IT Ruby SDK Family Tree

It has been mentioned to me at various times that our Ruby code base is difficult to contribute to when you don't know where to start, or how the pieces fit together. This document hopes to describe the logic of the codebase layout, deconstruct the 12+ year history (as of this writing) of its individual components, and hopefully make it feel significantly less intimidating to anyone newly exploring. This is **not** an exhaustive step by step, just a high level overview of component responsibility.

It's gonna be a read, buckle in!

Table of Contents

1. Overview
2. Base Utilities
    1. Configurability
    2. Loggability
    3. Pluggability
    4. Inversion
3. LDAP
4. PostgreSQL
5. RabbitMQ / AMQP
6. Web Framework
7. Putting it Together
8. Links

Overview

The overarching theme here is *separation of concerns*. Each component layers more logic on top of its parent via Class Inheritance. Ruby doesn't have multiple parent inheritance, but via Ruby's concept of Mixins, we can include and re-use arbitrary behaviors across any number of classes.

Using this architecture, we can segment code into logical pieces for maintainability and ease of testing, separate public release from internal/proprietary business logic, then stack them back together to form a cohesive whole. This allows us to package and install just what is needed for a given application, expressing exact dependencies between specific code bases. This can provide the outward impression of a single application environment – without any monolithic backends or code repetition.

Base Utilities

The following gems are public release, most sponsored by LAIKA. They provide a Ruby ecosystem for

performing routine duties, and are designed to work either individually or in lock-step with each other.

Almost all of the other software listed throughout this article utilizes these modules one way or another, usually in the form of a mixin. For detailed documentation, check docs for each individual module.

In practice, you don't need to require anything but the `laika` gem, as it carries all of these utilities, as well as methods to interact with default behaviors.

Configurability

This allows you to have configurations pushed into a class from a central YAML file. The simplest possible example:

```
require 'configurability'

class HappyFunBall
    extend Configurability

    configurability( :funball ) do
        setting :bounce, default: 10
    end
end

HappyFunBall.bounce #=> 10
```

Once a configuration is loaded, it is chopped up and set to all classes that have been extended with Configurability. The LAIKA base class has a loader function that defaults to a `~/.laikarc` file if present, then `/usr/local/laika/etc/config.yml`.

```
require 'laika'
LAIKA.load_config
```

Because of this, once configuration is loaded, more classes can be defined/required, and configuration doesn't need to be loaded again. Each class that is configured this way can have its own configuration key, keeping config for the entire SDK stack in a single place.

Loggability

Centralized logging for instances of Ruby classes. This allows logging levels and destinations to be defined on a per class basis – also set up via Configurability.

```
require 'loggability'

class HappyFunBall
    extend Loggability

    log_as :funball


```

```
        def initialize
            self.log.debug "I've been instantiated!"
            self.log.info "Don't touch the happy fun ball."
        end
    end
```

A configuration file that looks like the following would log HappyFunBall at a level of debug to stdout, using console colors, and everything else would be much more boring:

```
1  ---
2  logging:
3      __default__: warn STDERR
4      funball: debug (color)
```

There are a lot of additional tricks to Loggability, but we usually just spit everything out to `stdout`, and run things through `multilog` (part of the supervise package) to manage rotation and such.

Pluggability

In simple terms, this is a metaprogramming trick that we got tired of re-doing over and over again for each project. This allows you to make a parent abstract API, then define children (plugins) that implement the API specified. You can do this in ruby natively, but Pluggability adds automatic discovery and a well-known API for plugin architecture itself. You can instruct the plugins to be detected automatically for installed gems, so all the legwork of finding and requiring derivatives is handled for you.

A good example of this is in `Thingfish` (described later) – every part of that system is implemented as pluggable interfaces, so adding new Processor, Metastore, and Filestore behaviors just need to conform to the API, and can be installed as separate gems. The core `Thingfish` codebase can remain lean and mean, and the heavy lifting “I want my files stored in S3” logic can be entirely segregated and maintained.

> You may have noticed a theme in the naming conventions above. These gems are intended to soon be released as a single gem, the **ability** suite, that provides all this metaprogramming goodness (along with other stuff not mentioned).

