clj-v8

and the dieter asset pipeline

@paulbiggar

@circleci

Its March 2012, and 4 months after we started building it, our web site is at an architectural dead-end. The major part of CircleCI is written in Clojure, but its frontend is written in Rails. The integration between the two is poor, so we went in a new direction: replacing it with a one-page JS app, connected to the backend via an API.

CircleCI architecture

- Backend:
  - Running builds, configuring boxes, dev-ops
  - 100% Clojure
  - See "*Building a PaaS in Clojure*" by @arohner

CircleCI architecture

- Old Frontend:
  - Rails?
  - WTF?
  - See “*Clojure and JRuby: the good, the bad and the ugly*” by @arohner

CircleCI architecture

- New frontend:
  - One page JS app
  - Clojure REST API
  - Coffeescript, HamlCoffee, Less

The problem with one-page JS apps, is that they can be a pain to write. JS is not the most wonderful language in the world, nor is CSS, so we wanted to use CoffeeScript and Sass and Haml, and all the lovely DSLs that they use in the Rails world.

In the Rails world, they have an asset pipeline called Sprockets. An asset pipeline compiles and serves assets written in multiple languages. Part of serving them also means caching, adding hashes to file names.

Dieter

- https://github.com/edgecase/dieter
- @xandrews

I didnt start dieter, it was started by a guy called John Andrews, who worked for edgecase, which is a consultancy that recently became part of Neo.

Dieter use

```
(html5
 [:head
  (include-css (dieter/link-to-asset “css/app.less“ opts))
  (include-js (dieter/link-to-asset “js/app.js.dieter“ opts))]

 [:body
  [:div#app]])
```


Here's what it looks like in practice. (hiccup)

Dieter use

```
<html>
 <head>
   <link href="/assets/css/
app-21f212f94a9db6a0e3847c921842aa19.less"
rel="stylesheet" type="text/css">
   <script src="/assets/js/app.js-
c6891d0f2e304af6b71018d9903fafc6.dieter"
type="text/javascript"></script>
 </head>
  <body><div class=”app”></div></body>
</html>
```


Here's the result you get, the long names are for cache-busting: each time the contents change, a new file is fetched as it has a different name.

Dieter manifest file

```
[
  "../../components/jquery/jquery.js"
  "../../components/moment/moment.js"
  "bootstrap.js.dieter"

  "../views/"
  "libs/"
  "inner/"
  "outer/"

  "app.coffee"
]
```


Here's what app.js.dieter looks like. This is actually taken from our production site. Its slightly more complicated than this, but essentially we ship a single JS file called app.js.dieter, which is 124 HamlCoffee templates, 32 Coffeescript files, 10 JS files. app.less is 45 less documents.

Dieter features

- Supports
  - JS: CoffeeScript, HamlCoffee
  - CSS: Less
  - static files
  - manifests

So, some quick features. Compiles HamlCoffee, CoffeeSCript and Less. Concatentates many files together in a manifest. Serves static files.

Dieter features

- Cache-forever headers
- Cache-busting urls
- mime type headers

Adds cache busting, and adds cache-headers, url busting suffixes, and mimetypes, for JS, CSS, static files, and manifests.

