DRAFT: Airavata Deprecation/Migration Guidance

Key principles

- **Expand-migrate-contract** - https://www.martinfowler.com/bliki/ParallelChange.html
    - Summary: changes to an interface require first adding the updated interface (expand), then migrating clients to using the updated interface (migration), then finally removing the old interface (contract). Here "interface" might be, for example, a service API or database schema. As a simple example, consider a database table column name change. First, a column with the new name would be added, then all code referencing the old column would be updated to reference the new column name, then finally the old column could be removed.
- **Independently deployable services** - updates to one service shouldn't require a simultaneous update to clients of that service, allowing a service to be independently updated of other services. The situation we want to avoid is when two services must be updated/deployed simultaneously because changes in one service breaks the other one. Here "service" would be any network service that has clients such as a database server, microservices, REST API, web or native UI, etc.
- **Feature flags** - use feature flags to push new functionality into production for testing and validation without impacting existing clients
    - https://en.wikipedia.org/wiki/Feature_toggle

Release strategy

- We have to think about not only SciGaP operations, where we might release into production on a daily or weekly basis, but also official Apache releases and how migrations will impact users upgrading to a new release of Airavata.
- Breaking changes in public interfaces cannot be introduced from one release to another. The updated method should be added to the public interface in one release while the old method is marked as deprecated. The old method can only be removed in a future release.
- How long to wait until removing deprecated methods?
    - One we have major releases (i.e., 1.0, 2.0) we could remove functionality deprecated in version N in version N+1

        - This is similar to what Django does: https://docs.djangoproject.com/en/3.0/internals/release-process/#deprecation-policy
    - Until such time, perhaps we wait at least one year (especially if we move to time-based release like Galaxy)
- Deprecated functionality should log warnings. Ideally the warning message will let us know who/what is calling the deprecated functionality. The warning message should also clearly state what alternative should be used instead.
- Nice to have: client SDKs should also be updated to log warnings since API users will be more likely to notice these warnings than server-side warnings

Specific guidance

DB Migrations

- Adding a new table - this is fairly straightforward since the table is new.
- Renaming a table
    - Create a DB migration to add the new table
        - create the new table schema
        - populate the new table with values from the old table
        - (if appropriate) replace the old table with an updateable view with the old table's name -- this way inserts/updates to the old table will be reflected in the new table
            - CREATE VIEW old_table AS SELECT * FROM new_table;
        - (if appropriate) alternatively, create a trigger to synchronize changes to the old table to the new table
    - Update code using the old database table to use the new table
    - In a future release or after all clients are updated, create a DB migration to remove the old table
- Dropping a table
    - Remove code using the database table
    - In a future release or after all clients are updated, create a DB migration to remove the table
- Adding a new column
    - Create a DB migration to add the new column
        - populate the new column as appropriate
        - the column must have a DEFAULT value or be NULLABLE
            - alternatively create a trigger to populate the default value when none is specified by code that is not yet updated to use the new column
    - Update code to use the new database column

    - In a future release or after all clients are updated, create a DB migration to remove default value or trigger, if applicable
- Renaming a column
    - Create a DB migration to add a column with the new name
        - populate the column with values from the column with the old name
        - the column must have a DEFAULT value or be NULLABLE
            - alternatively create a trigger to populate the default value when none is specified by code that is not yet updated to use the new column
            - another idea: create an updateable view that maps the old and new column names to the new column
    - Update code to use the new database column
    - In a future release or after all clients are updated, create a DB migration to remove the column with the old name
        - also, if applicable, do a final sync of data from the old column name to the new column name
        - also remove default value or trigger on column with new name, if applicable
- Dropping a column
    - Remove code using the old column name
    - In a future release or after all clients are updated, create a DB migration to remove the column
- schema refactor/data migration
    - These are more complicated schema or data changes that don't neatly fit into the categories above. In general, these will need to be implemented with the principles listed above and in a way that is to a certain extent idiosyncratic to each migration.
    - In general the steps are:
        - create a DB migration for the new table structure
        - if applicable, also create a data migration script to move the data from the old table structure to the new table structure
        - if applicable, also create a DB migration that introduces updateable views or triggers that will synchronize data from the old table structure to the new table structure
        - Update code to use the new table structure
        - In a future release or after all clients are updated, create a DB migration to:
            - if applicable, run a final data migration
            - remove the old table structure