Architecture Decision Record: Datomic-based Configuration

Context

ADR-002 indicates that we will store the entire application config in a single rich data structure with a schema.

Config as Database

This implies that it should be possible to easily search, query and update the configuration value. It also implies that the configuration value is general enough to store arbitrary data; we don't know what kinds of things users or module authors will need to include.

If what we need is a system that allows you to define, query, and update arbitrary data with a schema, then we are looking for a database.

Required data store characteristics:

1. It must be available under a permissive open source license. Anything else will impose unwanted restrictions on who can use Arachne.
2. It can operate embedded in a JVM process. We do not want to force users to install anything else or run multiple processes just to get Arachne to work.
3. The database must be serializable. It must be possible to write the entire configuration to disk, and then reconstitute it in the same exact state in a separate process.
4. Because modules build up the schema progressively, the schema must be inherently extensible. It should be possible for modules to progressively add both new entity types and new attributes to existing entity types.
5. It should be usable from Clojure without a painful impedance mismatch.

Configuration as Ontology

As an extension of the rationale discussed in ADR-002, it is useful to enumerate the possible use cases of the configuration and configuration schema together.

- The configuration is read by the application during bootstrap and controls the behavior of the application.
- The configuration schema defines what types of values the application can or will read to modify its structure and behavior at boot time and run time.
- The configuration is how an application author communicates their intent about how their application should fit together and run, at a higher, more conceptual level than code.
- The configuration schema is how module authors communicate to application authors what settings, entities and structures are available for them to use in their applications.
- The configuration schema is how module authors communicate to other potential module authors what their extension points are; module extenders can safely read or write any entities/attributes declared by the modules upon which they depend.
- The configuration schema can be used to validate a particular configuration, and explain where and how it deviates from what is actually supported.

- The configuration can be exposed (via user interfaces of various types) to end users for analytics and debugging, explaining the structure of their application and why things are the way they are.
- A serialization of the configuration, together with a particular codebase (identified by a git SHA) form a precise, complete, 100% reproducible definition of the behavior of an application.

To the extent that the configuration schema expresses and communicates the "categories of being" or "possibility space" of an application, it is a formal Ontology. This is a desirable characteristic, and to the degree that it is practical to do so, it will be useful to learn from or re-use existing work around formal ontological systems.

Implementation Options

There are instances of four broad categories of data stores that match the first three of the data store characteristics defined above.

- Relational (Derby, HSQLDB, etc)
- Key/value (BerkelyDB, hashtables, etc)
- RDF/RDFs/OWL stores (Jena)
- Datomic-style (Datascript)

We can eliminate relational solutions fairly quickly; SQL schemas are not generally extensible or flexible, failing condition #4. In addition, they do not fare well on #5 -- using SQL for queries and updates is not particularly fluent in Clojure.

Similarly, we can eliminate key/value style data stores. In general, these do not have schemas at all (or at least, not the type of rich schema that provides a meaningful data contract or ontology, which is the point for Arachne.)

This leaves solutions based on the RDF stack, and Datomic-style data stores. Both are viable options which would provide unique benefits for Arachne, and both have different drawbacks.

Explaining the core technical characteristics of RDF/OWL and Datomic is beyond the scope of this document; please see the Jena and Datomic documentation for more details. More information on RDF, OWL and the Semantic web in general:

- Wikipedia article on RDF
- Wikipedia article on OWL
- OWL Semantics standards document.

RDF

The clear choice for a JVM-based, permissively licensed, standards-compliant RDF API is Apache Jena.

Benefits for Arachne

- OWL is a good fit insofar as Arachne's goal is to define an ontology of applications. The point of the configuration schema is first and foremost to serve as unambiguous communication regarding the types of entities that can exist in an application, and what the possible relationships between them are. By definition, this is defining an ontology, and is the exact use case which OWL is designed to address.
- Information model is a good fit for Clojure: tuples and declarative logic.

- Open and extensible by design.
- Well researched by very smart people, likely to avoid common mistakes that would result from building an ontology-like system ourselves.
- Existing technology, well known beyond the Clojure ecosystem. Existing tools could work with Arachne project configurations out of the box.
- The open-world assumption is a good fit for Arachne's per-module schema modeling, since modules cannot know what other modules might be present in the application.
- We're likely to want to introduce RDFs/OWL to the application anyway, at some point, as an abstract entity meta-schema (note: this has not been firmly decided yet.)

