JARS: Just Another Repository for Scholars

Software Requirements Specification

v1.0-rc

9 October, 2014

Revision History

Date	Description	Author	Comments
2014-10-08	v1.0-rc	Erick Peirson	
2014-10-15	v1.0-rc2	Erick Peirson	

Table of Contents

1. Introduction

This document describes the functional requirements (as they are currently understood) for a repository web application called *JARS: Just Another Repository for Scholars*. *JARS* is intended to provide a minimalistic repository solution to support text-analysis projects in digital humanities, especially those involving the Quadriga-Vogon text annotation platform. *JARS* is part of a larger effort to develop a federated system of scholarly repositories at Arizona State University.

1.1 Purpose

The purpose of this Software Requirement Specification is to clarify for all stakeholders the functionality and constraints of the *JARS* software package as it is presently conceived. This document will be the primary reference point for subsequent documents, stories, and releases related to *JARS*.

1.2 Scope

JARS: Just Another Repository for Scholars is a web application intended to provide online storage and structured access to digitized scholarly resources, especially (but not only) texts.

1.3 Definitions, Acronyms, and Abbreviations

API -- Application Programming Interface -- An intermediary that allows applications to interact with each other.

HTTPS - Secure Hyptertext Transfer Protocol -- When HTTP (Hypertext Transfer Protocol) is layered on top of SSL.

MySQL -- An open-source database application (see http://www.mysql.com).

REST API -- REpresentational State Transfer API -- A widely-adopted architecture for web APIs, emphasizing the separation of clients and servers, stateless communication, and uniform interfaces.

SSL -- Secure Socket Layer -- A protocol for securing messages sent over the internet.

URI - Uniform Resource Identifier -- A string of characters used to globally and unambiguously refer to a resource. An URI may or may not also be a URL.

URL - Uniform Resource Locator -- A string of characters that unambiguously describes the location of a resource on the internet.

WSGI -- Web Server Gateway Interface -- A specification that describes how web applications and web servers communicate with each other. Described in PEP 3333 (ref 1.4.3).

1.4 References

1.4.1 ASU Information Security Policy -- https://uto.asu.edu/node/7118

1.4.2 ASU Secure Web Development Standard -- https://uto.asu.edu/node/7099

1.4.3 PEP 3333 Python Web Server Gateway Interface v1.0.1 -- http://legacy.python.org/dev/peps/pep-3333/

1.4.4 JSON API Standard -- http://jsonapi.org/format/

1.5 Stakeholders

The following individuals and organizations have an interest of some form in *JARS* and/or the development process. These individuals should be a party to revisions of this document, as described in section *4. Change Management Process*. This list is subject to change.

(1) Manfred Laubichler (he's paying for it);

(2) Students in the Laubichler Lab at ASU (will use it);

(3) Michael Simeone (IHR Nexus Lab, coordinating digital humanities stuff at ASU);

(4) Stephen Savage (responsible for ASU repository infrastructure);

(5) Julia Damerow (consultant, A Place Called Up Consulting).

1.6 Versioning

Subsequent versions of this document will be labeled using something approximating Semantic Versioning:

- A Major version will be incremented when requirements change in a way that reverses or is in some way incompatible with requirements in earlier versions;
- A Minor version will be incremented when requirements are added that are compatible with requirements in the earlier version;
- A Patch version will be incremented when minor corrections are made (e.g. typos, incorrect references, unclear language) that do not impact the substance of the requirements.

The suffix '-rc' indicates that a version is a "release candidate." See also *4. Change Management Process*.

Version numbers will be included in document filenames. All revisions will be stored in the docs section of the *JARS* GitHub repository.

1.7 Overview

The body of this document is comprised of two main sections. Section *2. General Description* addresses the context for *JARS*, its main functions, and its intended audience. Section *3. Specific Requirements* addresses the specific functionalities and constraints of the *JARS* software as they are presently understood. Section *4. Change Management Process* describes how this document should be revised as requirements change.

2. General Description

This section describes the context for *JARS*, its main functions, and its intended audience.

2.1 Product Perspective

Text analysis projects in digital humanities increasingly involve the consumption of online resources, such as authority services and linked data. As those projects move beyond the confines of the individual scholar's computer, it is increasingly important to store and share resources in ways that promote sharing and interoperability. Digital repositories promote sharing by making resources discoverable and available over the internet, and they promote interoperability by providing stable identifiers for resources, and by supporting widely-used metadata standards that can be used to describe resources.

