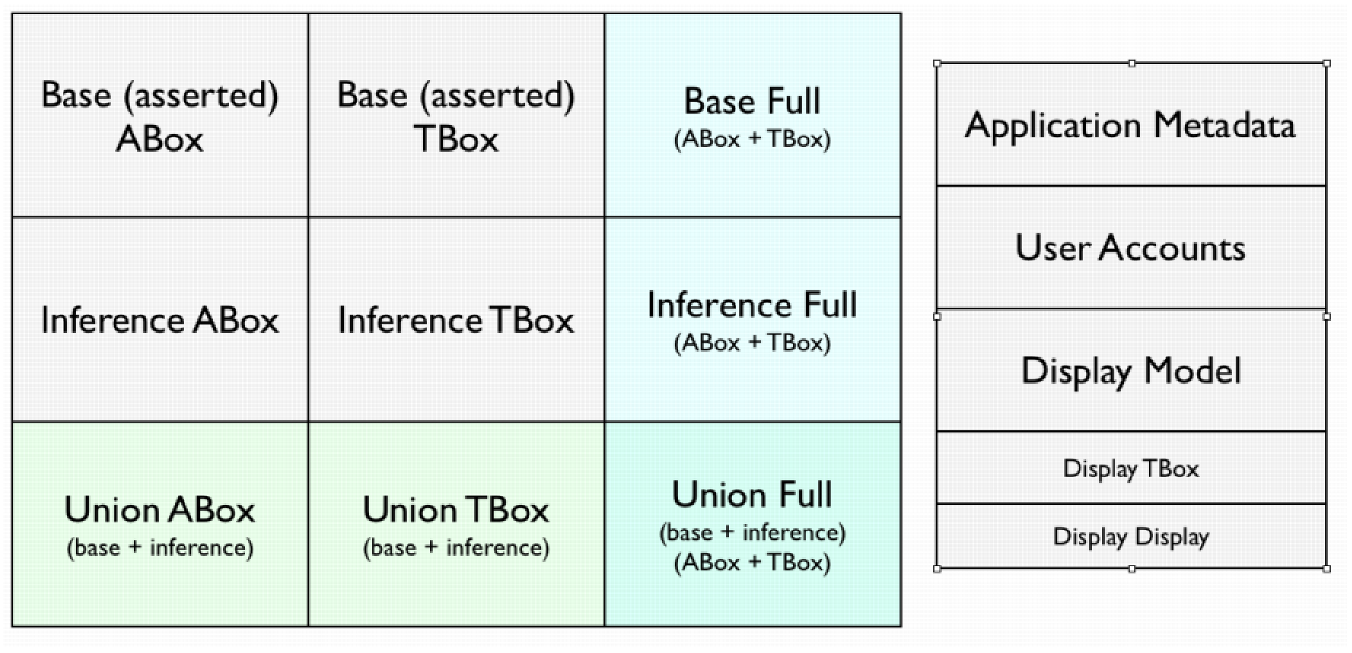
VIVO Data Models

Concepts



Frequently, we talk about "the data model" in VIVO. But this is an over-simplification which can be useful at times, but misleading at other times. In fact, VIVO contains a matrix of data models and sub-models, graphs, datasets and other constructs.

It might be more accurate to talk about the union of these data models as "the knowlege base". However, the terminology of "the data model" is firmly entrenched.

Beginning in VIVO release 1.6, we are attempting to simplify this complex collection of models, and to produce a unified access layer. This is a work in progress. Regardless of how clean the design might eventually become, this will remain an area with complex requirements which cannot be satisfied by simplistic solutions.

Divisions in the knowledge base

Depending on what you want to do with the data, it can be useful to sub-divide it by one or more of the following criteria:

Types of statements

An RDF model is often divided into ABox (assertions) and TBox (terminology). In RDF, there is no technical distinction between TBox and ABox data. They are stored separately because they are used for different purposes. The combination of the two is informally called the Full model.

