SWEET Developers' Guide

Introduction and Main Data Objects

This is a developers' guide to the eagle-i data tools. It will focus primarily on the SWEET (Semantic Web Entry and Editing Tool, formerly, and confusingly, referred to as "the datatools webapp" or "datatools"). Several of the other user-level data manipulation tools (bulk data management, aka datamanagement; bulk data import, aka ETL; and the extraction of resources from published research articles and the like, aka nlp) will come up in passing. They each should have a guide, and this developers' guide will not attempt to cover them in any detail.

This guide begins with the back end of SWEET, most of which can be found in org.eagle-i.datatools (the eagle-i-datatools-common module). The backend components are held in common by all of the datatools; they are not specific to the SWEET. The `SweetServlet` (`org.eaglei.ui.gwt.sweet`; in the `datatools.sweet.gwt` package), in this context, is the user-facing endpoint for the data tools backend.

EIInstance

All of the data tools rely on two key abstractions: the `EIInstance` and the `EIInstanceMinimal`. (In reality, the `EIBasicInstance` is also important, but datatools only sees the `EIBasicInstance` through the `EIInstance`, so their attributes are conflated for the purposes of this discussion.) An `EIInstance` is the java-side representation of the collection of RDF statements (from the repository and from the ontology) about a particular subject (resource) in the repository.The `EIInstance` only contains representations of those RDF statements that are relevant to eagle-i users. `EIInstances` are used only for the resources captured in eagle-i.

For example, a user of eagle-i would *not* be an `EIInstance`; eagle-i has no interest in capturing (more than the minimum) information about its users. On the other hand, a DNA sequencer *would* be an `EIInstance`.

As mentioned, the EIInstance only represents RDF statements that eagle-i users need. So a DNA sequencer would have the type "DNA sequencer", but it would also have a type of "Instrument." In an `EIInstance`, it would *not* have the type "Continuant" or "Thing" even though there will certainly be an inferred or inferrable statement in the repository to that effect. What types can be retrieved for (and cached in) an `EIInstance` depend on annotations in the application ontology; if the type is annotated as `ClassGroup_DataModelCreate`, it's included in the type hierarchy of the `EIInstance`. Every `EIInstance` must also have a label and (of course) a URI; these are encapsulated in its `EIEntity`.

An `EIInstance` also contains all the other relevant statements, grouped into types of properties. Properties have two relevant dimensions: whether they come from the eagle-i ontology or not, and whether their values are references to other subjects in the repository (or ontology) or are complete in themselves. Bear in mind that (almost) all properties can legitimately have multiple distinct values for the same property, so the `EIInstance` keeps them as multimaps. Since the values are all distinct and have no intrinsic ordering, the `EIInstance` keeps the values of a particular property as a Set.

Ontology properties are always displayed in a resource page, either through search or through the SWEET webapps. Non-ontology properties are extra, either because of ontology changes or because they are managed by the repository. Only the datatools applications bother to load the non-ontology properties.

- Datatype properties are the properties with values that are complete in themselves. They are represented as a `Map<EIEntity, Set<String>>` Boolean properties, date properties, and of course text properties fall into this category.

- Object properties are the properties with values that refer to other subjects in the repository or ontology. They are represented as `Map<EIEntity, Set<EIEntity>>`. In practice, the SWEET applications need to be able to distinguish between object properties that refer to terms from the ontology and ones that refer to other instances in the repository. Doing so requires a separate call to the server.

- Non-ontology datatype properties are datatype properties that don't appear in the eagle-i ontology. Many of the so-called "provenance metadata" properties added by the repository itself (creation date, last modified date, contributor) are non-ontology datatype properties. The `is_stub` property is another, as are the standard `note` and `curator note`. In addition, any datatype properties that are associated with an instance but are no longer relevant to the instance's type (either because of a change to the ontology or because the user changed its type) will appear here.

- Non-ontology object properties are the object properties that don't appear in the eagle-i ontology. The remaining "provenance metadata" properties, like workflow state and workflow owner, are non-ontology object properties.

EIInstanceMinimal

The `EIInstanceMinimal` is the core representation for listing resources in the webapps (both SWEET and search front ends). As the name suggests, it contains only the minimal information required to list the relevant instances.This includes:

- The label and URI
- The type
- The resource-providing organization (lab, center, ...) that contributed this resource
- All the supertypes up to the eagle-i base type (for filtering)
- Workflow state and owner
- Creation and modification dates
- Whether or not the resource is a "stub"

Datatools Back End

Configuration

Datatools backends (particularly for SWEET and ETL) need to know the URL of the repository to point to. Because eagle-i applications use `https`, it's not possible to point to `localhost`. Instead, the repository location is specified in a configuration file.An example file is found in `eagle-i/examples`.

Datatools relies on the `classloader` to find this property file.Developers and deployers need to put that file on an appropriate location in order for the SWEET webapp to find it. In the reference implementation, the file is found in `catalina.home/eaglei/conf`. `catalina.home/conf/catalina.properties` has been modified to contain these lines:

```
common.loader=${catalina.base}/eaglei/conf,${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.home}
/lib,${catalina.home}/lib/*.jarorg.eaglei.home=/opt/tomcat6/eaglei/
```

Datatools web applications will look in the loader's classpath to find the relevant configuration file.

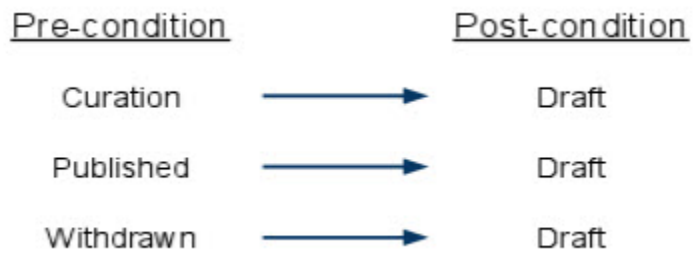
Workflow and access control

The repository provides a mechanism for controlling which users can edit which resources. Details can be found in the Repository Workflow Design Guide. It has four core notions: ownership, workflow state, transitions between states, and roles assigned to users (or types of users), which determine which transitions are legal for each user type. All workflow privileges are based on the URI of the user in the repository; therefore, all datatools operations (listing resources, editing them, making workflow transitions, etc.) require a repository username and password.

