Do it yerself, Rube!

Introduction

Hi everyone. My name is John Anderson. I've been writing code for a living for quite a long time. Ruby since 2004. Web development since HTML 3.2 was shiny and new, and enough c, c++ and java to be really, really tired of parentheses, braces and semicolons, especially the missing ones. I quite often ride motorcycles up and down steep and rocky inclines.

Da rulez: I don't have swag because my sponsor (me) also has to pay school fees. But I have box of little endorphin kicks here. And like any endorphin kick worth its weight in hormones, you have to do a little work for them. First person to shout out the name of the movie gets one.

This talk started off as “Do it yerself, sport” but we're not at a sport conference. So I changed it to 'Do it yerself, Rube'. Coincidentally, I later found out about these (point to video) which are sometimes called Rube Goldberg machines. Meaning unnecessarily complex arrangements which accomplish little other than amusement. Software can all to easily resemble that.

Rube can also refer to a mark, a con-victim, a not-so-smart person. A little while ago, I was stopped in a traffic waiting to turn right. This dude was gesturing at the front of my car. In a moment of misplaced sympathy, I opened the window. “It's my dream car” he says. “You look like Tom Cruise” “Your son looks like Ben 10”. Gives me a hat through the window and starts a patter about a promotion and his birthday. So I put the hat on my son's head. Then he says he's asking for a donation. I give him a few rands. He tells me he normally asks for a donation of R50. I give the hat back, and drive off cos the light has changed. Then I realised what a Rube I'd been.

It's called Cog. You've probably all seen it already, but it was new to me when I coincidentally discovered it while preparing this talk. Those are physical parts, not CGI. It took 6 months, 606 takes, and two18 wheeler trucks full of parts, and is exactly 2 minutes long. As most of us here already know, sometimes it takes a lot of work to make things just work.

Somebody once said “In Smalltalk, everything happens somewhere else.” Coincidentally, that was a certain Adele *Goldberg*. She must have been Rube's cousin.

The Problem

I have a project which grew up slowly over quite a long period. I wanted to take a piece of it out, and turn it into a gem so I could, naturally enough, use it on other projects. I pretty quickly ran into a little problem. I had some configuration: hostnames, tcp ports, memory limits. 8 lines of yaml. And a little code to read it. The little problem was that some of the configuration belonged to the app, and some belonged to the code I was gemifying. And I didn't want end up with 2 really small configuration files.

So, naturally enough, I went looking for configuration gems. Which was frustrating because for starters there were 29. Yes, you heard me right. 29 gems for handling configuration? You've gots to be pulling on my leg.

This has happened before. When I was a young man, it was nearly impossible to get started on a Java web project, because there were 50 frameworks that had to be evaluated first. 50 I tell you! 7Which is why the world was ripe for that famous rails video.

It's not hard to find this kind of proliferation in software. Javascript frameworks also seem to be heading in that direction. Backbone, ember, meteor, canjs, angular and that's just off the top of my head.

Search Cost

Libraries are traditionally regarded as a good way of getting functionality into your project without the accompanying cost of development, testing and maintenance. However this can easily get to be too much of a good thing. You find something useful or cool, and you want to get your name up in github lights, so you put it in a gem and release it with too little documentation. Then it languishes on rubygems at version 0.1.1 for the next several years. You can cover your eyes now and put your hand up if you've done that before. If you're feeling a strong need to do penance, I suggest writing an xml parser in c++.

Fortunately the opposite applies to web development in ruby and development in general in .net - there are a limited set of choices so in most cases you just start your project with the usual (rails or the ide, respectively) and get on with it.

So, ironically, if there are more than a handful of libraries, you might spend as much, or more, time finding the right library as you would just writing your own code.

Do it yerself, Rube!

Which, naturally enough, leads to the question: how did we end up with 29 configuration gems?

Leaky Abtractions

Well, naturally enough, or maybe coincidentally, I have theory. See, that's the nice part of doing talks at conferences. You get to bend *everybody's* ear about your pet theory. Leaky Abstractions. Some of you may know this already, in which case please bear with me.

A stream is a good example of a leaky abstraction. This one clearly leaked a bit. Streams are easy. You read from them, you write to them. They eventually come to an end. But that's half the story.

The other half of the story is that files, network sockets and strings can all be treated as streams. But each has properties which are not accessible through a stream abstraction. Those are the leaks. Files have metadata. Strings and files both have a fixes size support random access. Network ports are potentially infinite, cannot support random access (can never read the same byte from a stream twice, as long as it's a network port), and have a different kind of metadata. Neither strings nor files can be infinite, but in general files can be an order of magnitude or two larger than strings.

