Repository Design Specification and API Manual

Overview

This document is intended to be a thorough description of the data repository design; concentrating on its external interfaces. The intended audience is repository coders as well as implementers of other components that depend on the repository. It should be kept up to date with any API changes so it can serve as a reference manual.

The repository is essentially an RDF triple-store with some extra features dictated by the needs of data entry tools, search, dissemination, and the data collection and curation process. These features include the ability to:

- Bind an RDF graph to the contents of a serialized file allowing clean updates.

- Resolve URIs as linked data.
- Maintain provenance metadata.

Other features include:

- Workflow and lifecycle management with automated enforcement.
- Isolated administrative domains so separate groups can share a repository.
- Fine-grained access control for reading as well as modification operations.

Concepts and Internal Structure

This section describes the internal conceptual data model.

Resource Instance

The primary purpose of the repository is to store, edit, and retrieve resource instances for its clients. The term *resource instance* comes from the eagle-i data model: a *resource* is an indivisible unit of something described in the eagle-i database. A resource instance in the repository is the corresponding graph of RDF statements that represents this resource as abstract data. It is rooted at one *subject URI*.

A resource instance is defined as the *subject* URI, and the collection of statements in which it is the subject. Furthermore, there *must* be one statement whose predicate is `rdf:type` and whose object is a URI, preferably of type `owl:Class`, but this is not checked.

The significance of resource instances in the repository architecture is:

- The `/update` service creates, modifies, or deletes *exactly one single instance* transactionally.
- The repository maintains *metadata* about each instance, described in more detail below.
- Resource instances typically live in *workspaces*, described in more detail below.
- The `dissemination` service retrieves the contents of a single instance in various formats.
- The `/harvest` service reports on changes to instances.

Embedded Instances (within Resource Instance)

Early in the development process, the definition of a *resource instance* was extended to encompass ***embedded instances*** (EI). This change only affects the rules governing what statements belong in the instance's graph, although it has a profound effect on the behavior of the repository. *Embedded instances* are essentially regular resource instances which *are considered part of a "parent" resource instance*. The precise definition of an embedded instance is as follows:

1. It has a unique subject URI, and exactly one asserted `rdf:type` statement.
2. Its `rdf:type` (possibly an inferred type) is a member of the designated class group indicating embedded instances. In the eagle-i data model the URI of this class group is:

```
http://eagle-i.org/ont/app/1.0/ClassGroup_embedded_class
```

3. It has exactly one parent. There is exactly one instance, which is the subject of statements for which the EI is the object. This is an informal restriction (really an *assumption*) imposed on all instances of embedded types, although it is enforced by logic in the repository.
4. EIs may not be Orphaned or Shared. Any transaction that would result in an EI left without a parent, in other words it is *not* the object of any conforming statements, *or* it has multiple parents, is forbidden by logic in the repository. The only way to remove an EI from its parent is to delete all of its statements. You can *copy* an EI to another parent by creating a new EI under that parent, with a new unique URI for its subject.
5. No Broken Links to EIs. If an EI is removed, an instance cannot retain any statements of which it is the object. These must be removed.
6. EIs do not Have Metadata*.* The repository does not create metadata, Dublin Core, for example, about EIs. Any transactions on the EI are considered transactions on its parent and recorded as such, in the last-modified date in the metadata.
7. Transactional Integrity All *transactional* operations, such as `/update` and `workflow`, operate on the EI statements together with the parent instance's statements. For example, a workflow transition to published moves all the EIs to the published graph along with their parent's statements.
8. EIs reside in the same graph as the parent. Though it seems obvious, it's worth stating formally that the statements describing an EI must reside in the same named graph (workspace) as the statements of its parent.

Here is how EIs behave in repository operations:

Creation: An EI is created by adding a new URI with an appropriate `rdf:type` statement to a modification (including creation) of its parent. The type must belong to the embedded class group.

Modification/Deletion: Any modification of an EI *must* be done as a modification of its parent. The EI's properties, including type, may be changed; it may be deleted. These changes are recorded as a modification of the parent. The changes to the parent and its EIs can be driven by one HTTP request to the `update` service, and will be performed in a single transaction.

Dissemination: A dissemination request on the parent instance will include all of the statements about its EIs. The EIs will be filtered of hidden properties such as admin data and contact hiding, by the same rules as the parent, nd returned in the same serialized RDF graph. Dissemination requests on EIs are not supported. It is not recommended, the results are undefined.

Metadata Harvest:

See the description of the `/harvest` service for full details. Essentially, since EIs do not have an independent presence in the "instance" model of the repository, they are not reported on individually when the harvest service reports changes. A change to an EI, *even deletion of the EI*, is reported as a change to its parent. Likewise, creation of an EI is also reported as a change to its parent.

Resource Property Hiding

The repository is required to *hide* statements with certain predicates when exporting resource instances in these contexts:

1. The dissemination service (for example, the `/i` service and `/repository/resource`).
2. The `/harvest` service for populating search indexes

The set of predicates to be hidden is defined by the data model ontology, and identified by the data model configuration properties:

- `datamodel.hideProperty.predicate`
- `datamodel.hideProperty.object`

The hidden predicates are themselves subjects of statements whose predicate is the hiding predicate, and whose object is the hiding object, for example:

```
...for example this configuration would hide dm:someStupidProperty:
datamodel.hideProperty.predicate = dm:hasSpecialAttribute
datamodel.hideProperty.object = dm:cantSeeMe
....and then later on, in the ontology (shown in N3):
dm:someStupidProperty dm:hasSpecialAttribute dm:cantSeeMe.
```

The *mechanism* of hidden-property hiding is implemented through *access controls*. See that section for more details.

This has to be implemented in the repository in order to enforce a consistent security model, which would not be possible if content hiding were left up to each client application.

Properties are "hidden" for various reasons, such as:

- Properties that are effectively provenance metadata added by users in the resource acquisition and curation process, but are not suitable for public viewing. They may contain confidential information or comments that the curators do not want publicized.
- Properties whose values inherently contain confidential information, or information such as e-mail addresses, physical locations, and phone numbers that administrators have been directed to hide.

Contacts and E-mail

The "contact" issue is closely related to property hiding. The essential problem is that for every resource instance, it is desired to have a means for the agent viewing that resource. For example, a user in a Web browser viewing a semantic Web-style dissemination page of the resource instance to *contact* the agent who "owns" (or is otherwise responsible for) the resource. E-mail is one means of implementing this contact, but certainly not the only one. The contact could be in the form of a telephone number, street address, or even a redirect to another Web site which might include indirect contact info of its own. The purpose is to put a potential consumer of the resource in touch with its owner.

The repository only gets involved to mediate this contact process because it is also responsible for hiding all contact information from the agents who would use it. It must therefore implement some means of accepting a contact request or message from the outside agent, and forward it to the determined owner of the resource.

Contact properties are identified in the same way as hidden properties, only the relevant data model configuration keys are:

- `datamodel.contactProperty.predicate`
- `datamodel.contactProperty.object`

The *mechanism* of contact-property hiding is implemented through *access controls*, explained in more detail later in this document.

Internal Ontology and Metadata

There is a separate ontology document describing the repository's internal data model and administrative metadata, refer to repository-internal.n3 for more information. Note that some statements described by that ontology appear as publicly-readable metadata statements, while others are private and never exposed outside of the repository codebase.

The "ontology" graph is considered read-only in normal operation. All internal metadata, (administrative metadata) is stored in a separate, distinct, named graph which should only be available to the repository's internal operations.

Named Graphs

The repository design takes full advantage of the *named graph* abstraction provided by most modern RDF database engines (Sesame, Virtuoso, Anzo, AllegroGraph). *Every* statement in the RDF database belongs to exactly one named graph. Since this is typically implemented by adding a fourth column to each triple for the named graph's URI, these databases are often called *quad-stores* instead of triple-stores. The data repository design takes advantage of named graphs to:

- Restrict query results (for logic and/or access control) efficiently by defining a *dataset* made up of named graphs.
- Impose different access controls on subsets of the database defined by named graph.
- Group the statements loaded from a file or network ingest and keep administrative metadata about their source, so they can be checked and updated as new versions of that file come out.
- Apply different inference policies to graphs, according to their purpose (e.g. OWL-driven inference rules on ontologies).
- Implement the "workspace" abstraction efficiently.

Internally, we collect some metadata about each named graph including access control rules, and a *type* that documents the purpose of the named graph.

Named Graph Types:

1. **Ontology**---contains ontology descriptions, so reasoning may be required. The name of the graph is the namespace URI. Also implies that the graph is part of the "TBox" for inferencing.
2. **Metadata**---contains the repository's own metadata; public and private metadata are in separate graphs for easier containment of query results.
3. **Workspace**---holds resource instances still in the process of editing and curation. Not visible to the public, and access to some workspaces is restricted to a subset of users.
4. **Published**--Published resource description data.
5. **Internal**---Contains data that is only to be used internally by repository code and never exposed.

The repository is created with a few *fixed* named graphics for specific purposes such as internal metadata statements. Other named graphs are created as needed. Even the repository's own ontology is not a fixed graph since it can be managed like any other ontology loaded from a serialization.

Relationships---it would be helpful to record metadata about related named graphs, although the most compelling case for this is the ontologies that use **owl:includes** to embed other ontology graphs. Since Sesame does not interpret OWL by itself, and we have no plans to add this sort of functionality for the initial repository implementation, this will be considered later.