Most digital repository solutions are designed to support large digital archives, involving disparate data types, complex workflows, and elaborate presentation (e.g. DuraSpace, Fedora, Hydra). One disadvantage of these solutions is that their complexity can make them expensive to customize and maintain. Many text analysis projects do not require the extensive features that those solutions provide and, in some cases, may depend on features that are not present. For example, DuraSpace does not yet provide a functional REST API, a feature that is essential to text-annotation projects that use the Quadriga-Vogon platform.

The main objective of *JARS*, therefore, is to provide a minimalistic repository solution specifically tailored to the requirements of scholarly text-analysis projects. Those requirements can be summarized as follows:

(1) Store digital resources, especially digitized texts, along with standards-compliant descriptive metadata;

(2) Provide stable unique identifiers (e.g. handles) for all resources;

(3) Provide systematic search and retrieval of resources via a REST API;

(4) Allow curators to control access to resources (e.g. to comply with copyright laws).

Unlike other repository solutions *JARS* is **not intended to address the presentation of digital resources to human users**, beyond providing very minimal search and display views. Instead, *JARS* will support other web and desktop applications that consume, analyze, and display resources. The limited focus of *JARS* on the core functionalities described above will help to minimize the cost of maintenance.

Another departure from more sophisticated repository solutions is that *JARS* will not support complex or composite objects. A complex object or resource is comprised of multiple data streams tied together by a single metadata record. Resources in *JARS* may have no more than one associated data stream. While it is technically possible to create complex resources in *JARS* by defining relations among resources in metadata, the system is not designed with that in mind. The closest

2.2 Product Functions

The main function of *JARS* is to store and provide access to digital resources, especially for scholarly text analysis projects, without the hassle of a monolithic repository solution. A scholar possessing a basic familiarity with web servers and web applications should be able to:

(1) Deploy *JARS* in less than an afternoon on a modern UNIX-based web server;

(2) Easily add digital resources either individually or in bulk;

(3) Generate (or ingest) standards-compliant metadata via an intuitive web interface;

(4) Add and deputize additional users as curators;

(5) Organize resources into collections (which are themselves treated as resources);

(6) Access resources (and collections of resources) programmatically via a REST API;

(7) Control access to resources, e.g. to restrict access to copyright materials.

In order to support the functionality described above, *JARS* should do the following "behind the scenes":

(1) Generate a stable URI for each resource, e.g. by interacting with a Handle server;

(2) Understand formal descriptions of metadata standards (e.g. described in RDF);

(3) Interact with online authority services (e.g. Conceptpower) to support metadata creation.

Although *JARS* is conceived with text analysis projects in mind, there is nothing to preempt its use as a general repository solution for other media types and in service to other kinds of projects.

2.3 User Characteristics & Audience

The primary application of *JARS* is to support text-analysis projects that involve internet resources. The Quadriga-Vogon platform, for example, involves annotating text resources that are retrieved from repositories via APIs. Another potential application is to allow scholars conducting text-mining research to share and refer to resources. The primary audience for *JARS* is scholars who are engaged in text analysis projects, especially collaborative projects, and have access to server resources but not necessarily to programming support resources.

JARS will be designed with two main user classes in mind:

(1) **Curators** will add resources and metadata, and manage access restrictions;

(2) **Consumers** will develop scripts or applications that consume resources via a REST API.

A third class, (3) **Viewers**, are also considered in *3.3 Use Cases*. Viewers are users that search for and access resources and/or metadata via a minimal public web interface.

Finally, **System Administrator** is responsible for deploying and maintaining the *JARS* system. This person will keep software up to date on the server on which *JARS* is deployed, apply patches and updates to *JARS*, and perform maintenance tasks like managing backups, and starting and stopping services.

2.4 General Constraints

2.4.1 Django 1.7 Python Web Framework

JARS will be developed using the Django 1.7 Python Web framework, which should reduce the cost of extension and maintenance. Django provides almost all of the low-level functionality required for *JARS*, as well as a flexible and extensible user management and content administration interface. This means that *JARS* will have a much smaller code-base than most other repository solutions, and development efforts can focus directly on high-level functionality.

2.4.2 Does not preempt integration with federated repository environment

JARS will be designed with the broader context of the ASU federated research repository environment in mind. Given the natality of that environment, the main consequence of this constraint is that nothing in the *JARS* software should preempt future integrations with other services, e.g. central authentication or indexing. Wherever possible, *JARS* should take advantage of existing resources in this environment (e.g. database servers, authority services).