Inversion

A templating engine that is Ruby idiomatic, easily extensible, and most importantly follows the Inversion of Control design pattern. This separates areas of concern (again, MVC anyone?), and makes testing templates easy when using mocks.

```ruby
require 'inversion'

# Make a template from a string.
#
template = Inversion::Template.new <<-TMPL
Hi <?attr user ?>.
I have these items to show you.
<?for i in items ?>
  * <?attr i ?>

<?end ?>
TMPL

# Attach various objects to the template.
#
template.user  = "Mahlon"
template.items = [ 1, 2, 3 ]

# Render it, expanding all the pieces within.
puts template.render
```

```
Hi Mahlon.
I have these items to show you.
  * 1
  * 2
  * 3
```

Template files (and new tags) are automatically discovered using Pluggability, so putting them into a gem under the data directory `templates` allows `Inversion::Template.load()` to find them regardless of where they are on the system.

Okay, so now that the base behaviors are out of the way, I can give some more concrete examples of the “layering” I was describing at the beginning of this document. Lets just use a couple of systems in deep use here at LAIKA as examples, and explain the modules in use that interact with them – and how they interact with each other.

LDAP

LDAP is the heirarchial tree that stores all things People, Hosts, Network, and a pile of other miscellanous stuff. The *L* in LDAP stands for *lightweight*, and that’s a certain kind of hilarious, because it’s only an apt comparison against its predecessor, X.500. LDAP is a monster of complexity and weird legacy behaviors, but it all boils down to a key->value attribute store for entries. It is literally the original NoSQL.

On the positive side, it is rock solid, *astoundingly* fast, scalable, and the RFCs that govern it are solid and immobile. This is boring, borrring stuff. Reliable. Perfect!

Ruby has a couple of raw interfaces to LDAP. They are all varying degrees of awful – it’s as if authors decided to learn LDAP by releasing a module to the world, marketed as being *the most awesome evar* and *made with <3*. Ugh. Rather than reinvent that wheel, we chose the least awful path, which is `ruby-ldap`. It is a C layer implementation that uses the OpenLDAP headers on your system. Unfortunately, it doesn’t implement some of the advanced stuff (server side pagination, referral following), but given

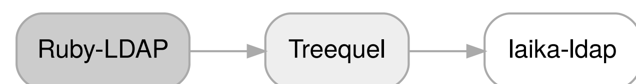
that it is essentially a C passthrough for Ruby, it does the primary job of talking to LDAP adequately, and *quickly*.

It became apparent early on that we needed a higher level abstraction for LDAP interaction – using the ruby-ldap objects and methods directly gets verbose and ugly pretty fast. There was one player in town at that time, `ActiveLDAP`, which was emulating `ActiveRecord`, a popular SQL ORM.

ActiveLDAP makes a critical architectural misstep, in that it attempts to treat LDAP as a relational environment, and trips all over itself trying. LDAP is heirarchial, remember? So, an attempt was made to cater to that core fact, using interface concepts from an ActiveRecord competitor, Sequel, called Treequel (get it?)

With this, you can create models of LDAP object classes, and have a fairly intuitive 1:1 between LDAP and Ruby Classes. (Ruby fits nicely into this, as each LDAP objectClass type is essentially a mixin for valid attributes on the entry.)

Here's the family tree. **Dark grey circles are public but not our code, light grey nodes are public but written but us, and white nodes are internal/proprietary LAIKA stuff.**



Module	Purpose
Ruby-LDAP	Low-level network communication and interface with LDAP.
Treequel	LDAP abstractions, modelling and association behaviors.
laika-ldap	LAIKA business logic, configuration, model validations.

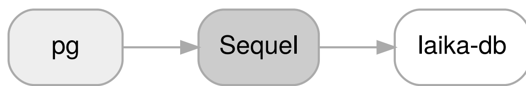
Here an example of loading my account record up and setting a description, assuming your configuration file knows the credentials to write to the LDAP master server:

```
1  require 'laika'
2  require 'laika/ldap'
3
4  LAIKA.load_config
5
6  mahlon = LAIKA::Account[ 'mahlon' ]
7  mahlon.description = "He just double clicks around"
8  mahlon.save
```

Succinct, huh? No boilerplate required, and all connection configuration is pushed to a separate YAML file.