	Data type	Example data

TBox	"Terminological data"
	
	Defines classes, properties, and relationships in your ontology.
		`foaf:Person`
		`   a owl:Class ;`
		`   rdfs:subClassOf owl:Thing ;`
		`   rdfs:label "Person"@en .`
		`ex:preferredName`
		`   a owl:DatatypeProperty ;`
		`   rdfs:subPropertyOf skos:prefLabel,`
		
		`foaf:name,`
		
		`rdfs:label ;`
		`   rdfs:domain foaf:Person ;`
		`   rdfs:label "preferred name"@en .`

ABox	"Assertion data"
	
	Enumerates the individual instances of your classes and describes them.
		`local:tobyink`
		`   a foaf:Person ;`
		`   ex:preferredName "Toby Inkster" .`

Full	The TBox and the ABox together, treated as a single model. For example, when you use the RDF tools to remove statements, you want them removed regardless of whether they are found in the TBox or the ABox.	

Source of statements

An RDF model can also be divided into Assertions and Inferences. The combination of the two is informally called the Union.

Statement type	Meaning	Example data
Assertions	Statements that you explicitly add to the model, either through setup, ingest, or editing.	`local:tobyink rdfs:type core:FacultyMember .`

Inferences	Statements that the semantic reasoner adds to the model, by reasoning about the assertions, or about other inferences.	`local:tobyink rdfs:type foaf:Person .`
		
		`local:tobyink rdfs:type foaf:Agent .`
		
		`local:tobyink rdfs:type owl:Thing .`

Union	The combination of Assertions and Inferences. For most purposes, this is the desired model. You want to know what statements are available, without regard to whether they were asserted or inferred.	

"Content" vs. "Configuration"

We sometimes distinguish between the data that VIVO is serving (Content) and the data that VIVO itself uses (Configuration). The Content is available for display, for searching, for serving as Linked Open Data. The Configuration controls how the content is displayed, who can access the data, and what VIVO itself looks like.

Model type	Purpose	Examples
Configuration	Data about the VIVO application itself.	Application parameters User Accounts Display options
Content	The payload - the data that VIVO is intended to distribute.	People data Publications data Grant data etc.

Model scope

The knowledge base exists for as long as VIVO is running. However, subsets or facets of the knowledge base are often used to satisfy a particular HTTP request, or through the length of a VIVO session for a particular user. These subsets are created dynamically from the full knowledge base, used for as long as they are useful, and then discarded.

Scope	Purpose	Example	Discarded when...
Application (Servlet Context)	Created for the life of VIVO.		Never discarded.
Session	Created for a particular logged-in user	Data that is filtered by what the user is permitted to view.	When the user logs out, or the session times out.
Request	Created for a single HTTP request	Data that is organized by the languages that are preferred by the browser.	When the individual request has been satisfied.

At present, the Session lifespan is almost never used. However, potential use cases do exist for it.

The Request lifespan is used extensively, since it provides a convenient way to manage database connections and minimize contention for resources.

Purpose vs. scope

It is tempting to think of the models of the Servlet Context as equivalent to the unfiltered models of the Request. They may even represent the very same data. However, they have different scope, which makes them very different in practice.

The unfiltered models in the Request go out of scope when the Request has been satisfied. The resources required by these models have short lifetimes and are very easily managed. The models of the Servlet Context never go out of scope until VIVO is shut down. It is difficult to reclaim resources such as database connections or processor memory from these models.

Filtering

To enable language filtering and internationalization of the user interface (translation of data labels and contents, localization, etc.), you should update the **runtime.properties** file:

```
RDFService.languageFilter = true
```

and then edit the list of needed languages:

```
languages.selectableLocales = en_US, es, de_DE, fr_CA, pt_BR
```

The Data Models

This is a summary of the data models:

The basic content	Base ABox, Base TBox, Inferred ABox, Inferred TBox	Named graphs from the RDF Service (optionally with sub-graphs).

Views of the content	Base Full, Inferred Full, Union ABox, Union TBox, Union Full	Views of the 4 basic content graphs in different combinations.
The configuration	Application Metadata, User Accounts, Display Model, Display TBox, DisplayDisplay	Named graphs from the application datasource.

Increasing complexity

The structure of the data models has grown as VIVO has developed. New models, new structures, and new means of accessing the data have been added as required by the growing code. The resulting data layer has grown more complex and more error-prone.

