SCAPE

Scalable Preservation Environments

Architecture Design

First Version

Authors
Frank Asseg, Finn Bacall, Stanislav Barton, Rui Castro, Matthias Hahn, Markus Plangg, Martin Schenck, Rainer Schmidt, David Withers

March 2013

This work was partially supported by the SCAPE Project. The SCAPE project is co-funded by the European Union under FP7 ICT-2009.4.1 (Grant Agreement number 270137).

Executive Summary

This document discusses the software architecture of the SCAPE Preservation Platform. The SCAPE Platform provides an infrastructure that targets the scalability of preservation environments in terms of computation and storage. The goal of the architecture is to enhance the scalability of storage capacity and computational throughput of digital object management systems based on varying the number of computer nodes available in the system. The SCAPE Preservation Platform also provides support for migrating existing applications to the parallel environment, and integration with other SCAPE services and components such as preservation planning and watch.

This document examines the platform architecture from a number of different perspectives. The architectural layers separate the design into tiers, each of which provides different functionality, and can be deployed in separate host environments. Each layer is based on existing, mature software components - like Apache Hadoop, the Taverna Workflow Management Suite, and the Fedora Digital Asset Management System. The SCAPE Preservation Platform provides additional services on top of these software components to support scalability and integration with digital preservation processes. The provided functionality includes scalable access to the digital object management system, support for the parallel processing of preservation workflows, plan management and reporting, and preservation component registration and semantic search.

The Platform architecture supports a number of applications to help users in interacting with the system. These include applications for creating, validating, and registering workflow components based on a graphical user interface; a compiler to translate from sequential workflows to parallel applications that comply with the platform's execution environment; a utility for scalable repository ingest and status monitoring, as well as graphical support for plan management.

The architecture document also identifies a set of general use-cases that are supported by the platform architecture in order to provide a closed view on the system from the user perspective. The use-cases cover processes such as workflow design, registration, managing dependencies to external tools, deployment on the parallel system, execution, and digital object management.

Table of Contents

1. Introduction

1.1. Context

Cloud and data-intensive computing technologies have introduced novel methods for developing virtualized and scalable applications. The SCAPE Preservation Platform leverages such technologies to overcome scalability limitations in digital preservation processes and workflows. This document describes the software architecture of the *Platform for Scalable Preservation* developed by the PT Sub-project as part of the EC FP7 project SCAPE.

1.2. Objectives

The SCAPE project is developing tools and services for the efficient planning and application of preservation strategies for large-scale, heterogeneous collections of complex digital objects. The SCAPE Platform provides an infrastructure that targets the scalability of preservation environments in terms of computation and storage. The goal of the architecture is to enhance the scalability of storage capacity and computational throughput of digital object management systems based on varying the number of computer nodes available in the system. Furthermore, the Platform targets scalability limitations of content repositories that manage large number of information objects and associated digital content. For this purpose, the Platform is designed to support the coordinated and parallel execution of existing preservation tools, to maintain large volumes of records of generated results, and to interact with various information and data sources and sinks. At its core, the SCAPE Platform provides a scalable execution and storage backend that can be attached to different object management systems through standardized interfaces. The architecture aims to address scalability limitations regarding the number, size, and complexity of the managed information objects and associated content.

1.3. Scope

This document provides a comprehensive overview of the SCAPE Preservation Platform with respect to its architecture. We describe the platform's system design from different points of views including a discussion on the general system layers (Section 1.4), the main architectural entities (Section 2), services that are specifically developed by the platform (Section 3), and the applications provided by the platform (Section 4). A list of the use-cases the platform has been designed to satisfy, as well as deployment considerations and a glossary are provided in Appendices.
The document describes only the platform architecture, and does not detail the architecture of the SCAPE project as a whole. Neither does the document describe the interactions between the SCAPE Preservation Platform and other SCAPE sub-systems such as the Planning and Watch components, beyond describing the services provided by the platform in general. For a more complete picture of the SCAPE architecture the reader is referred to deliverable D2.2 Technical Architecture Report[1]. It is also important to note that the document provides a complete picture of the platform's services and APIs solely from an architectural perspective but does not provide a complete (and final) technical specification of all services, some of which were still under development at the time of writing.

[1] http://www.scape-project.eu/wp-content/uploads/2012/11/SCAPE_D2.2_BL_V1.0.pdf

1.4. Layers of the Architecture

This section describes existing software components and interfaces that the SCAPE Preservation Platform adopts and builds upon. In general, the platform architecture consists of a set of four logically independent system layers that communicate through APIs and services. This allows a system administrator to choose between different ways to deploy the SCAPE Preservation Platform, for example to host the digital object repository remotely and physically distributed from the storage and computation environment. In principle, each layer of the architecture may be deployed within a separate hardware, or virtual environment; subsequently, applications in different layers must be able to run in separate host environments. For efficiency reasons however, it will be useful in many cases to deploy layers that host tightly coupled services (like storage and computation) within a shared environment.

Each of the layers is based on existing and mature software components such as Apache Hadoop[2], Taverna Workflow Management System[3], and Fedora Digital Asset Management System[4], which are briefly described in this section. The SCAPE Platform services, described in chapter 4, extend and integrate these components in order to meet the specific preservation challenges identified in the context of SCAPE.

The following provides an overview of the general structure of the platform architecture followed by a brief description of the software environments and services utilized on each layer:

- The **Storage Layer** hosts the main storage environment, which is accessible by the execution platform. At this layer, scalability, robustness, and throughput are the primary requirements. The Storage Layer may provide a pure storage environment for raw or semi-structured content and may also serve as the underlying environment for persistence of structured metadata, for example generated by the repository.
- The **Execution Layer** provides robust and scalable computational services. These include the provisioning of tools and applications required to carry out specific preservation operations. For efficiency reasons the deployment of this layer will be combined with the storage layer in many cases, i.e. the data will be stored on the same computer nodes which are used to perform computations on them.
- The **Data Management Layer** hosts the digital object repository that primarily maintains structured information about objects. Required functionality includes the creation, retrieval, update and search for information objects. Depending on the technologies used, this layer may be deployed on the storage layer for scalability and performance reasons.
- The **User Application Layer** hosts services and user applications that are only loosely coupled with the data management, execution, and storage layers. Examples are the workflow design environment and services for the registration and discovery of preservation components. Graphical user interfaces for the digital object repository are also hosted on this layer.

[2] http://hadoop.apache.org/
[3] http://www.taverna.org.uk/
[4] http://fedora-commons.org/

1.5. Storage Layer

The SCAPE Preservation Platform employs the Apache Hadoop framework to establish the distributed and scalable storage and execution layers. The Hadoop Distributed File System (HDFS), part of the Hadoop framework, provides the Platform's preferred storage technology as it offers scalability benefits with respect to storage and computation. The storage layer can be used for data archiving as well as for providing a data staging area used during ingest and data analysis. The integration of digital object management systems with HDFS has been implemented through both an HDFS storage adapter and the Data Connector API (Section 0), providing an efficient interface between the Digital Object Repository and the Execution Layer. Please note, it is recommended but not required that a SCAPE Preservation Platform instance makes use of HDFS as a file system (for processing and/or archival storage). SCAPE preservation services, described in section 3, are independent from the utilized file system implementation, and currently accept a number of protocols like HTTP and File URIs, as well as SCAPE Digital Object Identifiers.

HDFS provides a file-based interface that is used internally by platform components and externally by client applications that transfer data to the platform storage. HDFS is a distributed and horizontally scalable file system that provides a decentralized storage layer on top of local disks (also called *shared nothing architecture*). The file system provides reliability by replicating data across distributed cluster nodes. Data integrity is automatically checked and does not require manual intervention. The HDFS API is similar to those of existing file systems and supports basic file operations and streaming but is not fully POSIX compliant, hindering its direct use by a number of existing application. HDFS provides a JAVA API and a corresponding command-line interface. A POSIX-like FUSE mount HDFS extension is available for working with OS commands. APIs for Perl, Python, Ruby, and PHP are also available. The HDFS over HTTP (Hoop) project[5] provides a REST API that has been recently integrated with the Apache Hadoop project.

1.6. Execution Layer

SCAPE stakeholders demand scalable methods that support the processing of content as part of the preservation environment. The execution layer of the SCAPE Platform has been designed to satisfy this requirement. The MapReduce engine, residing on top of HDFS, provides this functionality for the Hadoop framework. It provides a parallel programming paradigm (MapReduce) and a distributed runtime environment capable of processing vast amounts of data on clusters of commodity hardware. The MapReduce paradigm is highly scalable and has proven to be widely applicable for processing structured data. In the context of the SCAPE Platform, this approach is utilized for the processing of a wide range of content types. This is primarily achieved by porting existing preservation tools and workflows originating from different communities to the MapReduce programming model and the Hadoop execution environment. General methods and tools like the SCAPE tool specification language, tool wrappers, and workflow compilers help the user to migrate preservation strategies from a desktop environment to the MapReduce-based execution environment.

The Hadoop framework provides a number of APIs as well as higher-level programming languages like Hive and Pig to help implement parallel applications. The execution layer performs data decomposition and resource allocation dynamically, however it requires development of specific

[5] http://cloudera.github.com/hoop/docs/latest/index.html

input and output format handlers hat comply with the Hadoop API in order to decompose complex input formats.

A number of approaches to enable the distributed processing of SCAPE data sets have already been developed. One example is archHD[6], a project developed by the SCAPE Web Content Testbed, which provides a custom Hadoop RecordReader, enabling the distributed processing of web archive data encoded in the ARC.GZ format. The PT-MapReduce Tool[7], developed in the context of the Platform Sub-project, provides a generic wrapper for command line tools that can be conveniently configured using the SCAPE Tool Specification Language[8]. Another example of support provided for practitioners porting SCAPE preservation workflows to the execution layer is the development of a library that supports the distributed processing of METS files[9].