Views

The repository provides *views* to give clients an easier way to query over a useful set of named graphs. A view is just a way of describing a *dataset* (a collection of named graphs). The repository server has a built-in set of views, each named by a simple keyword. You can use a view with the SPARQL Protocol and with a resource dissemination request. It is a equivalent to building up a dataset out of named graphs but it is a lot less trouble, and guaranteed to be stable whereas graph names could change. The views are:

- **published**---all resource instances and user records visible to the public, *and* all relevant ontologies and metadata.
- **published-resources** ---resource instances visible to the public, *and* all relevant ontologies and metadata.
- **metadata**---all named graphs of type Metadata visible to the authenticated user.
- **ontology**---all named graphs of type Ontology visible to the authenticated user
- **metadata+ontology**---all named graphs of types Metadata *and* Ontology visible to the authenticated user
- **null**---all statements in the internal RDF database *regardless* of named graph. Administrators only.**NOTE**: This is the ONLY way to see any statements that do not belong to any named graph, i.e. the "null context" in Sesame. If we are lucky this will be a small or empty set.
- **user**--- graphs that the current user has permission to read.
- **user-resources**---graphs containing or related to eagle-i resources that the current user has permission to read. Note that the graph containing instances of repository user,s of type `foaf:Person` is ironically, NOT part of this data set, since they are not *eagle-i* resources.
- **public**---all graphs that are visible to the public, to the anonymous user, plus inferred statements. Note that this is *not* the same as

'published'.

- **all**---all named graphs including the ones internal to the repository; administrators only.

Important Note: You may have noticed that according to the definition, the user view is the same as the all view for an administrator user, so why bother creating an **all** view? It is intended to be specified when you have a query that really must cover *all* of the named graphs to work properly; if a non-administrator attempts it, it will fail with a permission error, instead of misleadingly returning a subset of the graphs.

Workspaces

A *workspace* is just another way to describe a data set, by starting with a named graph. It is effectively a special kind of *view*. The name of a workspace is the URI of its *base* named graph, which must be of type workspace or published. When you specify that as the workspace, the repository server automatically adds these other graphs to the data set:

1. All graphs of type ***Ontology*** that are readable by the current user.
2. All graphs of type ***Metadata*** that are readable by the current user.
3. The graph of *inferred* statements.
4. The graph of *repository user instances* (referenced by metadata statements)

You can specify a workspace *instead of* a view in SPARQL Protocol requests, and in resource dissemination requests.

Inferencing

The repository supports inferencing in some very specific cases. Since the repository's RDF data is very frequently modified, it does only the *mini mal* inferencing needed by its users in order to keep the performance bearable.

Many inferencing schemes require inferencing to be re-done over the entire RDF database after every change because tracing the effects of a change through various rules would be at least as much computational effort as simply running the inferencing over. We have chosen a select subset of RDFS and OWL inference rules that makes incremental changes easy and efficient to re-compute.

See the RDF Semantics page for an overview of the greater body of inference rules (of which we implement only a small subset). The repository implements two different kinds of inferencing:

1. **TBox** inferencing:
   - *TBox* is the *terminology* source, i.e. the ontology graphs.
   - It consists of named graphs whose type is **ontology**.
   - Upon any change to a TBox graph, inferencing is redone on the entire graph, and on all the ABox graphs.
   - All inferred TBox statements are added as inferred statements directly to their TBox graph.
   - Inference is done independently on every TBox graph they are expected to be completely disjointed. Thus, there should only be one or two TBox graphs.
   - Following entailment rule **rdfs11**, all inferred `rdfs:subClassOf` relationships are created as direct statements.
   - Following entailment rule **rdfs5**, all inferred `subPropertyOf` relationships are created as direct statements.
   - Following entailment rule **rdfs9**, all inferred `rdf:type` properties are added.
2. **ABox** inferencing:
   - *ABox* is the body of *assertions*, i.e. the statements about resource instances.
   - All non-TBox named graphs are part of the ABox.
   - All statements created from inference on ABox statements are added to a special named graph, http://eagle-i.org/ont/repo/1.0/NG_Inferred.;
     - This makes it easy to include or exclude the inferred statements in the dataset of SPARQL query
     - You can detect whether or not a statement is inferred by adding a GRAPH keyword to the query and testing its home graph.
   - Any change to a TBox graph only causes re-inferencing of the *subject* of each statement where a named instance was changed. This is possible because the inferred statements only depend on the asserted `rdf:type` properties of a a subject and the TBox graphs.
   - Following entailment rule **rdfs9**, all inferred `rdf:type` properties are added to the graph for inferred statements.

The TBox graphs are configurable. You can set the configuration property `eaglei.repository.tbox.graphs` to a comma-separated list of graph URIs. By default, the TBox consists of:

1. The repository's internal ontology, http://eagle-i.org/ont/repo/1.0/
2. The eagle-i data model ontology, http://purl.obolibrary.org/obo/ero.owl

This inferencing scheme ensures very fast performance by assuming the TBox graphs never change under normal operations, which ought to be true. The data model ontology graph is only modified when a new version of the ontology is released. Likewise, the repository's internal ontology graph remains unchanged once the repository is installed. When the TBox graphs *are* changed, be aware that you will probably see a delay of many seconds or perhaps minutes, as all the TBox and ABox inferencing is re-done.

Inferred statements are not normally written when an entire graph is dumped. See the `/graph` service for details.

Users and Authentication

Authentication is managed entirely by the Java Servlet container. We rely on the container to supply an authenticated *user name* (a short text string) and whether that user has the "superuser" *role*. The container's role is only used for bootstrapping; normally roles are recorded in the RDF database and they take precedence over the container's role map.

Users Are Described in Two Databases

Each login user is (ideally) recorded in both the RDBMS used by the servlet container (or possibly some other external DB) and the RDF database. This is necessary because the servlet container, which is doing the authentication, only has access to the RDBMS through a plugin, but the repository authorization mechanism expects an RDF expression of the user as well. All of the services that modify users keep the RDBMS and RDF databases synchronized, and can cope with users found in one and not the other.

The RDBMS description of a user contains:

1. **Username---*present in RDF as well, this is the common key. Since it is the permanent description of a user this is *immutable**.
2. **Password**---only in RDBMS
3. Membership in **Superuser** role. No other roles.

The RDF description of a user contains:

1. **Username** of the corresponding RDBMS user.
2. **Roles** including **Superuser** if present in RDBMS
3. Various descriptive and provenance metadata.

When a user is present in RDF but not in the RDBMS, they are considered disabled and cannot log in. They can be reinstated through the Admin UI.

When a user is present in the RDBMS but not in the RDF, they are considered *undocumented*. Upon logging in, an undocumented user is given the URI corresponding to his/her highest known role: **:Role_Superuser** if the RDBMS indicates that role, or **:Role_Anonymous** otherwise. (Arguably, the default role could also be **:Role_Authenticated**, but without RDF data for the user they are not fully authenticated, and this is incentive to fix the discrepancy.)

To fix an undocumented user:

An Administrator (superuser); can become documented by logging in, and either running the `\whoami` service with `create=true` or using the Admin UI to edit and save their own user information. An Administrator can fix an ordinary undocumented user by using the Admin UI to save their descriptive metadata; even if it is all blank, a user record will be created. Importing users also straightens out the mapping automatically.

About Roles

Roles are a way to characterize a group of users, for example, to grant them some access rights in the access-control system. Functionally, the role is part of a user's authentication, i.e., "who" they are.
A *role* is defined by a URI, the subject of a **:Role** instance. It should also have a locally unique, short text-string name (the **:label** of its **:Role** insta nce).

Each Role is independent of other Roles. Roles cannot be "nested". This is a necessary limitation that simplified the implementation considerably.

The **Superuser** role is built into the system because its privileges are hard coded.

There are a couple of special ***Roles*** whose membership is implicit, that is, it never needs to be granted explicitly:

1. **Anonymous**---unauthenticated users are implicitly assigned this role so they can be given explicit access. This role is *never visibly asserted by a user*, it is *only* for describing access controls: For example, "The Published graph is readable by the Anonymous role".
2. **Authenticated** ---any user who logs in is *authenticated* and implicitly belongs to this role; the opposite of *anonymous*. This role is *never explicitly asserted by a user, it is only* for describing access controls.

Username and Password

A repository user is identified uniquely (within the scope of **ONE** repository instance) by a short symbolic *username*. This is a character string composed of characters from a certain restricted subset of the ASCII character set, in order to avoid problems of character translation and metacharacter interpretation in both the protocol layer (HTTP) and OS tools such as command shells. The *password*, which is paired with a username to serve as *login credentials*, is likewise restricted to the same range of characters as the username.

Character restrictions: The username and password * MUST NOT include* the character **':'** (colon). They can only include:

- ***Letter*** or ***digit*** characters in Unicode C0 and C1 (basic latin, latin-1)

- The punctuation characters: **~, @, #, $, %, _, -, .** *(dot)*

Note that although the HTTP *protocol* allows any graphic characters in the ISO-8859-1 codeset (modulo ':'), and linear whitespace, and even chars WITH special MIME RFC-2047 encoding rules, these are often implemented wrongly by HTTP clients and also invite encoding and metacharacter problems with OS and scripting tools.To avoid these troubles we simply restrict the available characters.

Authentication policy

All of the servlet URLs in this interface ***except*** the public dissemination request `/i` require authentication. The dissemination request makes use of an authenticated user and roles when they are available, to access data that would be invisible to an anonymous request, but it is never *required*.

Access Control (Authorization)

This is just an outline of the access control system. It is implemented as statements stored in the internal metadata graph. The access controls applying to an instance or other object are not to be directly visible in the repository API, except through administrative UI.

These types of access can be granted:

- **Read**---in context of named graph, allows the graph to be included in dataset of a query; for workflow transitions and hidden/contact property groups, this is the *only* relevant access grant.
- **Add**---add triples to a graph or instance.
- **Remove**---remove triples from a graph or instance.
- **Admin**---change the access controls on a resource. (Not really used in practice; only the Superuser role can change access.)

On the following types of resources:

- **Named Graphs**, including workspaces.
    - Read lets people download and run queries against the graph.
    - Add, Remove lets you modify it with `/graph` Usually reserved for admins.
- **Resource Instances** (ignores explicit Read access; that comes from it's home named graph.)
    - Add, Remove let you modify it with `/update`, this is usually granted *temporarily* to a specific user as part of the workflow process.
    - *ignores Read grants*, read access reverts to its *home graph*.
- **Property Groups** - sets of properties on a resource instance identified by statements in the data model ontology, see the data model configuration manual, particularly `datamodel.hideProperty.predicate`, `datamodel.hideProperty.object`.
    - Read lets the Dissemination and Harvest service report on these properties.
    - *All other access types are ignored*.
- **Workflow Transitions**
    - Read gives access to take (push) the transition.
    - *All other access types are ignored*.

