Architecture Decision Record: Data Abstraction Model

Context

Most applications need to store and manipulate data. In the current state of the art in Clojure, this is usually done in a straightforward, ad-hoc way. Users write schema, interact with their database, and parse data from user input into a persistence format using explicit code.

This is acceptable, if you're writing a custom, concrete application from scratch. But it will not work for Arachne. Arachne's modules need to be able to read and write domain data, while also being compatible with multiple backend storage modules.

For example a user/password based authentication module needs to be able to read and write user records to the application database, and it should work whether a user is using a Datomic, SQL or NoSQL database.

In other words, Arachne cannot function well in a world in which every module is required to interoperate directly against one of several alternative modules. Instead, there needs to be a way for modules to "speak a common language" for data manipulation and persistence.

Other use cases

Data persistence isn't the only concern. There are many other situations where having a common, abstract data model is highly useful. These include:

- quickly defining API endpoints based on a data model
- HTML & mobile form generation
- generalized data validation tools
- unified administration & metrics tools

Modeling & Manipulation

There are actually two distinct concepts at play; data *modeling* and data *manipulation*.

Modeling is the activity of defining the abstract shape of the data; essentially, it is writing schema, but in a way that is not specific to any concrete implementation. Modules can then use the data model to generate concrete schema, generate API endpoints, forms, validate data, etc.

Manipulation is the activity of using the model to create, read update or delete actual data. For an abstract data manipulation layer, this generally means a polymorphic API that supports some common set of implementations, which can be extended to concrete CRUD operations

Existing solutions: ORMs

Most frameworks have some answer to this problem. Rails has ActiveRecord, Elixir has Ecto, old-school Java has Hibernate, etc. In every case, they try to paper over what it looks like to access the actual database, and provide an idiomatic API in the language to read and persist data. This language-level API is uniformly designed to make the database "easy" to use, but also has the effect of providing a common abstraction point for extensions.

Unfortunately, ORMs also exhibit a common set of problems. By their very nature, they are an extra level of indirection. They provide abstraction, but given how complex databases are the abstraction is always "leaky" in significant ways. Using them effectively requires a thorough understanding not only of the ORM's APIs, but also the underlying database implementation, and what the ORM is doing to map the data from one format to another.

ORMs are also tied more or less tightly to the relational model. Attempts to extend ActiveRecord (for example) to non-relational data stores have had varying levels of success.

Database "migrations"

One other function is to make sure that the concrete database schema matches the abstract data model that the application is using. Most ORMs implement this using some form of "database migrations", which serve as a repeatable series of all changes made to a database. Ideally, these are not redundant with the abstract data model, to avoid repeating the same information twice and also to ensure consistency.

Decision

Arachne will provide a lightweight model for data abstraction and persistence, oriented around the Entity/Attribute mode. To avoid word salad and acronyms loaded with baggage and false expectations, we will give it a semantically clean name. We will be free to define this name, and set expectations around what it is and how it is to be used. I suggest "Chimera", as it is in keeping with the Greek mythology theme and has several relevant connotations.

Chimera consists of two parts:

- An entity model, to allow application authors to easily specify the shape of their domain data in their Arachne configuration.
- A set of persistence operations, oriented around plain Clojure data (maps, sets and vectors) that can be implemented meaningfully against multiple types of adapters. Individual operations are granular and can be both consumed and provided á la carte; adapters that don't support certain behaviors can omit them (at the cost of compatibility with modules that need them.)

Although support for any arbitrary database cannot be guaranteed, the persistence operations are designed to support a majority of commonly used systems, including relational SQL databases, document stores, tuple stores, Datomic, or other "NoSQL" type systems.

At the data model level, Chimera should be a powerful, easy to use way to specify the structure of your data, as data. Modules can then read this data and expose new functionality driven by the application domain model. It needs to be flexible enough that it can be "projected" as schema into diverse types of adapters, and customizable enough that it can be configured to adapt to existing database installations.

Adapters

Chimera *Adapters* are Arachne modules which take the abstract data structures and operations defined by Chimera, and extend them to specific databases or database APIs such as JDBC, Datomic, MongoDB, etc.

When applicable, there can also be "abstract adapters" that do the bulk of the work of adapting Chimera to some particular genre of database. For example, most key/value stores have similar semantics and core operations: there will likely be a "Key/Value Adapter" that does the bulk of the work for adapting Chimera's

operations to key/value storage, and then several thin *concrete* adapters that implement the actual get/put commands for Cassandra, DynamoDB, Redis, etc.

