**BAO Build & Integration Workflow (modularization-compatible)**

First step is complete the documentation I started with every step the way it actually should work - the way we want it to work moving forward. As a foundation for specification for the suite of tools we need

Second step is then to make a detailed plan.  
loading data from the flat files into the database  
Cleaning up and expanding and reorganizing the database  
From there build the files  
We want template files for the BAO Upper Level - sort of a BAO Upper Level ontology that holds together all the external and internal pieces  
We need to decide if we want to keep BAO core and of course we need to keep BAO complete.  
We need a separate expert bao axiom file that feeds into BAO complete. There maybe a bao axiom file that feeds into bao core.in one example this could look like:  
bao vocabularies and external modules feed into bao upper level and together with expert axioms into bao completeor we could add the bao modules in between; but I almost think that the bao module and bao core can be in the BAO upper level ontology - or we repurpose BAO core for that function.These all need to be autobuilt from the database, except the

Then we need to reverse engineer the simple axioms in current bao core and bao complete and see which ones are in the database and if how they are currently added to these files, and which axioms are expert (more complex nested) and not the the database and have to move into the expert axioms file

we then have to rebuild everything until we produce the current version exactly, but in a clean way

can you use this please as a starting point for a plan

Curator Spreadsheet Template

Add Term

|  |  |
| --- | --- |
| **Column field** | **Field definition** |
| label | A **label** is a human-readable name or descriptor assigned to a **class**. |
| parent BAO ID | The parent BAO ID establishes the "is-a" (subclass) relationship in the hierarchy. |
| definition | A **definition** is a formal, unambiguous description of the meaning of a **class**. |
| source | The **definition source** specifies where the definition originates to prevent ambiguity. |

Add Synonym

|  |  |
| --- | --- |
| **Column field** | **Field definition** |
| BAO ID | A **BAO ID** is a unique, machine-readable identifier assigned to terms within the **BioAssay Ontology (BAO)** |
| label | A **label** is a human-readable name or descriptor assigned to a **class**. |
| synonym | A **synonym** is an alternative label or term that refers to the same concept (class). |

Rename Term

Add Definition

Add Annotations(?)

Move Term

Deprecate

**1) Data Ingestion & ID Management (BAODB)**

* Inputs: New or revised classes, labels, definitions, and metadata in standardized tab-delimited CSV.
* Load & IDs: CSVs are ingested into BAODB, which assigns and manages BAO IDs and ID ranges (via the DB and ingest scripts). These IDs are authoritative and reused downstream.
* Axiom scope in DB: BAODB stores simple axioms only (e.g., direct subsumptions and other simple logical axioms with shallow AND/OR). Complex/nested axioms are intentionally not in BAODB.
* External references: BAODB records required IRIs for external classes to be imported later.

**2) BAO Module & Vocabulary Generation (OntoJog)**

* Reads BAODB, preserves IDs: OntoJog generates OWL modules and vocabularies from BAODB content, using the pre-assigned BAO IDs (it does not create or alter identifiers).
* What OntoJog emits from BAODB:
  + Vocabulary OWL files (taxonomy only; subsumption trees for BAO components).
  + Core BAO logical modules built from simple axioms in BAODB.
  + OntoFox input lists (IRIs of required external classes).
* Manual/complex axioms via templates:
  + User-generated complex axioms are maintained outside BAODB as template-based axiom files (“manual axioms”).
  + OntoJog respects and includes these axiom templates at build time; it does not overwrite them. This preserves expert modeling (e.g., nested intersections/restrictions) while keeping BAODB simple.
* QC: Diff/validation reports compare the generated build with the prior release (added/changed/deprecated classes & axioms).

**3) External Class Retrieval (OntoFox)**

* Inputs: OntoFox input lists (IRIs) produced by OntoJog.
* Process: OntoFox retrieves the specified external classes (e.g., GO, CLO, UO, DOID, ChEBI, UBERON, BFO/RO) with relevant annotations and property axioms, yielding external OWL modules.
* Modular alignment: Overlaps between internal BAO classes and external classes are resolved in mapping/“combinator” modules via equivalence or subsumption axioms, keeping BAO core stable while allowing external updates.