Access control *statements* grant access to either a specific ***user***, or to a ***Role***, which applies to all users holding that role.

Any user asserting the **Superuser** role is always granted access, bypassing all controls. This lets us bootstrap the system when there is no RDF database yet to describe grants. Repository administrators should always have the **Administrator** role, since most of the Admin UI and API requires it.

Access control is implemented by statements of the form:

```
Subject: resource, Predicate: access-type, Object: accessor
```

The *resource* is the URI of the instance, named graph, or workflow transition of interest. The *access-type* names one of the four types of access described above: read, add, remove, admin. Finally, the *accessor* is the URI of the *Principal* to be granted the access, either a **Role** or an **Agent** (user).

We anticipate having a relatively small number of these access grants. Although named graphs and workflow transitions need elaborate access descriptions, there are only a few of those--on the order of dozens. Resource instances are of course more numerous but most of them have no access grants, deriving their read/query access from the named graph they reside in. The workflow **claim** service adds temporary grants to give the claim owner read/write access to be able to edit the instance while it is claimed.

Provenance Metadata

The repository automatically records *provenance metadata* about objects when they are created and modified by users' actions. *Provenance* mea ns information about the history and origin of the data, in this case the authenticated identity responsible and time of the latest change. The following properties are recorded for these types of objects, and can be obtained by querying with a view or dataset that includes the named graph containing public administrative metadata.

Note that the there is at most one value of any of these properties for each subject. That means the "modified" properties are *updated* whenever a subject is modified and the record of the previous modification is lost. This is a simplification that may be remedied at some point in the future if we add versioning of data to the repository.

Named Graphs:

- `dcterms:modified`---literal date of last modification, encoded as `xsd:dateTime`
- `dcterms:contributor`---the URI of the Agent (authenticated user) who last modified it
- `dcterms:source`---description of the file or URI last loaded into this NG, if that is how it was created. This record is used to compare it against the source later to decide whether an update is necessary. It is a node (possibly blank node) with the following properties:
    - `dcterms:identifier`---the resolvable URI of the resource loaded, most likely a URL in either the file or http scheme.
    - `dcterms:modified`---last-modification date of the resource, for later comparison when deciding whether to decache the repository copy of an external file, a literal `xsd:dateTime`.
      Resource Instance:
    - `dcterms:created`---literal date when resource was created, encoded as `xsd:dateTime`
        - Set automatically by the Repository.
    - `dcterms:creator`---the URI of the Agent (*should* be an authorized user) who created the instance.
        - The *true meaning* is the user who *authored the data* in the inital version of this instance.
        - *Usually*, this is the same as the user directly responsible for *creating* the instance.
        - However, when a different user uploads, a spreadsheet created by other RNAVs, for example, the Repository user is a *mediator*. The actual value of `dcterms:creator` comes from the uploaded data.
    - *(optional)* `dcterms:mediator`---ONLY when `dcterms:creator` does *not* refer to the authenticated user who created the data, this is the URI of the Agent (authenticated user) who created this instance in the Repository.
        - Set automatically by the Repository.
    - `dcterms:modified`---literal date when resource was last modified, encoded as xsd:dateTime
        - Set automatically by the Repository.
    - `dcterms:contributor`---the URI of the Agent (authenticated user) who last modified this instance.
        - Set automatically by the Repository.

Sesame Triplestore Extensions

Some repository features are implemented as *extensions* to the Sesame RDF database (also known as *triplestore*). This means they are available both internally to the repository implementation and externally whenever an API to Sesame, its SPARQL query engine, is exposed.

1. Output formats

Additional output formats for both RDF serialization and SPARQL tuple query results allow output in:

- **HTML**, media type `text/html`
- **plain text**, media type `text/plain` (for SPARQL)
- **N-Quads,** RDF only, for comparing serialized RDF including context (graph).

2. SPARQL Query Function

The repository adds a custom function to Sesame's query engine: `repo:upperCaseStr`. It returns the `toUpperCase()` version of of the string value of an RDF value. Use it to sort values *ignoring* whether (a) the case of characters differs, (b) they are datatyped-literals or untyped literals (or other terms).

To invoke it you ***must*** have the repository's URI namespace defined as a prefix. For example,

```
PREFIX repo:<[http://eagle-i.org/ont/repo/1.0/]>
..query text...
 ORDER BYrepo:upperCaseStr(?label)
```

Workflow

The repository includes a "workflow" control system that directs the complete life cycle of each resource instance, and mediates access by users at each life cycle stage. The word "workflow" is often used to describe process-management and administration systems, but in this case it is really just a minimal implementation of states and extended access control.

It was implemented in the repository because it depends on persistent data and access control which are already available in the repository. It is also closely integrated with the access control system, which is easier to accomplish securely from within the repository codebase.

Workflow is manifested in RDF statements (of course) in the internal metadata graph. Although the Web API exposes some URIs and names of workflow objects, the ontology and access control details are intentionally hidden. There is no need for applications using workflow to see the model, all their access is through the API.

The model is a state map, with nodes and transitions between them. Elements of workflow are:

1. **Workflow State**---A node on the state map. Every resource instance has exactly one current state. A newly created resource is initialized to a fixed "New" state.
2. **Transition**---Description of an arc on the map, or a *transition* from an *initial* state to a *final* state.
3. **Claim**---Assertion on a resource instance that a specific user (the *claimant*) has taken possession of it, in order to prepare it for the next workflow transition.
    - Claiming typically causes some *side-effects* such as changing access controls.
    - A claim may be resolved by taking a transition to another state, or by *releasing* it.
4. **Pool**---The set of resource instances available for claiming *to a specific user*. A pool is always computed by a query, it is not materialized.

Bootstrapping and Configuration

Bootstrapping refers to how a new repository node first starts up. The process is not trivial, since so much of its operation depends on the RDF database (triple-store, actually a quad-store) which is completely empty when a new repository is launched.

The repository must be simple (ideally "foolproof", although that only breeds more destructive fools) to install and manage, since it is intended to be deployed at dozens or hundreds of sites, managed by administrators with varying experience levels. All the while, it must still maintain adequate security and data integrity.

See Cycle 1 Repository Documentation for more information on installing the repository and setting up its configuration.

The bootstrap process:

1. Install the webapp by dropping the WAR file into place, and adding configuration:
    a. Add one system property to the Web server's JVM to indicate the configuration properties.
    b. Create configuration properties file with at least the minimum required properties.
2. Create authentication DB and add the initial Administrator user.
3. Start the servlet container---the repository webapp automatically initializes the following named graphs if they are found to be empty:
    a. The internal Repository ontology (read-only)
    b. Internal Repository administrative metadata (e.g. Roles, Named Graphs, WF Transitions, etc.)
    c. SPARQL Query macros.
4. Create an RDF metadata wrapper for the initial administrator user - this is done automatically by the post-install procedure, `finish-install.sh`. See the Administrator Guide.
5. Load the eagle-i data model ontology. Although this is not necessary for any of the Repository's internal functions, the eagle-i apps will expect it to be there.
6. Browse to the Admin UI and log in as the Administrator user. You can create user logins and assign roles; modulo any customizations of workflow, workspaces and such. With that, the repository is open for business.

Configuration properties

The configuration properties are loaded from the file `configuration.properties` in the repository home directory. It is read by Apache Commons Configuration, which allows system properties and other variables to be interpolated into the values. See the Apache Web site for complete documentation.

See the Configuration section in the Administrator Guide for a complete list of configuration properties.

Basic Repository REST API

These are the possible HTTP requests in the repository API.

Webapp Mount Point

The repository's webapp must be mounted at the web server's root, so that it can resolve the canonical URI of resource data instances.

Format Identifiers and MIME Types

This section lists the formats that the repository can use for output and, in some cases, input, of data. The MIME type is how you describe it in the API, whether through explicit args or HTTP headers.

In a request, you can usually specify ***input format*** two ways:

1. Explicitly through the format arg, which takes precedence.
2. As the HTTP Content-Type header on the request body or arg entity (this only takes precedence in a request like `/update` which has multiple text entities with a possible content-type)

You can ask for an ***output format*** in two ways:

1. Explicitly through the `format` arg, which takes precedence.
2. As the HTTP Accept header in the request headers, which may be a list of formats and qualifiers; the repository implements full HTTP 1.1 content negotiation.

Note that the tabular (tuple) and boolean query result formats are output-only. There are no requests that take them as input formats.

RDF Serialization formats:

Name	Symbol	Default MIME type	Additional MIME types
RDF/XML	RDFXML	application/rdf+xml	application/xml
N3	N3	text/rdf+n3	
N-Triples	NTRIPLES	text/plain	
TriG	TRIG	application/x-trig	
TriX	TRIG	application/trix	
NTriples With Context [1,2]	Context-NTriples	text/x-context-ntriples	
HTML [2,3]	RDFHTML	text/html	application/xhtml+xml

[1] The Context-NTriples format is not an official RDF serialization; it was added for this repository, as a convenient way to export quads for testing. Note that it was formerly named NQuads, but there is already a different unofficial format known to the RDF community that is called "NQuads".
[2] This format only supports output, it cannot be read by the repository.
[3] HTML is for interactive viewing only, it cannot be parsed.

Tuple (Tabular, SPARQL SELECT) Query Result Serialization formats:

Name	Symbol	Default MIME type	Additional MIME types
SPARQL/XML	SPARQL	application/sparql-results+xml	application/xml
SPARQL/JSON	JSON	application/sparql-results+json	
TEXT	TEXT	text/plain	
HTML	HTML	text/html	application/xhtml+xml

Boolean Query Result Serialization formats:

Name	Symbol	Default MIME type	Additional MIME types
SPARQL/XML	SPARQL	application/sparql-results+xml	
TEXT	TEXT	text/boolean	

Identify and/or "Create" Current User (/whoami)

This request returns a tabular report on the current authenticated user's identity, to compose a friendly display in a Web UI. This is simply a SPARQL query wrapped in a servlet to hide the details of the internal data.