The `RDFService` interface, increases the flexibility of data sources, and promises to allow a more unified view of the knowledge base. However, the transition to `RDFService` is not complete, and so this adds another layer of complexity to the data issues. New structures have been added, but none removed.

Beyond the models

There is an incredible variety of ways to access all of these models. Some of this variety is because the models are accessed in different ways for different purposes. Additional variety stems from the evolution of VIVO in which new mechanisms were introduced without taking the time and effort to phase out older mechanisms.

Here are some of the ways for accessing data models:

Attributes on Context, Session, or Request

Previously, it was common to assign a model to the ServletContext, to the HTTP Session, or to the HttpSessionRequest like this:

```
OntModel ontModel = (OntModel) getServletContext().getAttribute("jenaOntModel");

Object sessionOntModel = request.getSession().getAttribute("jenaOntModel");
```

Occasionally, conditional code was inserted, to retrieve a model from the Request if available, and to fall back to the Session or the Context as necessary. Such code was sporadic, and inconsistent. This sort of model juggling also involved inversions of logic, with some code acting so a model in the Request would override one in the Session, while other code would prioritize the Session model over the one in the Request. For example:

```
public OntModel getDisplayModel(){
    if( _req.getAttribute("displayOntModel") != null ){
        return (OntModel) _req.getAttribute(DISPLAY_ONT_MODEL);
    } else {
        HttpSession session = _req.getSession(false);
        if( session != null ){
            if( session.getAttribute(DISPLAY_ONT_MODEL) != null ){
                return (OntModel) session.getAttribute(DISPLAY_ONT_MODEL);
            }else{
                if( session.getServletContext().getAttribute(DISPLAY_ONT_MODEL) != null){
                    return (OntModel)session.getServletContext().getAttribute(DISPLAY_ONT_MODEL);
                }
            }
        }
    }
    log.error("No display model could be found.");
    return null;
}
```

This mechanism has been removed in 1.6, being subsumed into the `ModelAccess` class (see below). Now, the `ModelAccess` attributes on Request, Session and Context are managed using code that is private to `ModelAccess` itself. Similarly, the code which gives priority to a Request model over a Session model is uniformly implemented across the models.

It remains to be seen whether this uniformity can satisfy the various needs of the application. If not, at least the changes can all be made within a single point of access.

The DAO layer

This mechanism is pervasive through the code, and remains quite useful. In it, a `WebappDaoFactory` is created, with access to particular data models. This factory then can be used to create DAO objects which satisfy interfaces like `IndividualDao`, `OntologyDAO`, or `UserAccountsDAO`. Each of these object implements a collection of convenience methods which are used to manipulate the backing data models.

Because the factory and each of the DAOs is an interface, alternative implementations can be written which provide

- Optimization for Jena RDB models
- Optimization for Jena SDB models
- Filtering of restricted data
- and more...

Initially, the `WebappDaoFactory` may have been used only with the full Union model. But what if you want to use these DAOs only against asserted triples? Or only against the ABox? This led to the `OntModelSelector`.

OntModelSelectors

An `OntModelSelector` provides a way to collect a group of Models and construct a `WebappDaoFactory`. With slots for ABox, TBox, and Full model, an `OntModelSelector` could provide a consistent view on assertions, or on inferences, or on the union. The `OntModelSelector` also holds references to a display model, an application metadata model, and a user accounts model, but these are more for convenience than flexibility.

Prior to release 1.6, `OntModelSelectors`, like `OntModels`, were stored in attributes of the Context, Session, and Request. They have been subsumed into the `ModelAccess` class.

Further, the semantics of the "standard" `OntModelSelectors` have changed, so they only act as facades before the Models store in `ModelAccess`. In this way, if we make this call:

```
ModelAccess.on(session).setOntModel(ModelID.BASE_ABOX, someWeirdModel)
```

Then both of the following calls would return the same model:

```
ModelAccess.on(session).getOntModel(ModelID.BASE_ABOX);
ModelAccess.on(session).getBaseOntModelSelector().getABoxModel();
```

Again, this is a change in the semantics of OntModelSelectors. It insures a consistent representation of `OntModels` across `OntModelSelectors`, but it is certainly possible that existing code relies on an inconsistent model instead.