Limitations and Drawbacks

Chimera is designed to make a limited set of common operations *possible* to write generically. It is not and cannot ever be a complete interface to every database. Application developers *can* and *should* understand and use the native APIs of their selected database, or use a dedicated wrapper module that exposes the full power of their selected technology. Chimera represents only a single dimension of functionality; the entity/attribute model. By definition, it cannot provide access to the unique and powerful features that different databases provide and which their users ought to leverage.

It is also important to recognize that there are problems (even problems that modules might want to tackle) for which Chimera's basic entity/attribute model is simply not a good fit. If the entity model isn't a good fit, do not use Chimera. Instead, find (or write) an Arachne module that defines a data modeling abstraction better suited for the task at hand.

Examples of applications that might not be a good fit for Chimera include:

- Extremely sparse or "wide" data
- Dynamic data which cannot have pre-defined attributes or structure
- Unstructured heterogeneous data (such as large binary or sampling data)
- Data that cannot be indexed and requires distributed or streaming data processing to handle effectively

Modeling

The data model for an Arachne application is, like everything else, data in the Configuration. Chimera defines a set of DSL forms that application authors can use to define data models programmatically, and of course modules can also read, write and modify these definitions as part of their normal configuration process.

Note: The configuration schema, including the schema for the data model, is *itself* defined using Chimera. This requires some special bootstrapping in the core module. It also implies that Arachne core has a dependency on Chimera. This does not mean that modules are required to use Chimera or that Chimera has some special status relative to other conceivable data models; it just means that it is a good fit for modeling the kind of data that needs to be stored in the configuration.

Modeling: Entity Types

Entity types are entities that define the structure and content for a *domain entity*. Entity types specify a set of optional and required *attributes* that entities of that type must have.

Entity types may have one or more supertypes. Semantically, supertypes imply that any entity which is an instance of the subtype is also an instance of the supertype. Therefore, the set of attributes that are valid or required for an entity are the attributes of its types and all ancestor types.

Entity types define only data structures. They are not objects or classes; they do not define methods or behaviors.

In addition to defining the structure of entities themselves, entity types can have additional config attributes that serve as implementation-specific hints. For example, an entity type could have an attribute to override

the name of the SQL table used for persistence. This config attribute would be defined and used by the SQL module, not by Chimera itself.

The basic attributes of the entity type, as defined by Chimera, are:

- The name of the type (as a namespace-qualified keyword)
- Any supertypes it may have
- What attributes can be applied to entities of this type

Attribute Definitions

Attribute Definition entities define what types of values can be associated with an entity. They specify:

1. The name of the attribute (as a namespace-qualified keyword)
2. The min and max cardinality of an attribute (thereby specifying whether it is required or optional)
3. The type of allowed values (see the section on *Value Types* below)
4. Whether the attribute is a *key*. The values of a key attribute are expected to be globally unique, guaranteed to be present, and serve as a way to find specific entities, no matter what the underlying storage mechanism.
5. Whether the attribute is *indexed*. This is primarily a hint to the underlying database implementation.

Like entity types, attribute definitions may have any number of additional attributes, to modify behavior in an implementation-specific way.

Value Types

The value of an attribute may be one of three types:

1. A **reference** is a value that is itself an entity. The attribute must specify the entity type of the target entity.

2. A **component** is a reference, with the added semantic implication that the value entity is a logical "part" of the parent entity. It will be retrieved automatically, along with the parent, and will also be deleted/retracted along with the parent entity.

3. A **primitive** is a simple, atomic value. Primitives may be one of several defined types, which map more or less directly to primitive types on the JVM:

    - Boolean (JVM `java.lang.Boolean`)
    - String (JVM `java.lang.String`)
    - Keyword (Clojure `clojure.lang.Keyword`)
    - 64 bit integer (JVM `java.lang.Long`)
    - 64 bit floating point decimal (JVM `java.lang.Double`)
    - Arbitrary precision integer (JVM `java.math.BigInteger`)
    - Arbitrary precision decimal (JVM `java.math.BigDecimal`)
    - Instant (absolute time with millisecond resolution) (JVM `java.util.Date`)
    - UUID (JVM `java.util.UUID`)
    - Bytes (JVM byte array). Since not all storages support binary data, and might need to serialize it with base64, this should be fairly small.

> This set of primitives represent a reasonable common denominator that is supportable on most target databases. Note that the set is not closed: modules can specify new primitive types that are logically "subtypes" of the generic primitives. Entirely new types can also be defined (with the caveat that they will only work with adapters for which an implementation has been defined.)

Validation

All attribute names are namespace-qualified keywords. If there are specs registered using those keywords, they can be used to validate the corresponding values.