So: some abstractions are leaky. Actually not. All abstractions necessarily leak. They wouldn't be abstractions if they didn't. Abstraction means you're taking a bunch of things, identifying what's in common between those things and giving that commonality an existence by naming it. We do this all the time, it's called thinking. Isn't there's a branch of philosophy dedicated to that? Scatology? Eschatology? It works pretty well (abstraction that is. Thinking ... sometimes not so much), and only really becomes dangerous when … you do it the car... or when you allow yourself to be lulled into a false sense of security. It's easy to forget that the thing you're working with through the lens of your abstraction is still a thing in its own right, and that other abstractions can equally well apply to it.

For example, when you're working with a stream (which is actually a file) you forget it's a file and you attempt slurp the whole thing into memory. Or worse, you attempt to slurp the mythical twitter firehose through an http socket into memory.

Or, you actually *need* the particularity. The leaky bits. Again using a stream to read a file, but then you realize hangonasecond, I actually need the name of this file when I throw an exception about why I can't parse it.

So I don't think it makes sense to differentiate between leaky abstractions and non-leaky abstractions. There are just abstractions, all of which leak to a greater or lesser extent depending on two things: firstly, the generality of the abstraction, and secondly the complexity of the concrete item you're applying it to. The more general the abstraction (and the more complex the concrete item), the more it's going to leak. And quite often the more it leaks, the more you have to write code to deal with the particularities of the leaks. But, you already have code to deal with the particularities, so effectively you're just adding another layer which, especially if you have the wrong abstraction, just redistributes the particularities in a different way. An example of this would be associating an instance of a Metadata class with a stream, just so you can get to the file-ness, network-port-ness or string-ness of the stream.

Configuration Gems

Right, back to configuration gems.

Every gem that I looked at, and I didn't look at anywhere near all of them, had a different idea of what set of behaviours should be abstracted in a configuration gem. Some used yaml. Most were hierarchical where some levels could override other levels, including data from environment variables and command line options. Some allowed for multiple pluggable backends, yaml, json, sql, nosql, dot-notation files, ini files. Some handled compression. A couple allowed for encryption of the whole file, or only the password values in the file.

Take the one with the pluggable backends. Was the author trying to solve his or her own problem, or trying to solve all potential problems in the problem domain? In other words all of our problems too? Which is kind and thoughtful to be sure. But most of us aren't even really sure what our own problems are until we're halfway through writing the code to solve them. So how could somebody else solve them with no knowledge of or particularities, in other words the stuff that leaks?

Well, the entire problem domain would have to be adequately covered. And with a problem that starts off with “I need to store a few values in a persistent hash”, the entire problem domain turns out to be a lot bigger than it appears. In a sense this goes to a polar tension at the core of software development: Do you try to code the most general solution you can think of, or so you do the simplest thing that could possibly work? Personally I prefer /etc to the Windows Registry. Because, let's face it, the real world is messy and continually changing. And the more you try to pin it down, the more it escapes from you. On the other hand, if you don't abstract at all, you end up with spaghetti.

Coincidentally, not so long ago, I saw this tweet: Sandi Metz @sandimetz The wrong abstraction is far more damaging than no abstraction at all. Waiting trumps guessing every time.

So if somebody else also said so, it must be true right!? Seeing it on twitter is like seeing it on TV, only more proofier.

Dimensions of Mutability

At a code retreat, I was talking to the person I was pairing with, and the phrase “Dimensions of Mutability” came out of my mouth. I was surprised. I guess you could also call that the set of orthogonal features. I think that covers most of them for configuration gems:

- type translation (mostly strings to other data types, but depends on source)

- different sources (yaml,xml,json,dot-syntax, ini, ENV)
- hierarchical
- defaulting (a kind of hierarchical)
- dynamic settings (erb enabled yaml)
- reloading, (on-change, when specified, on startup, on fork)
- transformation (whole file encryption, individual value encryption, Public/Private vs Symmetric encryption, compression)
- validation (make sure that values make sense, and won't exhaust resources)
- persistence (save the values)
- security (how much can you trust the people with write access to your config files)

You'll notice that combined file for one app and one gem is not there. Maybe the problem domain isn't adequately covered? Maybe I found a leak? Maybe I'm just expecting too much?

Not only that, but something about them left me feeling a bit uncomfortable. Everything was happening somewhere else and it felt a bit Goldbergish (not sure if it's Rube or Adele here) For quite a few of them, that was partly a result of DSLs aiming at syntactic sweetness, resulting in the replication of language features. Hey, waitasecond, that's the redistribution of particularities I was talking about.