This request can also function as a "login" mechanism to establish a session and cache authentication, while at the same time getting the user's displayable name to show in the UI.

Subtle Alternate Function: Note that the POST form of this request has a separate function when `create=true`: It creates the RDF metadata for a login user account. Normally this is only done as part of the initial bootstrap procedure, by the `finish-install.sh script`. When users are created through the Admin UI or `/import` their RDF metadata is created automatically.

NOTE: Perhaps it would be better to implement this as a redirect to the resolvable URI for the person, which would then yield a description compatible with the FOAF standard. That's how the Semantic Web wants us to manage it.

URL: `/repository/whoami (GET, POST)`

Args:
`format`---Same as for SPARQL result format, same default (SPARQL XML)
`create=(true|false)` When true, invokes alternate function of this service to create RDF metadata for current user (see explanation above).
`firstname=text`---**ONLY** when create=true, the first name value of the created User instance (optional).
`lastname=text`---**ONLY** when create=true, the last name value of the created User instance(optional).
`mbox=text`---**ONLY** when create=true, the `mbox` value of the created User instance(optional).

GET Result:
Response document is a SPARQL tuple result, format determined by the same protocols as for `/sparql`. It contains the following columns:

1. **uri**---resolvable URI of the :Agent instance for this person if any (see below).
2. **username**---short user name as given to the login process
3. **firstnamefirst**---(given) name, and middle names if any
4. **lastname**---user's last (family) name
5. **mbox**---user's email address

Note that the last 3 fields may be empty if that data is not available.

If there is no `:Agent` instance for the logged-in user, the URI will revert to their implicitly asserted Role, for example, `:Role_Superuser` for an administrator. This is the same URI that appears in provenance metadata entries like `dc:creator`.

POST Result:
When `create=true`, the result document is empty, and the status code indicates success:

- **201** when a new User instance was successfully created.
- **409** (conflict) when a User instance already exists. THIS SHOULD BE CONSIDERED SUCCESS!
- Any other code indicates failure.

Access:
Open to authenticated users.

Create New Instance Identifier(s) (/new)

This call creates one or more globally unique, resolvable, URIs for new resource instances. It does not add any data to the repository; the instances will not exist until a user inserts some statements about them. The URI namespace is the default namespace from the configuration properties, followed immediately by the unique identifier. Note that ETL tools may request thousands of URIs at once so the mechanism to produce unique IDs must be able to handle that.

URL: `/repository/new` (POST only)

Args:
`count`---number of URIs to return; optional, default is 1.
`format`---same as for SPARQL result format, same default (SPARQL XML)

Result:
The requested number of new URIs are returned, packaged as a SPARQL query result for a field named "new". Its encoding is determined by the `format` parameter or, if none specified, by the `Accept` header of the HTTP request. Default is SPARQL/XML.

Access: Requires an authenticated user.

Disseminate Instance (/i, /repository/resource)

The disseminate service returns the RDF content of an instance; it is how the URI is resolved to implement the Linked Open Data paradigm of the Semantic Web. Note that there are actually three valid ways to construct a request for any given data instance:

1. `/i/instance-ID`---assumes that URI prefix matches Web server's DNS address, in other words, the configured default URI namespace.
2. `/i?uri=instance-URI`---retrieves any instance URI whether the prefix matches the default namespace or not. This allows one repository to resolve multiple domains.
3. `/repository/resource?uri=instance-URI`---Just like the `/i` form, only with authentication required. This is the recommended URL for programs accessing resource contents through the REST API, since `/i` might not require or make use of authentication credentials.

URL: `/i/instance-ID` (GET or POST method)
`/i`
`/repository/resource`

Args:
`uri=uri`---optional, only if a URI is not specified as the tail of the request URI; an alternate way to explicitly specify the complete URI of the resource to disseminate. Allows any URI to be accessed, instead of assuming that the URI's namespace matches the hostname, context, and servlet path ("/i") to which the repository's webserver responds.
`format=mimetype`---optionally override the dissemination format that would be chosen by HTTP content negotiation. Note that choosing `text/html` results in a special human-readable result.
`view=view`---optionally choose a different view dataset from which to select the graph for dissemination. Mutually exclusive with workspace.
`workspace=uri`---URI of the workspace named graph to take the place of the default graph. Relevant metadata and ontology graphs are included automatically. Mutually exclusive with view.
`noinferred` (boolean)---Exclusive of all inferred statements from the generated results. This only applies to `rdf:type` statements; if the noinferred query argument is present (it need not have any value) then inferred types are left out of the results.
`forceRDF` (boolean)---forces a result of serialized RDF data. When the negotiated result format is text/html, the usual choice is to generate the human-readable view. This forces an HTML rendering of the RDF statements which can be handy for troubleshooting, especially when combined with noinferred. NOTE: This is the only way to see the Embedded Instance statements in an interactive HTML view, for example, in a web browser, so it is especially handy for generating a clean view for debugging EIs. Default false.
`forceXML` (boolean)---when an HTML format would be generated, output the intermediate XML document instead of transforming it to XHTML. This is mainly useful for obtaining examples of the intermediate XML for developing new XSLT stylesheets and testing/debugging. Default is false.

Result:
Returns a serialization, or optionally, human-readable HTML rendering of the graph of RDF statements describing the indicated resource instance. Note the deliberate choice of words: This graph includes not only the statements of which the URI is the subject, but also:

1. Statements describing embedded instances in the resource instance, for the same meaning of "describing" (but note that EIs are not recursive, an EI may not have its own EIs.)
2. Statements about the "Label" properties of all predicates and object-property values of the instance and its EIs. (The exact set of properties used for "label" is configurable.) See the SWAG page for more about label properties.
3. Provenance and some administrative metadata about the instance.

Note that depending on the authenticated roles of the requesting user and the configured access controls, some properties may be excluded from this result. For example, in some cases, unauthenticated users will not see certain properties which may contain confidential information.

About HTML dissemination:
When the negotiated format is text/html, and unless either of the `forceRDF` or `forceXML` args was given, the dissemination process creates an intermediate XML document and transforms it into XHTML with the configured XSLT stylesheet. See description of the `eaglei.repository.instance.xslt` in the Repoistory Administrator Guide.

If no XSLT stylesheet is configured, the intermediate XML document is delivered instead, with a media content type of application/xml. Note that this means, to obtain correct XHTML output, you MUST configure an XSLT stylesheet.

The content of the intermediate XML format is described in a W3C XML Schema document that may be downloaded from a running repository at for example,

```
[https://localhost:8443/repository/schemas/instance.xsd]
```

We provide an example transformation stylesheet that produces very simple HTML, intended to be the basis of custom stylesheets. It is available for download at:{{https://localhost:8443/repository/styles/example.xsl}}We manage the transformation within the repository, instead of adding an xml-stylesheet processing instruction to the XML, for compelling reasons:

1. Client-side transformation is a relatively new concept and we cannot trust it to be implemented reliably and consistently.
2. Some user agents, such as Web crawlers, depend on receiving true HTML content and so they should not be given XML. This might be determined by content negotiation but we do not expect that to be reliably implemented either.
3. Some aspects of the transformation depend on parameters in the stylesheet (see below) that are supplied at run-time from repository configuration values.

The transformation stylesheet is supplied with these parameters when it is invoked. They should be declared with `<xsl:param name="..."/>` directives in the XSL. Be sure your stylesheet can cope with parameters that are not set, by supplying default values.

- `__repo_version`---string containing Maven version spec of the running repository code. This is always set.
- `__repo_css`---configured value of `eaglei.repository.instance.css`, can not be set.
- `__repo_logo`---configured value of `eaglei.repository.logo`, cannot be set.
- `__repo_title`---configured value of `eaglei.repository.title`, cannot be set.

Property Filtering

The set of properties returned in the HTML view is based on the same result as RDF disseminations, which is automatically filtered as necessary for the requesting user's access level.
Access:

Requires read permission on all named graphs in the query's chosen view. Note that this is the ONLY service available to unauthenticated users, so it must be able to gather a useful result from named graphs readable by the Anonymous role. If you do access this service with credentials, you will be able to see instances and properties that would be invisible to an unauthenticated user, for example, instances in private workspaces that are still in unpublished workflow states.

Note that when the requesting user does not have read access to the requested instance, it will appear to him/her that it does not exist; the error returned is identical to one for a nonexistent resource, since it is essentially the same case.

There is also access control on some individual properties of the resource: those properties identified as hidden and contact properties by the data model ontology (and its configuration, see that separate document). The access controls on the resource URI configured as `datamodel.hideProperty.object` regulate hidden properties, and `datamodel.contactProperty.object` regulates contact properties.

Anyone with READ access on the URI gets to see the properties. To expose them to the world you'd give access to the Anonymous role. Normally only Curator, RNAV, and Lab User roles would be granted access to hidden and contact properties since they have to see and manipulate them through the data tools.

Update a Single Instance (/update)

This service actually implements three different kinds of requests:

1. Create a new resource instance,including embedded instances. It must specify its home graph.
2. Obtain the edit token with which to modify an existing resource instance.
3. Modify ("update") the properties of a resource instance, both delete and add in one transaction. This includes adding, deleting, and modifying any embedded instances.

The update operation does all its work in the instance's home named graph. For an existing instance being modified, it is computed as the named graph in which the asserted `rdf:type` statement(s) are found.

When creating new resources: Since the create operation doesn't have an instance from which to derive its home graph, its home graph must be specified as the workspace arg.

Workflow implications of creating new resources: Since the `/update` action that creates a new resource instance is effectively performing a transition from the New workflow state, the current user must have permission to make such a transition to the destination workspace; if there are multiple transitions, one is chosen arbitrarily.

Acquiring and use of edit tokens: The edit token is intended to "protect" the read-only copy of the instance that you (presumably) download as a basis for edits. The correct sequence of operations when modifying an instance is:

1. Obtain the edit token.
2. Get RDF for the resource instance.
3. Submit an `/update` request to modify the resource instance, with token (1).