1.7. Data Management Layer

Digital content comes in different flavours and levels of complexity running from simple text through books, images and maps to complex, compound and even dynamic objects like video, audio or numeric data sets with associated code. Not only has the variety of types increased over time, but the quantity of digital information produced has also increased dramatically, especially over the past decade. Commercial and non-commercial digital object repositories are available and are used in diverse communities. Fedora Commons[10] provides an open source product used widely by the digital library community. Fedora Commons is also used by three out of four SCAPE repositories as the underlying core repository component. Fedora Commons does not provide a full-fledged product but is designed to be used as a software component of a specific repository, built for a particular community (e.g. an institutional repository). Fedora Commons makes use of a very flexible content model that is easy to adapt to diverse use cases. The following list gives examples of material Fedora is able to support: Documents, articles, journals, electronic scholarly texts, digital images, complex multimedia publications, datasets, metadata and much more. In addition to being able to manage complex digital objects, Fedora also supports the creation and management of relationships between objects using RDF descriptions.

The user (as a producer or a consumer of content) interacts with Fedora via several APIs providing functionality such as ingest, retrieval, access, query of metadata or triples, and storage interfaces, as illustrated in Figure 1.

[6] https://github.com/openplanets/scape/tree/master/tb-wc-hd-archd
[7] https://github.com/openplanets/scape/tree/master/pt-mapred
[8] https://github.com/openplanets/scape-toolspecs
[9] https://github.com/openplanets/scape-platform/blob/master/metshadoop/README.md
[10] http://fedora-commons.org/

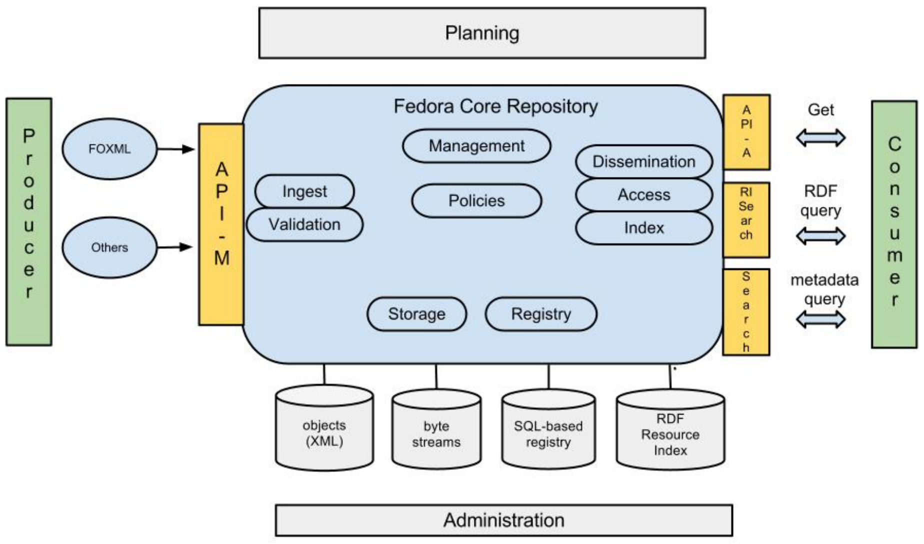


Figure 1: Simplified illustration of Fedora services

The following list provides a mapping between the APIs listed in Figure 1 and the APIs provided by Fedora Commons Version 3.x:

- API-A – Fedora Access Service (SOAP)
- API-A-LITE – Fedora Access Service (REST)
- API-M – Fedora Management Service (SOAP)
- API-M-LITE – Selected Operations (REST),
- REST API – Fedora REST API
- Basic Search – Repository Search via API-A-Lite (REST)
- Basic OAI – Simple OAI-PMH Provider (REST)
- RISearch – Resource Index Search (REST)
- Messaging API – JMS Messaging (ATOM)

Fedora is designed to be only one component of a Digital Object Repository (DOR). A number of SCAPE partners including KEEPS, SB, and FIZ have developed custom repositories that build upon Fedora. In order to integrate these various repository implementations with other SCAPE components like the computation cluster, or the SCAPE Planning & Watch Component[11], the repositories must implement a specific set of APIs (e.g. the Plan Management API; the Report API; and the Data Connector API) outlined in this document. The following graphic illustrates a high-level view of a SCAPE repository that uses Fedora at its core:

[11] The reader is referred to SAPE deliverable D2.2 for further information on the overall SCAPE architecture.

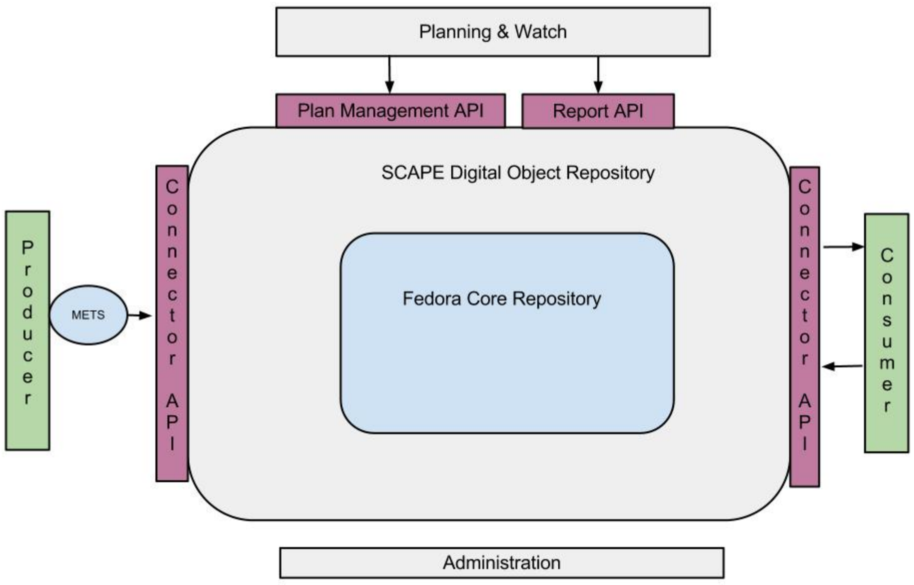


Figure 2: Simplified illustration of a SCAPE Digital Object Repository (not shown are the specific services already implemented by repositories like eSciDoc, RODA or DOMS).

This SCAPE Platform architecture defines these required APIs; the Plan Management API is described in Section 3.4, the Report API in Section 0 and the Data Connector API in Section 0. All three interfaces must be implemented by SCAPE repositories in order to allow fully integrate with the SCAPE components.

1.8. User Application Layer

The User Application Layer provides mostly graphical applications that enable the user to design, deploy, register, and execute preservation tools and workflows.

A majority of the Application Layer has been implemented using the Taverna software stack including the Taverna Workbench[12] and MyExperiment[13]. The resulting end-user applications provide a suite of graphical user interfaces for the definition, semantic description, and registration of preservation workflows. A tool enabling the semantic annotation and validation of SCAPE components, and realized as a Taverna plug-in is described in section 0. The SCAPE Component Catalogue used for registration and lookup of SCAPE components is described in section 4.2.

The Taverna workflow management system[14] is a suite of tools for designing and executing workflows including Taverna Workbench, Taverna Command Line, and Taverna Server. **Taverna Workbench** is a desktop application that supports design of workflows using a graphical workflow editor. Workflows are created by adding services and connecting together the input and output ports of these services, which can then be executed through the workbench; workflow progress is shown through the workflow diagram, with result values viewed in the results perspective. **Taverna Command Line** is an application for executing Taverna workflows from the command line for use

[12] http://www.taverna.org.uk/download/workbench/
[13] http://www.myexperiment.org/
[14] http://www.taverna.org.uk/

when a graphical interface is not available, such as when running a workflow on a node of a cluster. This allows Taverna workflows to be run on a remote server.

myExperiment is a web application that facilitates the publishing and discovery of scientific workflows. In the SCAPE Platform, it serves as the Component Catalogue: a repository for both preservation components and complete preservation plans, both of which are realised as Taverna workflows. Moreover, myExperiment allows users to create groups to assist with collaboration. When a user uploads a new resource, they can elect to share it with a group that they are a member of. Each group has a page displaying information on the topics and interests concerning the group, and lists of group members, and shared resources. A SCAPE group has been created to allow sharing amongst SCAPE project members, and dissemination of SCAPE preservation workflows to the wider community. As well as a web interface, myExperiment also provides a REST API which allows clients to programmatically upload, browse and retrieve workflows and other content. This API is used by the Taverna Workbench to allow convenient publishing of workflows and components directly to myExperiment, and it is this mechanism that provides the underlying functionality for the design and implementation of the SCAPE component catalogue.

2. Main System Entities

This section outlines the major system entities that constitute the SCAPE Preservation Platform describing provided services, interdependencies, and cardinalities. Although the SCAPE Platform is designed to support software components and services provided by other SCAPE Sub-projects, its core entities can operate independently from external systems and services. A particular preservation scenario, for example, can be carried out autonomously on an instance of the SCAPE Preservation Platform presuming all required prerequisites (like data, tools, and workflows) are available.