Clojure requires that a namespace be loaded before the specs defined in it are globally registered. To ensure that all relevant specs are loaded before an application runs, Chimera provides config attributes that specify namespaces containing specs. Arachne will ensure that these namespaces are loaded first, so module authors can ensure that their specs are loaded before they are needed.

Chimera also provides a `generate-spec` operation which programmatically builds a spec for a given entity type, that can validate a full entity map of that type.

Schema & Migration Operations

In order for data persistence to actually work, the schema of a particular database instance (at least, for those that have schema) needs to be compatible with the application's data model, as defined by Chimera's entity types and attributes.

See ADR-16 for an in-depth discussion of database migrations work, and the ramifications for how a Chimera data model is declared in the configuration.

Entity Manipulation

The previous section discussed the data *model*, and how to define the general shape and structure of entities in an application. Entity *manipulation* refers to how the operations available to create, read, update, delete specific *instances* of those entities.

Data Representation

Domain entities are represented, in application code, as simple Clojure maps. In their function as Chimera entities, they are pure data; not objects. They are not required to support any additional protocols.

Entity keys are restricted to being namespace-qualified keywords, which correspond with the attribute names defined in configuration (see *Attribute Definitions* above). Other keys will be ignored in Chimera's operations. Values may be any Clojure value, subject to spec validation before certain operations.

Cardinality-many attributes *must* use a Clojure sequence, even if there is only one value.

Reference values are represented in one of two ways; as a nested map, or as a *lookup reference*.

Nested maps are straightforward. For example:

```
{:myapp.person/id 123
 :myapp.person/name "Bill"
```

```
  :myapp.person/friends [{:myapp.person/id 42
                          :myapp.person/name "Joe"}]}
```

Lookup references are special values that identify an attribute (which must be a key) and value to indicate the target reference. Chimera provides a tagged literal specifially for lookup references.

```
{:myapp.person/id 123
 :myapp.person/name "Bill"
 :myapp.person/friends [#chimera.key[:myapp.person/id 42]]}
```

All Chimera operations that return data should use one of these representations.

Both representations are largely equivalent, but there is an important note about passing nested maps to persistence operations: the intended semantics for any nested maps must be the same as the parent map. For example, you cannot call `create` and expect the top-level entity to be created while the nested entity is updated.

Entities do *not* need to explicitly declare their entity type. Types may be derived from inspecting the set of keys and comparing it to the Entity Types defined in the configuration.

Persistence Operations

The following basic operations are defined:

- `get` - Given an attribute name and value, return a set of matching entity maps from the database. Results are not guaranteed to be found unless the attribute is indexed. Results may be truncated if there are more than can be efficiently returned.
- `create` - Given a full entity map, transactionally store it in the database. Adapters *may* throw an error if an entity with the same key attribute and value already exists.
- `update` - Given a map of attributes and values update each of the attributes provided attributes to have new values. The map must contain at least one key attribute. Also takes a set of attribute names which will be deleted/retracted from the entity. Adapters *may* throw an error if no entity exists for the given key.
- `delete` - Given a key attribute and a value, remove the entity and all its attributes and components.

All these operations should be transactional if possible. Adapters which cannot provide transactional behavior for these operations should note this fact clearly in their documentation, so their users do not make false assumptions about the integrity of their systems.

Each of these operations has its own protocol which may be required by modules, or satisfied by adapters à la carte. Thus, a module that does not require the full set of operations can still work with an adapter, as long as it satisfies the operations that it *does* need.

This set of operations is not exhaustive; other modules and adapters are free to extend Chimera and define additional operations, with different or stricter semantics. These operations are those that it is possible to implement consistently, in a reasonably performant way, against a "broad enough" set of very different types of databases.

To make it possible for them to be composed more flexibly, operations are expressed as data, not as direct methods.

Capability Model

Adapters must specify a list of what operations they support. Modules should validate this list at runtime, to ensure the adapter works with the operations that they require.

In addition to specifying whether an operation is supported or not, adapters must specify whether they support the operation idempotently and/or transactionally.

Status

PROPOSED

Consequences

- Users and modules can define the shape and structure of their domain data in a way that is independent of any particular database or type of database.
- Modules can perform basic data persistence tasks in a database-agnostic way.
- Modules will be restricted to a severely limited subset of data persistence functionality, relative to using any database natively.
- The common data persistence layer is optional, and can be easily bypassed when it is not a good fit.
- The set of data persistence operations is open for extension.
- Because spec-able namespaced keywords are used pervasively, it will be straightforward to leverage Spec heavily for validation, testing, and seed data generation.