This ensures that no matter how much time passes between (2) and (3), if, for example, a user dawdles over an interactive edit or forgets and leaves it overnight, the edit token is already in place to indicate his/her intention to make a change. It does not prevent another user from coming along and grabbing the token to make a change, but it will indicate that there is an edit in progress, and it will prevent a stale copy from being checked in.

Comparison with SPARQL/UPDATE: In case you are wondering why we chose to implement this complex special-purpose service instead of a general protocol like SPARQL/UPDATE, there were some compelling reasons:

1. Too difficult to impose the necessary fine-grained access controls on SPARQL/UPDATE. This service is defined to be a transaction that only affects one resource instance (and its embedded instances).
2. Major benefit of S/U protocol is accepting a list of deletes and a list of inserts, which we do here.
3. SPARQL/UPDATE is still not a recognized standard, and implementations are poor.

When `action=create`, there must be no existing statements in the repository with the given URI as a subject. The request must include an insert arg containing one or more statements whose predicate is `rdf:type`. (All of the subjects must match the request URI). It is an error to specify a delete arg.

When `action=gettoken`, an edit token is created if necessary, and returned along with the user who created it, time it was created, and a boolean value that is true if it was newly created by this request. This information is intended to help a UI service advise the user when there might be an edit in progress, if the boolean was false and the timestamp on the token is recent.

When `action=update`, there must be an existing instance matching the URI of the request. DO NOT specify the workspace arg, since the repository automatically finds the instance's home graph and makes all changes there.

Updating a resource instance's properties requires an edit token. The token lets the server check that your edits are based on the current valid state of the resource; if another update occurs before yours, its changes could be corrupted or lost. To update, first, run the `/repository/update` request with the arg, `action=gettoken` to obtain the current edit token, creating one if necessary. Then, get the content properties as before. When calling `/repository/update` again with `action=update` add the `token=token-uri` arg.

Note on file format and character set: The request specifies the file format and/or character set of the serialized RDF data as a Content-Type header value in the entity bodies of insert and delete args, for example, `text/rdf+n3; charset="ISO-8859-1"`. The character set defaults to Unicode UTF-8, so if your source data is not in that character set you must declare it. The content-type can be provided in two different places – they are searched in this order of priority, and the first one found is the only one considered:

- `Content-Type`---header on the value of the content entity in a POST request. This takes precedence because it allows for different content-types in insert and delete args.
- `format`---query argument value.

URL: `/repository/update` [*/instance-ID*] (POST only)

Args:
`uri`---optional way to explicitly specify the complete URI, instead of assuming that the URI's namespace matches the hostname, context, and servlet path ("/i") of this webserver.
`format`---the default expected format for insert and delete graphs. If the args specify a content-type header, that overrides this value. Only recognizes triples even if the format supports quads.
`action=(update|create|gettoken)`---Update to modify an existing instance, create adds a new one. See below for details about gettoken.
`token=uri`---When action is update or create, this must be supplied. The value is the URI returned by the last gettoken request.
`workspace=uri`---choose workspace named graph where new instance is created. Only necessary when action=create. Optional, default is the default workspace. DO NOT specify a workspace when action=update.
`delete`---graph of statements to remove from the instance; subject must be the instance URI. Deletes are done before inserts. Graph may include wildcard URIs in predicate and/or object to match all values in that part of a statement.
`insert`---graph of statements to add to instance; subject must be the instance URI.
`bypassSanity`---(boolean, default false, deprecated) **NOTE: It is best if you pretend this option does not exist**. When true, it skips some of the sanity tests on the resulting instance graph, mostly the ones checking the integrity of Embedded instances. Requires Administrator privilege. This was added to make the data migration from broken old EI data possible, it should rarely if ever be needed.

The delete wildcard URI is `http://eagle-i.org/ont/repo/1.0/MatchAnything`

Result:
HTTP status indicates success or failure. Any modifications are transactional; on success the entire change was made, and upon failure nothing gets changed.

When `action=update` is used to effect a change to a resource instance, this service automatically optimizes the requested change so the fewest actual statements are modified. For example, if the request deletes all statements by using the wildcard URI in the position of predicate and value, and then inserts all of the statements that were there before along with an additional new statement, the only change actually made is to add that new statement. Since a gratuitous change to an `rdf:type` statement results in extra time spent inferencing, it is best to avoid it when possible.

When an update fails because the edit token is stale, the HTTP status is always 409 (Conflict). If this occurs, the only solution is to get a fresh token and re-do the update. It is NOT advisable to have a client simply retry the update with the same data, at least inform the user that there has been an intervening edit and updating now would destroy somebody else's changes.

When `action=gettoken`, the response includes a document formatted (according to the chosen format) as a SPARQL tuple result. It includes the columns:

`token`---URI of the edit token. It has no meaning other than its use in an update transaction. This is the last edit token created (and not yet "used up" by an update) on this instance; or else a new one that was created if there wasn't one available.
`created`---literal timestamp at which the token was created. It may be useful to display the date to the user if there was an existing token of recent vintage.
`creator`---URI of the user who created the token
`new`---boolean literal that is true if the `gettoken` operation created a new token. When false, it means the token already existed, which indicates there MAY already be another user's update in progress and that update may be in conflict with yours.
`creatorLabel`---`rdfs:label` of the creator, if available.

Access:
Requires ADD access to either the instance itself or its home named graph if the insert argument was given and REMOVE access on the instance or the graph if the delete argument was given. When `action=create`, requires READ access on an appropriate Workflow Transition, out of the New state.

We may eventually decide to implement a quota on the count of statements that may be added or deleted, as a protection against DOS attacks and runaway clients.

SPARQL Protocol Endpoint (/sparql)

This servlet is a complete implementation of the standard SPARQL Protocol, implemented according to the description in `http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/`. It also extends the protocol to support specific features of the repository REST API, such as views and workspaces.

The most significant departure from the "standard" is the treatment of the dataset clauses in the SPARQL query language. Sesame's query engine ignores any dataset specified in the query if an explicit dataset is supplied in the API call. This SPARQL endpoint will always supply an explicit dataset in order to enforce access control: the dataset determines which named graphs the query is allowed to "see". If the HTTP request asks for certain named graphs by URI, they will be included in the dataset so long as the current user has read access to them.

Matching or restricting by Named Graph: If you want to use the GRAPH keyword as described in section 8.3 of the SPARQL Specification, Sesame's implementation requires named graphs (context) to be present in both the default-uri and named-uri lists of the dataset. The view and workspace abstractions manage this automatically, but if you are constructing a dataset yourself be sure to heed this advice if you use GRAPH.

Inferred Statements: The inferred option determines whether inferred statements are included in the query results. Its default is true. This is a nonstandard exstension to the SPARQL protocol.

Character Sets: The SPARQL query document is entered as the value of the HTTP `query-argument` query in a GET or POST request. When the query is embedded in the URL in a GET request, or in a body of `content-type application/x-www-form-urlencoded` in a POST, it MUST use the character set of the URL. This is supposed to be ISO Latin-1 (ISO-8859-1) according to the HTTP 1.1 spec, but in practice browsers seem to use UTF-8-encoded Unicode.
If you put multi-lingual characters in a SPARQL query document, which is perfectly legitimate in literal values, the best way to ensure they are interpreted correctly is to declare the content-type explicitly with a charset parameter. To do this you must send a POST request with the entity body in multipart/form-data format, so that each argument value is encoded as a separate entity. This adds complexity, but it is the best way to ensure your HTTP client conveys the character set identification correctly. Within the entity for the query arg, add the header: `Content-Type: text/plain;charset="UTF-8"`
If you're using UTF-8. The media type should always be `text/plain`.

URL: `/repository/sparql` (GET and POST)

Args:
`query`---the text of the SPARQL query document.
`default-graph-uri`---add this URI to the list of default graphs in the query dataset, overriding any built-in default. Read access required. Conflicts with other explicit methods of specifying datasets: the `*-graph-uri`, `view`, and `workspace` arguments are all mutually exclusive. Can be specified multiple times to include multiple graphs.
`named-graph-uri`---add this URI to the list of named graphs in the query dataset, overriding any built-in default. Read access required. NOTE: Sesame's query engine does not seem to pay attention to named graphs, just default graphs, so this argument should be deprecated.
`format`---explicitly names a result format, overriding the default from HTTP content negotiation. Value must be one of the specified MIME type strings. This is a nonstandard extension to the SPARQL Protocol.
`view`---symbolic name of a pre-packaged set of named graphs presenting a "view" of the data; see secton on Views in the Concepts section above. Conflicts with other explicit methods of specifying datasets: the `*-graph-uri, {{view`, and `workspace` arguments are all mutually exclusive. This is a nonstandard extension to the SPARQL Protocol.
`workspace`---URI of the workspace named graph to take the place of the default graph. Relevant metadata and ontology graphs are included automatically. Conflicts with other explicit methods of specifying datasets: the `*-graph-uri`, `view`, and `workspace` arguments are all mutually exclusive. This is a nonstandard extension to the SPARQL Protocol.
`time`---sets an explicit time limit on the query, overriding any built-in default. Value is in seconds. This is a nonstandard extension to the SPARQL Protocol.
`inferred`---boolean arg. When false, any inferred statements are excluded from the results. The default is true. This is a nonstandard extension to the SPARQL Protocol.

Result:
The HTTP response includes a result document, whose format depends on the requested format and on the type of query: SELECT queries produce tabular results, ASK and DESCRIBE return an RDF graph, and ASK writes a boolean result. See the MIME Types tables above for all the result formats. In the event of a failure, an HTTP error status is returned.

Timeouts: This service imposes a time limit on the query to prevent one requestor from monopolizing server resources with a time-intensive query. The time limit is taken from the first of these sources with a value:

1. The time argument to the service request
2. Configuration option `eaglei.repository.sparqlprotocol.max.time`
3. The built-in default is 600 seconds (10 min) if nothing is configured.