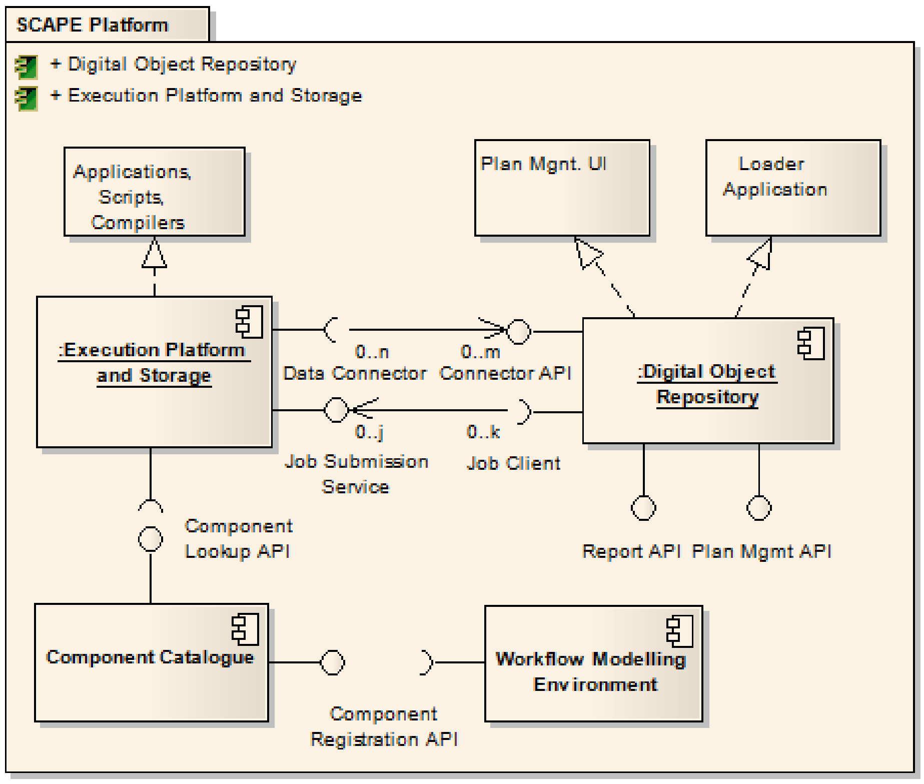

SCAPE Platform
+ Digital Object Repository
+ Execution Platform and Storage
Applications, Scripts, Compilers
Plan Mgnt. UI
Loader Application
:Execution Platform and Storage
0..n
0..m
Data Connector
Connector API
:Digital Object Repository
0..j
0..k
Job Submission Service
Job Client
Component Lookup API
Report API
Plan Mgmt API
Component Catalogue
Workflow Modelling Environment
Component Registration API

Figure 3: Overview of the Platform's main entities, provided services, and interdependency.

Figure 3 shows the interdependencies between the digital object repository and execution. The entities interoperate with each other using two defined interfaces: (1) the Data Connector API (see Section 0); and (2) the Job Submission API[15] (see Section 3.1). The services that implement these APIs provide the two core functionalities of the SCAPE Platform: data management and computation, respectively. Although a typical Platform deployment might involve only a single repository and a single execution platform, the system is not limited to this configuration[16]. Both, the Data Connector API and the Job Submission Service maintain an n:m relationship with their clients enabling the

[15] The Job Submission Service is referred to as the Job Execution Service in SCAPE D2.2.
[16] A SCAPE repository reference implementation that is integrated with the platform's storage and execution environment (supporting the Data Connector API as well as the Job Submission Service) is being distributed as part of the Platform software package.

Execution Platform to interact with multiple Digital Object Repositories, or indeed a Digital Object Repository to be used by many Execution Platforms.

The Component Catalogue (based on myExperiment) provides a central registry for SCAPE preservation components. Components are designed, semantically described, and registered using the SCAPE workflow modelling environment. The SCAPE Platform provides a number of mechanisms to convert or embed SCAPE components into parallel applications capable of running on the execution platform (using for example the tool wrapper or the PPL workflow compiler). Parallelized versions of SCAPE preservation components are installed and executed using the Platform's execution environment.

2.1. Execution Platform

The SCAPE Execution Platform (EP) provides a tightly coupled data storage and processing network (called a cluster) that forms the underlying infrastructure for performing data-intensive computations on the SCAPE Platform. The Execution Platform specifically supports the deployment, identification, and parallel execution of SCAPE tools and workflows, and integrates with different data sources and data sinks. The system provides a set of command-line tools that support users in directly interacting with the system, for example to carry out data-management and preservation actions on the cluster. The Execution Platform does not provide graphical user interfaces per se but provides a set of service APIs (section 1.6) to interact with client applications.

2.2. Digital Object Repository

A SCAPE Digital Object Repository (DOR) provides a data management system that interacts with the Execution Platform to carry out preservation actions. The SCAPE DOR exposes services to other system entities developed in the context of SCAPE, for example it enables the retrieval of information about events that take place within the DOR for use by Planning and Watch components – see the Report API in Section 0. A SCAPE Digital Object Repository exchanges information with the execution platform via the Data Connector API (see Section 0) and may store or copy sets of content directly to the Execution Platform's storage system. The DOR manages and triggers the execution of Preservation Plans through the Plan Management API (see Section 3.4), with the ability to preserve portions or the entire outcome of a workflow that has been executed against the content a DOR manages. A DOR therefore employs a defined data exchange model (i.e. the SCAPE Digital Object Model) as well as a scalable object store. The object repository is also responsible for helping its user community to deposit, curate, and preserve digital content.

2.3. Component Catalogue

The SCAPE Component Catalogue provides a registry for SCAPE components in a SCAPE environment. A SCAPE component consists of a workflow fragment that implements well defined interfaces, a description of contextual dependencies, and additional semantic descriptions. A workflow component must adhere to the SCAPE component profiles developed in the context of the Planning and Watch Sub-project. SCAPE components are registered and discovered using the SCAPE Component Catalogue. The platform's workflow modelling environment specifically supports the creation, registration, and discovery of SCAPE components, as described in Section 4. The Component Catalogue supports semantic discovery of SCAPE components through a specific query interface (the Component Lookup API – see Section 0) used by the Preservation Planning Tool[17]. Complex workflows may be composed from existing SCAPE components based on their interfaces (contracts). Workflow components also explicitly define dependencies on software packages (e.g. Linux packages), which is essential information to know when deploying a SCAPE component on the Platform’s execution environment.

3. Services Interfaces

The following section provides a technical overview of the application programming interfaces (APIs) that are provided by the services the SCAPE Preservation Platform exhibits. At the time writing, detailed API specifications are still under development and have not yet been publicly released. It is therefore important to note the services described below are in a draft state and do not provide complete or final API specifications.

3.1. Job Submission Service

The Job Submission Service (JSS) provides an interface for performing and monitoring parallel data processing operations (jobs) on the platform infrastructure. The Platform’s Digital Object Repository (DOR) acts as a client to this service in order to perform preservation operations, for example as defined by a preservation plan, against the data it manages. The service may also be utilized by applications running on a desktop computer like the workflow modelling environment provided by the Taverna workbench. Multiple clients may interact with a Job Submission Service concurrently.

Depending on the repository implementation and use-case, the processed data may or may not reside on the Platform's storage network prior to execution. For example, a Digital Object Repository may be loosely integrated in the sense that the Execution Platform is just used as an external data processing environment (perhaps at the point of ingest). An example would be the implementation of an active cache, where the repository can perform scalable preservation activities on data that has been cached on the platform during ingest, without directly exposing the repository’s storage layer. Thus, a SCAPE Digital Object Repository may maintain its own storage layer and, through the JSS, use the execution platform for external processing.

The man functionality of the Job Submission Service is job queuing and scheduling, as shown in Figure 4: Use-case diagram for the Platform Job Submission Services. The design follows the model of a Job scheduling Web service that typically works on top of a local job scheduler (like for example PBS[18]); a model commonly used in many Cluster- and Grid Computing infrastructures. The main contribution of the SCAPE Job Submission Service will be the design and implementation of the Job Submission Language, which supports SCAPE concepts like preservation workflows, tool dependencies, and different digital object identifiers and access protocols. Existing standards like the OASIS WSRF[19] framework and implementations like the Apache Oozie[20] Workflow Scheduler for Hadoop are being considered for the design and implementation of this service.

[17] http://www.ifs.tuwien.ac.at/dp/plato/intro.html
[18] http://en.wikipedia.org/wiki/Portable_Batch_System
[19] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
[20] http://oozie.apache.org/

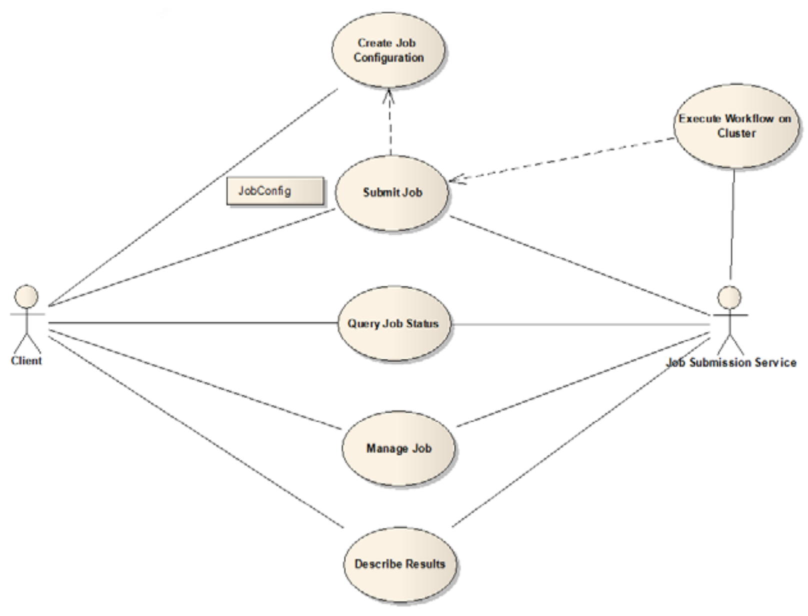


Figure 4: Use-case diagram for the Platform Job Submission Services.