2.4.3 Complies with ASU security policies and standards

JARS will comply with the ASU Information Security Policy (1.4.1), Secure Web Development Standard (ref 1.4.2), and any other ASU policies and standards pertaining to software, networks, or computer systems.

2.4.4 Can be deployed in a Linux, OSX, or Windows Server environment

Versions should be tested in all three environments before release or deployment.

2.5 Assumptions and Dependencies

Deviation from these assumptions may require revision of this SRS.

2.5.1 Server Characteristics and Software

The target deployment environment for *JARS* is a web server with the following software and characteristics:

(1) Python 2.7 (**not** 3.x);

(2) Django 1.7;

(3) Access to a MySQL database server;

(4) WSGI HTTP server (e.g. Gunicorn);

(5) Secure HTTP server that can act as a proxy for the WSGI HTTP server (e.g. Nginx, Apache).

2.5.2 A Handle server is available

Requirement 3.1.2.5 assumes that a Handle server is available to register/update Handles that can act as resource URIs.

2.5.3 A Conceptpower authority service is available

Requirement 3.1.2.4 assumes that a Conceptpower authority service is available to retrieve actor URIs.

3. Specific Requirements

This section describes the specific development requirements (functional and otherwise) for *JARS*. Wherever possible, functional requirements are written as use cases that describe functionality from the user's point of view. Each requirement is accompanied by a reference number that should be used in subsequent documents. Every effort has been made to develop requirements that are verifiable and unambiguous. These requirements may change as the project proceeds.

3.1 External Interface Requirements

3.1.1 User Interfaces

These requirements pertain to web-based interfaces for human users.

3.1.1.1 Should provide an administrative interface for managing users, resources, and other aspects of the system.

This interface should be designed to support the use cases described in *3.3.1 Curator Use Cases*.

3.1.1.2 Should provide minimal public views for searching and retrieving resources and metadata.

This interface should be designed to support the use cases described in *3.3.2 Viewer Use Cases*.

3.1.1.3 All web interfaces involving authentication should use HTTPS/SSL.

See also *2.4.3 Complies with ASU security policies and standards*, and ref 1.4.2.

3.1.2 Software & Communication Interfaces

These requirements pertain to interactions between *JARS* and other software and services.

3.1.2.1 Should be deployable as a WSGI application.

JARS should be designed to run behind a WSGI web server, such as Gunicorn.

3.1.2.2 Should store data in a MySQL database.

The administrator should be able to configure *JARS* to access a MySQL database server via the network, and reliably store and retrieve data.

3.1.2.3 Should provide a public JSON-based REST API that conforms to the JSON API Standard (ref 1.4.4).

Remote applications (e.g. web or desktop applications) should be able to search for and retrieve resources via that API. See also *3.3.3 Consumer Use Cases*.

3.1.2.4 Should access remote authority services via their APIs.

In order to support metadata creation (Use Cases 3.3.1.3 and 3.3.1.6) *JARS* should be able to connect to remote authority services to retrieve specific entity types, such as actors, institutions, or locations. At a minimum, *JARS* should be able to retrieve actor names and URIs from a Conceptpower authority service. This requirement assumes that *2.5.3 A Conceptpower authority service is available*.

3.1.2.5 Should interact with a Handle server to manage handles for resources.

As a mechanism for requirement 3.2.2, *JARS* should be able to connect to a remote Handle server to generate and manage URIs. This requirement assumes that *2.5.2 A Handle server is available*.

3.1.2.6 Should use OAuth 2.0 to provide access to restricted resources via the API.

See *3.3.3.4 Use case: Authenticate API Session via OAuth2*.

3.2. Functional Requirements

3.2.1 Provide Metadata Values from Remote Authority Service.

If an authority service (e.g. Conceptpower) is available and configured, *JARS* should provide the user with easy and appropriate access to entries from that service when the user is generating metadata.

"Easy" means that the user should not be required to perform more than one extra step to use an authority entry as a metadata value. "Appropriate" means that, wherever possible, the entities provided to the user should be of the correct type for the selected metadata field.

3.2.2 *JARS* should assign a URI for each resource.

When a user creates a new resource, *JARS* should automatically generate a URI for that resource. If available and configured, that URI should be a Handle generated by a Handle server. See also *3.1.2.5 Should interact with a Handle server to manage handles for resources*.

3.3 Use Cases

3.3.1 Curator Use Cases

Curators will add resources and metadata, and manage access restrictions.