Workflow states are configurable in the repository; the default set is:

- New
- Draft
- In Curation
- Published (in the Published graph, which makes the resource accessible to search)
- Withdrawn (in the Withdrawn graph, which makes the resource inaccessible to search and marks it as being no longer valid for one reason or another)

Transitions are *also* configurable, and are specified by a URI and label. Each has a precondition: the workflow state the resource must be in for this transition to succeed, and a postcondition: the workflow state this transition will put the resource in when it succeeds. As a result, there are 3 separate "Return to Draft" transitions:



A list of transitions can be found at `url` of a machine with an eagle-i `repository]/repository/workflow/transitions`.

The repository returns only a list of the transitions that are allowed for the current user, based on the user's role in the repository. (User roles are also configurable in the repository, and again, the repository has a default set of roles with permissions. These are largely beyond the scope of this document.) For historical reasons, the transition query returns with a boolean indicating whether it is legal for the current user; in practice, a user will only see transitions they're allowed to perform.

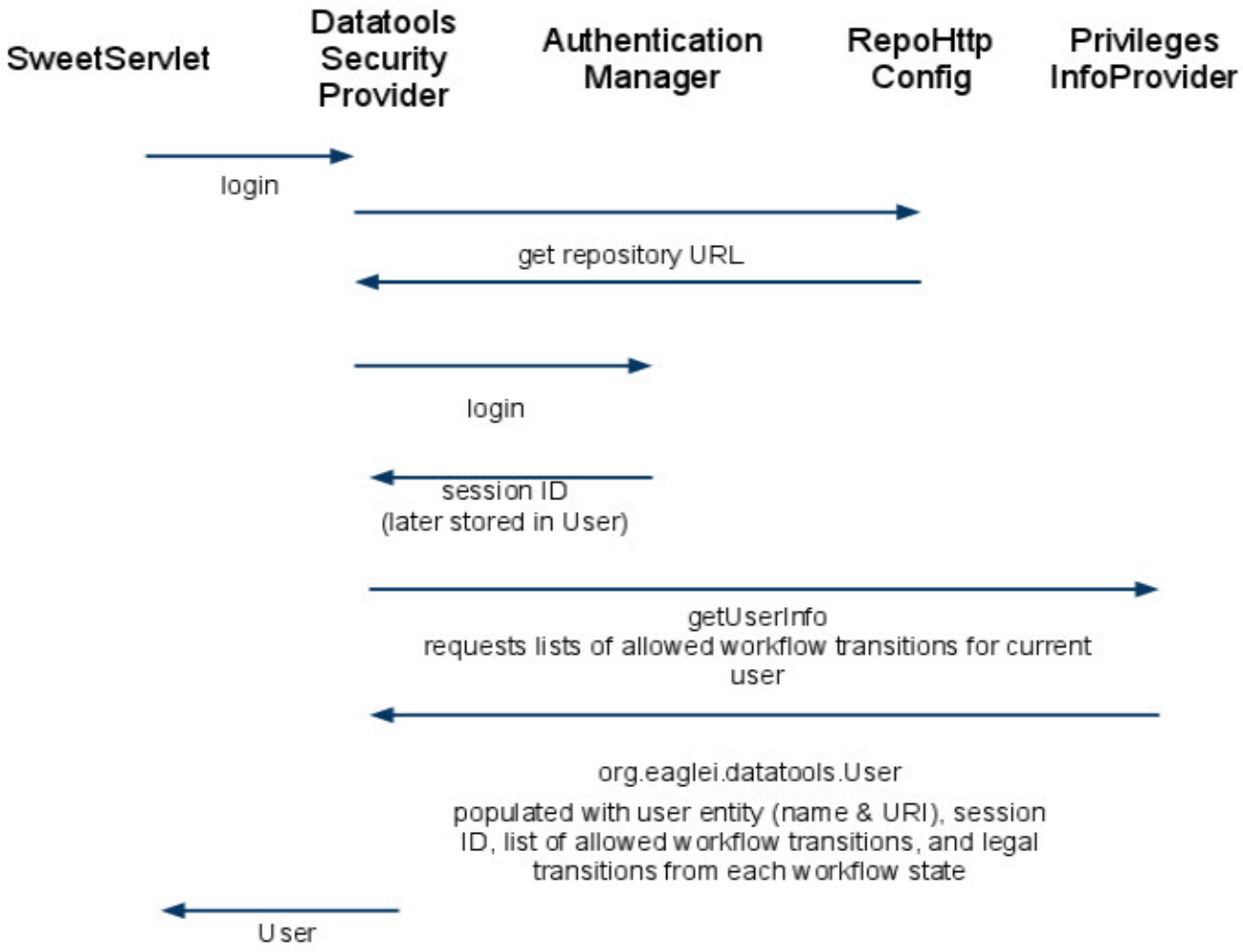
In the SWEET, these transitions are represented by `org.eaglei.ui.gwt.WorkFlowTransition`. These transitions [will be] loaded from the repository, as are the states. The `PrivilegesInfoProvider` is responsible for this loading and managing.

In order to prevent conflicting edits, each resource can only be edited by a single "owner". In order to edit a resource, a user must first claim the resource. The user is then set as the resource's workflow owner in the repository. No one else can claim the resource until it has been shared, either explicitly via the "share" button, or by a workflow transition. All workflow transitions clear existing ownership.

Security and logging in

Obviously, then, datatools requires a user-specific login to the repository, while search can make do with a single generic user with no privileges (and access only to the Published graph). Furthermore, datatools needs to retrieve the user's valid transitions in order to be able to present their options correctly. Both applications, though, need to have logged into the repository in some fashion, and both need to keep track of a user's activity and get rid of connections when a user has been inactive for too long. The `AuthenticationManager` (and `AuthenticationProviders`) in `org.eaglei.services.authentication` have the job of handling logging in and stale sessions. Because the datatools operations all go to the eagle-i repository, all SWEET (and other datatools) connections use the `StandardAuthenticationProvider` and the `Apache4xHttpConnectionProvider`.

Below is an interaction diagram for a SWEET login.