The following service API provides an idea of the basic functionality provided by the JSS with respect to the overall architecture design. A detailed specification of the Job Submission Service API has to be defined and will be delivered as a separate document.

Job Configuration

A job configuration is an XML descriptor that specifies all the information required to execute a particular application on the cluster. Jobs may refer to different types of applications including Java libraries (jars), scripts (pig/hive), and file-system operations (HDFS), which may be referenced in different ways, using paths or an identifier schema. In general, a job might consist of a single workflow activity or specify a workflow graph (Directed Acyclic Graph) containing multiple activities. The SCAPE Job Submission Service focuses on the execution of entities specific to the SCAPE architecture; hence it is important to provide mechanisms to execute SCAPE components based on the identification schema used by the SCAPE Component Catalogue. This means that the JSS should be capable of testing whether a specified component is available on the execution platform and, if necessary test that all tool dependencies of a component can be resolved. The SCAPE job configuration must also support SCAPE Digital Object Identifiers, as implemented by the Data Connector API (Section 0). This allows a user to execute workflows that operate on Digital Objects managed by a SCAPE Digital Object Repository. SCAPE Digital Objects may be input and/or output to a job and can be retrieved from and stored to a DOR based on the operations defined by the Data Connector API. Additional job description parameters like wall clock time, priority, or resource reservation are supported by the execution platform's job scheduler. Authentication will be implemented using an available standard such as HTTP basic authentication or a TLS/SSL infrastructure.

Job Submission

This endpoint accepts a job configuration based on an HTTP POST request. The job will be created and scheduled for execution based on the information provided by the descriptor.

Path:	/jobs/
Method:	HTTP/1.1 POST Content-Type: application/xml
Consumes:	XML job descriptor
Produces:	HTTP/1.1 201 CREATED | Error Code Content-Type: text/xml A URL providing the Job identifier.
Content-Type:	text/xml

Job Status

This endpoint will return the status information for a submitted job.

Path:	/jobs/<id>/status
Method:	HTTP/1.1 GET
Produces:	HTTP/1.1 200 OK | Error Code A list with parameters that describe the current status of a job. Examples are status code (like QUEUED, RUNNING, FINISHED), wall clock time, or a URL of a graphical user interface provided by the job scheduler.
Content-Type:	text/xml

Job Management

This endpoint performs a management action on a particular job. Examples are removing a job from the job queue or terminating a running job.

Path:	/jobs/<id>
Method:	HTTP/1.1 PUT
Parameters:	action: *Describes the action to be performed like TERMINATE | REMOVE*
Produces:	HTTP/1.1 200 OK | Error Code
Content-Type:	text/xml

Job Results

This endpoint returns the results of finished/archived job execution. Examples are job and error logs as well as URIs to the input data and the output data sets.

Path:	/jobs/<id>/results
Method:	HTTP/1.1 GET
Parameters:	action: *RESULTS*
Produces:	HTTP/1.1 200 OK | Error Code
Content-Type:	text/xml

3.2. Data Connector API

The Data Connector API is a RESTful API implemented on top of a repository. This API relies on the well-defined format of Intellectual Entities, which are being implemented based on the SCAPE Digital Object Model specification. The specification uses the METS schema – a standard for encoding descriptive, administrative, and structural metadata[21]. The METS format is widely used by digital libraries, but is agnostic of the metadata schema contained within the document. The Digital Object Model documentation provides a specific METS profile for SCAPE.

Use Cases addressed by the API are:

- **Batch ingest via Loader Application:** To ingest of large amounts of data into a repository it is recommended that the Loader Application (see Section 4.3) is used. The Loader Application acts as a client of the repository using the Data Connector API.
- **Request for Intellectual Entities by the computation cluster:** When processing large sets of Intellectual Entities, for example performing identification and characterization, access to the metadata and the binary content of an Entity is provided through the Data Connector API.
- **Update Intellectual Entities with provenance information:** Preservation actions executed on the computation cluster that need to update the provenance metadata of an Intellectual Entity representation can do so via the Data Connector API.

Based on these primary use cases, the Data Connector API has two main purposes: (1) Provide a defined interface to facilitate the integration of a Digital Object Repository within the SCAPE infrastructure. (2) Expose defined REST endpoints to the user (e.g. developer or an application) to ingest, update and retrieve digital objects.

Retrieve an Intellectual Entity

Retrieval of entities is done via a GET request. Specific versions of Intellectual Entities can be requested using an optional version identifier, which when omitted defaults to the most current version of the Intellectual Entity. When successful the response body is a METS representation of the Intellectual Entity. The parameter useReference controls whether the response is created using references to the metadata via <mdRef> elements or if the metadata should be wrapped inside <mdWrap> elements in the METS document.

Path:	***/entity/<id>/<version-id>?useReferences=[yes|no]***
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the requested Intellectual Entity*
	version-id: *the version of the requested entity (optional)*
	useReferences: *Whether to wrap metadata inside <mdWrap> elements or to reference the metadata using<mdref> elements. Defaults to yes.*
Produces:	A XML representation of the requested Intellectual Entity version

[21] http://www.loc.gov/standards/mets/

Content-Type:	text/xml

Retrieve a metadata record

Retrieval of a single metadata record is done via a GET request. Since Intellectual Entities can have multiple versions there is an optional version identifier, which when omitted defaults to the most current version of the Intellectual Entity. When successful the response body is an XML representation of the corresponding metadata record.

Path:	***/metadata/<id>/<version-id>***
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the requested metadata record*
	version-id: *the version of the requested entity (optional)*
Produces:	A XML representation of the requested metadata record according to the corresponding metadata's schema
Content-Type:	text/xml

Retrieve a set of Intellectual Entities

In order to make fetching a whole set of entities feasible this POST method consumes a list of URIs sent in the body of the request. It resolves the URIs to Intellectual Entities and creates a response consisting of the corresponding METS representations. If any URI could not be resolved the implementation returns a HTTP 404 Not Found status message.

Path:	***/entity-list***
Method:	HTTP/1.1 POST
Consumes:	A text/uri-list of the entities to be retrieved
Produces:	METS representations of the requested entities.
Content-Type:	Multipart

Ingest an Intellectual Entity

Ingestion an Intellectual Entity is performed by providing a METS representation the Entity in the body of a HTTP POST request, which gets validated and persisted by the repository. If validation does not succeed, a status message HTTP 415 “Unsupported Media Type” is returned. When successful, the response body is a plain text document consisting of the ingested entity's identifier.

Path:	***/entity***
Method:	HTTP/1.1 POST
Consumes:	An XML representation of the entity
Produces:	The Intellectual Entity identifier
Content-Type:	text/plain

Ingest an Intellectual Entity asynchronously

Sending a METS representation of an Intellectual Entity (a SIP) to this endpoint queues it for ingest. The method returns instantly and supplies the User with an ID, which can be used to request the ingest status.

Path:	***/entity-async***
Method:	HTTP/1.1 POST
Consumes:	An XML representation of the entity
Produces:	An identifier, which can be used to request the ingest status of the digital object, ingested.
Content-Type:	text/plain

Update an Intellectual Entity

In order to allow updating of Intellectual Entities the implementation exposes this HTTP endpoint. The mandatory parameter <id> identifies the Intellectual Entity to be updated. The request must include the updated METS representation of the entity in the request body.

Path:	***/entity/<id>***
Method:	HTTP/1.1 PUT
Parameters:	id: *the id of the Intellectual Entity to update*
Consumes:	A digital object's XML representation.

Retrieve a version list for an Intellectual Entity

To retrieve a list of all versions of an Intellectual Entity a plain GET request can be sent to this endpoint with the <id> parameter indicating which entity's versions to list. If successful the response consists of the Intellectual Entity's version identifiers in an XML representation

Path:	***/entity-version-list/<id>***
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the Intellectual Entity to update*
Produces:	An XML representation of all the entities version ids.
Content-Type:	text/xml

Retrieve a File

For fetching the files associated with Intellectual Entities the implementation exposes a HTTP GET endpoint. Requests sent to this endpoint must have a <id> parameter indicating which File to fetch. The parameter <version-id> indicating the version to fetch is optional and defaults to the most current version of the File. Depending on the Storage Strategy the response body is the binary file with the corresponding Content-Type set by the repository or a HTTP 302 redirect in the case of referenced content.

Path:	*/file/<id>/<version-id>/*
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the requested file*
	version-id: *the version of the Intellectual Entity (optional)*
Produces:	The file requested or a HTTP 302 redirect to the file when using referenced content.
Content-Type:	Depends on File's metadata, but defaults to application/octet-stream.

Retrieve named bit streams

For fetching a named subset of Files, such as an entry in an ARC container, the implementation exposes a HTTP GET method. The mandatory parameter <id> is the identifier of the requested bit stream in the Intellectual Entity. Depending on the Storage Strategy the implementation returns the bit stream directly in the response body, or it redirects the request using HTTP 302 to the referenced content[22].

Path:	*/bitstream/<id>/<version-id>*
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the requested binary content*
	version-id: *the version of the requested bit stream's parent Intellectual Entity (optional)*
Produces:	The binary content associated requested or a redirect to the binary content.
Content-Type:	Depends on content's type, but defaults to application/octet-stream.

