SCAM
STANDARDIZED CONTENT ARCHIVE MANAGEMENT

2002-05-17
Jan Danils, Jöran Stark

SCAM - STANDARDIZED CONTENT ARCHIVE MANAGEMENT

(draft)

1 Abstract

During your time in school, university and work, you produce and acquire a lot of valuable information in the form of documents, web links, book references and other resources. But how do you archive, organize and bring with you this information? You have increased difficulty locating specific information as your resource library grows. As the amount of information increases, so does the need to organize and describe that information in order to keep control over it. What you need is a way to add information about the information - metadata.

SCAM is, as the name implies, a web-based content archive with many similarities with a distributed filesystem, but with the ability to attach metadata to named resources. A resource can be almost anything from a file to a data object, or simply a reference to a document or a book. A resource's metadata describes additional information such as creator, keywords, conclusions, annotations, etc., which are properties that an ordinary filesystem lacks the ability to express. SCAM supports all metadata vocabularies denotable by Resource Description Framework (RDF).

SCAM highly emphasizes on portability and flexibility, which in turn relies on standardization of both design and implementation. Efforts on this subject have been one of main challenges during the development. Standards involve metadata vocabularies, content packaging, authentication of users and access control, system interfaces, etc.

2 Introduction

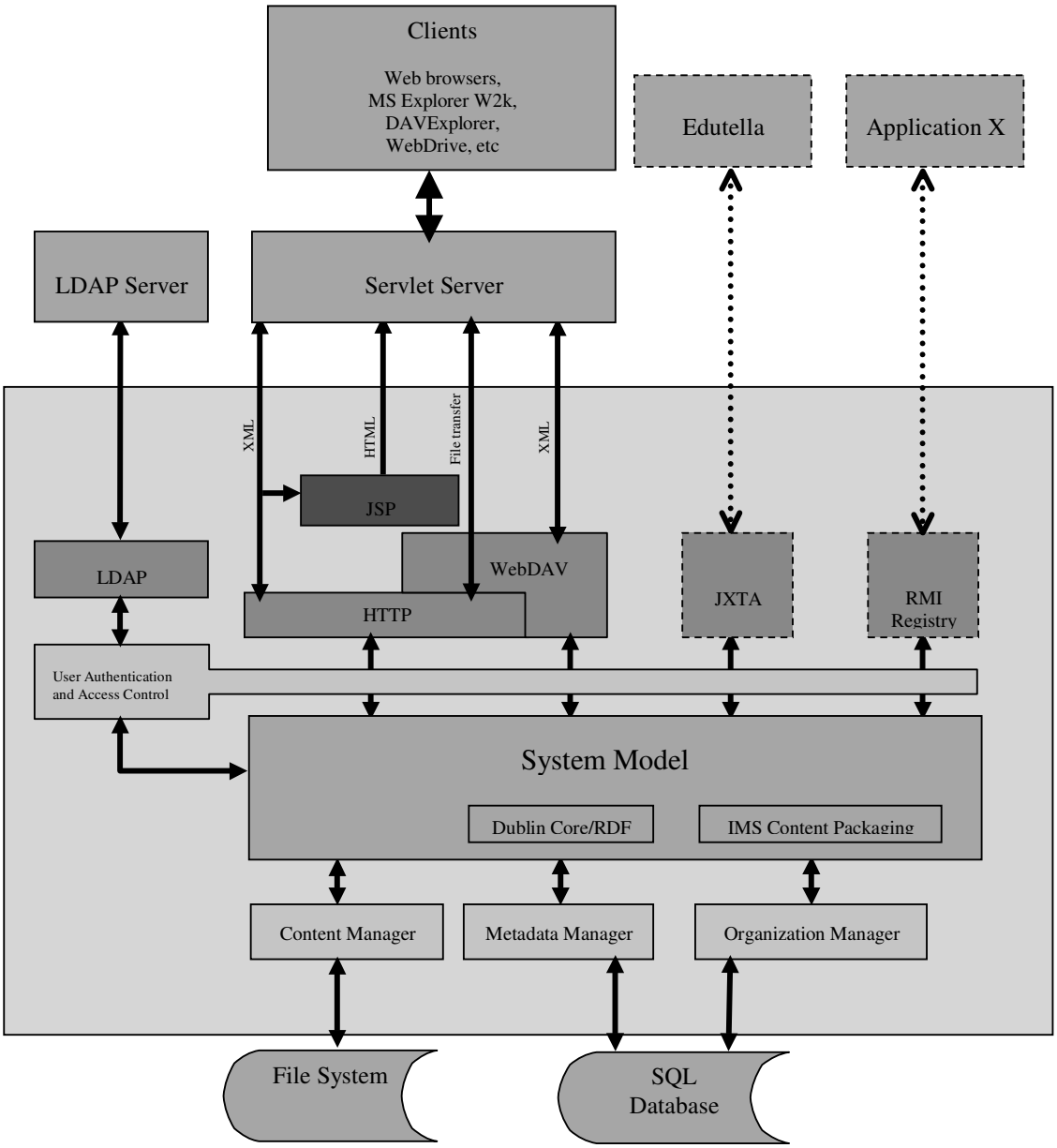
SCAM is entirely implemented in Java and Java Server Pages. One of the main goals is to use standards where applicable to increase interoperability and portability. SCAM is purely based on open-source software; an object-relational database manager *PostgreSQL* (acts as an archive-backend) and Apache's servlet server *Tomcat* (acts as a frontend). The choice of Postgre, besides the fact that it is free, is based on its performance, reliability and its ability of row locking. The ability of row locking is vital since it increases concurrent database access.

