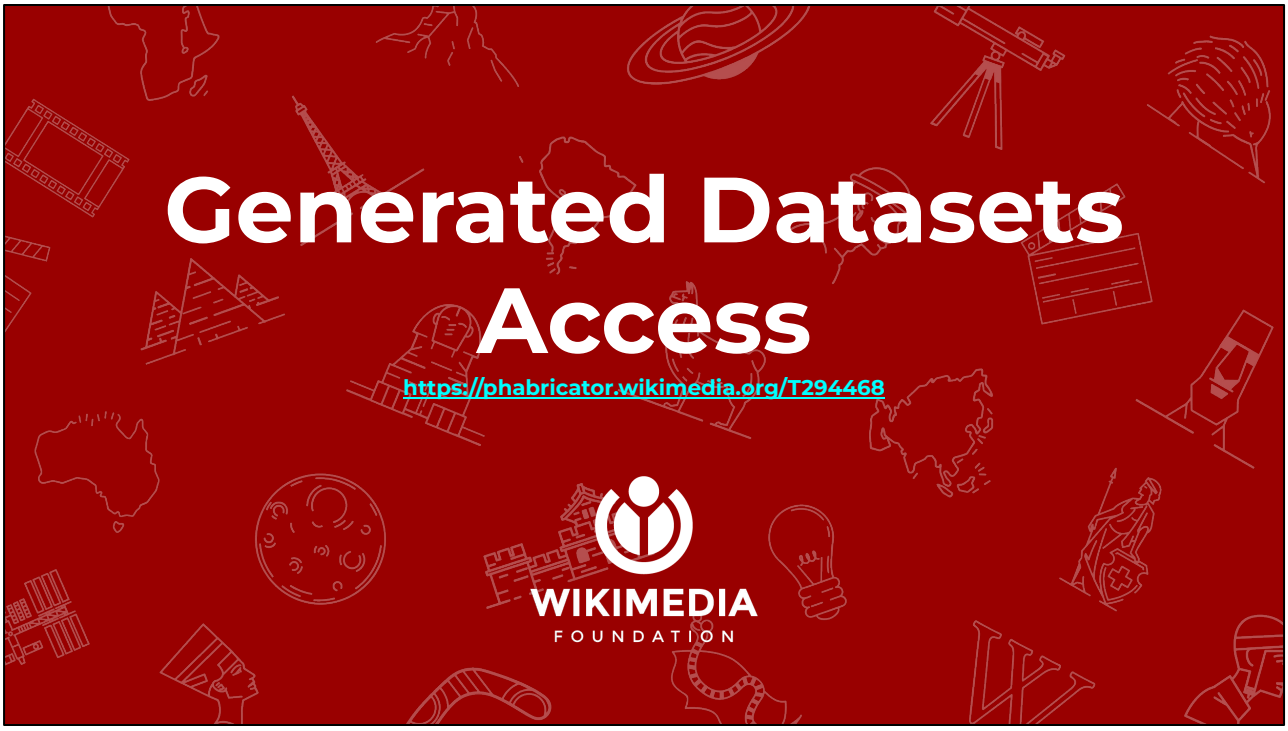
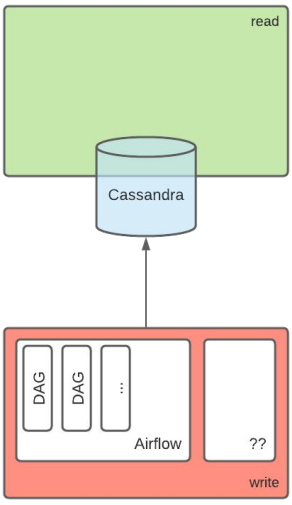
Generated Datasets Access

https://phabricator.wikimedia.org/T294468





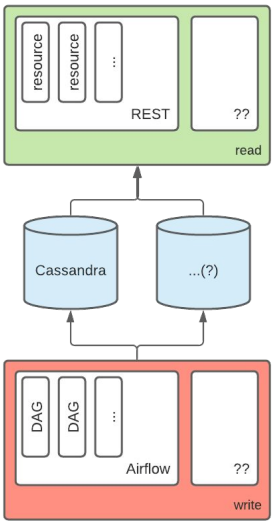
WIKIMEDIA FOUNDATION

So far we've talked about building pipelines to create/update datasets, and move them to persistence, but we haven't spent much time talking about how clients will query those results. If nothing else, a service could use a Cassandra driver and CQL to query the datastore directly, but would it make sense to create an abstraction, instead?