Retrieve the lifecycle status of an entity

In order to access the lifecycle status of an Intellectual Entity, without having to fetch the whole METS representation, an endpoint is exposed by the repository.

Path:	*/lifecycle/<id>*
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the Intellectual Entity to get the lifecycle status of*
Produces:	A XML representation of the lifecycle status
Content-Type:	text/xml

[22] Special care is required when using Referenced Content as a Storage Strategy since the implementation is only able to redirect to referenced bit streams, making the redirect target responsible for answering the request properly.

Retrieve a Representation
For fetching single representations, without having to retrieve the METS data of the whole Intellectual Entity, a dedicated endpoint is exposed by the repository.

Path:	***/representation/<id>***
Method:	HTTP/1.1 GET
Parameters:	id: *the id of the requested Representation*
Produces:	A XML representation of the requested Representation
Content-Type:	text/xml

Update a Representation of an Intellectual Entity
For updating a single representation of an Intellectual Entity, without sending a METS representation of the entire Intellectual Entity, an endpoint is exposed by the repository. The repository *has to* create a new Version of the Intellectual Entity with the updated Representation.

Path:	***/representation/<id>***
Method:	HTTP/1.1 PUT
Parameters:	id: *the id of the Representation to update*
Consumes:	A Representations' XML representation.

Update the metadata of an Intellectual Entity
For updating only the metadata of an Intellectual Entity. An endpoint is exposed to clients for updating the metadata of an Intellectual entity that consumes a METS representation of an Intellectual Entity.

Path:	***/metadata/<id>***
Method:	HTTP/1.1 PUT
Parameters:	id: *the id of the Intellectual Entity to update*
Consumes:	An Intellectual Entity's XML representation.

Search Intellectual Entities in a collection
Digital object discovery is performed through an SRU[23] search endpoint. The endpoint implements the SRU specifications by the Library of Congress for Internet Search queries, utilizing CQL[24], a standard syntax for representing queries.

Path:	***/sru/entities***
Method:	HTTP/1.1 GET
Parameters:	See SRU specification

[23] http://www.loc.gov/standards/sru/
[24] http://www.loc.gov/standards/sru/specs/cql.html

Produces:	A XML representation as specified by SRU
Content-Type:	text/xml

Search Representations in a collection
For discovering Representations the implementation exposes a SRU search endpoint.

Path:	***/sru/representations***
Method:	HTTP/1.1 GET
Parameters:	See SRU specification
Produces:	A XML representation as specified by SRU
Content-Type:	text/xml

Search Files in a collection
For discovering Files the implementation exposes a SRU search endpoint.

Path:	***/sru/files***
Method:	HTTP/1.1 GET
Parameters:	See SRU specification
Produces:	A XML representation as specified by SRU
Content-Type:	text/xml

3.3. Report API

The Report API provides the Watch Component, Scout with an adapter for integration with the Platform's Digital Object Repository (DOR). This facilitates retrieval of the relevant information from the DOR. Without this API, Scout would have to create unique adapters for each new repository's internal information structure and naming schemes. The API has been developed as part of the Planning and Watch Sub-project.

Goal of the API
The OAIS reference model structures a digital preservation repository's main activities into macro components, namely: Ingest, Management, Preservation Planning, Data Management, Archival Storage and Access. Two of these are used internally (Data Management and Archival Storage) while the other four usually interact with users. The interactions between these repository components and users are the events exposed via the Report API. Table 1 shows the details of the events that are part of the Report API.

OAIS unit	Event type	Description	Parameters	Parameter Type
Ingest	IngestStarted	Ingest started	SIP ID [1,1]	String
	IngestFinished	Ingest finished	SIP ID [1,1]	String
			Outcome [1,1]	Boolean

			Representation ID [1,∞]	String
Access	ViewDMD	View descriptive metadata	DMD ID [1,1]	String
	ViewRepresentation	View representation	Representation ID [1,1]	String
	DownloadRepresentation	Download a representation	Representation ID [1,1]	String
Planning	PlanExecuted	Plan executed on an Object[25]	Plan ID [1,1]	String
			Object ID [1,∞]	String
			Outcome[26] [1,1]	String

Table 1: Report API event details

Besides the information presented in Table 1, the repository must expose more information describing each event, such as the time the event occurred, and the identity of the agent that triggered the event. Essentially, for each event a repository must be able to answer three questions:

- Who triggered the event?
- When did the event occur?
- What are the details of the event?

Events will have three main attributes that will answer the previous questions:

- **Agent** - will contain information about who triggered the event. Table 2 shows all the sub-attributes of Agent.
- **Date/time** - will contain the date and time at which the event happened and optionally the duration of the event.
- **Details** - will contain additional information about what happened. Table 1 shows all the sub attributes (i.e. Event types) that can be used to characterize the details of an event. Besides the information presented in Table 1, more details can be retrieved for each object involved in the event. For instance, for event IngestFinished, the details about the ingested representations could be retrieved together with the event.

Agent detail attributes		Type	Description
User [0,1]	Role [1,1]	String	The role of the user in the repository.
	Language [0,1]	String - RFC 1766	The language of the user.
Endpoint [1,1]	IP hash [1,1]	String	IP address hash
	Network hash [0,1]	String	Network address hash
	Session ID [0,1]	String	HTTP session identifier

[25] Object refers to any type of PREMIS [1] Object (Representation, File or Bitstream).

[26] Should contain the value of PREMIS Event eventOutcome field (Semantic unit 2.5.1 eventOutcome of PREMIS Data Dictionary [1]).

	Geo IP [0,1]	Country code [0,1]	String	ISO 3166-1[27] alpha-2[28] code
		Country name [0,1]	String	ISO 3166-2[29] country names
		Region name [0,1]	String	ISO 3166-2 region name
		City name [0,1]	String	City name
		Zip code [0,1]	String	Zip code
		Latitude [0,1]	Decimal	Latitude[30] geographic coordinate
		Longitude [0,1]	Decimal	Longitude[31] geographic coordinate
		Time zone	String	Time zone part of ISO 8601[32]
User agent[33] [1,1]	User agent ID [1,1]		String	User agent identifier (e.g. browser User-Agent)
	Language [0,∞]		String - RFC 1766	Supported languages of user agent
	Plugin [0,∞]	Name [1,1]	String	User-Agent plugin name
		Version [1,1]	String	User-Agent plugin version
	OS [0,1]		String	Operating system identifier
	Device [0,1]	Screen height [1,1]	Integer	User device screen height
		Screen width [1,1]	Integer	User device screen width
		Colorspace [1,1]	String	User device screen color space

Table 2: Agent details

Events exposed through the Report API will be encoded in XML following the PREMIS schema[34].

Technical overview

The Report API enables a client to retrieve a list of all the events recorded for the repository, to retrieve a single event, and filter returned events by time and type. There's an established standard in the archive community that provides this functionality, OAI-PMH[35]. OAI-PMH was created to enable repositories to expose their metadata to other parties via a standard API. It establishes a protocol between two entities, the Repository[36] and the Harvester[37]. The Repository holds Items[38],

[27] http://www.iso.org/iso/home/standards/country_codes.htm
[28] http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2
[29] http://en.wikipedia.org/wiki/ISO_3166-2
[30] http://en.wikipedia.org/wiki/Latitude
[31] http://en.wikipedia.org/wiki/Longitude
[32] http://en.wikipedia.org/wiki/ISO_8601
[33] http://en.wikipedia.org/wiki/User_agent
[34] http://www.loc.gov/standards/premis/
[35] http://www.openarchives.org/OAI/openarchivesprotocol.html
[36] http://www.openarchives.org/OAI/openarchivesprotocol.html#Repository
[37] http://www.openarchives.org/OAI/openarchivesprotocol.html#harvester
[38] http://www.openarchives.org/OAI/openarchivesprotocol.html#Item

identified by Unique Identifiers[39], and each Item may have multiple metadata Records[40] associated with it. Each Record held in a Repository may be associated with Sets[41] of Items. The client applications which collect metadata from Repositories, through OAI-PMH requests, are known as Harvesters[42]. In the context of SCAPE, the Harvester will be Scout's repository source adaptor.

OAI-PMH restrictions

OAI-PMH is a protocol that allows a Harvester to retrieve metadata Records from a Provider. In the context of the Report API, a Record is an event and a Set is an event type. This allows Harvesters to retrieve only events of specific types (i.e. events from a Set) using OAI-PMH's selective harvesting feature[43], and is implemented through use of the 'set' parameter in the **ListRecords** and **ListIdentifiers** methods. Although a Set (event type) can have multiple Records (events) belonging to it, since an event can only have one type, a Record (event) can only belong to one Set (event type). This restriction doesn't exist in the OAI-PMH specification, but must be guaranteed by Report API implementers.

Metadata formats

OAI-PMH supports the dissemination of Records in multiple metadata formats from a repository. Using this feature, the Report API can provide information about events with different levels of detail, avoiding problems when an event has too much associated information, making retrieval inefficient. The Report API establishes two metadata formats; one consisting of minimal information, just the PREMIS event with the attributes presented in Table 1 and the agent information; the other consists of the PREMIS event together with all the information associated with it.

The OAI-PMH specification also states that for reasons of interoperability, repositories must disseminate Dublin Core[44]. To comply with the standard, Report API providers must disseminate events in the Dublin Core format.