And, in terms of syntactic sugar I have to say I'm kinda surprised that this:

```
def friendly_name
```

is considered too verbose. We forget so quickly:

```
const std::string & Settings::friendlyName() const
```

Which, come to think of it, may be part of the reason for text configuration files. One implication of compiled languages is that obviously you can't execute a non-compiled file. Data cannot become Code in that world. I'm guessing here, but perhaps one of the reasons that configuration has been regarded as something done in a text file is as a result of that being the only way to do it with compiled languages. Coincidentally, there is another very good reason for text configuration files:

Security

I seem to have spent a lot of time lately patching rails servers. And it's all yaml's fault. Or is it JSON's fault? Nono it's eval's fault. Well, actually it's an implication of a non-compiled language. Code is Data.

And these are also because of something else that often happens with abstractions.

Misconception: YAML is hashes.

Abstraction: serialization formats instantiate objects.

The yaml parser can instantiate arbitrary ruby objects which is a really nice feature. Although that does imply that it can execute arbitrary code under the right conditions, which in this case is string substitution. It's kindof a truism in software that if something can happen, sooner or later it will.

In a sense, compiled languages have quite a high level of security by accident. Data is necessarily not Code. Interpreted languages have to work a bit harder. If you really do have to work with untrusted files with an interpreted language, you have to separate them out, and be very sure that Data does not become Code.

safe_yaml prevents deserialisation of anything but basic types. So much for everything is an object.

Reasons

Anyway, once again, the 29 gems. I shoulda called this talk Ali Baba and the 29 gems. Here are some reasons why, in my opinion, there are 29 configuration gems:

- The problem domain is more complex than it seems, considering the Dimensions of Mutability.
- Each gem has its own (mis)conception of what the “configuration” abstraction means.
- There's a ruby-culture-wide quest for syntactic sweetness. A DSL is an abstraction. And therefore leaks.
- Peer recognition is always nice. I'm guessing it triggers endorphins or something like that.

Plain Ruby

So, ironically, or perhaps naturally or coincidentally, that all pushed me to do just enough to satisfy my own particular needs. Which means I accidentally got rid of all the unnecessary dimensions of mutability. The ones that were not relevant to my particular problem. For my app + gem situation, I don't need encryption, for example. But I do need load-on-modify, and a combined file. In another app, I need encryption, but the app would break if it tried to re-read its config halfway through a run. Neither of those has to mistrust the config file.

I started thinking, partly driven by the app+gem combined file problem, why am I putting what boils down to a nested set of value objects, or a nested hash which mostly amounts to the same thing, in a separate file and then going through all kinds of parsing and data type transformation issues and overridable convention-over-configuration when Ruby already has a parser, a concise syntax, and multiple abstraction mechanisms?

I can put values in modules, or classes, or objects or singletons or hashes. I could use inheritance for hierarchies. I have built-in flow control for when namespacing is not sufficient (y'know when you have uat, staging and production servers all using the production environment but with different from addresses in outgoing mails). I can use ||= for defaulting. I could use an assertions for validation if that made sense.

Even better, I'm not limited to strings and other simple values, I could use regular expressions or dates. I can do calculations as needed and I don't need a special syntax to defer them. erb enabled ruby is an oxymoron. And it's right here, not all happening somewhere else.

It's amazing how simple the code becomes when you just leave out the dimensions you don't need. So simple, in fact, that it's barely enough to put in a gist, let alone make an entire gem. This happens because you can make assumptions, and the more assumptions you make, the fewer dimensions of mutability you have to deal with. Of course it is very easy to make the wrong assumption.

Closing

So I finally found a solution to my gem + app configurable by one file. I have this nagging feeling that a LISP machine somewhere is laughing at me. Ironically, or maybe coincidentally, I got here by deliberately not being as general as possible. Sometimes, writing more code is not the answer. But writing more code can help to show you what not to do.

If there is only one thing you take away from this talk (other than a chocolate) I would really like it to be this: We have a bunch of really powerful tools right here in ruby, which I think are quite often obscured by sugar. There's nothing wrong with with builtin libraries and there's nothing wrong with def, class and module. So in spite of the common wisdom that says use a gem first, sometimes you might be surprised and save yourself some time by just doin' it yerself, Rube!

What I want to do now is run through some really simple code for reading various kinds of configuration files. Without using any configuration gems. And I'm deliberately going to make absolutely no attempt whatsoever to pull them all together into one all-encompassing configuration framework.

So, just to shake things up, before we look at the code, what questions do you have so far? Remember, even if it feels like a stupid question to you, there are probably 10 other people here who're feeling exactly the same way you are.