Also note:

- Any user can override this setting to impose a ***shorter*** timeout by giving a value for the nonstandard time argument.
- Only the Administrator can override with a longer timeout.
- If a SPARQL Protocol request cannot be complted within the timeout, it returns an HTTP 413 status (Result too large)

Access:
Requires Read access to all of the named graphs included in the dataset. When `workspace` or `view` arg with a dynamic view is supplied, the dataset is automatically constructed to include only readable named graphs out of a set that depends on the chosen view or workspace.. If a set of graphs is explicitly requested, it is an error if any of them are not readable.
Access NOTE: It MAY be desireable to configure the servlets to allow unauthenticated access to `/sparql`. This would only allow query access to named graphs which the anonymous role can read.

List Named Graphs (/listGraphs)

Returns an enumeration of all named graphs in the repository matching the search criteria, and which the current authenticated user can read. This request is equivalent to a SPARQL tuple query, except it hides the details of internal administrative metadata structures and bypasses most users' lack of read access to that metadata.

This is intended to be used for UI tasks such as populating a pick list of workspaces available to a user creating a new instance.

URL:`/repository/listGraphs (GET, POST)`

Args:
`type=(ontology|metadata|workspace|published|internal)`---Restrict results to graphs of the indicated type. Default is to return all graphs, this arg is optional.
`format`---same as for SPARQL result format, same default (SPARQL XML)

Result:
Response document is a SPARQL tuple result, format determined by the same protocols as for `/sparql`. It contains the following columns:

`namedGraphURI`---URI naming the graph
namedGraphLabel}}---Literal, its textual label (rdfs:label) in the admin metadata if any.
`typeURI`---the URI of its named graph type
`typeLabel`---String literal, the textual label of the type.
`version`---Last loaded version, or just value of owl:versionInfo property of the graph's name URI, which is NOT necessarily the correct version, for example, not for the eagle-i data model ontology.
size - count of asserted statements. Note that the inferred graph always shows size of 0. (Long integer literal.)
`read`--- true if current user has read permission (Boolean literal)
`add`--- true if user can add statements to this graph.(Boolean literal)
`remove`--- true if user can remove statements from this graph (Boolean literal)

NOTE on "undocumented" graphs: It is possible that named graphs will get created in the underlying RDF database (Sesame) without the corresponding repository metadata, such as type and label. You can detect an undocumented graph because there will be no value for typeURI. You can use the admin UI to set the type and label, and add access grants.

Access:
Requires authenticated user; reports on all graphs and contexts regardless of your access..

Dump Contents of a Named Graph (/graph - GET)

Creates a serialization of all statements in the indicated graph (or, optionally, all named graphs in the repository). This is typically used for administrative tasks such as backing-up and migrating the contents of the repository.

Non-administrative users should never need to dump an entire graph, and in fact should never use this request. Since the data returned by e.g. a dissemination request on an instance comes from multiple named graphs, the dump of just one graph will be missing some relevant statements.

Request must include either the name argument or `all`.

URL: `/repository/graph` (GET)

Args:
`format`---output serialization format, overrides what HTTP content negotiation would decide.
`name=URI`---Name of graph to dump - mutually exclusive with **all**.
`all`---dumps all (accessible) graphs. Format *must* be one that encodes quads, e.g. TriG or TriX, or an error will occur.
`inferred`---include inferred statements as well. By default they are left out. This argument does not need a value, it is boolean so just including its name as a query arg will assert it. (SEE WARNING BELOW)

Result:
Output document is the serialization of the selected RDF statements. Any failure returns an appropriate HTTP status.

WARNING: When you dump a graph with the inferred option on, be sure it does NOT get re-loaded into the repository because then the inferred statements will become explicit and will not get re-computed when necessary. It was mainly intended to be used in testing.

Access:
Requires read access on the selected graph(s). When all is specified, non-readable graphs are ignored.

Load Serialized RDF into Named Graph (/graph - POST, PUT)

Import a serialized RDF document into a specific named graph, optionally replacing all previous contents. The repository records some provenance metadata about when and who did the import, and a description of the source material.

If the name parameter is not specified, and the format supports named graphs or quads (e.g. TriG or TriX), then the graph name is taken from the ingested data. This is very dangerous, but is useful for restoring an entire repository from a backup, for example. It requires administrator role.

The action argument lets you choose whether to add to or replace the graph contents. To remove all contents of a graph, upload an empty serialization in replace mode.

Note on File Format and Character Set: The request specifies the file format and/or character set of the serialized RDF data as a Content-Type value, for example:

```
text/rdf+n3; charset="ISO-8859-1"
```

The character set defaults to Unicode UTF-8, so if your source data is not in that character set you must declare it. This can be provided in several different places,they are searched in this order of priority, and the first one found is the only one considered:

`format`---query argument value. This takes precedence because some clients may not allow complete control of the content-type headers.
`Content-Type`---header on the value of the content entity in a POST request.
`Content-Type`---header in the body of a PUT request.

URL: /repository/graph (PUT or POST)

Args:
`name=named-graph-URI`---required name of the affected graph. Mutually exclusive with all.
`all`---loads serialized data into all (accessible) graphs. Format must be one that encodes quads, e.g. TriG or TriX.
`action=(add|replace|delete)`---required, determines whether new statements are added to existing contents of the graph, replace the contents, or are deleted from the graph.
`format`---MIME type input serialization format, overrides the Content-Type in HTTP protocol.
`type=(metadata|ontology|workspace|internal|published)`---keyword representing the type of content in this named graph.
`label=text`---value for rdfs:label property of the named graph, which appears as its title.
`content=`---The immediate content of serialized RDF to load into the graph. ONLY used when method is POST, but then it is required.
`source=string`---The file or URL to record as the source of the data being ingested into the graph. Only applies to add and replace actions. Overrides any filename specified with the content, which would otherwise be recorded as the source identifier. It is recorded as the dcterms:identifier property of the dcterms:source node in the metadata.
`sourceModified=dateTimeStamp`---last-modified time for the source of data being ingested. Value must be parsed as XSD date-time expression. It is recorded as the `dcterms:modified` property of the `dcterms:source` node in the metadata.

Result:
HTTP status indicates success.

Access:
If the graph already exists, requires add access (and remove access action is replace or delete). If there is no existing graph by that name, only the administrator can create a new one. For multi-graph (all) load, requires superuser access.

Logout (/logout)

Destroys any lingering session state and credentials for the current user. Users of shared or public computers need a way to positively end a session so that subsequent users do not get their access to the repository.

NOTE: Beware of this if you are using a Web browser to access the repository The repository uses HTTP Basic Authentication to identify users. Most web browsers cache the last Basic Authentication credentials for a site and don't offer an easy way to clear it.

For Mozilla Firefox 3, try the Web Developer add-on, and select Miscellaneous-> Clear Private Data-> HTTP Authentication from its menu.

URL: `/repository/logout` (POST only)

Args: none.

Result:
HTTP status indicates success. Succeeds even if there was no session to destroy.

Access:
Requires a login and established session.

Check or Load Data Model Ontology (/model) - *Admin Only*

This service either reports on or loads the Data Model Ontology. The `GET` request returns a report of the version of the ontology loaded, and the version that is available to be automatically loaded from within the repository's codebase. The `POST` request give a choice of selective update or forcibly replacing the ontology.

URL: `/repository/admin/model` (GET)

Args:
`format`--- MIME-type of desired output format

Returns a tabular result of two columns:

1. **loaded**---the version of the data model ontology currently loaded, if any. For example, "0.4.6"
2. **available**---the version of data model ontology that is available to be loaded.

The value of loaded is the result of this query over the **ontology** view:

```
select * where {<http://purl.obolibrary.org/obo/ero.owl> owl:versionInfo
?loaded}
```

URL: `/repository/admin/model` (POST)

Args:
`action=(update|load)`
`source=(jar)`

Loads a new copy of the data model ontology, replacing the entire existing contents (if any) of its named graph. For more details about how the graph name and data model source are determined, see The Data Model Configuration Properties Guide.

When `action=update`, only load the ontology if the version of the current ontology is different from the version of the available one (also if there is nothing loaded yet). Note that it does **not** attempt to judge if the available version is *later*, just *different*.

When `action=load`, the "available" ontology is always forcibly loaded.

The `source` argument determines where the ontology data comes from. When `source=jar`, which is recommended, the ontology is the collection of all OWL files in the maven package `eagle-i-model-owl}}as it was loaded into the repository's source base. This ought to reflect the version of the ontology for which the sofware was designed. When {{source=model`, the `EIOntModel` is used as the data source. Although there *should* be no difference between them, because the `EIOntModel` is loaded in to Jena from the same source files, in practice there appears to be some differences.

Manage Internal Graphs (/internal)

URL: `/repository/internal` (GET, POST)

This service manages the *internal* named graphs which are (usually) automatically initialized by the repository upon bootstrap. Each of these graphs has an *initializer* data file of seralized RDF that is a versioned part of hte source and gets built into the webapp. This API service lets you get the initializers and read or query them so that e.g. an upgrade procedure can integrate differences into a locally-modified internal graph.

Extra unrelated action: When `action=decache` is specified, this service tells the running repository that it should *decache* any in-memory caches of internal RDF metadata because internal graphs have been changed behind its back, e.g. by the `/internal` request.

This service is intended for repository internal maintenance and testing procedures ONLY. If{ you are planning to use it in a user application, you are very likely making a bad mistake. Its API is unstable and will probably change without any notice.

GET Method:

name=*uri* - optional, URI of the internal graph to report on, default is to list all known intenral graphs.**format**=*mimetype* - optional, default is negotiated, must be tuple query result format.

Result is a tabular query response with the following columns:

1. `name`--- URI of the graph
2. `loaded`---version of initializer that was loaded, if any
3. `available`---version of initializer that is avaialble and would be loaded.