To accomplish this, implementers must guarantee the following:

- The method **ListMetadataFormats** must return at least three <metadataFormat> elements for the following metadata prefixes: **oai_dc[45], premis-event-v2[46]** and **premis-full-v2[47]**.
- Methods **GetRecord** and **ListRecords**, when provided with parameter metadataPrefix **oai_dc**, Dublin Core metadata must be returned with at least the following elements <identifier>, <type> and <date>. When provided with parameter metadataPrefix **premis-event-v2**, a PREMIS event inside <metadata> element must be returned, together with the PREMIS agent inside <about> element. When the parameter metadataPrefix is **premis-full-v2**, besides returning a PREMIS event and agent, can also return PREMIS objects for representations, files or bit streams inside repeatable <about> elements which are placed after the <about> element for the PREMIS agent.

[39] http://www.openarchives.org/OAI/openarchivesprotocol.html#UniqueIdentifier
[40] http://www.openarchives.org/OAI/openarchivesprotocol.html#Record
[41] http://www.openarchives.org/OAI/openarchivesprotocol.html#Set
[42] http://www.openarchives.org/OAI/openarchivesprotocol.html#harvester
[43] http://www.openarchives.org/OAI/openarchivesprotocol.html#SelectiveHarvestingandSets
[44] http://www.openarchives.org/OAI/openarchivesprotocol.html#MetadataNamespaces
[45] Schema URL for oai_dc is http://www.openarchives.org/OAI/2.0/oai_dc.xsd
[46] Schema URL for premis-event-v2 is http://www.loc.gov/standards/premis/v2/premis.xsd
[47] Schema URL for premis-full-v2 is http://www.loc.gov/standards/premis/v2/premis.xsd

Authentication and authorization

The OAI-PMH protocol doesn't define any authentication or authorization mechanism, but since the protocol is HTTP based, Basic Authentication Scheme[48] can be used together with HTTPS to provide a secure authentication mechanism. For authorization, it's up to the metadata Records' Provider (i.e. the DOR) to implement a suitable mechanism, such as maintaining a list of authorized users, checking the identity of the requester, and returning HTTP error code 401 (Unauthorized) or 403 (Forbidden) in cases where the user is not on the list of authorized users.

3.4. Plan Management API

The Plan Management API is a set of HTTP endpoints for serving content in the SCAPE environment. Its purpose is to integrate the platform's Digital Object Repository with the preservation planning pool Plato.

The plans used by the SCAPE platform are part of a digital objects provenance and need to be stored somewhere; the repository therefore acts as a storage system for preservation plans. The Plan Management API also provides a bridge between the Workflow Execution Environment (part of the SCAPE Execution Platform) and the SCAPE planning agent, relaying execution of plans as requested by the agent and supplying information about preservation plan execution state to the agent.

The Use Cases for the Plan Management API are the following:

- **Deploying new plans in the Repository** - An agent that created a new preservation plan using the Planning and Watch component needs to be able to put the new preservation plan into the repository, in order to have them execute on the workflow execution environment.
- **Get state information about plans** - A Planning agent needs to be able to get state information about a preservation plan from the repository to monitor the plan's execution, as well as for deciding on eligibility of preservation plans for execution.
- **Change existing plans** - If a plan is erroneous or the set of parameters for a given preservation plan has to be changed, it has to be updated. Because the provenance of a digital object may reference the old plan, a versioning system for is required.
- **Get a specific plan based on some criteria** - In the simplest case the agent has a preservation plan identifier and wants to fetch this plan, for example, for manipulation using Plato. In more complex cases, the agent might want to search for plans based on some plan properties like a description.
- **Reservation of Identifiers** - In order to create new Plans the agent has to be aware of the identifier of the plan. Therefore a facility to request a reserved identifier is required, which is used when the preservation plan gets deployed on the repository.
- **Execute a preservation plan on the Workflow Execution Environment (the Execution Platform)** - An agent requires the ability to trigger plan execution on the Workflow Execution Environment.

The Plan Management API is a RESTful[49] API enables DORs to incorporate preservation plans that conform to the Plato plan schema[50]. The Plan Management API provides the following REST endpoints:

[48] http://tools.ietf.org/html/rfc1945#section-11.1
[49] http://en.wikipedia.org/wiki/Representational_state_transfer

Retrieve a plan

The repository exposes an endpoint for plan retrieval. Plans are returned according to the Plato Version 4 XML schema that is currently in development.

Path:	***/plan/<id>***
Method:	HTTP/1.1 GET
Parameter:	id: *the id of the plan to be fetched from the repository*
Produces:	A XML representation of the plan

Deploy a new plan

An endpoint for deployment of new plans in the repository is exposed via HTTP PUT. An agent can upload new preservation plans for later use or reference by sending a request structured as follows.

Path:	***/plan/<id>***
Method:	HTTP/1.1 PUT
Consumes:	A XML representation of the plan

Search plans

In order to be able to search plans based on their significant properties an endpoint for the repository exposing SRU searching is provided. The endpoint implements the SRU specifications by the Library of Congress for Internet Search queries, utilizing CQL, a standard syntax for representing queries, and exposes this functionality via a HTTP GET endpoint. Pagination is done via the SRU parameters startRecord and maximumRecords. A level 1 implementation, according to the SRU standard, is implemented in order to accommodate various use cases.

Path:	***/sru/***
Method:	HTTP/1.1 GET
Parameter:	see SRU specification
Produces:	A URI list referencing the plans found by the search request

Retrieve plan execution states

In order to supply an Agent with feedback about the execution of preservation plans, the repository exposes an Endpoint for retrieving lists of execution states.

Path:	***/plan-execution-state/<id>***
Method:	HTTP/1.1 GET
Parameter:	id: *the id of the preservation plan*

[50] https://github.com/openplanets/plato/blob/master/planning-core/src/main/resources/data/schemas/plato.xsd

Produces:	A XML representation of all the execution states associated with a preservation plan.

Add a plan execution state

The workflow execution environment needs the possibility to add new execution states to preservation plans, in order for this state to be available to the Plan Management GUI user.

Path:	***/plan-execution-state/<id>***
Method:	HTTP/1.1 POST
Parameter:	id: *the id of the plan for which the execution state should be updated*
Consumes:	A XML representation of the execution state

Update plan lifecycle status

An agent requires the possibility to change the life cycle state of a preservation plan to enable and disable specific preservation plans. Only enabled plans can be executed on the Workflow Execution Environment.

Path:	***/plan-state/<id>/<state>***
Method:	HTTP/1.1 PUT
Parameter:	id: *the id of the plan which is to be updated*
	state: *the new state of the plan, one of {ENABLED, DISABLED}*

Retrieve a reserved plan identifier

An agent can request a preservation plan identifier, which gets reserved by the repository. The reserved identifier has timely limited validity, so that a preservation plan using the reserved identifier has to be deployed within a certain time frame or the identifier will be invalidated and unusable.

Path:	***/plan-id/reserve***
Method:	HTTP/1.1 GET
Produces:	A XML representation of the Identifier reserved for up to 24h.

3.5. Component Registration and Lookup API

The SCAPE Component Catalogue extends the myExperiment REST API[51] with an endpoint that provides semantic query functionality for SCAPE components.

Authentication

The myExperiment REST API uses HTTP basic authentication through which the credentials of the user's myExperiment account can be used.

Querying Components

This endpoint returns a list of components in response to a HTTP GET request with a set of parameters detailing required properties. Parameters are passed as variables representing workflow features (an input port, output port, processor, or the workflow itself), followed by one or more predicate-object pairs. Parameters have an index number, to differentiate between multiple inputs, outputs, etc.

Path:	***/components.xml***
Method:	HTTP/1.1 GET
Parameters:	input[x]: *Match components that have an input with the given semantic annotations. "x" is an index, and multiple inputs can be specified by using different numeric values for x.* output[x]: *As above, but for component outputs.* processor[x]: *As above, but for component processors.*
Produces:	A list of URIs of components that match the given criteria.
Content-Type:	text/xml

Examples:

```
/components.xml?input[0]="http://purl.org/DP/components#portType
http://purl.org/DP/components #FromURIPort"
```

This will find all components that have an input port annotated with http://purl.org/DP/components #portType http://purl.org/DP/components#FromURIPort.

To specify multiple annotations for a single workflow feature, add additional pairs separated by a comma, e.g.

```
/components.xml?input[0]="<uri1> <uri2>","<uri3> <uri4>"
```

Multiple parameters can be specified, separated by an ampersand (&). If multiple parameters for the same type of feature (e.g. "input") are required, give each one a distinct index, e.g.

```
/components.xml?input[0]=...&input[1]=...&input[2]=...&output[0]=...
```

[51] http://wiki.myexperiment.org/index.php/Developer:API

4. Applications

This section describes the parts of the Platform that interact directly with users. Specifically these are the Taverna Workflow Modelling Environment (Section 0), the Component Catalogue (Section 4.2), the Loader Application (Section 4.3), the Plan Management Component (Section 4.4), and the Workflow Compiler (Section 4.5).

4.1. Component Support by the Workflow Modelling Environment

SCAPE Components are implemented as Taverna workflows that follow a set of conventions. The Taverna workflow-modelling environment has been extended to provide graphical support for creating, validating and registering SCAPE preservation components. Workflow components allow users to easily compose complex Workflows by combining SCAPE components. SCAPE components are registered with the SCAPE preservation catalogue. The Taverna Workbench is used to execute workflows while in the development and testing phases. When the workflow is ready for production use on the Platform, it is published to the workflow repository. A SCAPE component can be parallelized for example by using the workflow compiler described in section 4.5, and subsequently installed on the SCAPE Execution Platform.