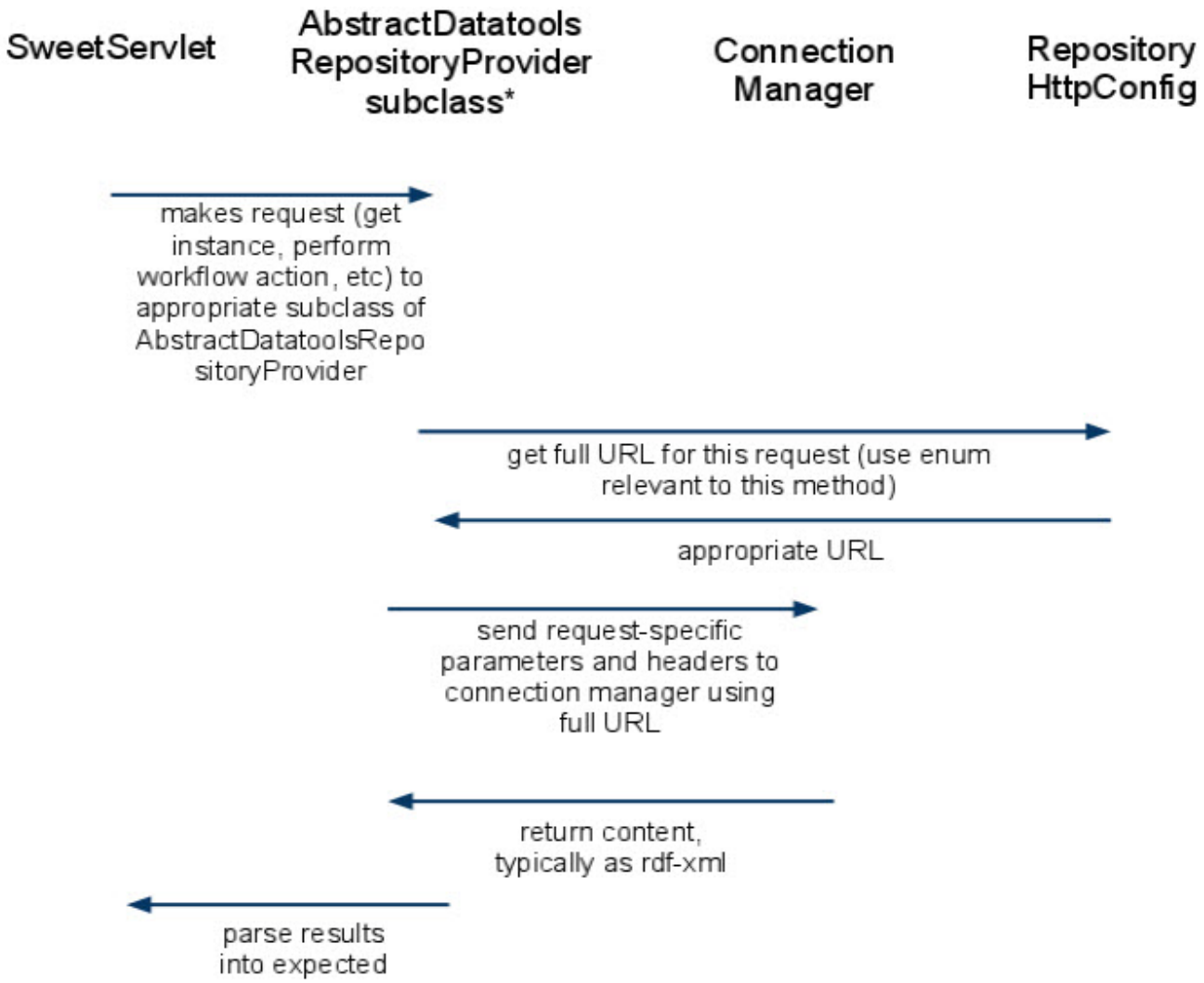
A login request goes to the `DatatoolsSecurityProvider`, which logs in through the `AuthenticationManager`, which has been configured to use a `RepositoryAuthenticationProvider` (and therefore present credentials to the eagle-i repository specified by the `DatatoolsConfiguration`). The `DatatoolsSecurityProvider` then requests a User from the `PrivilegesInfoProvider`, using the sessionId from the `AuthenticationManager` login. The `PrivilegesInfoProvider` requests information about the current user, including the workflow transitions (if any) this user is authorized to perform. It populates a User object with that information (which is also parceled out into a map from workflow state to the list of allowed transitions, to facilitate determining what if any actions a user is allowed to perform).

Getting ontology information

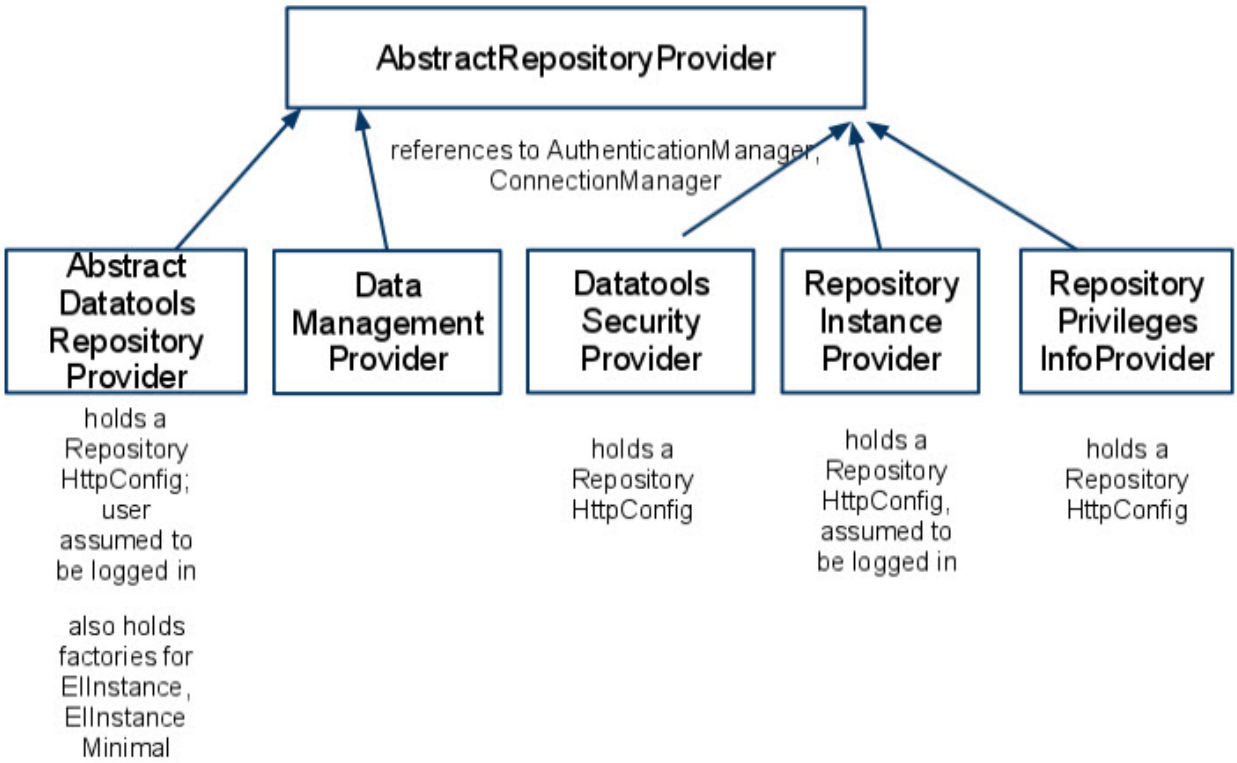
Many operations require information from the application and domain ontologies. Making that information available is the role of the `JenaEIOntModel` (`eagle-i-model-jena: org.eaglei.model.jena`) at the back end. The front-end equivalent is the `ModelServlet` (`eagle-i-common-ui-model-gwt: org.eaglei.model.gwt.server`). The details of the servlet and the `JenaEIOntModel` are beyond the scope of this document. For the purposes of datatools and SWEET, all information about the ontologies is encapsulated in these classes.