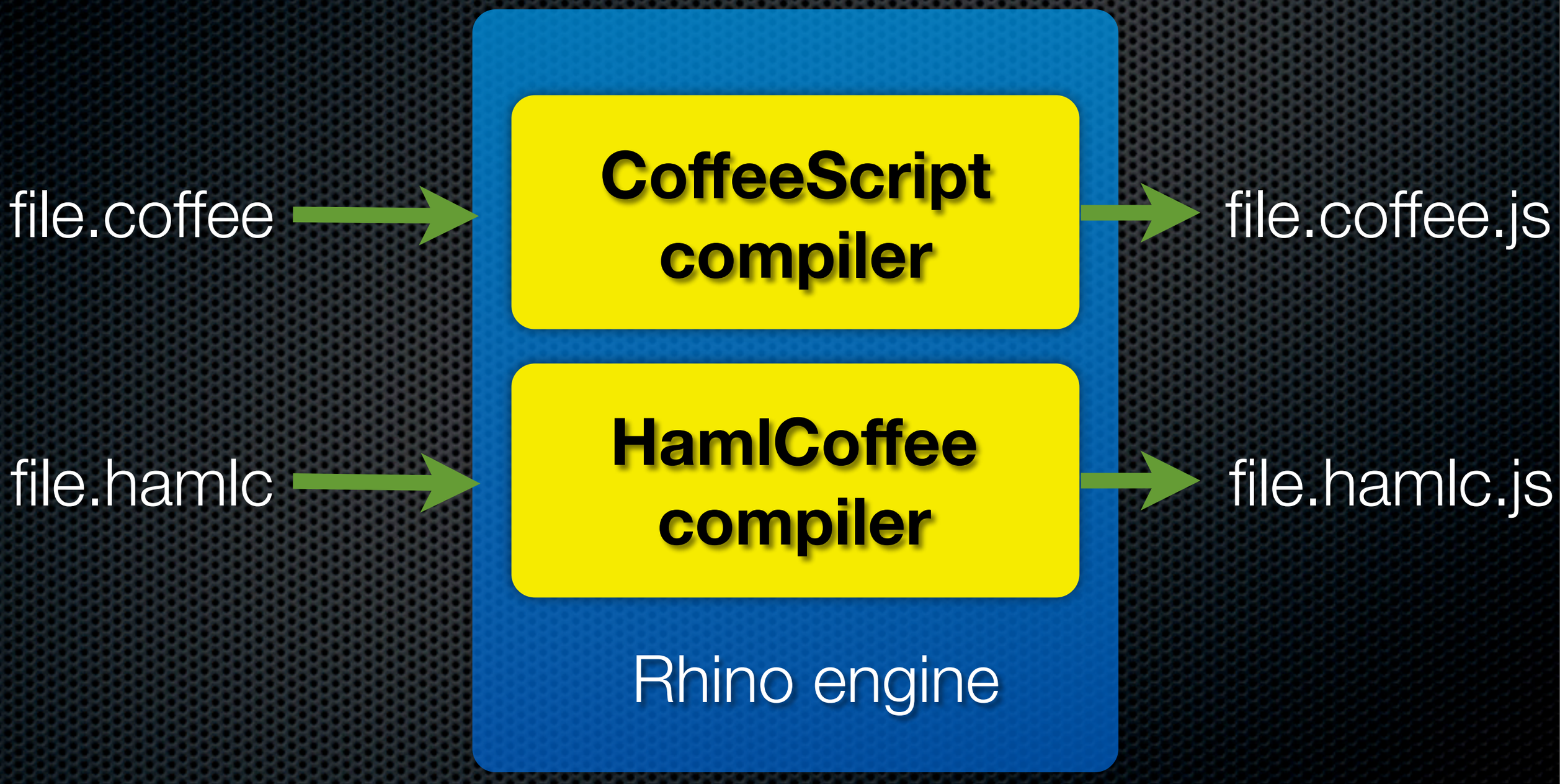
OK, so that's a brief overview of how dieter works and why its useful. Now I want to tell you about why I hated using dieter.

Slooooooooowwwwww

- 124 HamlCoffee files
- 10 JS files
- 32 coffeescript files
- 1.2MB asset
- **Compilation time: 90s**

And that's because it was slow. For our app, we eventually got to the point where it took 90s to refresh the page! We currently have about 160 files we're compiling, and each time we refreshed, they were all compiled. And that makes for unhappy developers. By which I mean me.

Compiling an asset



So the way that the compilation works is pretty simple. Less, HamlCoffee and CoffeeScript have compilers written in Javascript. You use the same compiler whether your app is in Node, Rails, or Clojure, you just run them from a JS engine running within your asset pipeline.

So that’s exactly what dieter does, using Rhino.

Rhino is a Javascript engine written in Java. So obviously its very very fast. Or it was for its day, which was 2008. Times have changed, and Rhino is not fast.

Replacing Rhino

- Package v8 for clj
- Write interface
- Make clj-v8 work well

V8 is fast though. Very very fast - its the engine in Chrome. Lots of people use V8 because its so fast, including Node.js and Sprockets, which are other users of Coffeescript, etc. So we decided to add v8 to dieter.

Here are the rough steps: package the v8 libraries, write the interface to them, and then of course make sure that it works. The last one turned out to be by far the hardest, and I suffered through sgfaults for a few months, which crashed our VM in development.

Packaging v8

- clj-v8
  - src/clj-v8/core.clj
- clj-v8-native
  - native/linux/x86/libv8wrapper.so
  - native/linux/x86/libv8.so.clj-v8
  - native/linux/x86-64/libv8wrapper.so
  - native/linux/x86-64/libv8.so.clj-v8
  - native/macosx/x86-64/libv8wrapper.dylib
  - native/macosx/x86-64/libv8.so.clj-v8

Packaging v8 was pretty straightforward. We compiled v8 for each of the platforms, and then wrote a wrapper in C which exposed exactly the right functions. Take the compiled libraries, and put them in a jar with this directory structure, and you have something that leiningen will treat correctly.

