Repository Workflow Design Notes

Contents

Requirements

The workflow system provides two important services to repository clients. These services are implemented *within* the repository and only exposed through the REST API. Hiding the implementation details is good modularity and security practice.

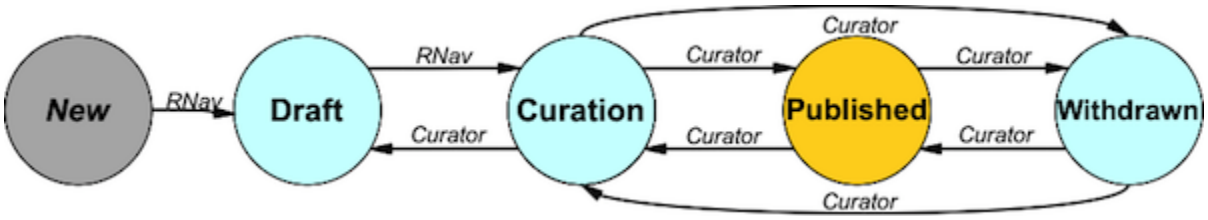
1. Mark and track each resource instance through a series of ***states*** in a certain order; this is the "flow" part of workflow.
2. Regulate when a user is allowed to gain ***write access*** to a resource instance and modify its properties.

These are the ***requirements*** driving the design:

- Remove the requirement for all users to have Administrator access to the repository:
  - Use Workflow transitions and claims to grant each *individual* user *write* access to an *individual* resource instance.
  - Rely on existing graph-based access control to manage *read* access to unpublished resources.
- Employ role-based access control to manage access to each workflow transition
- Allow separate, isolated, streams of workflow. For example, Lab A and Lab B can share a repository without being able even to see each other's unpublished resources.
- Produce user-sensitive reports on:
  - Transitions available to current user
  - Resources visible to current user with workflow implications
    (This *might* be used for auto-complete result sets)
- Highly configurable and extensible data-driven design, so some unanticipated needs can be met simply by changing internal data and configuration

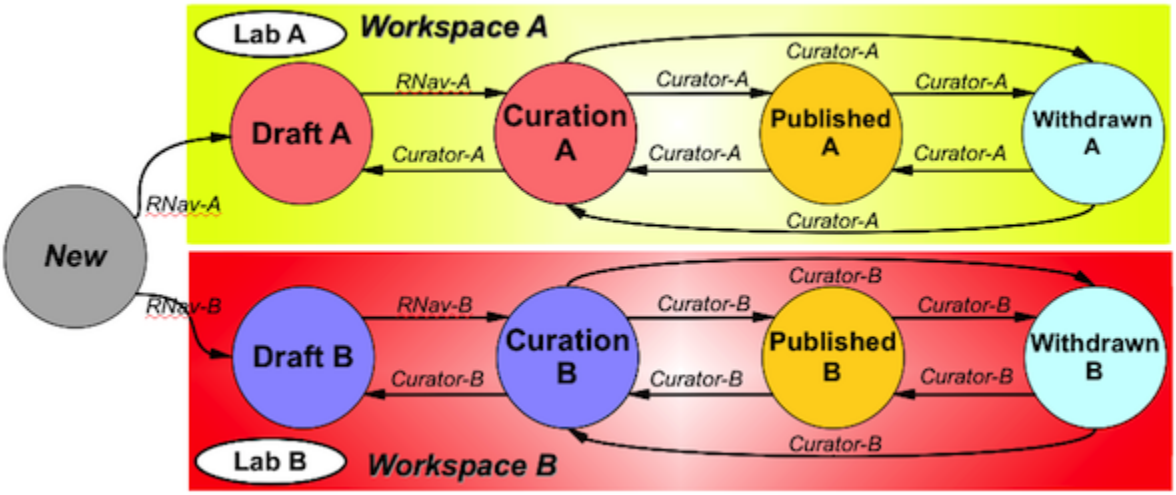
Use Cases

Here is the typical expected use of workflow. The blue states are *unpublished* (and thus not indexed nor visible to the public); only the gold state is published. Lines with arrows show the possible transitions, and the role(s) permitted to take them. The *New* state is implicit, it is where new instances come from.