Getting data of various kinds (`AbstractRepositoryProvider` and its subclasses)

In order to allow a user to enter and edit data, SWEET must retrieve data in various forms from the ontology and repository. The relevant interfaces are found in `org.eaglei.datatools.provider`, and implementations in `org.eaglei.datatools.jena`.



These requests may also be made to the `org.eaglei.services.repository. RepositoryInstanceProvider`, which is NOT a subclass of `AbstractDatatoolsRepositoryProvider`. Its behavior is identical for the purposes of this interaction diagram.



`AbstractDatatoolsRepositoryProvider` and its subclasses make a few assumptions: there's a known `RepositoryHttpConfig`, and the user has logged in to the repository, so that the `ConnectionManager` can provide connections from the user's session ID.

Subclasses of `AbstractDatatoolsRepositoryProvider` include:

- `RepositoryCrudProvider`: implements `CrudProvider`. Performs such functions as getting a new instance ID from the repository, creating /updating/deleting an instance in the repository, making a new instance with the same properties as an existing instance except for the label (deep copy), etc. It does *not* have the ability to fetch an instance from the repository: that's the job of the `RepositoryInstanceProvider`. (Creation and deletion can happen using only the type or URL, respectively; update requires having fetched the instance first.)
- `RepositoryListResourcesProvider`: implements `ListResourceProvider`. Allows several ways of listing resources. These methods all return lists of `EIInstanceMinimal`.
- `RepositoryQueryProvider`: implements `QueryProvider`. Allows for arbitrary SPARQL queries to the repository.
- `RepositoryWorkflowProvide`: implements {{ WorkflowProvider}}. Allows user to claim or share resources, or to request transitions from the current workflow state to another.

The datatools backend shares instance retrieval functionality with the search applications. The `RepositoryInstanceProvider` (`org.eaglei.services.repository`) performs this function.

SWEET Front End

The SWEET is built in GWT. Documentation for GWT can be found at http://code.google.com/webtoolkit this guide assumes some familiarity with GWT.

Servlet

The `SweetServlet` is a thin wrapper around a collection of `AbstractRepositoryProviders` as described above. Each call to the servlet checks for a valid `sessionId`, then dispatches the call to the appropriate provider.

RPC

As usual for a GWT application, much of the `org.eaglei.ui.gwt.sweet.rpc` package is taken up by definitions of the services and their asynchronous counterparts. `ClientSecurityProxy` and `ClientSweetProxy` are different, and important. The `ClientSecurityProxy` encapsulates authentication and session-related behavior for the `ClientSweetProxy`. A number of front-end classes register as listeners for changes to sessions; the `ClientSecurityProxy` is responsible for notifying them when a session becomes valid or invalid. Similarly, the `ClientSecurityProxy` detects when a user is not authorized to access the SWEET webapp (currently, if that user is not permitted to create resources).

`ClientSweetProxy` is a single point for all of the UI classes to talk to the backend as needed. In addition to the `SweetServlet`, the `ClientSweetProxy` talks to a `ModelService` in order to fetch ontology information that certain front-end operations require. In a few cases, the `ClientSweetProxy` makes multiple server calls for a single user operation, in order to be sure to have the most up-to-date data. Examples include claiming (where first the proxy verifies that the resource is not out of date), and sharing, which re-fetches the instance after a successful share. For now, creating a new instance forces the workflow state to Draft, so that the instance has a valid workflow state; the alternative is to re-fetch the instance (or instances, when creating a resource also creates stub resources).