The RDF Service

Interface for API to write, read, and update Vitro's RDF store, with support to allow listening, logging and auditing. Moreover, it is an intefrace for API to perform a SPARQL select query against the knowledge base. The query may have an embedded graph identifier. If the query does not contain a graph identifier the query is executed against the union of all named and unnamed graphs in the store.

At the end, implementation of this interface should enable serialization of the contents of the named graph to the supplied OutputStream, in N-Triples format.

Model makers and Model sources

The ModelAccess class

The root access point for the RDF data structures: RDFServices, Datasets, ModelMakers, OntModels, OntModelSelectors and WebappDaoFactories. Enables getting a long-term data structure by accessing from the context.

```
ModelAccess.on(ctx).getRDFService(CONFIGURATION);
```

Moreover it enables getting a short-term data structure by accessing from the request.

```
ModelAccess.on(req).getOntModel(ModelNames.DISPLAY);
```

The elaborate structure of options enums allows us to specify method signatures like this on RequestModelAccess:

```
getOntModelSelector(OntModelSelectorOption... options);
```

Which can be invoked in any of these ways:

```
ModelAccess.on(req).getOntModelSelector();
ModelAccess.on(req).getOntModelSelector(LANGUAGE_NEUTRAL);
ModelAccess.on(req).getOntModelSelector(INFERENCES_ONLY);
ModelAccess.on(req).getOntModelSelector(ASSERTIONS_ONLY, LANGUAGE_NEUTRAL);
```

The compiler insures that only appropriate options are specified. However, if conflicting options are supplied, it will only be caught at runtime.

Initializing the Models

When VIVO starts up, `OntModel` objects are created to represent the various data models. The configuration models are created from the datasource connection, usually to a MySQL database. The content models are created using the new RDFService layer. By default this also uses the datasource connection, but it can be configured to use any SPARQL endpoint for its data.

Some of the smaller models are "memory-mapped" for faster access. This means that they are loaded entirely into memory at startup. Any changes made to the memory image will be replicated in the original model.

The data in each model persists in the application datasource (usually a MySQL database), or in the RDFService. Also, data from disk files may be loaded into the models. This may occur:

- the first time that VIVO starts up,
- if a model is found to be empty,
- every time that VIVO starts up.

depending on the particular model.

Where are the RDF files?

In the distribution, the RDF files appear in `[vivo]/rdf` and in `[vitro]/webapp/rdf`. These directories are merged during the build process in the usual way, with files in VIVO preferred over files in Vitro.

During the VIVO build process, the RDF files are copied to the VIVO home directory, and at runtime VIVO will read them from there.

The "first time"

For purposes of initialization, "first time" RDF files are loaded if the relevant data model contains no statements. Content models may also load "first time" files if the RDFService detects that its SDB-based datastore has not been initialized.

Initializing Configuration models

Application metadata

Function: Describes the configuration of VIVO at this site. Many of the configuration options are obsolete.

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-applicationMetadata

Source: the application Datasource (MySQL database) (memory-mapped)

If this is the first startup, read the files in rdf/applicationMetadata/firsttime.

- In Vitro, there are none
- In VIVO, `initialSiteConfig.rdf, classgroups.rdf and propertygroups.rdf`

User Accounts

Contains login credentials and assigned roles for VIVO users.

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-userAccounts

Source: the application Datasource (MySQL database) (memory-mapped)

If this model is empty, read the files in rdf/auth/firsttime.

- In Vitro, there are none (except during Selenium testing)
- In VIVO, there are none

Every time, read the files in rdf/auth/everytime

- In Vitro, `permission_config.n3`
- In VIVO, there are none.

The Display model

This is the ABox for the display model, and contains the RDF statements that define managed pages, custom short views, and other items.