Two Isolated Labs

The other proposed use case is a pair (or more) of labs which share a repository, but they must not be allowed to see each other's unpublished resources. In this picture, Lab A's resources are stored in separate workspaces - including a separate published graph and separate withdrawn graph - from Lab B's.



The labs also have separate resource-navigator and curator roles, RNav-A, etc. When a user in the RNav-A role creates a new instance, she must choose (or take by default, if it is the only one available) the transition out of ***New*** into the **Draft** state in Workspace A. Note that the states are labelled A and B but they are really the same states, it is the workspace or other graph which separates A from B.&

Presumably the repository's access controls are configured so only Lab A's personnel have read access to the **Workspace A** and **Withdrawn A** graphs. The two **Published** graphs must be world-readable, and of type *Published*, but they also have to be separate: Resources are not "owned" by any user or role. The workflow transitions available on a resource instance are governed by a combination of the *resource's home graph* and the user's role, so keeping Lab A's published resources in e.g. the **Published A** graph will let you configure workflow transitions on that graph so they are only available to Lab A's users.

Ontology and Design Principles

Resource Instance Properties

Workflow is expressed as two properties that *may* be asserted on resource instances. The statements using these predicates live on the *internal metadata* (`:NG_Metadata`) graph, so they are visible to the world but can only be modified by administrators and internally-mediated actions. The properties are:

- `:hasWorkflowState`---identifies the current state this instance is "in"; value is one of the WorkflowState object URIs
- `:hasWorkflowOwner`---names the repository user who is currently asserting a claim on the instance. Value is URI of that user account.

WorkflowState

This is the existing repository class; no ontological changes, it is essentially a controlled vocabulary with which to enumerate the possible states. Unlike fake workflow, the *state* does not grant access, it is transitions which have access controls. An instance has the following properties:

- `rdfs:label`
- `rdfs:comment`---more verbose description
- `:order`---literal used to determine sort order in visual presentation, required.

The set of States is fixed and relatively immutable. Normally a repository administrator never needs to add or remove states. They *can* always do so by uploading RDF with the `/graph` service, but it is not expected to be necessary. States live in the `NG_Internal` (internal metadata) graph.

We will add a state, named `New`", representing a resource instance in the process of being created. The state only exists momentarily inside the creation process to drive a workflow transition.

WorkflowTransition

This class represents a possible *transition* from an *initial* state to a *final* (relatively) in the workflow engine. Think of workflow as a graph, with states as nodes, the transitions are the edges. An instance has the following properties:

- `rdfs:label`
- `rdfs:comment`---more verbose description
- `:workspace`---URI of target workspace or wildcard URI if unlimited
- `:initial`---URI of WorkflowState from which this starts
- `:final`---URI of WorkflowState to where this ends
- `:action`---fully qualified class name of Java class implementing action
- `:actionParameter`---an optional single parameter which gets passed to action method, useful when an action implementation is shared by multiple transitions.
- `:order`---literal used to determine sort order in visual presentation, optional. Recommend using a typed literal, xsd:integer, and putting a group of transitions within the same 1000 range, e.g. 1010, 1020, 1030, etc with spacing to allow insertions.

- (implied) access control list - implemented by repository ACL mechanism, only users with READ access are allowed to take this transition.

Transitions are actively managed by the repository administrator through an admin UI page. They live in the `NG_Internal` (intenral metadata) graph. There is a sample set of transitions loaded at repository initialization but they are expected to be modified locally.

Actions

Actions are instances of a Java class implementing the `WorkflowAction` interface. Its `onTransition()` method gets called with the resource instance, and a transition-specific *parameter* (an RDF value object specific to the workflow transition, provided so the same action class can be re-used with different transitions). For example, a transition that moves a resource instance to a different named graph would use a common move-to-graph action with the destination graph as its parameter. Planned stock actions include:

- New instance in a designated graph
- Move instance to a new graph

About Access Control

Workflow ***manipulates*** the Repository's internal access control system to grant write (insert/delete) access to a resource when a user establishes a ***claim*** on it. Since this mechanism circumvents the established rules of access control, it is absolutely essential that it only grant write access in the correct circumstances. The repository administrator ought to be easily tell who can obtain write access to what resources by examining a small, manageable, amount of data.