**4) Ontology Integration & Final Assembly (self-assembly)**

* No programmatic merge: The final ontology self-assembles via owl:imports, shared IRIs, and axioms. Loading the release in Protégé or a reasoner resolves all imports and relationships automatically.
* What “fits together”:
  + BAODB-derived modules (OntoJog) + Vocabulary files (terms/taxonomies).
  + Manual (template) axiom files (user-generated complex axioms, preserved as-is).
  + External OntoFox modules (imported classes + supporting axioms).
  + Combinator/mapping modules (equivalence/subsumption to align external with internal).
* Release artifacts (perspectives/views):
  + bao\_core.owl — BAO core (internal vocabularies + core axioms; minimal imports)
  + bao\_complete.owl — BAO complete (imports core + external modules + mappings)
  + Internal BAO vocabulary OWL files (taxonomy/terms only)
  + Internal BAO module OWL files (logical structure)
  + External BAO OWL modules (OntoFox outputs)
  + (Optional) BFO/RO-aligned perspective when needed for alignment work

**Understanding BAO Modularization: Notes for Students**

The **BioAssay Ontology (BAO)** is built in layers, much like assembling different parts of a complex system so that each piece can be developed, tested, and reused independently. This design is called **modularization**, and it ensures that BAO remains clear, scalable, and easy to maintain.

**1. The Layered Architecture (BAOP modularization paper, Figure 3)**

BAO is structured in layers that build on one another:

* **Vocabularies** – contain *terms only* (class names, labels, and definitions).
* **Modules** – combine vocabularies with *logical constraints* (relationships and hierarchy).
* **Axioms** – stored in separate files, define the logic that connects classes (e.g., *is a*, *part of*, *equivalent to*).
* **Perspectives or Views** – import and connect multiple modules to create complete “views” of the ontology (e.g., a **core** or a **complete** version).

Think of vocabularies as *the words*, modules as *the grammar rules*, and views as *the assembled story*.

**2. Internal vs. External Modules**

* **Internal modules** are the core of BAO—stable, curated, and carefully managed inside the project.
* **External modules** are imported from other ontologies (e.g., OBI, CLO, CHEBI).
* A special type of file called a **combinator module** connects external classes to BAO classes using *mapping axioms* (for example, *equivalentClass* or *subClassOf*).

This design ensures that BAO’s internal logic stays stable even when external ontologies change. Only the imported parts need to be refreshed, not the whole ontology.

**3. Why Modularization Matters for Scalability and Reasoning**

* **Reasoning** is how ontology software (like Protégé with a reasoner) automatically infers new relationships.
* Modularization keeps reasoning **tractable** – you can load only what you need.
  + Example: Load bao\_core.owl for a lightweight session.
  + Load bao\_complete.owl for full reasoning with all external content.
* This approach makes BAO flexible, efficient, and maintainable.

**4. Governance and Modeling Discipline**

* **Central ID management:** BAODB (the BAO database) assigns and tracks all class identifiers and ID ranges. This ensures every class has a stable, reproducible ID.
* **Modeling discipline:**
  + BAODB stores only *simple axioms* (basic “is a” and “part of” relations).
  + *Manual or complex axioms* (involving nested logic) are defined separately in curated **template files** by ontology experts.
  + OntoJog never overwrites these manual axioms – this preserves expert knowledge while keeping the automated build reliable.

**5. Robust Integration through OWL Self-Assembly**

* When all modules (internal, external, and templates) are loaded together, they **self-assemble** through shared IRIs and import declarations – no manual merging is needed.
* This **OWL-based self-assembly** makes the ontology deterministic (always builds the same way) and modular (easy to extend or update individual parts).

**Key Takeaways**

* **BAO’s modular design** mirrors good software engineering: separate components, clear dependencies, and version control.
* **Core principle:** Keep the database simple, preserve expert modeling externally, and let OWL imports do the heavy lifting.
* The result is a **stable, extensible, and FAIR** ontology that supports reasoning, reuse, and collaboration.