SCAM is developed as a core repository for different types of resources. The term *resource* is not well defined which creates a need of flexible storage handler. RDF can express any data structure that can be represented by a graph. Storing native RDF in the database provides a flexible mechanism for storing different sets of metadata. The problem with such flexibility is not in the storage, but rather in the complexity of the presentation of the metadata. Standardization of vocabularies and organization decreases this complexity, but to visualize this information is still a challenge. Different resources and vocabularies may require different presentations and the context in which the resource is accessed may also influence presentation.

A separate login handler provides user authentication and also grants portfolios to registered users. This means that no administration of new users is necessary as long as the users in question are included in the user repository. An Access Control List (ACL) describes what permissions users (or groups of users) have on a specific resource, allowing cooperative work and/or privacy.

3 Design

The implementation of SCAM employs a multi-level tier design. In this chapter we define the responsibilities of each significant tier and hint to vital Java classes. Refer to JavaDoc for a more extensive documentation.



3.1 Storage Tier

3.1.1 Responsibilities

Provide storage mechanisms.
Provide transaction control.

3.1.2 Description

A *Database* object acts as an adapter to the database application that is configured for the system. This object is responsible for providing *Connection* objects when accessing the database, and to provide mechanisms for keeping the database consistent during a request, *Transaction Control. Pool design pattern* is used for handling database connections, implemented by *ConnectionPool*

objects. The main advantage with this solution is the ability of keeping track of the total numbers of concurrent active connections, since databases has some limit in concurrent active connections. Another advantage is the ability to reuse connections between serialized queries since generating new connections is time consuming. [See package scam.repository.database.pool]

A *Resource* is defined by its *state information* and *metadata*, some resources are also defined to have a *content* (i.e. files). A request involves one or more resources in the requested operation, so the transaction control has to be able to include several resources. The transaction controller provided by the database could implement the transaction control mechanism. But due to the fact that databases are not well suited for storing file content we have chosen to separate *content* from *state information* and *metadata*. This introduces a new problem; the transaction control mechanism has to keep consistency for two separate sub system. A *Content Store* is delegated to handle the content the same way as the Database handles *state information* and *metadata*, i.e. implementing transaction control functionality. [See package scam.repository.content and scam.repository.database.Database]

Normally a request (from a client) is a request to perform an operation on a specified Resource**,** this Resource responds by executing a series of commands. If these commands involve the Content Store and the Database, or other Resources, the Resource starts a transaction by issuing *startTransaction*. If all commands completes successfully the Resource ends the transaction by issuing a *commit***,** otherwise the Resource issues a *rollback* to undo the transaction.

