Architecture Decision Record: Database Migrations

Context

In general, Arachne's philosophy embraces the concepts of immutability and reproducibility; rather than *changing* something, replace it with something new. Usually, this simplifies the mental model and reduces the number of variables, reducing the ways in which things can go wrong.

But there is one area where this approach just can't work: administering changes to a production database. Databases must have a stable existence across time. You can't throw away all your data every time you want to make a change.

And yet, some changes in the database do need to happen. Data models change. New fields are added. Entity relationships are refactored.

The challenge is to provide a way to provide measured, safe, reproducible change across time which is *also* compatible with Arachne's target of defining and describing all relevant parts of an application (including it's data model (and therefore schema)) in a configuration.

Compounding the challenge is the need to build a system that can define concrete schema for different types of databases, based on a common data model (such as Chimera's, as described in ADR-15.)

Prior Art

Several systems to do this already exist. The best known is probably Rails' Active Record Migrations, which is oriented around making schema changes to a relational database.

Another solution of interest is Liquibase, a system which reifies database changes as data and explicitly applies them to a relation database.

Scenarios

There are a variety of "user stories" to accomodate. Some examples include:

1. You are a new developer on a project, and want to create a local database that will work with the current HEAD of the codebase, for local development.
2. You are responsible for the production deployment of your project, and your team has a new software version ready to go, but it requires some new fields to be added to the database before the new code will run.
3. You want to set up a staging environment that is an exact mirror of your current production system.
4. You and a fellow developer are merging your branches for different features. You both made different changes to the data model, and you need to be sure they are compatible after the merge.
5. You recognize that you made a mistake earlier in development, and stored a currency value as a floating point number. You need to create a new column in the database which uses a fixed-point type, and copy over all the existing values, using rounding logic that you've agreed on with domain experts.

Decision

Chimera will explicitly define the concept of a migration, and reify migrations as entities in the configuration.

A migration represents an atomic set of changes to the schema of a database. For any given database instance, either a migration has logically been applied, or it hasn't. Migrations have unique IDs, expressed as namespace-qualified keywords.

Every migration has one or more "parent" migrations (except for a single, special "initial" migration, which has no parent). A migration may not be applied to a database unless all of its parents have already been applied.

Migrations are also have a *signature*. The signature is an MD5 checksum of the *actual content* of the migration as it is applied to the database (whether that be txdata for Datomic, a string for SQL DDL, a JSON string, etc.) This is used to ensure that a migration is not "changed" after it has already been applied to some persistent database.

Adapters are responsible for exposing an implementation of migrations (and accompanying config DSL) that is appropriate for the database type.

Chimera Adapters must additionally satisfy two runtime operations:

- `has-migration?` - takes ID and signature of a particular migration, and returns true if the migration has been successfully applied to the database. This implies that databases must be "migration aware" and store the IDs/signatures of migrations that have already been applied.
- `migrate` - given a specific migration, run the migration and record that the migration has been applied.

Migration Types

There are four basic types of migrations.

1. **Native migrations**. These are instances of the migration type directly implemented by a database adapter, and are specific to the type of DB being used. For example, a native migration against a SQL database would be implemented (primarily) via a SQL string. A native migration can only be used by adapters of the appropriate type.
2. **Chimera migrations**. These define migrations using Chimera's entity/attribute data model. They are abstract, and should work against multiple different types of adapters. Chimera migrations should be supported by all Chimera adapters.
3. **Sentinel migrations**. These are used to coordinate manual changes to an existing database with the code that requires them. They will always fail to automatically apply to an existing database: the database admin must add the migration record explicitly after they perform the manual migration task. *(Note, actually implementing these can be deferred until if or when they are needed)*.

Structure & Usage

Because migrations may have one or more parents, migrations form a directed acyclic graph.

This is appropriate, and combines well with Arachne's composability model. A module may define a sequence of migrations that build up a data model, and extending modules can branch from any point to build their own data model that shares structure with it. Modules may also depend upon a chain of migrations specified in two dependent modules, to indicate that it requires both of them.

In the configuration, a Chimera **database component** may depend on any number of migration components. These migrations, and all their ancestors, form a "database definition", and represent the complete schema of

a concrete database instance (as far as Chimera is concerned.)

