Eyre HTTP Caching

Mark Staarink ~palfun-foslup,
Ted Blackman ~rovyns-ricfer,
Joe Bryan ~master-morzod
Tlon Corporation, Urbit Foundation, Urbit Foundation

Abstract

Eyre is the server vane for the Urbit OS, and is responsible for handling all HTTP requests. Eyre supports caching responses to GET requests, which can improve performance by reducing the number of times that the server must generate a response from scratch. This historical report and response were published as advisory gists on GitHub, and encapsulate design decisions that have gone into modifying and improving Eyre's caching performance.

Contents

Urbit Systems Technical Journal I:2 (2024): 105–113.

Address author correspondence to ~palfun-foslup.

1 Improving Eyre HTTP Caching

This proposal for Eyre's HTTP caching mechanism was released by `~palfun-foslup` *on* `~2022.8.15` *and revised through* `~2022.8.16`.

1.1 Introduction

Urbit's primary role is to function as a webserver, but it is not as fast at that operation as it could be. It would be good if Urbit could comfortably serve hundreds or even thousands of pageviews per minute. (Benchmarks for the status quo are left as an exercise to the reader.)

1.2 Approaches to Improving Performance

There are two primary ways in which the scry namespace can be utilized to make Eyre more performant:[1]

1. **Publication cache**: Eyre can track a publication cache with static, known-ahead-of-time responses bound to specific endpoint paths. Eyre would tell the runtime about these known responses, which the runtime would use to serve up responses to `GET` requests.
2. **Stateless reads**: The runtime, when receiving a `GET` request, can scry into Eyre to retrieve a response for it. Eyre might scry into agents to further resolve the read. This way, with `GET` requests handled as pure reads, they could theoretically be served in parallel.

This document focuses on the publication cache. The latter will not deliver performance gains by itself, may actually incur a performance hit in the "scry miss" case, and has unanswered questions around referential transparency. Some prior art can be found in an old draft PR (#4674).

The publication cache also aligns closely with the imagined future of "solid state subscriptions" and the "shrub namespace",

[1] Truthfully, there are probably more than two ways, but these specific two have highest relevance in Eyre's recent history.

wherein new or changed data is explicitly published into the namespace.

1.3 Kinds of Content

Before talking about the publication cache proper, we must be aware of what kinds of content may get served through Eyre. We identify three kinds:

1. **Static content** is fully self-contained and only changes when the data within changes; for example: a blog post without comments, or an image.

2. **Dynamic content** changes based on the current state of an agent (or some other datum); for example: a blog post *with* comments or a list of pals.

3. **Procedural content** is generated from the request itself. While the response for any given path may be known ahead of time, it may not be possible to enumerate all the valid paths for which we have responses; for example: a parameterized `/sigil.svg` endpoint, or a calculator API.

Note that for dynamic and procedural content, it may not always be possible to publish a known response at all. If the response depends on the timestamp of the request in some way, a cache entry would be busted before it even got stored. We simply ignore this case.

1.4 Sources of Content

Briefly, take note that for both dynamic and procedural content, Hoon code *must* be executed to generate the (original) response. Most commonly, this takes the form of a Gall userspace agent. In rare cases, generators serve this function.

For static content (only) can we consider another source: Clay, the file system vane. For things that are already files, it makes sense to stick these in Clay. (Think JavaScript blobs, image files, and other "earth" content.) For data that originates

within agents, however, in most cases it is unergonomic and unsound to store that data in Clay, especially if the agent may want to refer back to the data later.

In practice, both the "dynamic/prodecural response from agent" and "static file from Clay" cases are common, though the latter is often handled through Docket's "glob" system instead (for, at this point, largely historical reasons).

1.5 Static Publication Cache

Accounting for the fact that agent-driven responses are already common, and weighing the fact that Eyre does not currently have any connection to or dependency on Clay, we propose the following model for a publication cache in Eyre:

```
+$  task
  $%  to-cache
      etc...
  ==
::
+$  gift
  $%  to-cache
      etc...
  ==
::
+$  to-cache
  $%  [%save endpoint data=simple-payload:http]
      [%dump endpoint]
  ==
::
+$  endpoint  ::  exact binding
  $:  binding
      tail=(unit @t)
  ==
```

1. Eyre gets a new task and gift, `%save`, which can be used to publish responses for a specific endpoint (optional site, plus path, plus extension if any) into the cache.

2. Eyre stores the cache within itself. Whenever something gets added, it notifies the runtime. On-`%born`, it notifies

the runtime of all existing entries.

3. The runtime tracks the cache as per Eyre's notifications. Whenever it receives a `GET` request, and an entry for that exact path exists in the cache, it serves the stored response instead of injecting the request as an event.

4. `%save` may be used to overwrite existing entries. `%dump` may be used to remove existing entries.

5. When resolving an incoming `GET` request, Eyre checks the cache for an exact match. If there is, it serves that. If there is none, it falls back to the regular binding matching that path, if any.

This is sufficient to let agents publish responses into the cache, and keep those updated as the underlying data changes.

The "static file from Clay" case can be implemented using the affordances here. To avoid repetition of boilerplate patterns regarding this, we might ship a small piece of userspace infrastructure, an `/app/file-server` if you will, that is responsible for bindings of this kind.

1.5.1 Authentication

The above does not account for authentication, limiting cached responses to fully public content. Presently, we do not have the affordances needed to handle private content properly.