A problem with keeping multiple sub-systems consistent is that if one of them fails to commit when others already has committed successfully, you cannot undo the complete transaction. Committing the Database before the Content Store reduces the effect of having this system partially committed; if Database fails to commit then both is rolled back, if the Database successfully commits but the Content Store fails then only the Content Store can be rolled back. The possibility for a failure in committing the Content Store can be considered small since it operates on the local file system. The consequence of such a failure is, in the worse case, an "orphan" in the Content Store. By matching entries in the database with entries in the Content Store these orphans can be detected and manually removed.

A design goal is to include the ability to handle custom related metadata sets in addition to the internal metadata set. *Dublin Core* (DC) metadata set with *Canberra Qualifiers* (DCQ) defines the internal (native) metadata set used by SCAM. XML could be used to express this internal metadata set and many other metadata sets. One of the strength with XML is that data is represented in a hierarchy of elements stating beginning and end of data. The XML model is well suited for sending data in a serialized form. But storing hierarchies, such as XML, in a relational database is not trivial since the hierarchy itself implies the relations between the elements.

Resource Description Framework (RDF) provides mechanisms for describing relations between elements. RDF can express the same metadata sets and can easily be stored in a database for searches; it expresses relations as a set of triples, *{subject, predicate, object}*. Each triple is a *statement* about a resource; the resource is defined by the *subject,* what is stated is defined by the *predicate* and the value of the statement is defined by the *object.* The *object* can be represented by another statement. Since the RDF model is represented by a set of triples it is an "easy" task to

serialize the model into a database; store each statement as a triple using a table of three columns. [See scam.repository.ModelHandlerImpl]

3.2 Application tier

3.2.1 Responsibilities

business logic and rules
access control
user authentication
transaction handling (atomicity/consistency)

3.2.2 Description

A resource, as defined by SCAM, consist of the following parts:

- Access Control List (ACL); defines which principals having what permissions.
- Locks; user defined locking of resources (see WebDAV specification for more details).
- Metadata Model; the model is an implementation by Stanford University, which complies with the W3C/RDF API.
- Internal data such as name, owner, identifier (id).

A ResourceFactory fetches a Resource from the database according to an URL, e.g. "/users/documents/index.html". The *virtual* organization of the database is similar to that of an ordinary filesystem, i.e. a strict hierarchical tree. In other words we have resources that can contain other resources (containers) and other resources that can have content (files) and so forth. SCAM currently implements the following semantically different types of resources:

1. Root
2. Library
3. Portfolio
4. Folder
5. File
6. Url

Resources of type 1-4 are container resources similar to a folder in a file system. The Root-resource is the grandparent of all other resources. A user's personal collection of resources is denoted Portfolio. A Library is a collection of Portfolios and other Libraries. The difference between a File- and a Url-resource is that a File can have *content*. A Url is simply a reference to another Resource, either locally stored in the system or to any other component denotable by an URL. A Resource contains a set of atomic operations/methods as defined in scam.share.Resource.

3.2.3 User Authentication

The application tier supports multiple user authentication schemes. You can define either local or remote authentication depending on in which environment the system is located. SCAM is designed to be either a stand-alone application or to be a part of another system that has external user authentication capabilities. For instance an LDAP authentication design is easily introduced. [See scam.webdav.login.Login].

3.2.4 Access Control

The ResourceFactory is responsible for controlling read-permission of the caller before returning the resource. After this, the resource itself performs permission control accordingly. The ACL functionality is defined according to specifications of java.security.Acl. [See scam.repository.AclHandlerImpl]

3.2.5 Locking

These user defined locks is not the same as the row locking in the database. This functionality exists to prevent the *lost-update problem*. If a user A downloads a certain file for editing and another user B does the same, we get two instances of the same file. Then A uploads her file with changes made, followed by B. B will then overwrite the changes made by A. Instead A first lock the resource telling all others that "I am currently working with this". If B then tries to lock it, the system denies this request. Locks have timeout, which means that locks have to be refreshed before timeout occurs or the lock is automatically removed.