To abstract, or not...

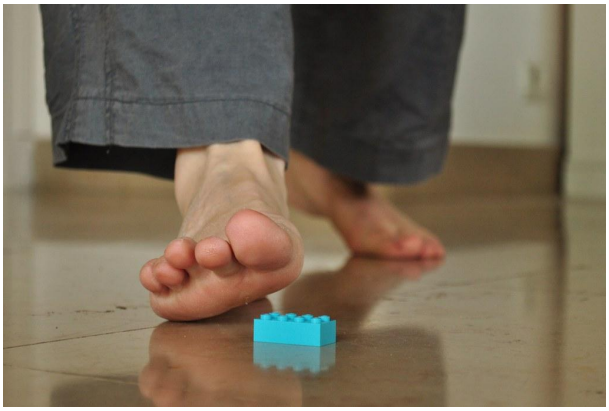
- Decouple
- Narrow interface
- Hide or suppress (unnecessary) details
- Generalize
- Simplify use
- ...

There are many reasons why one might use an abstraction, and some do seem applicable here. For example, decoupling clients from Cassandra could give us the freedom to (later) implement caching layers, or transparently migrate datasets between backend stores. Generalizing the interface might open the possibility of using different backing stores depending on circumstance.



WIKIMEDIA
FOUNDATION

...but watch out!



WIKIMEDIA
FOUNDATION

But this is also a space where we should tread carefully. Designing good interfaces is hard, and requires thorough understanding of your use cases. We always feel we possess that understanding, but rarely do until much later on. Getting these abstractions wrong won't prevent people from using them (unfortunately).

To explore this subject: a small experiment/proof-of-concept to create a (very) lightweight HTTP data gateway framework.

Assumptions

- Not an end-user API; Data access for services
- Convention over code (and configuration)
- Read-only
- Data stored matches expected query results (or nearly so), and always either:
  - Single JSON object (think: row)
  - JSON array of objects (think: rows)
- Minimum viable dose

```
router.GET("/imagerec/articles/:wiki/:page_id", func(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
        // Simple: SELECT * ...
        var s *Select
        s = SelectBuilder.From("image_recommendations", "articles").Bind(ps).Session(session).Build()
        s.Handle(w)
```

WIKIMEDIA FOUNDATION

A REST endpoint in 3 lines:

- A builder with a fluent API constructs an object of type `Select`
  - For the applicable Cassandra keyspace and table
  - While binding the REST API parameters (to serve as query predicates)
  - And supplying a session object for the database
- `Select#Handle` is invoked to process the HTTP request and send JSON-encoded response

```
var router = httprouter.New()
router.GET("/imagerec/articles/:wiki/:page_id", func(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
        // Simple: SELECT * ...
        SelectBuilder.
                From("image_recommendations", "articles").
                Bind(ps).
                Session(session).
                Build().
                Handle(w)

        // More flexible: Specifying the query projection
        /*
                SelectBuilder.
                        Select(
                                Field{Column: "wiki"},
                                Field{Column: "page_id"},
                                Field{Column: "dataset_id"},
                                FieldBuilder.Column("dataset_id").CastTo("timestamp").Alias("timestamp").Get(),
                                Field{Column: "images"},
                        ).
                        From("image_recommendations", "articles").
                        Where("wiki", EQ, ps.ByName("wiki")).
                        Where("page_id", EQ, ps.ByName("page_id")).
                        Session(session).
                        Build().
                        Handle(w)
        */
})
```

WIKIMEDIA FOUNDATION

Same query, but gratuitously in-lined (for maximum effect).

The commented example that appears below demonstrates that finer-grained control of the resulting query (projection and predicates) *is* possible.

Of course, for any use-case where this type of approach broke down, there would be nothing preventing us from coding an entirely bespoke solution.

It's worth noting: The code that drives this is quite small. This is possible because it performs no processing of results, relying upon Cassandra to serialize directly to JSON (which the framework simply returns verbatim).

This would seem to demonstrate that a minimal framework could remove almost all of the tedium to creating new endpoints that serve datasets. It could be a great starting point for an iterative design, giving us the space to defer decisions that would be better made as concrete use-cases present.