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-displayMetadata

Source: the application Datasource (MySQL database) (memory-mapped)

If this model is empty, read the files in `rdf/display/firsttime`

- In Vitro, `application.owl, menu.n3, profilePageType.n3, pageList_editableStatements.n3`
- VIVO contains its own copy of menu.n3, which overrides the one in Vitro `aboutPage.n3 menu.n3 PropertyConfig.n3 PropertyConfigSupp.n3`

Every time, read the files in rdf/display/everytime

- in Vitro, `dataGetterLabels.n3  permissions.n3 displayModelListViews.rdf  searchIndexerConfigurationVitro.n3 pageList.n3  vitroSearchProhibited.n3`
- In VIVO `homePageDataGetters.n3  vivoConceptDataGetters.n3 localeSelectionGUI.n3  vivoListViewConfig.rdf n3ModelChangePreprocessors.n3  vivoOrganizationDataGetters.n3 orcidInterfaceDataGetters.n3  vivoQrCodeDataGetter.n3 searchIndexerConfigurationVivo.n3 vivoSearchProhibited.n3`

Display TBox

The TBox for the display model.

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-displayMetadataTBOX

Source: the application Datasource (MySQL database) (memory-mapped)

Every time, read the files in rdf/displayTbox/everytime.

- In Vitro, `displayTBOX.n3`
- In VIVO, there are none

DisplayDisplay

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-displayMetadata-displayModel

Source: the application Datasource (MySQL database) (memory-mapped)

Every time, read the files in rdf/displayDisplay/everytime

- In Vitro, `displayDisplay.n3`
- In VIVO, there are none.

Initializing Content models

base ABox

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-2

Source: named graph from the RDFService

If first setup, read the files in `rdf/abox/firsttime`

- In Vitro, there are none
- In VIVO, `geopolitical.ver1.1-11-18-11.individual-labels.rdf`

Every time, read the files in `rdf/abox/filegraph`, and create named models in the RDFService. Add them as sub-models to the base ABox. If these files are changed or deleted, update the RDFService accordingly.

- In Vitro, there are none
- In VIVO `documentStatus.owl academicDegree.rdf  geopolitical.abox.ver1.1-11-18-11.owl  us-states.rdf continents.n3  validation.n3 dateTimeValuePrecision.owl  vocabularySource.n3`
- Plus whatever data packages you may have added. See Managing Data Packages

base TBox

Name: http://vitro.mannlib.cornell.edu/default/asserted-tbox

Source: named graph from the RDFService (memory-mapped)

If first setup, read the files in rdf/tbox/firsttime (without subdirectories)

- In Vitro, there are none
- In VIVO, additionalHiding.n3 initialTBoxAnnotations.n3

Every time, read the files in `rdf/tbox/filegraph`, and create named models in the RDFService. Add them as sub-models to the base TBox. If these files are changed or deleted, update the RDFService accordingly.

- In Vitro, `vitro-0.7.owl, vitroPublic.owl`
- In VIVO `education.owl   personTypes.n3 agent.owl   event.owl   process.owl appControls-temp.n3 geo-political.owl publication.owl bfo-bridge.owl   grant.owl   relationship.owl bfo.owl   linkSuppression.n3 relationshipAxioms.n3 classes-additional.owl location.owl   research-resource-iao.owl clinical.owl   object-properties.owl research-resource.owl contact-vcard.owl object-properties2.owl research.owl contact.owl object-properties3.owl role.owl data-properties.owl objectDomains.rdf sameAs.n3 dataDomains.rdf   objectRanges.rdf service.owl dataset.owl   ontologies.owl   skos-vivo.owl date-time.owl   orcid-interface.n3 teaching.owl dateTimeValuePrecision.owl other.owl   vitro-0.7.owl documentStatus.owl outreach.owl   vitroPublic.owl`
- Plus whatever ontology extensions you may have added

base Full

Source: a combination of base ABox and base TBox

inference ABox

Name: http://vitro.mannlib.cornell.edu/default/vitro-kb-inf

Source: named graph from the RDFService

inference TBox

Name: http://vitro.mannlib.cornell.edu/default/inferred-tbox

Source: named graph from the RDFService (memory-mapped)

inference Full

Source: a combination of inference ABox and inference TBox

union ABox

Source: a combination of base ABox and inference ABox

union TBox

Source: a combination of base TBox and inference TBox

union Full

Source: a combination of union ABox and union TBox