We do not use the database's internal locking of rows because of deadlock/starvation problems. If the client does not unlock the resource, the resource will never be unlocked denying all further locking requests. [See scam.repository.LockHandlerImpl]

3.2.6 Metadata Model

SCAM uses RDF as a carrier of metadata, which means that any metadata that can be expressed with RDF can be stored in the database. The system currently has no validation of RDF schemas, nor does it validate the metadata itself. This is considered to be the responsibility of a metadata editor.

The system natively understands and uses the Dublin Core metadata set. Other vocabularies can be stored, but not natively understood by SCAM. [See scam.repository.ModelHandlerImpl]

We have chosen to represent the organization of resources separately outside the RDF model. This is due to the complexity of managing consistency between resources using the W3C RDF API. The only native organizational relation between resources is "parent of". This choice is subject of change.

3.2.7 Business rules

SCAM defines a set of organizational rules that must be valid before execution of operations can take place. This involves issues like consistent relations between resources, access- and lock-control, and "garbage collection". The latter covers things like removing children of a container resource when the container resource itself is removed, in other words removing resource orphans (unrelated resources). [See scam.share.Resource].

3.2.8 Transaction Control

If a resource is fetched for the purpose of editing/changing, the ResourceFactory extracts it in a locked state (locked in the database) to preserve consistency. If a transaction deals with multiple resources, problems like *deadlock* and *starvation* must be considered. The order in which the resources are locked is important to prevent this. We have solved this issue by always locking

them top-down, i.e. parent before children. However, this method is only applicable when a resource is part of **one** organization.

This factory-design induces a rather complicated transaction-design. Resources are only allowed to be locked in a "known environment" in order to ensure that no resource is left locked in the database due to some failure. In other words, a resource must never leave the repository-package in a locked state. This means that a proxy retrieves an unlocked resource from RF, perform the requested operation on the resource that first locks itself before actually executing, and finally unlocks itself before returning. The same is true in the case of multiple resource operation. [See scam.repository.ResourceImpl and scam.repository.ResourceFactoryImpl].

3.3 Presentation tier

3.3.1 Responsibilities

Interface between client and repository

3.3.2 Description

Different clients and protocols require different presentations and interfaces. In SCAM we define each of these as *proxies*, request interceptors that translates client-specific protocols to SCAM's internal object model and back, separating system logic from client capabilities. Several proxies can be used concurrently to access the repository.

SCAM defines a HTTP proxy with two branches, one for the lightweight browser client, and the other for a WebDAV client. In the heart of this proxy we have a Java Servlet that acts as a Controller. The proxy-methodology is as follows:

1. The Controller receives a HTTP request.
2. A login-handler is deployed by the Controller to identify and authenticate the caller. If no user can be extracted from the request or session, we have an anonymous user. This handler can also be used to prompt the caller to identify herself.
3. A method factory is used to identify and create the appropriate method, both for WebDAV and core HTTP requests. A class represents each operation defined by SCAM.
4. The Controller executes the method created.
5. The method parses the request for operation parameters. The repository package is used to create and manipulate resources. Some methods forwards requests to Java Server Pages.
6. The response from the repository is translated to the appropriate response format.

A vision of SCAM is to implement several other proxies when demand arises. You could think of RMI-, SOAP- or CORBA-clients wanting access, or perhaps a wrapper for FTP-clients. SCAM will also adapt the JXTA-based peer-to-peer *Edutella* network for metadata queries in future releases.

4 Further reading

SCAM is still considered to be in its initial stage and the implementation has a prototype status.

Further documentation and software can be found at SCAM's official homepage at KTH-KMR [http://kmr.nada.kth.se/scam] and at SourceForge [http://sourceforge.net/projects/scam].