fine. We should aim to be able to shade jclouds and avoid conflicting on guava completely.

Api interfaces

(Stop) Cargo-Culting aka blindly copy/pasting annotations

There are numerous cases where people have copy/pasted @Fallback annotations which either don't make sense, don't make sense for the api, or actually cause problems. For example, I've seen fallbacks to false on methods that return void, or a list. Also, we fallback on 404 to list methods whose documentation say 404 is not possible. Read the api documentation before making a fallback. If you have a fallback, test it.

Value (Domain) Types

(Start) Using Auto-Value for all value types

Auto-Value covers aspects such as null-checking, hashCode and toString. Use of auto-value reduces the work needed to review and maintain code, and avoids guava incompatibility with things that generate hashCode and toString.

(Stop) Declaring top-level types for subordinate classes.

If you have a status object that's only valid for Server, make it a static inner class of Server. Similarly for single-field types. This makes it easier to understand the top-level, most important types, and prevents us from having to use naming conventions like ServerStatus to disambiguate.

(Undo) Writing builders for output-only types

Just adds work to us and no value to users. Use the canonical factory for that type and name the parameters in your tests.

(Continue) Writing builders for user-input objects with more that 3 parameters.

(Stop) Writing complex builders

Builders will be generated in the future. Stop writing builders with add to collection, convert type functionality, or null checking functionality. Rely on factory methods or auto-value for things like this. Ex. just set the field and continue.

Tests

(Continue) Writing MockWebServer (MockTests)

MockWebServer is used to test that our Api interfaces create expected http requests and parse responses properly. It decouples us from accidentally asserting true-true by using a real http endpoint in tests. Yes, eventhough MockWebServer is named like that, it opens a port and is a real http server!

Use MockWebServer to test all Apis (under the feature package) and any Views (like ComputeService and BlobStore).

MWS basic setup

* Until we switch to JUnit (where we'd use test rules), use a base test class. This should setup a field for the server and tear it down, and hold assets that are shared across tests. This makes the subtypes easier to read and also reduces errors. [Example in GCE](https://github.com/jclouds/jclouds-labs-google/blob/master/google-compute-engine/src/test/java/org/jclouds/googlecomputeengine/internal/BaseGoogleComputeEngineApiMockTest.java).
* Only use MockWebServer's port; Do not use its host. Test hosts in the cloud often botch hostnames. Just point anything that uses the mock server's url. Ex. `"http://localhost:" + server.getPort() + path`.

MWS advanced setup

MockWebServer can do many things, including listen on ssl, spdy, you can run multiple ones to test auth separately, etc. This section covers some non-trivial test concerns.

Handling links in server responses

When your api returns links, you need to correct these to point to the mock server. Otherwise, a compound operation will first go to the Mock server, then go to the real one!

The easiest way is replace well-known base urls with one from the mock server in the base test class.

```
protected String stringFromResource(String resourceName) {

try {

return toStringAndClose(getClass().getResourceAsStream(resourceName))

.replace("https://www.googleapis.com/compute/v1/", url("/"));

} catch (IOException e) {

throw propagate(e);

}

}
```

MWS do's and don'ts

* Do test annotations you add, especially custom ones. If you add a `@Fallback` test it. Do not add test default fallbacks as this clutters code.
* Do test response parsing, but don't repeat yourself. If you are using a default response parser like json, and you've already a "parse test", just check that the response is the correct type and not null. All you need to do is access the result, as that would imply a ClassCastException if it were the wrong type. Ex. `assertTrue(aggregatedList.instances().hasNext());`
* Test anything "interesting" carefully. Particulary with pagination, test the second page, not just the first. It is "ok" to pin this to a separate class while working on other things, but at least have one class that tests pagination using multiple http responses. For example, in [GCE](https://github.com/jclouds/jclouds-labs-google/blob/master/google-compute-engine/src/test/java/org/jclouds/googlecomputeengine/internal/ToIteratorOfListPageExpectTest.java).
* Don't do multiple "api tests" in the same method. Ex. doing a list followed by a get in the same test method makes the test harder to understand than making different tests. The only reason to load multiple responses is if there's an implicit extra request due to authentication, or there's a continuation like pagination, or you are testing a complex type such as `BlobStore` which needs multiple http requests for the same command.