Tradeoffs for Arachne (with mitigations)

- OWL is complex. Learning to use it effectively is a skill in its own right and it might be asking a lot to require of module authors.
- OWLs representation of some common concepts can be verbose and/or convoluted in ways that would make schema more difficult to read/write. (e.g, Restriction classes)
- OWL is not a schema. Although the open world assumption is valid and good when writing ontologies, it means that OWL inferencing is incapable of performing many of the kind of validations we would want to apply once we do have a complete configuration and want to check it for correctness. For example, open-world reasoning can never validate a `owl:minCardinality` rule.
    - Mitigation: Although OWL inferencing cannot provide closed-world validation of a given RDF dataset, such tools do exist. Some mechanisms for validating a particular closed set of RDF triples include:
        1. Writing SPARQL queries that catch various types of validation errors.
        2. Deriving validation errors using Jena's rules engine.
        3. Using an existing RDF validator such as Eyeball (although, unfortunately, Eyeball does not seem to be well maintained.)
    - For Clojure, it would be possible to validate a given OWL class by generating a specification using `clojure.spec` that could be applied to concrete instances of the class in their map form.
- Jena's API is aggressively object oriented and at odds with Clojure idioms.
    - Mitigation: Write a data-oriented wrapper (note: I have a working proof of concept already.)
- SPARQL is a string-based query language, as opposed to a composable data API.
    - Mitigation: It is possible to hook into Jena's ARQ query engine at the object layer, and expose a data-oriented API from there, with SPARQL semantics but an API similar to Datomic datalog.
- OWL inferencing is known to have performance issues with complex inferences. While Arachne configurations are tiny (as knowledge bases go), and we are unlikely to use the more esoteric derivations, it is unknown whether this will cause problems with the kinds of ontologies we do need.
    - Mitigation: We could restrict ourselves to the OWL DL or even OWL Lite sub-languages, which have more tractable inferencing rules.
- Jena's APIs are such that it is impossible to write an immutable version of a RDF model (at least without breaking most of Jena's API.) It's trivial to write a data-oriented wrapper, but intractable to write a persistent immutable one.

Datomic

Note that Datomic itself does not satisfy the first requirement; it is closed-source, proprietary software. There *is* an open source project, Datascript, which emulates Datomic's APIs (without any of the storage elements).

Either one would work for Arachne, since Arachne only needs the subset of features they both support. In, fact, if Arachne goes the Datomic-inspired route, we would probably want to support *both*: Datomic, for those who have an existing investment there, and Datascript for those who desire open source all the way.

Benefits for Arachne

- Well known to most Clojurists
- Highly idiomatic to use from Clojure
- There is no question that it would be performant and technically suitable for Arachne-sized data.
- Datomic's schema is a real validating schema; data transacted to Datomic must always be valid.
- Datomic Schema is open and extensible.

Tradeoffs for Arachne (with mitigations)

- The expressivity of Datomic's schema is anemic compared to RDFs/OWL; for example, it has no built-in notion of types. It is focused towards data storage and integrity rather than defining a public ontology, which would be useful for Arachne.
    - Mitigation: If we did want something more ontologically focused, it is possible to build an ontology system on top of Datomic using meta-attributes and Datalog rules. Examples of such systems already exist.
- If we did build our own ontology system on top of Datomic (or use an existing one) we would still be responsible for "getting it right", ensuring that it meets any potential use case for Arachne while maintaining internal and logical consistency.
    - Mitigation: we could still use the work that has been done in the OWL world and re-implement a subset of axioms and derivations on top of Datomic.
- Any ontological system built on top of Datomic would be novel to module authors, and therefore would require careful, extensive documentation regarding its capabilities and usage.
- To satisfy users of Datomic as well as those who have a requirement for open source, it will be necessary to abstract across both Datomic and Datascript.
    - Mitigation: This work is already done (provided users stay within the subset of features that is supported by both products.)

Decision

The steering group decided the RDF/OWL approach is too high-risk to wrap in Clojure and implement at this time, while the rewards are mostly intangible "openness" and "interoperability" rather than something that will help move Arachne forward in the short term.

Therefore, we will use a Datomic style schema for Arachne's configuration.

Users may use either Datomic Pro, Datomic Free or Datascript at runtime in their applications. We will provide a "multiplexer" configuration implementation that utilizes both, and asserts that the results are equal: this can be used by module authors to ensure they stay within the subset of features supported by both platforms.

Before Arachne leaves "alpha" status (that is, before it is declared ready for experimental production use or for the release of third-party modules), we will revisit the question of whether OWL would be more appropriate, and whether we have encountered issues that OWL would have made easier. If so, and if time allows, we

reserve the option to either refactor the configuration layer to use Jena as a primary store (porting existing modules), or provide an OWL view/rendering of an ontology stored in Datomic.

Status

Proposed

Consequences

- It will be possible to write schemas that precisely define the configuration data that modules consume.
- The configuration system will be open and extensible to additional modules by adding additional attributes and meta-attributes.
- The system will not provide an ontologically oriented view of the system's data without additional work.
- Additional work will be required to validate configuration with respect to requirements that Datomic does not support natively (e.g, required attributes.)
- Every Arachne application must include either Datomic Free, Datomic Pro or Datascript as a dependency.
- We will need to keep our eyes open to look for situations where a more formal ontology system might be a better choice.