ApplicationState and how the front end redraws

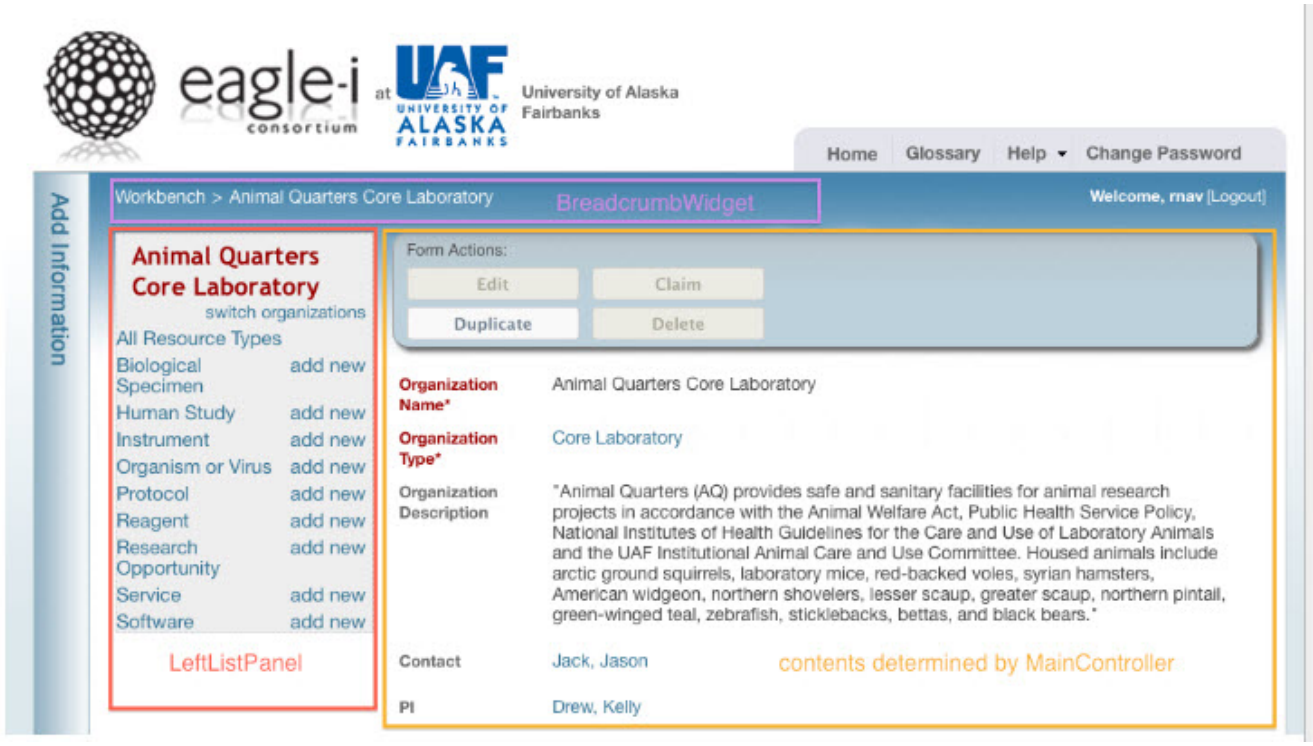
`ApplicationState` object is a central (singleton) location for various general bits of information the SWEET webapp needs. It tracks several selections a user has (or has not) made, and allows the UI to fetch and draw the correct information. It holds a list of resource-providing organizations fetched from the repository, as well as a cache of the `EIClasses` that are known to be embedded and those that are associated with labs (and other resource-creating organizations) and those visible for overall browsing. It's also where the client caches class definitions for use in tooltips.

The core of `ApplicationState's` behavior is in the `QueryTokenObject`.Through the `QueryTokenObject`, the `ApplicationState` handles history navigation, browser refreshes, session timeouts, and bookmark sharing. The `QueryTokenObject` converts between a #history url string and specific `EIEntities` (and a few other settings) to determine what will be drawn. The `QueryTokenObject` handles certain rules (whenever you start showing lists of resources, reset the pagination to the default), and maintains:

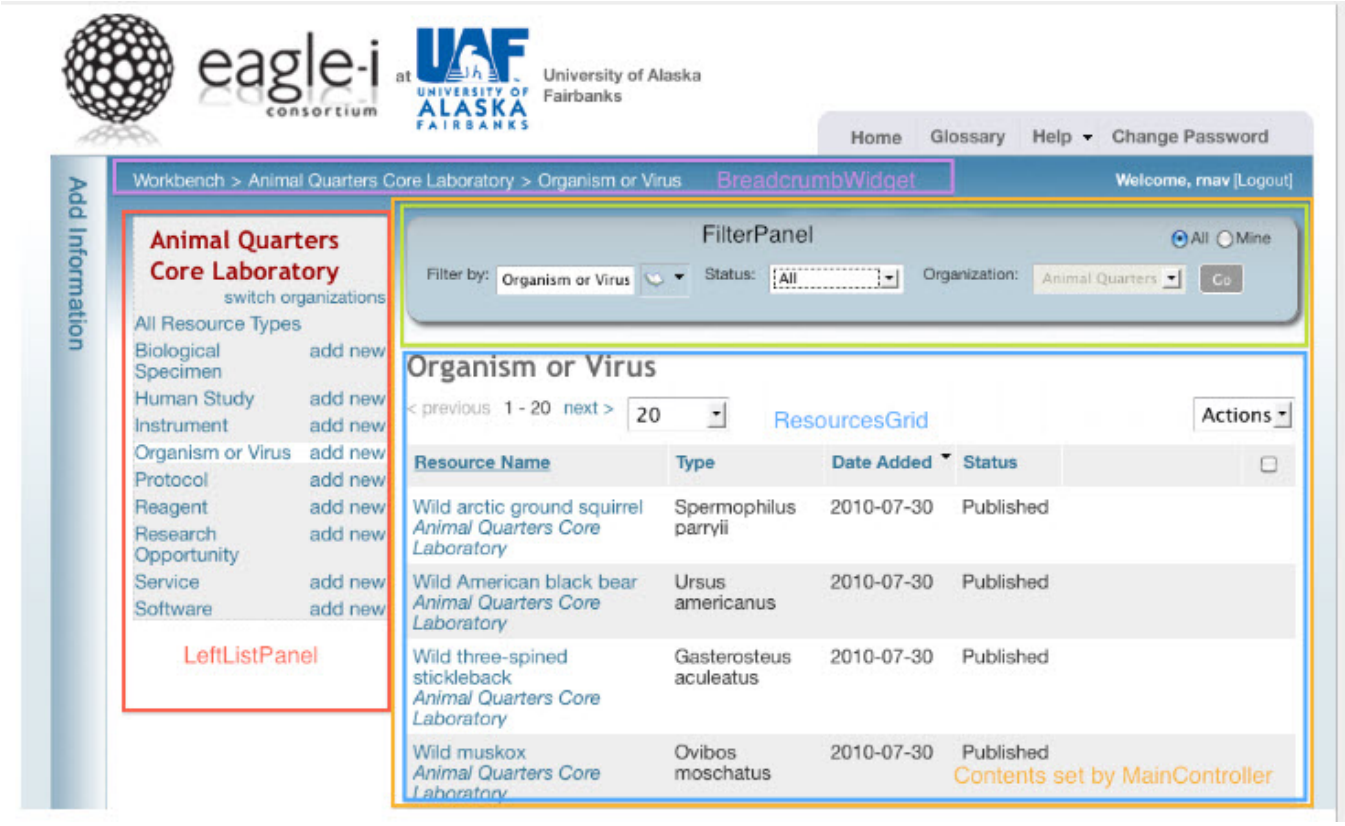
- some entities for use by the other front-end classes (type entity, instance entity)
- a map from keys to values for going back and forth to the url history string ("mode", "typeUri", etc). The entities must be built out of two entries in the map; for each entity, uri and label are stored separately.

The `ApplicationState` maintains a list of `ApplicationStateChangeListeners`; whenever it gets an update that should change the history, it updates the `QueryTokenObject`, then writes `QTO.toString()` into the GWT history mechanism and notifies its listeners. Bookmarks and browser refreshes work by first parsing the #history component of the url into the `ApplicationState's QueryTokenObject`, and then treating it as an `ApplicationStateChange` event.