**Figure caption plus legend**

**A diagram of an owl

AI-generated content may be incorrect.**

**Figure X. Modular workflow for BAO generation, integration, and assembly.** The BioAssay Ontology (BAO) is developed and maintained through a modular, database-driven workflow that integrates internal ontology classes and axioms with external ontology content, ensuring semantic consistency, traceability, and scalability while maintaining compatibility with the published BAO modularization framework [ref]. The pipeline separates four major functions: **(A)** Data ingestion and ID management (BAODB) – new or revised classes, labels, and definitions are imported from standardized tab-delimited files into BAODB, which assigns and manages BAO identifiers and stores simple axioms (direct subsumptions and shallow logical constructs) along with external class IRIs. **(B)**Ontology module and vocabulary generation (OntoJog) – OntoJog reads BAODB to generate modular OWL files, including vocabulary and core logic modules, and produces OntoFox input lists for external IRIs. Complex, user-defined axioms are managed externally in curated template modules, which OntoJog preserves without overwriting. **(C)** External class retrieval (OntoFox) – OntoFox retrieves external ontology classes (e.g., from BFO, OBI, CLO, CHEBI, and UO) and their annotation and property axioms to create modular external OWL files, maintaining synchronization with reference ontologies. **(D)** Ontology integration and self-assembly – all BAODB-derived, template-based, and OntoFox modules interconnect through shared IRIs and import declarations, allowing the ontology to self-assemble in OWL environments such as Protégé. The final deliverables include ***bao\_core.owl*** (core modules with minimal imports), ***bao\_complete.owl*** (the full ontology including external modules and annotations), internal vocabulary and logic modules, and external OntoFox modules. This modular, FAIR-aligned architecture supports reproducible ontology evolution and scalable reasoning.

**Structured figure caption**

The BioAssay Ontology (BAO) is developed and maintained through a modular, database-driven workflow that integrates internal ontology classes and axioms with external ontology content. The process ensures semantic consistency, traceability, and scalability while maintaining compatibility with the previously published BAO modularization framework [ref BAO modularization paper].

**Panel A. Data Ingestion and ID Management (BAODB)**

* Input: New or revised classes, labels, definitions, and metadata prepared in standardized tab-delimited CSV files.
* Process: Files are ingested into BAODB, which automatically assigns and manages BAO identifiers (IDs) and ID ranges.
* Axioms in DB: BAODB stores simple axioms (direct subsumptions and shallow logical constructs). Complex axioms are intentionally excluded.
* Output: A curated, versioned database of classes, annotations, simple axioms, and external class IRIs.

**Panel B. Ontology Module and Vocabulary Generation (OntoJog)**

* Process: OntoJog reads BAODB to generate modular OWL files while preserving BAO IDs and hierarchical structure.
* Outputs:
  + Vocabulary OWL files: Containing class labels, definitions, and synonyms.
  + Core module OWL files: Containing logical structure and simple axioms from BAODB.
  + OntoFox input files: Text files listing external IRIs to be imported.
* Manual (template-based) axioms: Complex, user-generated axioms are managed externally in template modules. OntoJog respects these templates, ensuring they are preserved and not overwritten during regeneration.
* Quality Control: OntoJog generates diff reports to track added, modified, or deprecated classes and axioms.

**Panel C. External Class Retrieval (OntoFox)**

* Input: OntoFox input lists generated by OntoJog.
* Process: OntoFox retrieves external ontology classes (e.g., from BFO, OBI, CLO, CHEBI, UO) and their associated annotation and property axioms.
* Output: Modular external OWL files containing imported classes, properties, and relationships.
* Purpose: Keeps BAO synchronized with external ontology updates while maintaining internal stability.

**Panel D. Ontology Integration and Self-Assembly**

* Mechanism: The final ontology is not programmatically merged. Instead, all modules—BAODB-derived, template-based, and OntoFox-derived—self-assemble via shared IRIs, import declarations, and logical axioms.
* Integration: OntoJog axioms reference external IRIs, and OntoFox modules provide the corresponding external classes, forming a coherent structure when loaded in an OWL environment (e.g., Protégé).
* Modular Fit: Vocabulary, logical, external, and combinator modules align automatically through the OWL import hierarchy.
* Final Outputs:
  + bao\_core.owl — Core BAO modules and minimal imports.
  + bao\_complete.owl — Full BAO including all external modules and annotation content.
  + Internal vocabulary and module OWL files.
  + External (OntoFox) OWL modules.

**Summary:**

BAO’s modular build pipeline separates data storage, ontology generation, external term retrieval, and logical assembly. BAODB governs identifiers and simple axioms; OntoJog generates internal modules and respects manually curated template axioms; OntoFox imports external content; and the ontology self-assembles through OWL semantics, ensuring a consistent, extensible, and FAIR-aligned ontology architecture.