3.3.1.1 Use case: Create Local Resource

Brief Description

The Curator creates a new Local Resource by uploading a file from their computer.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Resource change list.

1. The Curator selects Add Resource.
2. The system presents a choice of adding a local or remote resource.
3. The Curator selects to add a local resource.
4. The system presents a blank form.
5. The Curator enters a resource name, selects a local file, fills optional fields, and submits the form.
6. The system validates the form data and creates a new Local Resource object in the database.
7. The system returns the Curator to the Resource change list.

Postcondition

The Local Resource has been added to the database.

Other

Optional fields include Type, and whether the Resource is restricted.

3.3.1.2 Use case: Create Remote Resource

Brief Description

The Curator creates a new Remote Resource by entering the URL of a resource on a remote service.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Resource change list.

1. The Curator selects Add Resource.

2. The system presents a choice of adding a local or remote resource.
3. The Curator selects to add a **remote** resource.
4. The system presents a blank form.
5. The Curator enters a resource name, enters the URL of the remote resource, fills optional fields (can assign Type, flag as restricted), and submits the form.
6. The system validates the form data and creates a new Remote Resource object in the database.
7. The system returns the Curator to the Resource change list.

Postcondition

The Remote Resource has been added to the database.

Other

Optional fields include Type, and whether the Resource is restricted.

3.3.1.3 Use case: Update Metadata

Brief Description

The Curator updates the metadata for a Resource by selecting fields and entering values for those fields.

Step-by-Step Description

The Curator has already accessed the Resource change list.

1. The Curator selects a Resource from the Resource change list.
2. The system displays a form with existing metadata pre-filled, and an option to add a metadata Field to a resource.
3. The Curator selects the desired Field.
4. The system adds the field to the Resource metadata, and presents an input for the Field's value.
5. The Curator enters a value for the Field, and submits the form.
6. The system validates the form data and creates a new metadata Relation object in the database.
7. The system returns the Curator to the Resource change list.

Alternate Paths

A. At step 3, instead of adding a new Field the Curator can change the values of metadata Fields already added to the Resource.
B. At step 3, the Curator can also delete existing metadata Fields.
C. At step 5, the Curator can add additional metadata Fields before submitting the form.

Postcondition

A new metadata Relation has been added to the database.

Alternate path A: the values of the modified Relation(s) have been updated in the database.

Alternate path B: the modified Relation(s) have been dropped from the database.

Alternate path C: multiple Relations have been added to the database.

3.3.1.4 Use case: Restrict Access to Resource

Brief Description

The Curator restricts access to a Resource by selecting a "restricted" checkbox on the Resource change view.

Step-by-Step Description

The Curator has already accessed the Resource change list.

1. The Curator selects a Resource from the Resource change list.
2. The system displays a Resource form with pre-filled values, including a checkbox labeled "restricted".
3. The Curator checks the "restricted" box, and submits the form.
4. The system updates the Resource, and returns the Curator to the Resource change list.

Postcondition

The content of the Resource is no longer available to public users.

Bulk-update Resources use cases

3.3.1.5 Use case: Bulk-create Resources

Brief Description

The Curator creates multiple Resources by uploading a zip archive.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Resource change list.

1. The Curator selects to Bulk-create Resources.
2. The system presents an upload form.
3. The Curator selects a zip file to upload and submits the form.
4. The system unpacks the zip archive and looks for valid files, and (optionally) a metadata file.
5. The system creates a new Resource for each valid file and (if available) updates the metadata for each Resource.
6. The system returns the User to a filtered Resource change list that displays only the Resources just added.

Postcondition

The Resources have been saved to the database and (if metadata was available) metadata Relations have been saved to the database.

Other

If the available metadata did not contain names for the Resources, names are automatically generated using filenames.

3.3.1.6 Use case: Bulk-add Metadata

Brief Description

The Curator adds metadata Relations for multiple Resources by selecting Resources from the change list, and specifying a Field and value.

Step-by-Step Description

The Curator has already accessed the Resource change list.

1. The Curator selects multiple Resources in the change list, and selects to bulk-add metadata.
2. The system presents a list of available metadata Fields that can be applied to the selected Resources.
3. The Curator selects the desired metadata Field.
4. The system presents an input for the metadata Field.
5. The Curator enters a value in in the provided input, and submits the form.
6. The system creates metadata Relations for all of the selected Resources.

Postcondition

A new Relation has been added to the database for each of the selected Resources.