Single-resource (in this case lab) view:



Listing a particular type of resources for a specific lab:

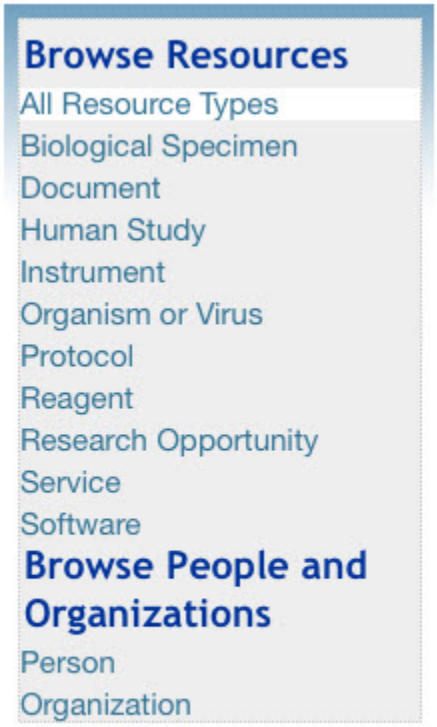


Three classes implement `ApplicationStateChangeListener`: the `MainController`, the `LeftListPanel, and the {{BreadcrumbWidget`.

BreadCrumbWidget
The `BreadcrumbWidget` displays a trail of resource provider (if any) and resource type selected (if any), with an initial link back to the workbench always.

LeftListPanel
The `LeftListPanel` is responsible for displaying the current resource providing organization (lab, division, etc), and a list of available resource types.The `LeftListPanel` has two modes: when a lab or other resource provider is selected, and when the user is instead browsing people and resources. Its two modes are shown below.

Lab (or other resource provider) selected



Browsing people and resources

When the user has selected a lab (above left), the selection is stored in the `ApplicationState`.The `LeftListPanel` then retrieves the selected lab from the `ApplicationState`, and displays the top-level types from the `ApplicationState`'s `resourceTypesForProvider` list. When there's no lab selected (above right), the `ApplicationState` instead uses `resourceTypesForBrowse`.In either case, if the `ApplicationState`'s `typeEntity` is populated, the `LeftListPanel` highlights the selected resource type (Protocol in above left); otherwise, it selects the "All Resource Types" entry{ (above right)

MainController

The `MainController` uses the `Mode` (an enum from `QueryTokenObject`) from the `ApplicationState` to determine what belongs in the main area of the window. Whenever the application state updates, the `MainController` checks first for a valid user (if there's none, it clears everything), and then checks the mode.

The possible modes are broken down into three sub-groups: workbench, editing/viewing single resources and listing resources.
The workbench group contains only one mode that shows a landing page (Workbench).

- WORKBENCH: shows a landing page with a number of standard options. For other application domains, it will probably make sense to create a new Workbench class for this mode to invoke.

Editing/Viewing single resources contains three modes:

- *EDIT*: edit a single resource; show the edit form (including all possible properties as supplied by the domain ontology)
- *VIEW*: shows all the properties of a single resource, but only those that have been annotated by a user. In other words, all the *actual* properties (including the properties not shown in search) but not the *potential properties shown in *EDIT* mode.

- *DUPLICATE*: placeholder mode; application state change should *not* be invoked for this mode. Show an edit form with a new instance, populated from the values of an existing instance. Clears the label field, to force users to add a new label.

Listing resources contains seven modes, which are differentiated based on having certain criteria met:

- *LIST*: list all resources meeting the criteria (including the filter criteria) from the `ApplicationState`. If a `typeEntity` exists, show only resources with that as a base type (otherwise, show all resources). If a `providerEntity` exists, show only resources belonging to that lab (or other resource provider). The `ApplicationState` is set to this mode by the `LeftListGridRowWidget`, or after a resource has been deleted.
- *FILTER*: the `ApplicationState` mode used when the user clicks the "go" button in the `FilterPanel`. Shows only resources meeting the description from the `ApplicationState`, further refined by the `FilterPanel` options (resources belonging to a subclass, only resources in Draft mode, etc).
- *RESOURCES*: a (hackish) way to get an empty resources grid shown. In highly-populated repositories, trying to show all resources of all types from all labs is prohibitively slow. Resources mode short-circuits the process.
- *REFERENCES*: List all the resources that refer to the `instanceEntity` from the `ApplicationState`. Resource A refers to resource B if resource B is the value of a some property on resource A. (In rdf-speak, we're looking for all the subjects A where B is the predicate for some statement about A, and A is an instance of one of the types we consider as eagle-i resources.)
- *STUBS*: List all the resources that were created through a "create new" mechanism in the edit form.
- *PROVIDERS*: Shows a list of all organizations to which resources can or should be added. [Note: currently broken--shows all organizations]
- MYRESOURCES: Show exactly the resources for which the current user is the editor/has claimed the resource.

For any of the modes that involve listing resources, the `MainController` clears its panel, fetches any relevant resources in the form of a list of `EIInstanceMinimal`, and draws a `ResourcesGrid`. For the edit, view, and duplicate modes, it clears its panel and invokes the `FormsPanelFactory` to fetch the appropriate resource (and possibly ontology model information) and draw the page.

ResourcesGrid

For any mode in the listing resources group, the `MainController` retrieves a list of `EIInstanceMinimals`, and then draws a `ResourcesGrid`. The `ResourcesGrid` is responsible for drawing lists of resources, of whatever sort. The `ResourcesGrid` has a `FilterPanel` at the top, which allows filtering the list by any combination of subtype, current owner (claimed by user, or all), and workflow state. It also has two `PaginationWidgets` (one at the top and one at the bottom), which determine how many resources to show per page and which page to view.Note that no total counts are available (and page sizes may be slightly off) because a single resource may be returned multiple times from the repository, but is filtered out at the front end.