SCAPE Components are small workflows that perform specific tasks and conform to a component profile. A number of component profiles are defined based on the performed preservation action, e.g. migration, characterisation, or quality assurance. The component profile specifies the component interface (input and output ports) as well as supported annotations (such as dependencies on external tools). In order to construct a component that conforms to the component profile, the Taverna Workbench allows semantic annotations to be added to the corresponding workflow. The SCAPE ontology for preservation components which is used to annotate SCAPE components, is outlined described below. An annotation might describe the file format migration paths supported by a particular file migration component.

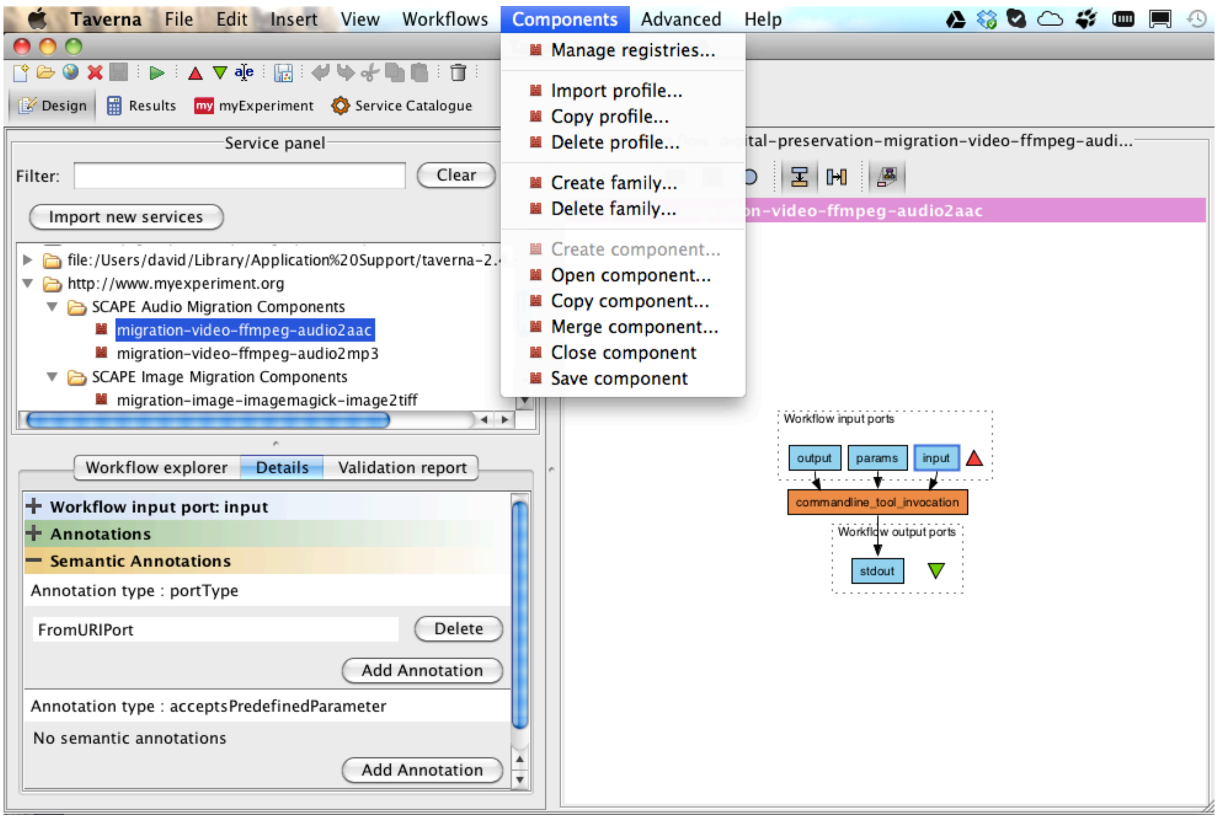


Figure 5 Component support in the Taverna Workbench.

Component Support in Taverna Workbench has been developed as a Taverna plug-in module shown in *Figure 5*. Using the Workbench, components can be saved to either a local or remote registry. The local component registry is stored on the user file system and is used while the component is in development and testing. When the component is ready for publication it can be added to the remote component registry (the SCAPE Component Catalogue) and made available to the SCAPE community. SCAPE components can use Taverna's Tool activity to execute preservation tools that are available on the Platform and include semantic annotations to specify the required tools. Using these semantic annotations, a SCAPE component's dependencies can be checked on the Platform to ensure the preservation tools required to execute the workflow have been installed.

SCAPE Component profiles[52] define the interface and required metadata for each type of component as an XML document conforming to the Component Profile schema[53]. The intension is to enable composition and interoperability among SCAPE components and provide the necessary metadata for discovery and execution. Taverna supports the annotation of workflows and validation of components against these profiles through its component support. Metadata is added to Taverna workflows by creating free-text based annotations for simple properties and semantic annotations based on OWL ontology[54] for complex data.

[52] https://github.com/openplanets/scape-component-profiles/tree/master/profiles
[53] http://ns.taverna.org.uk/2012/component/profile/ComponentProfile.xsd
[54] http://purl.org/DP/components

Example 1: For input and output ports, a profile defines the cardinality, as well as the free-text and semantic annotations that must be defined (for example port type and accepted parameters).

```
<inputPort maxDepth="0" minOccurs="0" maxOccurs="unbounded">
        <annotation>Example</annotation>
        <annotation>Description</annotation>
        <semanticAnnotation ontology="components"
                predicate="http://purl.org/DP/components#portType"
                class="http://purl.org/DP/components#PortType">
                http://purl.org/DP/components#ParameterPort
        </semanticAnnotation>
        <semanticAnnotation ontology="components"
                predicate="http://purl.org/DP/components#acceptsPredefinedParameter"
                class="http://purl.org/DP/components#PredefinedParameter"
                minOccurs="1" maxOccurs="unbounded" />
</inputPort>
```

Figure 6 Snippet of the migration action profile showing an input port for parameters

Example 2: If a component calls an external command line tool, that tool dependency must be specified as a semantic annotation, as shown in Figure 7.

```
<activity type="Tool" minOccurs="0" maxOccurs="unbounded">
        <semanticAnnotation ontology="components"
                predicate="http://purl.org/DP/components#hasDependency"
                class="http://purl.org/DP/components#Dependency"
                minOccurs="1" maxOccurs="unbounded" />
</activity>
```

Figure 7 Snippet of the migration action profile showing an external tool activity

Example 3: Supported migration paths of a migration action component.

```
<annotation>Title</annotation>
<annotation>Description</annotation>
<annotation minOccurs="0">Author</annotation>
<semanticAnnotation ontology="components"
        predicate="http://purl.org/DP/components#migrates"
        class="http://purl.org/DP/components#MigrationPath"
        maxOccurs="unbounded" />
<semanticAnnotation ontology="components"
        predicate="http://purl.org/DP/components#fits"
        class="http://purl.org/DP/components#Profile">
        http://purl.org/DP/components#MigrationAction
</semanticAnnotation>
```

Figure 8 Snippet of migration action profile showing general required metadata

SCAPE Component Ontology: The SCAPE component profiles reference the SCAPE component ontology[55], which has been developed in the context of the SCAPE Planning and Watch Sub-project. The Component Ontology is shown in *Figure 9* below:

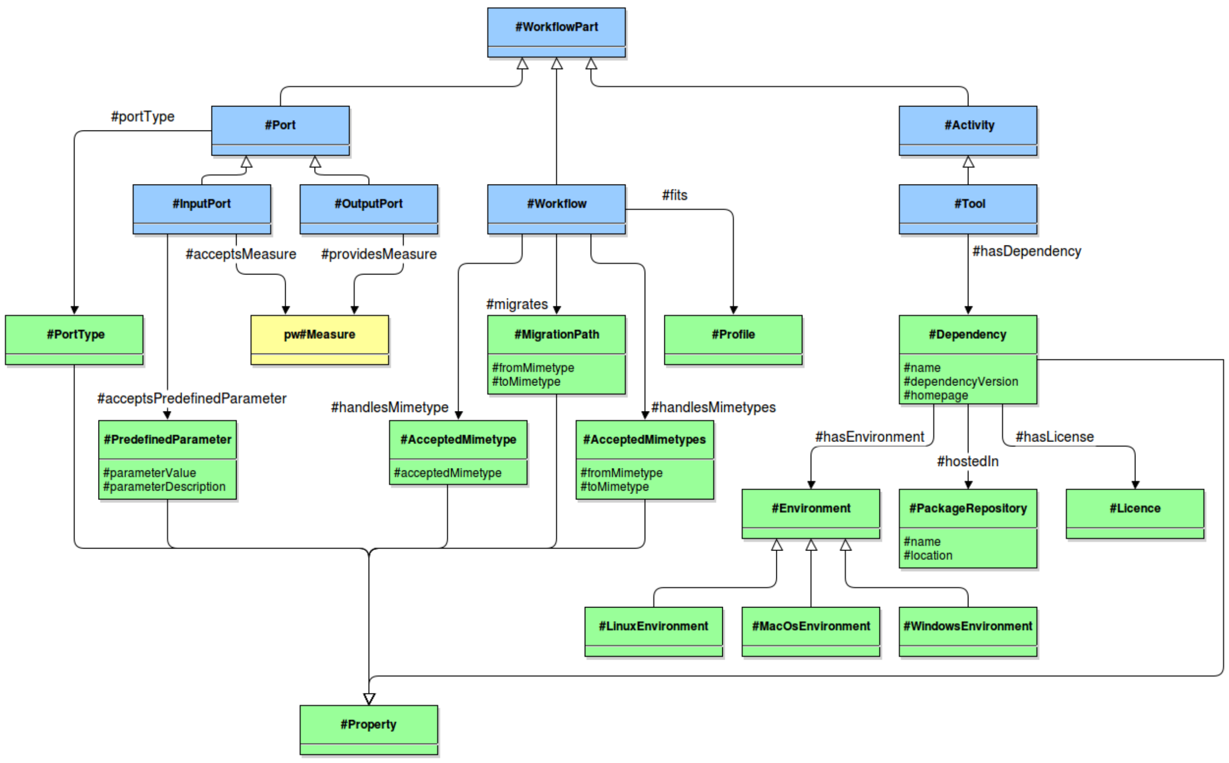


Figure 9 Overview of the SCAPE component profile ontology

4.2. SCAPE Component Catalogue

Components can be published to the SCAPE Component Catalogue from the Taverna Workbench using the MyExperiment Component Registration API. The SCAPE Component Catalogue adds support for SCAPE component profiles and provides an endpoint for advanced search functionality based on semantic descriptions, as described in Section 0.