Other

The Curator can indicate whether the add operation should be “aggressive” or not. If the add operation is aggressive then if Resources are encountered that already have a value for the selected Field, and that field is unique (i.e. a Resource can only have one value), then the value will be replaced with the new value. If the operation is not aggressive, the Resource will be ignored.

3.3.1.7 Use case: Create Collection

Brief Description

The Curator creates a Collection by entering a name for the Collection and selecting zero or more Resources.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Collection change list.

1. The Curator selects to add a Collection.
2. The system presents a blank form.
3. The Curator enters a name for the Collection, and selects zero or more Resources.
4. The system creates a new Collection in the database, and returns the User to the Collection change list.

Postcondition

The Collection has been added to the database, and the selected Resources are associated with that Collection.

Use case: Create Metadata Schema

The Curator creates a new Metadata Schema by entering a name for that Schema.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Schema change list.

1. The Curator selects to add a Schema.
2. The system presents the option to create a schema manually, or from an RDF document.
3. The Curator selects to the manual option.
4. The system presents a blank form.
5. The Curator enters a name for the Schema, and submits the form.
6. The system creates a new Schema in the database, and returns the User to the Schema changelist view.

Postcondition

The Schema has been added to the database.

Alternate Paths

See *3.3.1.9 Use case: Add Metadata Schema from RDF*.

3.3.1.8 Use case: Add Metadata Field to Schema

Brief Description

The Curator adds a new metadata field by entering a name for the Field, and optionally selecting domain and range Types, and/or a parent Field. The Curator adds the Field to a Schema by selecting a Schema while adding or editing a Field.

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Field change list.

1. The Curator selects to add a Field.
2. The system presents a blank form.
3. The Curator enters a name for the field, and optionally selects domain and range Types, and/or a parent Field.
4. The Curator selects a Schema to which the Field should be added, and submits the form.
5. The system creates the new Field in the database, and associates it with the selected Schema.

Postcondition

The Field has been added to the database, along with a reference to the selected Schema.

3.3.1.9 Use case: Add Metadata Schema from RDF

Brief Description

The Curator adds a metadata schema by specifying a remote RDF file (or uploading an RDF file).

Step-by-Step Description

The Curator has already accessed the JARS main administrative interface, and may have accessed the Schema change list.

1. The Curator selects to add a Schema.
2. The system presents the option to create a schema manually, or from an RDF document.
3. The Curator selects the RDF option.
4. The system presents a blank form.
5. The Curator enters a URL for the RDF document, or a local RDF file to upload, and submits the form.
6. The system reads and interprets the RDF document, and creates a new Schema and Fields accordingly.

Postcondition

The Schema and Fields have been added to the database.

3.3.2 Viewer Use Cases

Viewers are users that search for and access resources and/or metadata via a minimal public web interface.

3.3.2.1 Use case: View Resource

Brief Description

The Viewer can view a Resource's metadata and content.

Step-by-Step Description

1. The Viewer request a Resource detail view via its URL, a Resource search (see 3.3.2.2), or a Collection view (see 3.3.2.3).
2. The system presents a page with Resource metadata, and an inline display of the Resource content.

Postcondition

The Viewer is on the Resource detail view.

3.3.2.2 Use case: Search for Resources

Brief Description

The Viewer can search for Resources by name using a simple search form.

Step-by-Step Description

1. The Viewer has already accessed the system's public views.
2. The system provides a simple search form in the header of all public views.
3. The Viewer enters some characters in the search form, and submits the form.
4. The system searches for Resources by name and URI using the Viewer's query, and displays a list or table of results.

Postcondition

The Viewer sees a list or table of Resources corresponding to their search queries.

Other

The table of results should be paged, so that only a limited number of Resources are shown at one time.

3.3.2.3 Use case: View Collection

Brief Description

The Viewer can view a Collection's metadata and constituent Resources.

Step-by-Step Description

1. The Viewer request a Collection detail view via its URL, or via a Collection search (see 3.3.2.4)
2. The system presents a page with Collection metadata, and a list or table of Resources in that Collection.

Postcondition

The Viewer sees Collection metadata and a list or table of Resources.

3.3.2.4 Use case: Search for Collection

Brief Description

The Viewer can search for Collections by name using a simple search form.

Step-by-Step Description

1. The Viewer has already accessed the system's public views, and selects to search for a Collection.
2. The system presents a search form.
3. The Viewer enters some characters in the search form, and submits the form.
4. The system searches for Collection by name and URI using the Viewer's query, and displays a list or table of results.