POST Method:
`action=read|load|query|decache*`---required, specifies what to do
`name=uri`---required when `action=load,read,query`; in fact, can be repeated for `query`.
`format=mime`---mime type of response: when `action=read` must be a serialized RDF format; when `action=query` must be suitable for query result type.
`query*=_sparql-text`---text of query to run over initializer for graph(s)

This is a multi-function service, depending on the value of `action`: When `action=load`, replaces contents of an internal graph with hte initializer in the webapp. For `action=read`, returns contents of *initializer* for given graph in the result docuemnt. For `action=query`, runs given SPARQL query over contents of initializer(s) for given graphs (this is the ONLY form that can take multiple graphs). For `action=decache`, tells repository server to decache all in-memory data derived from RDF content in case, for example, the internal graphs have been modified or updated. All of these actions require *Administrator* privilege.

Harvest Resource Metadata (/harvest)

This service harvests the "metadata", which is actually the resource instance data from the repository. It is intended to be used by an external search index application to keep its index up-to-date by incremental updates, since that is much more efficient than periodically rebuilding the index.

It returns a result in any of the SPARQL query tuple response formats, selected by the `format` arg. The columns are:

1. `subject`---the resource instance's URI
2. `predicate`---predicate of a statement about the resource (only present when `detail=full`)
3. `value`---value of a statement about the resource (only present when `detail=full`)

Note that *deleted* resource instances are only indicated when a report over a time span is selected. When the `from` argument is not specified, the complete current contents of the repository are returned so deleted instances are simply not indicated. We assume the caller is rebuilding its index from scratch so there is no need to remove instances that have been deleted.

Property Filtering

The set of properties returned when `detail=full` is automatically filtered to remove *hidden* and *contact* properties as dictated by the access privileges of the user making the request. The filtering follows the same access-control rules as the Dissemination service.

URL: `/harvest`; (GET or POST method)

Args:
`format`---optionally override the dissemination format that would be chosen by HTTP content negotiation. Note that choosing `text/html` results in a special human-readable result.
`view`---optionally choose a different view dataset (see Views in concepts section) from which to select the graph for dissemination. Mutually exclusive with `workspace`. Default is the `published-resources` view. Be careful to choose a view that does not include repository user instances.
`workspace`---URI of the workspace named graph to take the place of the default graph. Relevant metadata and ontology graphs are included automatically. Mutually exclusive with `view.`
`inferred`---When true, includes all inferred statements from the generated results. This really only applies to `rdf:type` statements; default is *false*, so inferred types are left out of the results. Must be false when `detail=identifier.`
`from`---(optional) only resource instances last modified from (on or later than) this timestamp or later are shown. See Date/Time Format below.
`after`---(optional) does the same thing as `from`, with which it is mutually exclusive, only the time is exclusive rather than inclusive. That is, only resources modified *later than* this timestamp are shown.
`until`---(optional, *not implemented* yet) only resource instances last modified `until` this timestamp or before are shown. See `Date/Time Format` below. **NOTE:** this may not be fully implemented until versioning is done.
`embedded`---(boolean) include URIs of Embedded Instances (see Concepts section) in a report at the *detail=identifier* level. Has no effect at `detail=full`. This is usually not even a good idea, was included for troubleshooting.
`detail=(identifier|full)`---adjust level of detail.*Required* This affects the way deleted items are represented in the results:

- When `identifier` is chosen, only the identifier of each resource is returned in the `subject` column, AND deleted subjects are indicated by prefixing the URI with a fixed URI prefix such as "`info:deleted`" to represent that the URI in the fragment is deleted.
- For **full** results, a deleted item is indicated with a synthetic statement: the predicate `:isDeleted` and a boolean true value.

About Date/Time Format:
The date/time expression for the **from** and **until** args must be in the XML date format, which is a slightly restricted version of the ISO8601 date format. Note that if the time component is included, hours, minutes *and* seconds must be specified. Here are some examples:

- **2010-06-28** - June 28, 2010, the start of the day (just past midnight).
- **2010-06-28T13:45:06** - June 28 2010, 13:45:06 in the local time zone
- **2010-06-28T13:45:06.123** - June 28 2010, 13:45:06 and 123 milliseconds, in the local time zone
- **2010-06-28T17:45:06Z** - June 28 2010, 17:45:06 in UTC (GMT, Zulu) time
- **2010-06-28T13:45:06-04:00** - June 28 2010, 13:45:06 in the time zone 4 hours west of Greenwich

Result:
Response document is a SPARQL tabular ("select") query result, formatted according to the chosen or negotiated format.

Only resources in the named graphs selected by the view or workspace arg in the query are shown. For time-span queries, deleted instances in *all* graphs will be returned, so some of the delete notifications may not correspond to items in the caller's index. (On the other hand, it is also possible for a resource instance to be created and deleted *in between* harvests, so there is *always* the possibility of getting a deleted notice for a resource that the caller never indexed.)

About Incremental Updates:
The 'from' argument is intended to let you get incremental changes to keep a search index up to date. However, it is *essential* that you give each incremental request a timestamp that accurately reflects the actual time *on the server* of the last harvest. Since there are delays in transmission and server and client may not have synchronized clocks, you *should not* rely on the client's time for this. Each `/harvest` response includes an HTTP header **Last-Modified** which has the time of the last modification to the repository.This is *guaranteed* to give you all of the relevant changes in the next harvest. Unfortunately, since that timestamp comes back in HTTP format, you will have to parse it and translate it to XML format; also, it only has 1 second granularity so you *may* see some "false positives" for resources updated less than a second after that last-modified time, which also appeared in the last incremental harvest.

Precise last-modified timestamp: the `/harvest service` also adds a non-standard HTTP header to its responses to describe the last-modified time in full precision down to the millisecond level. This allows an application to give that precise time as the **after** argument value on the next incremental harvest to be sure of capturing any changes since that last /harvest call. The header name and date format looks like:

```
X-Precise-Last-Modified:Mon, 10 Jan 2011 20:49:10.770 GMT
```

Note that the time format is essentially the HTTP time format with ".mmm" appended to the time for milliseconds. It must be converted to the XML timestamp format to serve as an argument value to "**after**". By the time it was obvious what a stupid choice this was, it was too late in the development cycle to change easily, but perhaps in the future an XML-format precise timestamp header can be added.

Access control:
Read permission is required on all desired graphs.

Export and Import of Users, Resource Instances (/export, /import)

This service lets you export the users OR data contents of one repository and import them into a different one, or a later instance of the "same" repository. It is intended to migrate and mirror data between different releases of the software and ontology, or different deployments created for testing. Although the same service can handle both "user" and "resource" instances, these are two separate operations and cannot be combined.

Each export operation creates a single file of serialized RDF data. DO NOT MODIFY OR EDIT IT, or the results will be undefined. Especially in the user-export function, there are some statements included ONLY as cues to the import service, so altering or removing them will have unpredictable results.

Import Roles before Users: When importing Users who might have some different Roles from the Roles already existing in your repository, if you wish to preserve those role memberships, import Roles first.

In general, importing data into an existing repository poses some problems of which you need to be aware:

- If the URI prefix is different in the destination, do you transform the subject URIs so they are resolvable, or keep the original URIs to match the source?
- How do you want to handle a duplicate instance in the import (i.e. when the same instance already exists in the repository)?
- Should a resource instance import be allowed to create new named graphs?

URL: `/export` (GET or POST method)

Args:
`format`---MIME type of the serialization format to be output. By default, it is chosen by HTTP content negotiation. Must be capable of encoding quads; default is TRiG (application/x-trig).
`type=(user|resource|transition|role|grant)`---which type of data is to be exported, one MUST be chosen and there is no default.
`view`---(only when type=resource) optionally choose a different view dataset from which to select the graph(s) for export. Mutually exclusive with workspace. Default is the published-resources view. Be careful to choose a view that does not include repository user instances.
`workspace`---URI of the workspace named graph to take the place of the default graph. Relevant metadata and ontology graphs are included automatically. Mutually exclusive with view.
`include=list..`---Explicit list of user or resource instances to include in the export.Format is a series of resource URIs separated by commas (",") and optional spaces. Users may also be referenced by username.
`exclude=list..`---Explicit list of user or resource instances to exclude from the export. Mutually exclusive with include, format is the same as the include list.

Result of an export request is a document of serialized RDF data in the requested format, unless the HTTP status indicates an error. Note that inferred statements are never included. This document is ONLY meaningful to an /import request by the same or another data repository. It is not intended to be human readable or meaningful. Its contents may not necessarily reflect the source repository's content directly.

Access control: Only the superuser may export users. Read access on the resident named graphs is required to export resource instances.

About Export/Import of Access Grants: For most types, the export includes access grants since these are considered administrative metadata. The type of "grant" is included so you can export and import just the grants on any URI, mainly to give you a way to transfer the access grants on Named Graphs (Workspaces). Also see the ignoreACL option on /import.

About Format and Charset of Imported Document: By default, format is indicated by the content arg's Content-Type header, at least when it is a separate entity in a POST entity body of type multipart/form-data. However, this is not always easy to control in an HTTP client. You can override it with the format arg. If a charset parameter is not specified in the content-type value, it defaults to UTF-8. Also note that the format must be capable of encoding quads; e.g. TRiG (application/x-trig).

URL: /import (POST method only)

Args:
`format`---the MIME type (and charset) of the input document's serialization format. This overrides any content-type on the content arg's entity.
`type=(user|resource|transition|role|grant)`---which type of data is to be imported, one MUST be chosen and there is no default. MUST match the type of the exported data.

`content=stream..`---the serialized RDF data to import, must have been generated by export of the same type of data.
`transform=(yes|no)`---Required, no default. When 'yes', URIs of imported instances are transformed into new, unique URIs resolvable by the importing repository. When 'no', URIs are left as they are.
`duplicate=(abort|ignore|replace)`---how to handle an import that would result in a duplicate object. Default is abort.
`newgraph=(abort|create)`---how to handle a resource instance import that would result in creating a new named graph for the data. Must be superuser to choose 'create'. Default is abort.
`include=list..`---Explicit list of user or resource instances to include in the import. Format is a series of resource URIs separated by commas (",") and optional spaces. Users may also be referenced by username. IMPORTANT NOTE: The URIs in the include (and exclude) lists are matched against URIs in the imported source file, NOT the repository.
`exclude=list..`---Explicit list of user or resource instances to exclude from the import. Mutually exclusive with include, format is the same. Useful for avoiding conflicts, for example, when loading a user export into a repository where the initial admin user already exists.
`ignoreACL=(true|false)`---Ignore all access control grants on imported objects. The imported objects will not have any specific access grants.