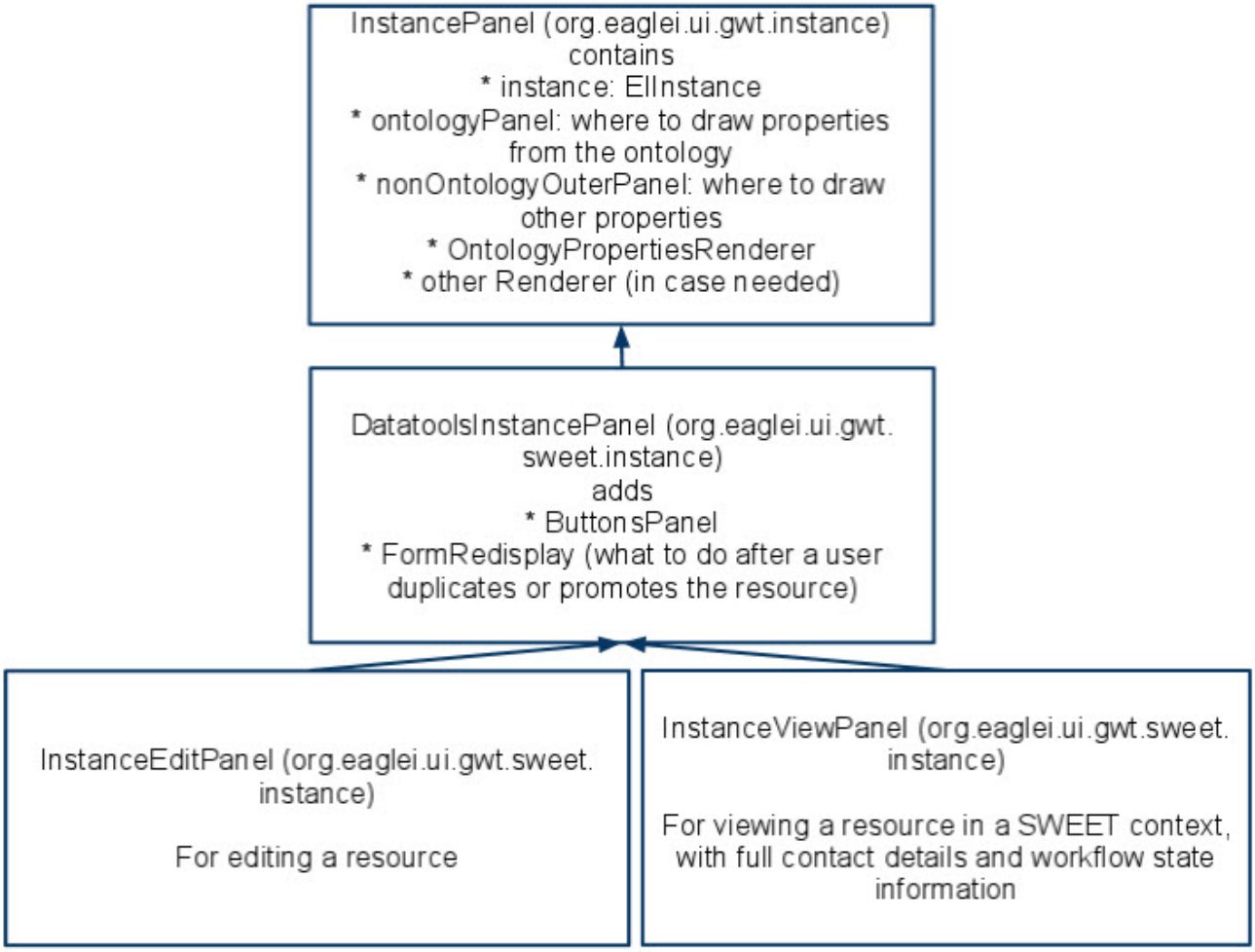
The `ResourcesGrid` displays a list of `GridRowWidget`, each of which wraps an `EIInstanceMinimal`.The `GridRowWidget` displays relevant information from the `EIInstanceMinimal`, and provides links to allow the user to claim or share the resource (if it's in a workflow state that the user can change), and to edit or delete it once it's been claimed. The `GridRowWidget` also has a checkbox which interacts with the `ResourcesGrid`'s actions drop-down to allow bulk workflow transitions.

Displaying and editing single resources

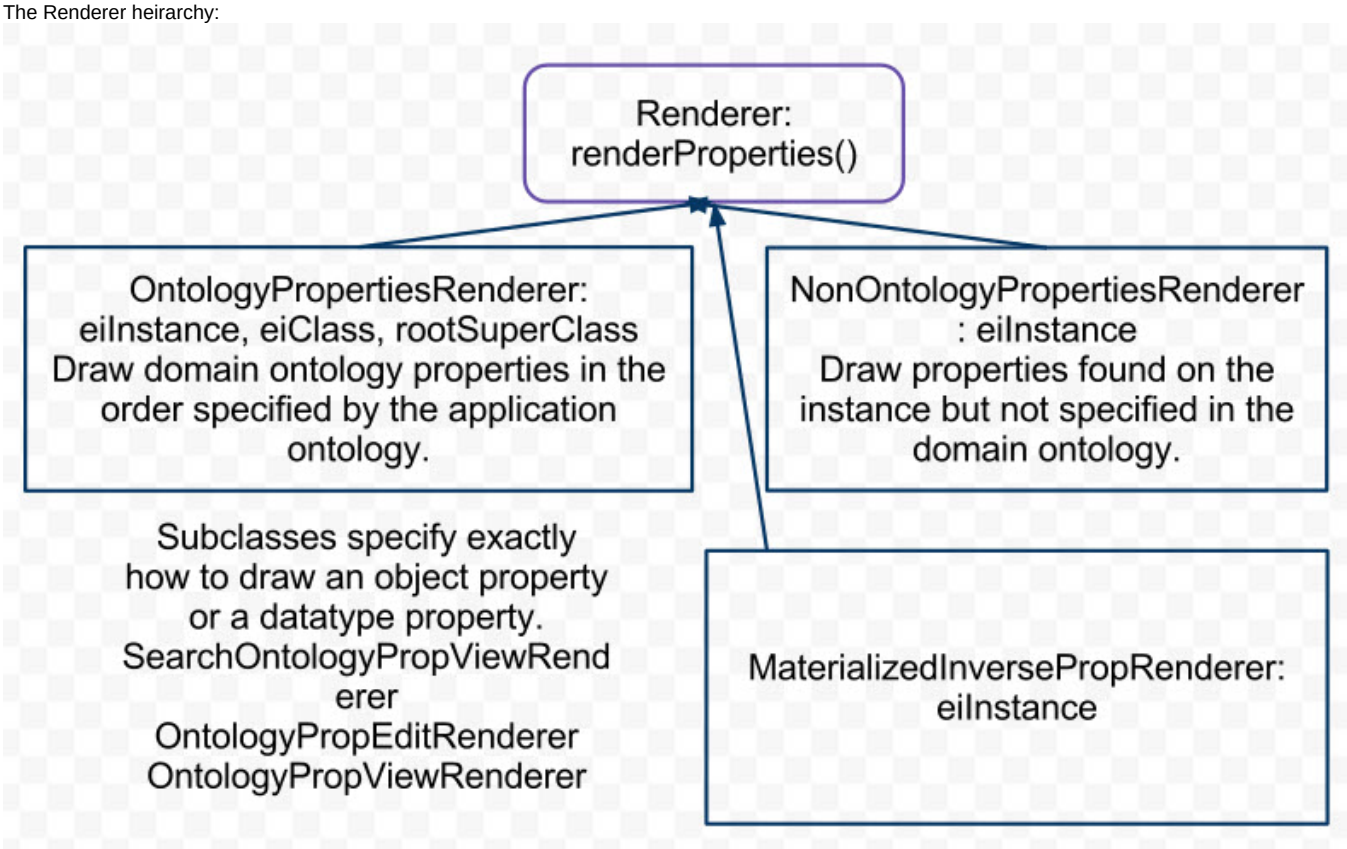
Panels for editing and viewing single resources are constructed by the `FormsPanelFactory`, which is responsible for fetching the instance (or getting a new, empty instance) and constructing a `DatatoolsInstancePanel` to display it.