Postcondition

The Viewer sees a list or table of Collections corresponding to their search queries.

Other

The table of results should be paged, so that only a limited number of Collections are shown at one time.

3.3.3 Consumer Use Cases

3.3.3.1 Use case: Search for Resources via API

Brief Description

The Consumer can search for Resources (including Collections) by sending a request to the JSON REST API.

Step-by-Step Description

1. The Consumer sends a request to the search API endpoint, including search parameters (e.g. name, URI, or metadata Field values).
2. The system generates a set of matching Resources that are not restricted.
3. The system returns a JSON response.

Alternate Paths

At step 1, the Consumer can include an authentication token (see *3.3.3.4 Use case: Authenticate API Session via OAuth2*). If the token is valid, the system includes results (step 2) that are restricted.

At step 1, the Consumer can include instructions for paging the results (e.g. page size, which page). Then at step 3, the system returns a JSON response limited by the paging parameters provided in step 1. The Consumer can make multiple requests to retrieve multiple pages for a particular search.

Postcondition

The Consumer has received a JSON response describing Resources matching their search parameters.

3.3.3.2 Use case: Access Resource Metadata via API

Brief Description

The Consumer can retrieve metadata about a Resource by sending a request to the JSON REST API.

Step-by-Step Description

1. The Consumer sends a request to the Resource metadata API endpoint, including the URI or internal ID of a Resource.
2. The system checks to make sure that the Resource is not restricted.
3. If the Resource is not restricted, the System returns a JSON response containing metadata about the Resource.

Alternate Paths

At step 2, if the resource is restricted the system will return a 401 Unauthorized status code and a JSON response containing an error description.

At step 1, the Consumer can include an authentication token (see *3.3.3.4 Use case: Authenticate API Session via OAuth2*). If the token is valid, the system will return metadata even if the Resource is restricted.

Postcondition

The Consumer has received a JSON response containing Resource metadata.

3.3.3.3 Use case: Access Resource Content via API

Brief Description

The Consumer can retrieve Resource content by sending a request to the JSON REST API.

Step-by-Step Description

1. The Consumer sends a request to the Resource content API endpoint, including the URI or internal ID of a Resource.
2. The system checks to make sure that the Resource is not restricted.
3. If the Resource is not restricted, the System returns the raw content of a Resource (e.g. full text content).

Alternate Paths

At step 2, if the resource is restricted the system will return a 401 Unauthorized status code and a JSON response containing an error description.

At step 1, the Consumer can include an authentication token (see *3.3.3.4 Use case: Authenticate API Session via OAuth2*). If the token is valid, the system will return content even if the Resource is restricted.

Postcondition

The Consumer has received a JSON response containing Resource content.

3.3.3.4 Use case: Authenticate API Session via OAuth2

Brief Description

First, a Curator must create a new Client. Then, the Consumer can obtain an API authentication token by sending a request to the authentication endpoint using the Curator's credentials and the Client's tokens.

Step-by-Step Description

The Curator has already accessed the JARS administrative interface, and may have accessed the Client change list.

1. A Curator selects to create a new Client.
2. The system presents a blank form.
3. The Curator enters a name for the Client.
4. The system generates a Client ID and a Client secret key, and adds them to the Client before storing the Client in the database.
5. The system presents the Curator with the Client's ID and secret key.

6. The Consumer sends a request to the API authentication endpoint, including the Client ID, Client secret key, Curator username, and Curator password.
7. The system generates an access token, and returns it to the Consumer in a JSON response.

Alternate Paths

At step 6, if the credentials in the Consumer's requests are incorrect, the system responds (step 7) with a 401 Unauthorized error code and a JSON response containing error details.

Postcondition

The Client has been saved in the database. The Consumer has received a JSON response containing an access token.

4. Change Management Process

The requirements and constraints for *JARS* will undoubtedly change as the project proceeds. Those changes should be reflected in revisions to this document. The revision process should proceed roughly as follows:

1. New requirements arise from conversations among project stakeholders, who communicate those requirements to the developer;
2. The developer will work with those stakeholders to refine and clarify those requirements, and then communicate the proposed changes to all stakeholders for comment;
3. If there are no objections within a reasonable timeframe, the changes will be incorporated into a new version of this document, which will be circulated among all of the stakeholders for review. This is the "release candidate."
4. Manfred Laubichler will approve the revised document, or stipulate necessary changes before approval.

See *1.6 Versioning* for details about how revisions of this document will be versioned and released.