Send Contact E-mail (/emailContact)

This service has not been rigorously designed, so it does not have a specification of behavior. This section only documents how to call it, the internal actions it takes are still under discussion.

URL: `/emailContact` (POST method only)
Args:
`uri=subject-URI`---required
`client_ip=IP addr`---required
`from_name=personal name`---required
`from_email=mailbox`---required
`subject=text`
`message=text`
`test_mode`---default false, true if present.

Sends an email message to the designated contacts for the resource identified by URI. The exact rules for determining that contact address have not been formally specified yet. When a contact cannot be determined, the mail is sent to the configured postmaster for the repository. See the Repository Installation and Administrator Guide for details about configuring that and the rest of the e-mail service.

Workflow Services REST API

Show Transitions

`/repository/workflow/transitions`

Method: GET or POST
Args:
`workspace=URI`---restrict results to transitions applying to given workspace (includes wildcards). Default is to list all.
`format=mime`---type of result.

Returns the selected workflow transitions as SPARQL tabular results.
Result columns are:

- Subject URI
- Label
- Description
- Workspace URI
- Workspace Label
- initial-state URI
- initial-state Label
- final-state URI
- final-state Label
- allowed - boolean literal, true if current user has permission on this transition

Show Resources

/repository/workflow/resources

Method: GET, POST
Args:
state=URI|all---workflow state by which to filter, or 'all' for wildcard.
type=URI--- include only instances of this type (by inference counts)Default is no restriction.
unclaimed=(true|false)---when true, unclaimed resources are included in the report. Default is true.
owner=(self|all|none)---show, in addition to any selected unclaimed resources:

- self---those claimed by current user
- all---instances claimed by anyone
- none---includes no claimed resources. Default is self. Note that the combination unclaimed=false and owner=none is illegal.
  workspace=URI---restrict results to resources in a given workspace; optional, default is all available worksaces.
  format=mime---type of result, one of the SPARQL query formats.
  detail=(brief|full)---columns to include in results, see description below.
  addPattern=SPARQL-text---add this triple pattern to SPARQL query. May refer to variables as described below. Value is a literal hunk of triple pattern ready to be inserted into the patterns of a SELECT query. It may refer to the result variables mentioned below, e.g. ?r_type.
  addResults=query-variable..---add these variables to results columns. value is expected to be a space-separated series of query variables (which must appear in the triple pattern), e.g. "?lab ?labname"
  addModifiers=modifiers..---add a SPARQL modifier clause to the query. Modifers include ORDER BY, LIMIT, and OFFSET and can be used to implement pagination.

This request returns descriptions of a selected set of resources in SPARQL tabular result form. It is intended to be used to populate menus and autocomplete vocabularies, as well as more complex table views.

- The brief level of detail includes columns:
  - r_subject - URI of resource instance
  - r_label - label of resource instance
  - r_type - URI of instance's asserted type
- The full level of detail adds the fields:
  - r_created - created date from provenance
  - r_owner - URI of workflow claimant if any
  - r_ownerLabel - the rdfs:label of r_owner if it is bound (and has a label)
  - r_state - URI of workflow state
- Any query variables you specified in the addResults list are added to either result.

Claim Resource Instance

/repository/workflow/claim

Method: POST
Args:
uri=URI---subject to claim
user=UserURI---optional, user who asserts the claim; default is authenticated user.

Asserts a claim on the given instance. This requires the following:

- user must have access to a transition out of the current state.
- user must have read access to instance
- there is no existing claim on the instance.

There is no result document. The status indicates success with 200. As side-effects, this actino also:

1. Adds insert and delete access to the instance for the claiming user.
2. Sets :hasWorkflowOwner property to the claimant (in an internal metadata graph)

Release (claimed) Resource Instance

/repository/workflow/release

Method: POST
Args:

`uri=URI`---subject to release

Requires that current user is the owner, or Administrator role.

Side effects:

Removes insert and delete access to the instance for the current user.
Removes :hasWorkflowOwner property (in an internal metadata graph)

Transition on Resource Instance

`/repository/workflow/push`

Method: POST
Args:
`uri=URI`---subject to transition
`transition=URI`---indicates transition to take

Requires:

1. read access to instance.
2. must be claimed
3. user should be either claim owner or Administrator
4. user must have READ access on the chosen Transition.

Side-effects include:

- Implied release of any current claim (and all applicable side-effects of that)
- Executes action associated with this transaction, if any.
- Resource's workflow state will become the final state of the transition.

Admin UI Support Services

There are also some REST services created to support the administrative GUI (the JSP pages). They are NOT intended to be part of the regular API since their exact function and parameter structure is dictated by the JSPs and is thus still somewhat fluid. Most of them also require Administrator access.

If you use these services, it is at your own risk, and thus they are not documented. See the JSPs for usage examples if you are really curious. The URLs are as follows:

```
/repository/admin/updateUser
/repository/admin/updateGrants
/repository/admin/updateRole
/repository/admin/updateTransition
/repository/admin/updateNamedGraph
```

Site Home Page

Since the repository webapp is installed as the ROOT webapp (so it can have control over the URL space for the purpose of resolving all of its resource instances), it also controls the site's top-level "home" (index) page. The site home page is not part of the API, however, so it does not need to control the content of that page.

In order to give the site administrator control over the content of the site home page, the repo lets you redirect it to any URL, under the control of a configuration property. See eaglei.repository.index.url in the configuration properties.

Services to Support Admin GUI Pages

These REST services were included to support the administrative GUI, which is implemented as JSPs. They are targeted precisely at serving as the back end of the JSPs, so their range of function is strictly limited and they are effectively "write-only". Typically you call on the service and the HTTP status says whether it succeeded or failed; it does not return a detailed report.

Another feature of these services is that they look at the referring URL, and if it ends in ".jsp", they redirect back to that page upon a successful completion, passing through selected parameters and adding a "message" parameter to display the status of the operation. See comments in the code for details about these parameters.

Access Control: All services except updateUser require superuser privilege. Since a user may update his/herself (except for roles) that is also allowed by the back-end service.

The services follow. Red marks required args.

Update User Account

`/repository/admin/updateUser`

Method: POST
Args:
username=`text`---account name to change, only not required for create
`only_password=(boolean)`---do not process changes to user metadata, just update password
`password`---new password
`password_confirm`---must match new password
`old_password`---current (old) password, required for non-admin user
`first`---first name of person (metadata)
`last`---last name of person (metadata)
`mailbox`---email address of person
`role`---URI of role to add to user, may be multiple; when specified, this list of roles replaces the user's current roles.

If current user is not an Administrator, there are these restrictions:

- Cannot create new users
- Can only change your own user account
- Must supply old password to change password
- Cannot change roles

Update Role

`/repository/admin/updateRole`

Method: POST
Args:
`action= create | update | delete`---(keyword)
`uri=URI`---required except for create, identifies the role
`label= text`---immediate label of role
`comment= text`---lenghty description of the role

Update Grants

`/repository/admin/updateGrants`

Method: POST
Args:
`action`= add | remove---(keyword)
`uri` = `URI`---subject of the access grant
`access`= `URI`---access type to be added or removed
`agent` = URI of person or Role in the grant

Update Workflow Transition

/repository/admin/updateTransition

Method: POST
Args:
`action`= create | update | delete}}---(keyword)
`uri` = URI---required except for create, identifies the role
`uri=URI`--subject to release
`label`=text---required to create
`comment=text`---description
`initial`= URI of initial state (required to create)
`final`=URI of final state (required to create)
`tAction`=java class of transition action provider
`tActionParameter` = text or URI param to pass to action method
`order`= sorting key, usually a decimal integer

Update Named Graph

/repository/admin/updateNamedGraph

Method: POST
Args:
`uri`*=`URI` names the graph or context
`label=text`
`type`=URI of a NamedGraph Type

Interactive Web UI Pages

Although the repository is mostly accessed by other programs through the HTTP REST API, there are a few pages available to interactive users.

HTML Dissemination View of a Resource Instance

The dissemination servlet that makes resource instances resolvable also generates an HTML page for interactive users when content negotiation selects the text/html format. The HTML renders properties and the values of object properties as links to their URIs, so they can be resolved easily.

The page should also include an accurate **Last-Modified** date header, and respond appropriately to the **If-Modified-Since** HTTP header, which is used by Web proxies and search engine crawlers to update their caches.

See the eaglei.repository.instance.xslt configuration property for instructions on adding an XSLT transformation stage into the pipeline that delivers HTML content. (Sorry, we haven't washed away all of the Cocoon kool-aid yet..)

Administrative UI pages (/admin)

Visit this page for a directory of all administrative UI functions. Users without superuser privilege will only see a partial list. It also displays server-status and configuration details.

```
http://_hostname_/repository/admin
```

Although almost all repository administration is done by manipulating RDF statements in an internal administrative-metadata graph, there is a need for a Web-based UI to make it easier for site administrators to see and change the current state. Except as noted, these pages are only available to users with the **Administrator** role:

- Interactive **SPARQL query** workbench (all auththenticated users).
- List Named Graphs with the option to edit their descriptive metadata and ACLs.
    - Note that *new* Named Graphs are only added with the `/graph` service.
- List, add, delete, and modify access Roles - edit the metadata of each Role.
- List, add, and modify User login accounts:

    - Display and change membership in roles.
    - Edit user metadata and change password.
    - Disable a user from logging in at all, or reinstate disabled user.
- List, add, delete, and modify Workflow Transitions.
- Report versions of Data Model Ontology: loaded, and available.
- Get a list of Published Resources. This is handy for troubleshooting (all users).

Search Engine Control

- `/robots.txt`---direct robots to a site map, warn them away from other pages.
- `site maps`---HTML and XML site maps to direct crawlers to pages we want them to index.