PostgreSQL

I'm not going to spend a lot of time in this one, because it's largely identical to LDAP in terms of usage, conceptual layout, etc.



Module	Purpose
pg	Low-level network communication and interface with PostgreSQL.
Sequel	Database ORM abstractions, modelling and association behaviors. This library is really amazing, and the author is exceptionally responsive.
laika-db	LAIKA business logic, configuration, model validations.

Here an example of loading a LAIKApedia wiki page, and programmatically changing the owner.

```
1  require 'laika'
2  require 'laika/db'
3  require 'laika/pedia' # Where the wiki models are defined
4
5  LAIKA.load_config
6
7  page = LAIKA::Pedia::Page[ '/it/isg/some-weirdo-page' ]
8  page.owner = "mahlon"
9  page.save
```

The similarity in usage with LDAP here can't be understated – it's the same interface to wildly differing systems. Certainly things like searching are going to be different, but the core setups and usage all parallel each other. Nice? Nice.

RabbitMQ / AMQP

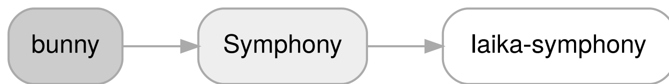
Once machine messaging reached critical mass at the studio, and RabbitMQ was the clear winner between the various available backends, some time was spent evaluating the higher-level AMQP options for Ruby. (It was definitely more promising than the LDAP landscape at the time.)

A popular one was JRuby only. Another had a heavy and fragile dependency we didn't want to introduce everywhere. We thought we had found a decent one called "Sneakers", and started moving forward with it – but quickly hit some walls after doing anything outside of the default setup. Sneakers just didn't expose enough AMQP behaviors to the user, instead catering to only the most common use cases. If you wanted to do something outside of what Sneaker supported, you were out of luck. We began writing patches to support these features under Sneakers, but architectural decisions with the library made us realize that it would take significantly less time to write our own, called Symphony. (It should be noted that since that time, Sneakers has improved considerably, but still doesn't support some behaviors we've come to appreciate.)

Symphony has solid default behaviors, understands server-side created queues vs client created, cleans up after itself with Rabbit, and most importantly, doesn't hide Rabbit/AMQP concepts from the developer. Simple and common defaults are great, but if you let Rabbit do the logical heavy lifting (offline retry, policies, dead letter queues, shovelling, etc) – Symphony just gets out of your way.. It also handles content negotiation automatically, so you can publish and receive ruby objects directly,

and the serialization is handled transparently.

A symphony worker is discovered via Pluggability, so you can include a worker in any other gem, under the gem data directory at `symphony/tasks`.



Module	Purpose
bunny	Low-level network communication and interface with RabbitMQ.
Symphony	Higher level abstractions for job workers.
laika-symphony	Symphony actually uses Pluggability to find qualified workers, which usually live in the repositories that require them. This library holds a singleton that allows easy message publishing and queue visualization.

Here's a quick example worker, that simply accepts an event called `trigger`, and publishes a new message back to the exchange with a key of `notify`. It'll automatically create a queue called `example` and hook up the event subscription from the configured exchange:

```
1  require 'laika'
2  require 'laika/symphony'
3  require 'symphony/routing'
4
5  class Example < Symphony::Task
6      include Symphony::Routing
7
8      on 'trigger' do |payload, metadata|
9          LAIKA::Symphony.instance.send_message( 'notify', "I got an event and presumably am leaping
10         return true
11     end
12 end
13
14 LAIKA.load_config
15 Example.run
```

The `run` method is a blocking call, waiting for new events. Fire up multiples of these across any number of machines, and you've got a immediate distributed work crew. When the last one exits, the queue is automatically cleaned up on the server.

Web Framework

This is a larger backstory, but I'll make it as brief as I can. There's a glut of frameworks out there, why on earth would anyone make another? Both myself and my counterpart Michael Granger had a long history with serving web content, and had both written multiple frameworks over time. We've seen first hand mod_perl environments that served tens of millions of hits a day before "facebook scale" was a

thing, and in the mid 2000s, Michael took it upon himself to translate all we learned from LiveJournal into a mod_ruby framework called *Arrow*.