Much of the instance display logic is shared with the eagle-i search applications, so the base Renderer interface, the abstract `OntologyPropertiesRenderer`, and the abstract `InstancePanel` are all found in `org.eaglei.ui.gwt.instance`.

The InstancePanel heirarchy:



The Renderer heirarchy:



The job of an `OntologyPropertiesRenderer` is to draw the properties specified for the instance by its type in the domain ontology. It draws them in an order specified by the application ontology. Specific subclasses are responsible for picking exactly how to draw the properties--as labels, as links, or as widgets for editing.

The `NonOntologyPropertiesRenderer` draws properties found on the given instance but not specified in the domain ontology. Important examples are the workflow owner and workflow state, creation and modification dates, and the like. The eagle-i application ontology also specifies comments and curator notes for each resource; these are also non-ontology properties.

The `OntologyPropViewRenderer` is the ontology renderer for viewing resources in a datatools context. It creates a `LabelValuesWidget` for each property that actually has a value on the instance.

The `OntologyPropEditRenderer` is the ontology renderer for editing resources in a datatools context. Since it needs to present all possible properties, it loads the properties from the `EIOntModel` as well as the values that currently exist on the instance (if any). The `OntologyPropEditRenderer` draws widgets that are subclasses of `EditWidget` (usually wrapped in `EditWidgetCollections`).

EditWidget and EditWidgetCollection

As previously mentioned, the values for a given property of an `EIInstance` is (generally) a set of values (collection that is unordered and for which each value must be unique). As a result, when the user changes a value using an edit form, there needs to be a mechanism for tracking which value is being replaced.The `EditWidget` hierarchy performs that function.

The base `EditWidget` class holds a reference to the `EIInstance` it is editing, the `EIProperty` for which it is the widget, and, crucially, an `oldValue` field. Since properties can be either objects (`EIURI}}s) or data types (strings, booleans, dates, etc), the {{oldValue` is a string; if the property is an object property, the subclass must use `getOldEIURI` value to get the correct `oldValue`.

Whenever its value field changes, the subclass is responsible for removing the old value (if any) from the `EIInstance` and setting `oldValue` to the current value. Ideally, `EditWidget` would be responsible for this behavior in just one place, but only the subclass has the context to determine if the old value will be found in ontology properties or non-ontology properties, object or datatype. Thus, the first abstract method of `EditWidget` is `removeValue`.

The other abstract method of `EditWidget` is `duplicateBlank`, which is needed by the `EditWidgetCollection`.

The SWEET handles multiple values for a property through the `EditWidgetCollection`. The `EditWidgetCollection` holds a list of `EditWidgets`, plus the `EIInstance` and the `EIProperty` that all the widgets in its collection edit. The `EditWidgetCollection` is responsible for adding a new `Edit Widget` of the same type as the previous widget in its list, using the `duplicateBlank` method on that `EditWidget`.It can also remove a value (the '-' link) by calling the relevant `EditWidget`'s `removeValue` method.Again, because the values of an `EIInstance`'s property are a set, if two `EditWidgets` in the same `EditWidgetCollection` are ever set to the same value, changing or removing one will change or remove the value from the `EIInstance` entirely. A desirable extension to `EditWidgets` or `EditWidgetCollection` would be to prevent a user from ever selecting duplicate values for the same property.

The subclasses of `EditWidget` include:

- `TextWidget`: a simple widget for datatype values; property value goes in a text box
- `TextAreaWidget`: a widget for longer datatype values; property value goes in a text area
- `TermWidget`: a widget for displaying and selecting domain ontology terms; values are constrained to valid values from the domain ontology (usually subtypes of a specified type)
- `ResourceListWidget`: a selection widget for legal values from the repository. Also allows a "create new" option. Selecting the "create new" option inserts a StubWidget under the `ResourceListWidget`. By default, a `ResourceListWidget` is populated only with resources of the appropriate type from the current lab; the "See choices from all organizations" link queries for all the resources of that type in the repository.
- `StubWidget`: a widget with label and type fields. Only drawn when the user has selected "create new" from a `ResourceListWidget`. Creates a new stub instance with the specified label and type; all stubs are saved as part of saving the main instance. (First the stubs are saved; once the first save has succeeded, the main instance is saved.)
- `ObjectWidget`: a complex widget to allow the user to select a value properly among several allowed ranges (essentially, types).See below.
- `EmbeddedResourceEditWidget`: a widget to draw any embedded instances in the current `EIInstance`.

The `ObjectWidget` handles properties that have multiple valid ranges for their values. For example, the Manufacturer property of an Instrument can be either a Person or an Organization; in order to see the correct values, the user must first select which. Then the `ObjectWidget` adds a `ResourceListWidget` populated with the instances of that type.

A more complex case is the Topic of a Protocol. It can be an Organism or Virus, a Disease, or a Biological Process. The Biological Process and Disease ranges are both types from the domain ontology; when the user selects one of them, they need to see a `TermWidget`. An Organism or Virus range, however, is an instance range--the user needs to see a `ResourceListWidget` with all the relevant organisms and viruses.