But it doesn't have to be that way. The `%save` task could simply include an `auth=?` flag alongside the `data`, indicating whether authentication is required or not. The runtime would then, where needed, check the incoming request for a valid authentication cookie, and either give the response or serve a simple 403.

… except that the runtime does not presently know how to check an incoming request for authentication. And making it do so is outside the scope of the grant. The changes here aren't too big though, and in some ways similar to the behavior outlined above. (Most likely, just Eyre telling the runtime about creation/expiry of session identifiers, and teaching the runtime

how to check for its presence in any `Cookie` headers.) Considering the very-nice-to-have nature of support for private endpoints, we (read: `~palfun-foslup`) may offer to implement this in the short-term, so that the grant work may make use of it.

Caching may not seem as relevant for private content, since it's significantly less likely to get requested many times a minute. But being able to eagerly cache there still provides tangible benefits. Urbit-generated web UIs can get served faster, and private endpoints stop being surface area for DOS attacks.

1.6 Procedural Publication Cache

At this point, we have accounted for serving static and dynamic content from Eyre, but are not yet able to serve procedural content. This requires a slightly different approach.[2]

```
+$  to-cache
  $%  [%prep =binding =work]
      [%drop =binding]
      etc...
  ==
::
+$  work  $-(inbound-request simple-payload:http)
::
+$  action
  $%  [%work work]
      etc...
  ==
```

1. Eyre gets a new task and gift, `%prep`, which can be used to publish a response generation function for a specific binding (optional site, plus top-level path) into the cache.

2. In addition to a cache entry, Eyre stores this among the normal bindings. Whenever a `%work` binding gets added, Eyre notifies the runtime.

[2] The approach outlined in this subsection is still tentative and under discussion, pending solid state subscriptions becoming more "real". Certainly the static publication cache above should be sufficient for most cases.

3. The runtime tracks the cache as per Eyre's notifications. Whenever it receives a `GET` request, it checks to see if a `%prep` binding matches. If one does, it runs the gate and serves the generated response, instead of injecting the request as an event.

4. When resolving an incoming `GET` request, Eyre resolves from the bindings as normal.

However, an important caveat holds at this point in the discussion: how procedural content forces one to bind on non-exact paths, in turn forcing the runtime to bind resolution. We could reduce the friction here by moving Eyre's funtion to find the binding for a given request path into `/sys/lull` or the ivory pill. Alternatively, Eyre just publishes one overarching cache resolution function to the runtime, instead of letting it implement its own logic. Concern with any of these is keeping old copies of the kernel around within these functions. Eyre could re-publish the function(s) on-upgrade, but might also need agents to do the same.

2 Further Thoughts

This document is an alternative proposal by `~rovyns-ricfer` *and* `~master-morzod` *on* `~2022.8.16`. *It particularly emphasizes the role that the bound scry namespace and remote scry play in Urbit's prioritization of referential transparency.*

This represents a vision for how HTTP handling in Urbit could work in the long-run:

1. First check if the URL is immutable, mapped to a fully qualified scry path.

2. Then check if the URL is in a mapping from mutable URL to runtime cache value, which is either:

   (a) an HTTP 307 temporary redirect to a fully qualified scry path, or

   (b) a direct HTTP response value

3. If neither of these, then inject the request into Arvo as an event.

Most read requests should use the namespace rather than directly cached values. This promotes having as much of the system as possible built on referential transparency, which facilitates scaling. However, if a request would be better served without an HTTP redirect (such as a request for a top-level webpage where the URL should not include a revision number or other scry-related details), then an application can ask Eyre to serve the response directly (and the runtime can cache this value, as long as it invalidates it properly when it changes).

If we want Eyre to serve a login page without external dependencies, for example, it should use the mapping from mutable URL to direct value – but if it's fine to put it in Clay, then it can redirect to the namespace.

In the future, we'll want to allow the runtime to handle authentication with minimal Arvo activation. This would prevent unnecessary events being enqueued – which could be used for denial of service attacks – and it also could be used to enable access to private scryable data. A user could authenticate and retrieve a (potentially mutable) piece of scryable data without activating Arvo, or only activating it to validate an authentication token.

Given this context, the mutable mapping we're trying to implement in this PR could be thought of as a subset of the second check here, namely mutable values that map directly to HTTP response values stored in Eyre and cached in the runtime.

Alternatively, we could treat this as the other version of the second check, involving a redirect: Eyre could maintain a referentially transparent mapping from mutable URL to scry path, and the runtime could mirror this mapping and also have a scry cache that it uses to serve HTTP scry requests after the redirects.

There are multiple considerations pointing us toward using direct HTTP responses for now:

1. The ergonomics of Clay:

   (a) We need a separate desk full of marks for all served files.
   (b) We need mark conversions from all filetypes to HTTP response.
   (c) We need to name each desk, with ad-hoc namespacing.
   (d) We need to establish tombstoning policies ("norms").

2. Limitations on the namespace for Gall agents:
   (a) Only scry at current date.
   (b) No solid-state publications yet.
   (c) Lack of permissioning.
3. Desire to have full control over the URL shown to a user in a browser.

Given all of these concerns, Eyre development should prioritize implementing direct responses first and the subsequent pieces at a later time.

References

~palfun-foslup (~2024..) "urbit/urbit #4674: king: scry on GET requests (WIP)". URL: https://github.com/urbit/urbit/pull/4674 (visited on ~2024.9.11).