Arrow took a bunch of what we were missing in LiveJournal environments and solidified that into the Ruby ecosystem. Over time however, the monolithic web application gave way to a desire for smaller, more focused handlers for specific URI paths. Microservices? Call it what you want. The idea was to carve up your URI space into individual applications, and not be locked down into any one particular technology.

So we set about to find a backend platform that supported those ideals. Rails was out of the question. Zed Shaw had just evolved his Ragel-based HTTP state machine into a new, non-ruby-centric backend called “Mongrel2” – it’s design felt perfect for what we were searching for.

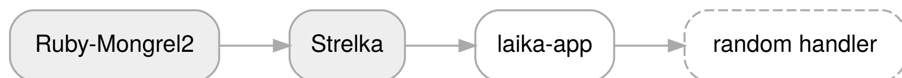
- Small C-based server that performs only the specific task required of it and nothing more
- The Ragel HTTP parser, for a proven track record of safety and security
- ZMQ for horizontal scalability
- tnetstring request/response cycle for easy handler handoff and communication
- Completely language agnostic – if a specfic portion of the URI path requires a different tool in the toolbox, it should be a relatively simple task to integrate.

Mongrel2 is not under heavy development, largely because it doesn’t need to be. It is basically an HTTP parser that wraps a request, converts it to a tnetstring, and sends it to a ZMQ socket. That’s it. What happens afterward is all in the domain of the handler – accepting a request, and providing a response.

LAIKA the company was named after a well-known dog cosmonaut. A lesser known fact – Laika wasn’t the only one. There was also a dog named *Strelka*, which means “little arrow”. Unlike Laika, Strelka returned alive. The coincidental link between the Arrow Web Framework and the company that was sponsoring this development was too good to pass up, and a Mongrel2 specific framework was born.

Strelka exposes all of it’s functionality via Pluggability (shocker), down to the request/response. Need a new authentication backend? Plug in. Log requests to elastic search? Plug in. Here’s a small portion of what Strelka provides:

- Sinatra style routing DSL for REST frameworks
- Global authentication defaults with per-route overrides
- Inversion templating integration, with look and feel layout wrapping
- Automatic content negotiations - the same URI can return different results based on HTTP `Accept` headers, the way nature intended
- Strict parameter validations with global definitions and per-route overrides, with automatic typecasting
- Easy hooks for extending it beyond what we’ve already thought of, and plugins can describe ordering dependencies



Module	Purpose
Ruby-Mongrel2	ZMQ and tnetstring parsing layer to the Mongrel2 C daemon, which beautifully handles all the HTTP parsing gruntwork
Strelka	Takes a raw Mongrel2 request object and passes it through a bunch of friendly interfaces to produce a response, following a ruby idiomatic path.

Module	Purpose
laika-app	A base Strelka handler that includes the base plugins we'll always be using, the look and feel, LDAP and PG automatic reconnection code, and LAIKA specific plugins. Essentially all the Strelka handler boilerplate without any logic.
random handler	Inheriting from `laika-app`, this represents a handler that contains the core business logic for a collection of URIs.

Inheriting from `laika-app` provides LAIKA template pathing and content negotiation. Here's a *Hello World* handler that responds with the regular theme if viewed in a browser, and with a structured payload with the proper **Accept** header:

