# General Design and Development Notes

## Main Behaviors

- Define, Edit, Explore, Modify, Annotate a Schema-Ontology.
- Check the Ontology for errors, inconsistencies, gaps.
- Generate SQL code from the Ontology.
- Generate graph code from the Ontology.
- Generate key-value (Redis) database code from the Ontology.
- Execute code to create, modify databases like:
    - SQLite
    - Redis
    - Postgres
- Check code. Optimize code. Store code. Edit code.
- Functions enabled by the code:
    - Create objects.
    - Backup and Drop objects.
    - Insert, Update, Remove data content.
    - Query (Get, Join) data content.

## Design and Architecture

- Nginx (maybe), some Tornado (maybe), Asyncio, Qt5/PySide2 (definitely), Sqlite, Redis
- 100% Python (with JavaScript, HTML, CSS as needed)
- Design first and foremost as a local app. Worry about web *components* later.
- Think in terms of middleware, of services, of messages, of reactive systems.
- Nginx puts up a public IP. All other IPs are private.
    - The "work" (backend, middleware) happens on private servers, even with public-facing app or app components.
- All behaviors (as much as feasible and sensible) are messages, are event-based.
- Logger, monitor, wiretap are componentized and used everywhere.
- Specialized Saskan FileIO is a class.
- Think events, think CLI, then think GUI.
- Major components should have their own Git Repo; all interactions with other major components via message layers, queues, etc.
- Minor components should have their own Classes, as much as feasible.
- Intra-class communication can be more tightly bound. However, avoid creating layers of sub-classes.
- Use BowQuiver for major shared functions, but don't get overly obsessed with avoiding redundancy.

## Messaging

- Every functional component has one or more Listeners and Senders, and is associated with a Message Server.
- Consider explaining messages using AsyncAPI. See: https://www.asyncapi.com/
    - This creates nice documentation. Same niceness as OpenAPI, etc. for REST but aimed at messaging.

    - Could store in Redis.
- Redis can handle Message Queues. But don't be in a big rush to use Redis for messaging. Learn the ropes. Explore everything that asyncio has to offer first.
    - May eventually want to check out Nginx/Nchan messaging.
    - Still kind of intrigued by rabbitmq too.
    - Message streams are interesting too. I think Redis supports them. Also see RxPy/ReactiveX.
    - And Kafka.

## Heuristics

- Identify what layers to work on when working on a given feature.
- Always:
    - Try an event-based and message-based approach first.
    - Prototype simple cases before developing more complex ones.
    - Keep thinking in terms of message/service-based reactive design.
    - Deploy to cloud/Digital Ocean only what can be done safely, securely.
        - Double-that for web-enabled.
    - Keep code modular.
        - "Do one thing well."
    - Strive to avoid procedural, sequential dependencies at both macro and micro levels.
        - Checking state is better than setting flags or making assumptions.
    - Keep docs, diagrams, issues in synch with code as much as is useful.

## Feature List / Hit Parade

- Follow on/repeat/enhance work done earlier. Build on good ideas and existing work.
- Use best tools for the job. Don't feel like I have to use everything.
- Ontology/Schema is the core of the app.
- DB-generation and generated for dynamic enhancements, configurations to services is A+.
    - That means making good use of metadata.
- A GUI Framework that I actually know how to use and works is very desirable.
- Event-based, non-blocking, non-JavaScript architecture please.
- Message-based, asychronous services that can handle pub-sub? Yes, that too.
- Monitoring and notifications to enhance performance, detect streeses? Unh-hunh. Yup.