Accessing VIVO Data Models

Accessing 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");

ctx.setAttribute("jenaOntModel", masterUnion);
```

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

Model makers and Model sources

The ModelAccess class

TBD - Show how it represents all of these distinctions. Describe the scope searching and masking, wrt set and get. Include the OntModelSelectors and WADFs.

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 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, permissions_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
- VIVO contains its own copy of menu.n3, which overrides the one in Vitro

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

- in Vitro, displayModelListViews.rdf
- In VIVO, homePageDataGetters.n3, localeSelectionGUI.n3, vivoDepartmentQueries.n3, vivoListViewConfig.rdf, 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 restart, 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, geopolitical.abox.ver1.1-11-18-11.owl, academicDegree.rdf, continents.n3
  us-states.rdf, dateTimeValuePrecision.owl, validation.n3, documentStatus.owl, vocabularySource.n3

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 restart, 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, 44 files:

/usr/local/vivo/home/rdf/tbox/filegraph

```
README.md                                    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
```

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

Transition from previous methods

TBD - What are we transitioning from? Check out VIVO-82.

- Semantics have changed: saves code, but may alter some uses.
    - Always searches the stack
    - OMS are facades with no internal state
        - There is no way to set an OMS - set the models instead
        - Keeps consistent

	prior to ModelAccess	using ModelAccess
User Accounts Model	ctx.getAttribute("userAccountsOntModel")	ModelAccess.on(ctx).getUserAccountsModel()
	ctx.setAttribute("userAccountsOntModel", model)	ModelAccess.on(ctx).setUserAccountsModel(model)
DisplayModel	req.getAttribute("displayOntModel")	ModelAccess.on(req).getDisplayModel()
	session.getAttribute("displayOntModel")	ModelAccess.on(session).getDisplayModel()

	ctx.getAttribute("displayOntModel")
	ModelContext.getDisplayModel(ctx)
		ModelAccess.on(ctx).getDisplayModel()

	ctx.setAttribute("displayOntModel", model)
	ModelContext.setDisplayModel(model, ctx)
		ModelAccess.on(ctx).getDisplayModel()

	req.setAttribute("displayOntModel", model)	ModelAccess.on(req).setDisplayModel(model)
"jenaOntModel"	ctx.getAttribute("jenaOntModel")	ModelAccess.on(ctx).getJenaOntModel()
	session.getAttribute("jenaOntModel")	ModelAccess.on(session).getJenaOntModel()
	req.getAttribute("jenaOntModel")	ModelAccess.on(req).getJenaOntModel()
	ctx.setAttribute("jenaOntModel", model)	ModelAccess.on(ctx).setOntModel(ModelID.UNION_FULL, model)

	req.setAttribute("jenaOntModel", model)	ModelAccess.on(req).setOntModel(ModelID.UNION_FULL, model)
		ModelAccess.on(req).setJenaOntModel(model)

"baseOntModel"
"assertionsModel"
Base Full Model
	ModelContext.getBaseOntModel(ctx)
	ctx.getAttribute("baseOntModel")
	session.getAttribute("baseOntModel")
		ModelAccess.on(ctx).getOntModel(ModelID.BASE_FULL)
		ModelAccess.on(ctx).getBaseOntModel()

	ModelContext.setBaseOntModel(model, ctx)	
"inferenceModel" Inference Full Model	ctx.getAttribute("inferenceOntModel")	ModelAccess.on(ctx).getInferenceOntModel()

Notes:

- "jenaOntModel" is a previous term for the Union Full model. The convenience methods `getJenaOntModel()` and `setJenaOntModel(m)`support this use.
- "baseOntModel" and "assertionsModel" are both previous terms for the Base Full model. The convenience methods `getBaseOntModel()` and `setBaseOntModel(m)`support this use.

	prior to ModelAccess	using ModelAccess

ontModelSelector
unionOntModelSelector
	ModelContext.setOntModelSelector(model, ctx)
	ModelContext.getUnionOntModelSelector(ctx)
	ctx.getAttribute("ontModelSelector")
	ctx.getAttribute("unionOntModelSelector")
		no mutator methods
		ModelAccess.on(ctx).getOntModelSelector()
		ModelAccess.on(ctx).getUnionOntModelSelector()

baseOntModelSelector	ctx.getAttribute("baseOntModelSelector")	ModelAccess.on(ctx).getBaseOntModelSelector()
inferenceOntModelSelector	ctx.getAttribute("inferenceOntModelSelector")	ModelAccess.on(ctx).getInferenceOntModelSelector()

- The default WebappDaoFactory is the one backed by the unionOntModelSelector. On the request level, this is also known as the "fullWebappDaoFactory". The convenience methods`getWebappDaoFactory()` and `setWebappDaoFactory(wdf)` support this use.
- "baseWebappDaoFactory" and "assertionsWebappDaoFactory" are both previous terms for the WebappDaoFactory backed by the baseOntModelSelector. The convenience methods `getBaseWebappDaoFactory()` and `setBaseWebappDaoFactory(wdf)` support this use.
- Nobody was using the "deductionsWebappDaoFactory", so we got rid of it.