(Undo) Testing default fallbacks

Tests that cover fallback annotations you didn't add clutters the code, and drowns out important test cases. Only test for code you added.

(Undo) Subclassing BaseRestApiExpectTest

"Expect Tests" were created before we knew about MockWebServer. This is a highly guice-ified, custom way of loading http requests and responses. Due to how resources are loaded, it is possible to get NPEs unless you add a testName attribute to every subclass. Moreover, debugging these has proven extremely difficult. Since ExpectTest mocks an http driver, we have missed bugs in the past due to implementation concerns in ExpectTest. Finally, ExpectTest uses jclouds internals, such as our http request and response objects. As such, any major refactoring to core implies rewriting *all* expect tests. We must stop writing these, and please start converting them.

(Undo) Subclassing BaseRestAnnotationProcessingTest

RestAnnotationProcessor and friends expose the internals of jclouds aging core. It is not possible to simplify core while tests depend on internals. Convert BaseRestAnnotationProcessingTest derivatives to MockTests or delete them.

Pagination

For each method that returns paginated results, you will need two methods in the api: one without parameters that returns the entire list, and the other one, with the pagination parameters, that is able to return a single page. In jclouds:

- The first one, returning a single page, must return an `IterableWithMarker<T>`, which is just an Iterable with a marker that points to the next page.
- The second one, the entire list, must return a `PagedIterable<T>`, which is an Iterable that has a single page (an `IterableWithMarker<T>`) and knows how to get the next page given the next page marker.

Having this in mind, you need to figure out two things:

- How to build the `IterableWithMarker<T>` so it contains a marker with all the info required to fetch the next page.
- How to build the `PagedIterable<T>` to automatically fetch pages given a marker.

Building the first one should be pretty straightforward. You just need to create a class that extends the`IterableWithMarker<T>` abstract class and make sure the returned marker has all the info to fetch the next page. In your example, the "meta" fields could serve this purpose. Once you have your implementation, you just need to create a ResponseParser that builds your class given an HttpResponse. That's what theParseImages class in Glance does.

Next thing you have to do, is build a `PagedIterable<T>` given an `IterableWithMarker<T>`. Since the APIs always return a single page, in the end, the HttpResponses for the "listAll" and "listSinglePage" methods will be the same. That's why most "listAll" methods use the same response parser than above, and then use a transformation function to transform the resulting `IterableWithMarker<T>` into a `PagedIterable<T>`(see the ImageApi annotations).

So, that `ToPagedIterable` is just a function that transforms an `IterableWithMarker<T>` into a`PagedIterable<T>`. This can be done in many ways, depending on the API. This functions usually build the iterable using the helper methods in the PagedIterables class, such as the advance one. If you take a look at that method you'll see that the PagedIterable is build given an `IterableWithMarker<T>` and an advancing function. That advancing function transforms an `Object` to an `IterablewithMarker<T>` where the Object is the current marker to the next page, and the result should be the next page. That advancing function will only be invoked if the marker to the next page is present.

So, in the end, you just need to create a function to transform an `IterableWithMarker<T>` into a`Pagediterable<T>`. How you build it is up to you. Here are a few examples you can take a look at, to get the whole idea:

- In Abiquo, for example, each page returns a link to the next one, so there is a generic function that simply performs a GET call to fetch the next page, given the marker (the link).
- In Glance, there is the ParseImages#ToPagedIterable that extends an existing helper class to build a new API call. The class it extends populates the caller arguments to the function, but this might not be the best example in your case and could bring confusion.
- There is a helper class that you could also use (and perhaps this is what I'd recommend if there is no generic way to do pagination for all API calls). You could create a function that extends the existingArgsToPagedIterable. That class already has the common logic and you only have to implement its abstract method. That method must return a function that returns the next page given an`IterableWithMarker<T>`, but also receives as a parameter the list of arguments (the arguments of the invoked java method) used in the previous API call, which can be useful to build the call to the next page. You can take the ParseImages#ToPagedIterable as an example to get the idea, but apply that to the ArgsToPagedIterable class.