The rules for granting write access are straightforward and fairly intuitive. To gain write access by asserting a claim on a resource instance, the user must:

1. Have ***read access*** to the instance (which seems obvious, but isn't necessarily..)
2. Have ***access to a workflow transition*** leading out of the instance's **current state**.
    - The transition's **workspace** must match the instance's home graph if it is workspace-limited.
    - The transition's ***in state*** matches the instance's current state.
    - Current user has **READ** access to the transition.

This is intuitive, if you consider one of the twin purposes of workflow is to ***progress*** instances from state to state. You must have access to the state-changing aspect of workflow to take advantage of its access-control circumvention.

For example, consider the simple workflow pattern developed for release 1.0: When a resource is in ***Draft*** state, RNavs are allowed to edit it. When an RNav pushes it to the ***Curation*** state, Curators can edit it (because they can push it to ***Published***, or back to **Draft**) but RNavs cannot, since they have no access to workflow transitions. Since the RNavs have turned it over to the Curators, the Curators *expect* that they are ***done*** editing it. It would be surprising to have an RNav edit an instance in **Curation**; so it is *not allowed*. If the instance needs more attention from an RNav, the curator ***signals*** that by sending it back to **Draft** state. At that point, the RNav can claim and edit it, but the curator cannot touch it until the RNav signals he is done by pushing it back to the **Curation** state.

For each user, workflow defines a ***pool*** of resource instances he/she can claim. Depending on the user's role and the configuration of the system, these may only appear in one state and one workspace, or many states and/or graphs.& For example, the typical curator has access to resources in the **Curation** , **Published**, and **Withdrawn** states.

QUESTION: What if the Claim operation is bound to a specific transition? The user would have to make the claim in the context of a transition, and we could then have a customizable method associated with the claim. This would let us configure, e.g., whether or not to make the instance writable. When claiming a Withdrawn or Published instance, for example, you might not want to allow it to be written until it goes back to Curation. Is this useful or necessary?

API

The REST API requests are:

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 tyep (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`---restirct results to resources in a given workspace; optional, default is all avaialble 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 modifiers 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.

Create New Resource `Instance/update?action=create`

The `/update` action that creates a new resource instance now includes some *implied* workflow behavior: There is an transition into the **New** workflow state. This means there must **be** a transition available to the current user and chosen destination workspace; if there are multiple transitions, one is chosen arbitrarily.

Note that `/update?action=create` *still* requires that the user have **Add** access to the workspace graph in which the instance is created. This might be an anachronism since workflow transitions bound to workspaces (provided there is no wildcard workspace) can *also* enforce access control. Also, there is also no way to create an instance as a proxy for another user.

Claim Resource `Instance/repository/workflow/claim`

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

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

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

There is no result document. The status indicates success with 200.

Side-effects:

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: Current user is the owner, ***or*** Administrator role.

Side effects:

1. ***Removes*** insert and delete access to the instance for the current user.
2. 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:

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

Administrative UI

The workflow mechanism also requires a couple of administrative UI pages, so admins can configure and examine the workflow system, and manage instances. All of these functions require the **Administrator** role.

Manage Workflow Transitions

This page lists all workflow transitions, along with enough salient details of each to select the one to edit. For example, label, initial/final states, and workspace. The master page also includes a link to create a new transaction (by filling in the detail page).

Each transition has a link to a "detail" page that lets the administrator examine and change all details of the transition. There is also a **delete** function.

Possibly add a link to export selected transitions (see next section).

Export and Import Transitions

Dump a serialized-RDF representation of the transitions which can be imported later in *any* repository. Options to select transitions, include access controls or not, etc. This may only appear as a REST-type HTTP transaction, not an interactive Web form.

Manage Claims

List resource claimed instances (restricted by workspace, user, etc) and allow the admin to ***release*** the claim, either individually or on a group. Includes sorting and pagination since results may be large.