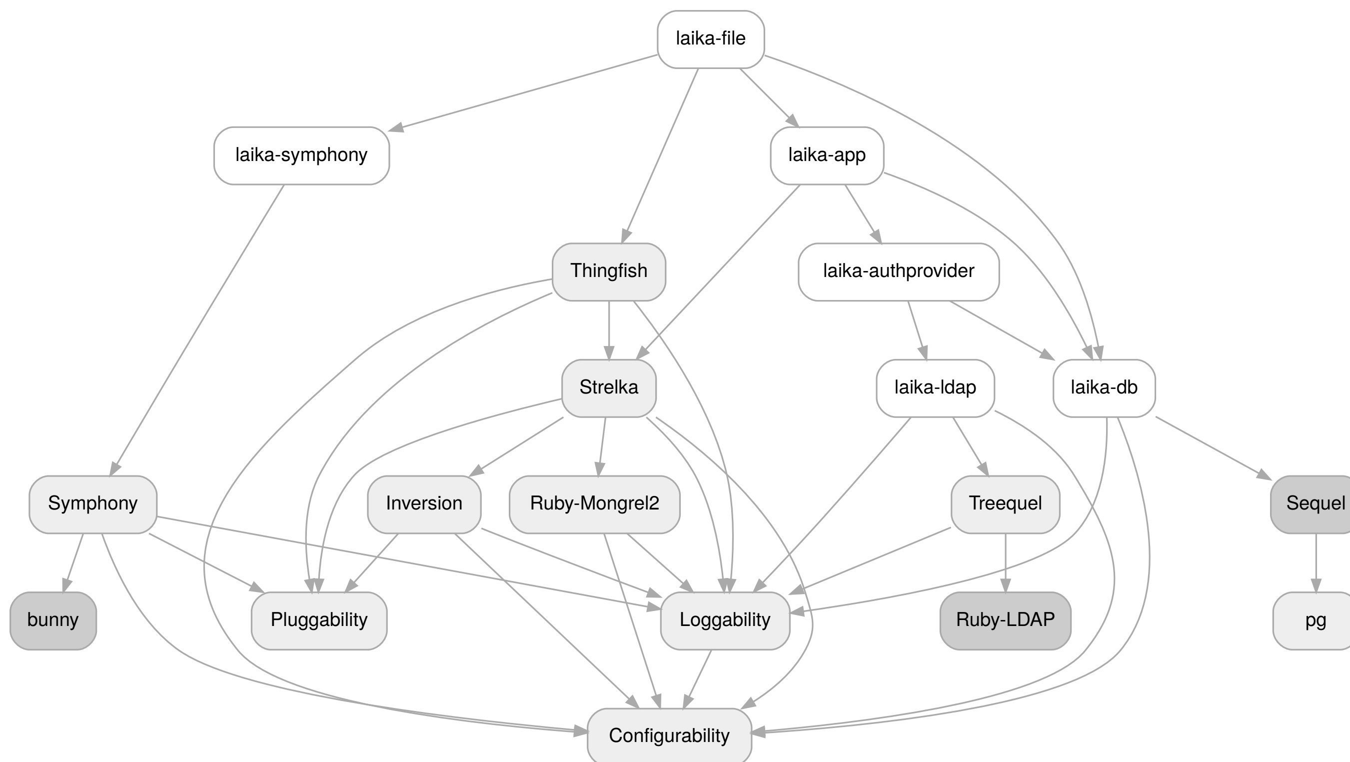
```
1  require 'laika'
2  require 'laika/app'
3
4  # An example web handler
5  class Example < LAIKA::App
6      ID = 'example'
7
8      layout 'layout.tmpl' # Main look-and-feel template
9      default_type 'text/html'
10
11     get '/' do |request|
12         response = request.response
13
14         response.for( :html ) do
15             "Hello, world!"
16         end
17
18         response.for( :json, :yaml ) do
19             { "hello" => "world" }
20         end
21
22         return response
23     end
24 end
25
26 LAIKA.load_config
27 Example.run
```

Just like the **Symphony** event handlers, starting more of these up (on the same hardware, or other machines across the network) will evenly distribute requests automatically.

Putting it Together

Ok, with the primary components behind us, here's the dependency graph for a real "random handler"

from above. This handler is called `laika-file`, and it's responsible for arbitrary file storage and retrieval for the environment. It inherits from a project called Thingfish, that's a framework using Strelka for storing, filtering, extracting metadata, and fetching files over a REST interface. Like the other stuff, `laika-file` adds specific LAIKA flavoring to it, teaching it our authentication methods and database backends, then emits events to AMQP when files are stored (for asynchronous processing.) Notes on its architecture can be found here.



laika-file advertises itself to Mongrel2 under the `/file` URI path, alongside a good number of other handlers. To end users, it appears as a single environment, and all elements of it are configured in a single place.

Hopefully this has helped demonstrate the ecosystem, and the "small parts" philosophy that drives it.

Links

- Configurability
- Loggability
- Pluggability
- Inversion
- Bunny
- Symphony
- Ruby-Mongrel2
- Strelka
- Thingfish
- Ruby-LDAP
- Sequel
- Treequel

- pg
- laika-ldap
- laika-db
- laika-app
- laika-symphony
- laika-file
- laika-authprovider

In addition, machine generated documentation for the SDK can be found here.

normal