When a database component is started and connects to the underlying data store, it verifies that all the specifies migrations have been applied. If they have not, it fails to start. This guarantees the safety of an Arachne system; a given application simply will not start if it is not compatible with the specified database.

Parallel Migrations

This does create an opportunity for problems: if two migrations which have no dependency relatinship ("parallel migrations") have operations that are incompatible, or would yield different results depending on the order in which they are applied, then these operations "conflict" and applying them to a database could result in errors or non-deterministic behavior.

If the parallel migrations are both Chimera migrations, then Arachne is aware of their internal structure and can detect the conflict and refuse to start or run the migrations, before it actually touches the database.

Unfortunately, Arachne cannot detect conflicting parallel migrations for other migration types. It is the responsibility of application developers to ensure that parallel migrations are logically isolate and can coexist in the same database without conflict.

Therefore, it is advisable in general for public modules to only use Chimera migrations. In addition to making them as broadly compatible as possible, and will also make it more tractable for application authors to avoid conflicting parallel migrations, since they only have to worry about those that they themselves create.

Chimera Migrations & Entity Types

One drawback of using Chimera migrations is that you cannot see a full entity type defined in one place, just from reading a config DSL script. This cannot be avoided: in a real, living application, entities are defined over time, in many different migrations as the application grows, not all at once. Each Chimera migration contains only a fragment of the full data model.

However, this poses a usability problem; both for developers, and for machine consumption. There are many reasons for developers or modules to view or query the entity type model as a "point in time" snapshot, rather than just a series of incremental changes.

To support this use case, the Chimera module creates a flat entity type model for each database by "rolling up" the individual Chimera entity definition forms into a single, full data structure graph. This "canonical entity model" can then be used to render schema diagrams for users, or be queried by other modules.

Applying Migrations

When and how to invoke an Adapter's `migrate` function is not defined, since different teams will wish to do it in different ways.

Some possibilities include:

1. The application calls "migrate" every time it is started (this is only advisable if the database has excellent support for transactional and atomic migrations.) In this scenario, developers only need to worry about deploying the code.

2. The devops team can manually invoke the "migrate" function for each new configuration, prior to deployment.
3. In a continuous-deployment setup, a CI server could run a battery of tests against a clone of the production database and invoke "migrate" automatically if they pass.
4. The development team can inspect the set of migrations and generate a set of native SQL or txdata statements for handoff to a dedicated DBA team for review and commit prior to deployment.

Databases without migrations

Not every application wants to use Chimera's migration system. Some situations where migrations may not be a good fit include:

- You prefer to manage your own database schema.
- You are working with an existing database that predates Arachne.
- You need to work with a database administered by a separate team.

However, you still may wish to utilize Chimera's entity model, and leverage modules that define Chimera migrations.

To support this, Chimera allows you to (in the configuration) designate a database component as "**assert-only**". Assert-only databases never have migrations applied, and they do not require the database to track any concept of migrations. Instead, they inspect the Chimera entity model (after rolling up all declared migrations) and assert that the database *already* has compatible schema installed. If it does, everything starts up as normal; if it does not, the component fails to start.

Of course, the schema that Chimera expects most likely will not be an exact match for what is present in the database. To accomodate this, Chimera adapters defines a set of *override* configuration entities (and accompanying DSL). Users can apply these overrides to change the behavior of the mappings that Chimera uses to query and store data.

Note that Chimera Overrides are incompatible with actually running migrations: they can be used only on an "assert-only" database.

Migration Rollback

Generalized rollback of migrations is intractable, given the variety of databases Chimera intends to support. Use one of the following strategies instead:

- For development and testing, be constantly creating and throwing away new databases.
- Back up your database before running a migration
- If you can't afford the downtime or data loss associated with restoring a backup, manually revert the changes from the unwanted migration.

Status

PROPOSED

Consequences

- Users can define a data model in their configuration

- The data model can be automatically reflected in the database
- Data model changes are explicitly modeled across time
- All migrations, entity types and schema elements are represented in an Arachne app's configuration
- Given the same configuration, a database built using migrations can be reliably reproduced.
- A configuration using migrations will contain an entire, perfectly reproducible history of the database.
- Migrations are optional, and Chimera's data model can be used against existing databases