JNA things about jna.library.path

clj-v8-native

- Simple interface
  - create_context:
    - returns struct {v8::Isolate, v8::Context}
  - cleanup_context
  - run_script_in_context
- Hard to get threading right
- Exception handling
- String conversion

Wednesday, March 20, 13

So a lot of this work was on the C/C++ side of the interface. We exposed functions to create a context, delete a context, and run code in a context. A context is kinda like a global scope. Funnily enough, all the JS engines use similar terminology here: v8, spidermonkey, rhino.

The hardest thing to get right is the threading. If you don't get it exactly right, it'll crash when multiple requests arrive at the same time. Like all good things, threading in V8 is very well documented, if you look at exactly the right header and squint in exactly the right way. In the end, I wrote a test which called my functions in thousands of simulatenous threads, using boost::thread. If it makes it through that, you've almost certainly got the bugs out.

There were also plenty of problems with getting exception handling and string conversion just so. They were of the more straightforward bang-you-head-against-the-wall, why-did-I-ever-learn-unmanaged-languages sort of problems, and were handled in the usual manner: alcohol, valgrind, cursing and tears.

clj-v8/core.clj

```
(-> “v8wrapper”
    com.sun.jna.NativeLibrary/getInstance
    (.getFunction “create_context”)
    (.invokePointer (into-array [])))
```


clj-v8 handles the actual interfacing of the C/C++ functions to clojure. This was surprisingly easy, check out the code. The entire package is 60 lines, and the meat of it is about 10 lines. Here's how you call create-context. From top-to-bottom, this is only 4 lines.

EXPLAIN

JNA isn't perfect: there's some fiddling with jna.library.path, LD_LIBRARY_PATH, etc, but overall its wonderful. In clojure, between leiningen's great support for native libs, and clojure's java support, its basically a few trivial lines to make it all work.

Optimizations

- Unoptimized: 90s

One thing I noticed when experimenting, was that the latest version of Rhino was 20% faster, so we got a decent speedup from it, while the v8 stuff was still cooking. But we don't use that anywhere now.

Optimizations

- Unoptimized: 90s
- new Rhino: 70s

Optimizations

- Unoptimized: 90s
- new Rhino: 70s
- v8: 7s

Adding V8 got us down to about 7s, a 10x speedup. This was the most naive wy possible. Each time we compile, we start up v8, compile coffeescript, then run all files through coffeescript.

However, this is the first time we had anything like productivity. Going from 70s to 7s is a godsend. But like most things, when you suspect you can go faster

Optimizations

- Unoptimized: 90s
- new Rhino: 70s
- v8: 7s
- memoization: 1s

Memoization.

Instead of compiling 160 files every single time, memoize it. Store a timestamp, or use an md5 add a timestamp, and only recompile the ones that are necessary. Dieter uses a timestamp for that. So now, we're serving 99% of files from the cache, and only recompiling one of them.

I find this interesting from optimization, because I spent a lot of time on making clj-v8 work, and very little time making this work. But it would nearly have been tolerable to keep using rhino with this fix.

Optimizations

- Unoptimized: 90s
- new Rhino: 70s
- v8: 7s
- memoization: 1s
- reuse v8 instance: 250ms

And the final optimization was reusing v8 multiple times. Instead of doing the entire work each time: create a context, load coffeescript, run the file, then destroy v8, do as much as possible the first time, then reuse it. So now we do the first two steps only once: creating the context and compiling the coffeescript compiler. Each asset page load, we do the rest.

So this got it down to 0.25s, which is pretty damn fast, and feels totally instantaneous.

Optimizations

- Unoptimized: 90s
- new Rhino: 70s
- v8: 7s
- memoization: 1s
- reuse v8 instance: 250ms
- `lein dieter-precompile`: 50ms

So this got it down to 0.25s, which is pretty damn fast, and feels totally instantaneous. Its almost fast enough to use in production, but not quite. We serve production assets from S3, using a lein-plugin I wrote called `lein dieter-precompile`. It takes a list of files you need, and compiles them from the command line, which we then upload to S3, and then serve those files.

The latency here is about 50ms when I tested it from the hotel this morning, but I think that's about as fast as we're going to get!

Help wanted

- sourcemaps
- minification
- deeper pipelines (asset.erb.coffee)
- image inlining and spriting
- parallelize manifest compilation
- upload to S3

Help wanted

- https://github.com/edgecase/dieter
- https://github.com/circleci/clj-v8
- https://github.com/circleci/clj-v8-native

We're hiring

jobs@circleci.com