Components are grouped into Component Families. These are collections of components that share the same component profile and perform similar functions. For example, in SCAPE we may a create component family for Image Migration and another for Audio Migration, both based on the Migration Component Profile. Component families may also contain documentation common to all components in the family. Component Families can be created and managed from the Taverna Workbench.

[55] https://github.com/openplanets/policies

The Taverna Workbench can discover components published to SCAPE Component Catalogue and displays them in the service panel, allowing them to be placed into workflows in the same was as other Taverna activities.

4.3. Loader Application

Ingesting a large amount of data into a repository should be handled by a loader application that is capable of keeping tracking the progress of the ingestion process, reading data from the file system and creating reports about the loading progress. The SCAPE loader application is designed to be used by any repository that exposes the SCAPE Data Connector API (see Section 0) and releases the burden of a repository provider to implement a specific Loader Application for their repository.

The loader application's input is known as a SIP (Submission Information Package) using OAIS terms[56]. SIP files must be generated either manually or via a specific SIP Generator, which is presently contained with the Loader Application. A SIP is comprised of the metadata and the content of a digital object. The metadata of a digital object is described by a METS[57] profile which can be found in a separate document describing the SCAPE Digital Object Model document[58]. METS files the source information for the loader application. METS files are picked up by the loader from a source location and registered for ingested into the repository. Alternatively, METS files can be stored in a Hadoop Sequence File (that may reside on HDFS or on the local file system) which the loader application can use to ingest the files into the repository. The command line interface of the Loader Application offers a way to configure the path of the source directory, as well as the REST endpoints of the repository to be used. A more enhanced graphical user interface may be developed if required, but is not part of the reference implementation. A command line interface summary is given below:

```
usage: java -jar loader-app-0.0.1-SNAPSHOT-jar-with-dependencies.jar [-c
       <arg>] [-d <arg>] [-h] [-i <arg>] [-l <arg>] [-p <arg>] [-r <arg>]
       [-t <arg>] [-u <arg>]
 -c,--checklifecycle <arg>   activate the periodic lifecycle retrieval.
                             [default: true]
 -d,--dir <arg>              Local input directory (Required). If a
                             sequence file is given, an extraction into a
                             local sips directory will be performed
 -h,--help                   print this message.
 -i,--ingest <arg>           ingest REST endpoint [default: entity-async].
 -l,--lifecycle <arg>        lifecycle REST endpoint [default: lifecycle].
 -p,--password <arg>         password of the repository user.
 -r,--url <arg>              base URL of the repository (Required).
 -t,--period <arg>           Period in min to fetch lifecycle states
                             [default: 1 min]
 -u,--username <arg>         username of the repository user.
```

Figure 10: Overview of the SCAPE component profile ontology

The binary files, like jpg, pdf, wav, mp3 etc., will not be directly ingested into the repository, merely referenced in the METS description of each intellectual entity; Neither is it the Loader Applications

[56] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=24683
[57] http://www.loc.gov/standards/mets/
[58] The document has not been made publicly available at the time writing this deliverable.

responsibility to copy binary files to a storage device, but to deal with the correct ingestion of the intellectual entities into the repository. As such, it is not part of the Loader Application to take care of copying binary files to a storage device, but to deal with the correct ingestion of the intellectual entities into the repository.

The Loader Application offers three distinct services that are as follows:

1. 'Add': The SIPs gets added to the Loader Applications registry (e.g. HSQLDB) while reading a source directory
2. 'Ingest': for each SIP a POST request will be triggered to ingest the digital object asynchronously into the repository
3. 'Get State': this service retrieves the status of the SIP during the ingest process. The state gets recorded in the registry of the Loader Application so the user is kept up-to-date about the progress.

The Loader Application implements the following ingest process, as described by the sequence diagram in Figure 11:

1. Read the file URIs of the SIPs and register them in the data store of the Loader Application.
2. Successfully authenticate the Loader Application against the repository
3. Ingest the digital object (as METS) into the repository by using the asynchronous REST endpoint of the Data Connector API (section 3.2).
4. Retrieve the SIP ID of each digital object and store the ID in the Loader Applications registry.
5. Retrieve the lifecycle state of the ingested objects on a regular basis (configurable).
6. Update the registry with the status of the object.
7. Create a summary of the status of the ingested objects.

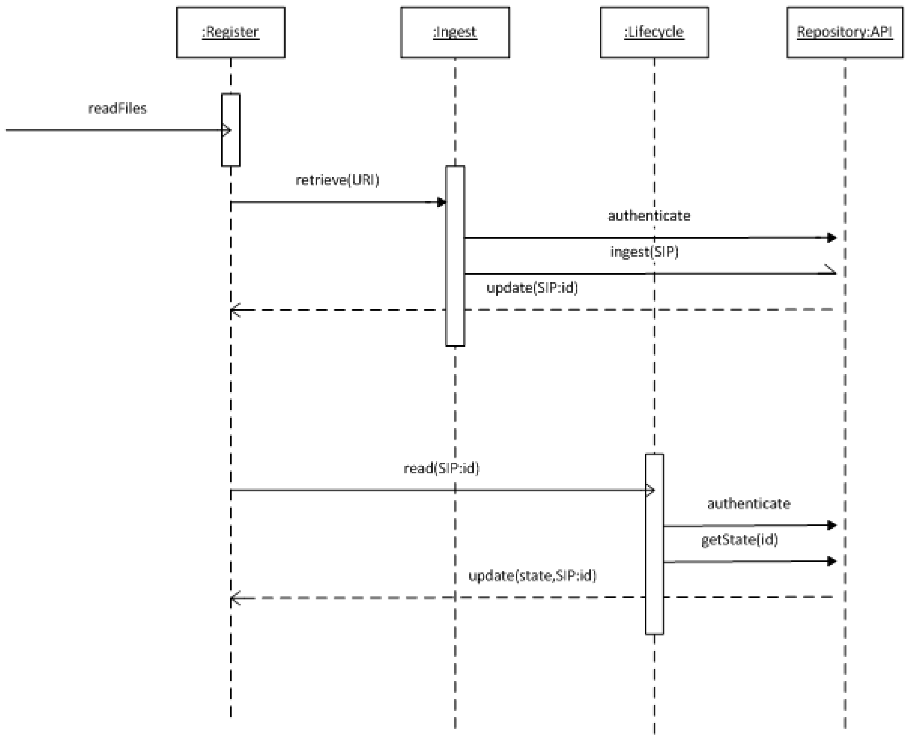
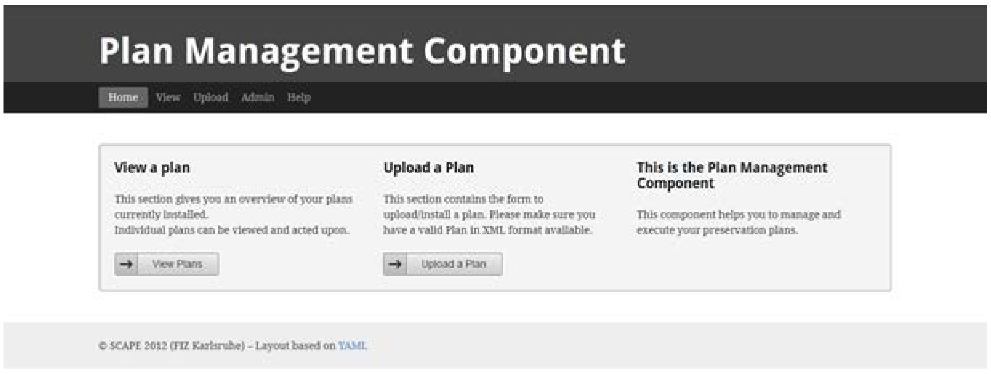


Figure 11 Sequence diagram of the ingest process via the Loader Application.

4.4. Plan Management Component

The Plan Management Component is implemented by the Digital Object Repository (DOR) and enables the management and execution of preservation plans. Using the Plan Management Components, Preservation Plans developed using the preservation planning tool developed in SCAPE can be stored within the Digital Object Repository, as they are a part of the provenance information of a digital object. The Plan Management Component also implements a client application to the Platform's Job Submission Services (section 3.1) in order to trigger the execution of a particular preservation plan.



View a plan

This section gives you an overview of your plans currently installed.
Individual plans can be viewed and acted upon.

Upload a Plan

This section contains the form to upload/install a plan. Please make sure you have a valid Plan in XML format available.

This is the Plan Management Component

This component helps you to manage and execute your preservation plans.

© SCAPE 2012 (FIZ Karlsruhe) – Layout based on YAML

Figure 12 Design of the Plan Management Component user interface.

The Plan Management Component provides a user interface to upload, manage and submit a plan for execution, as shown in Figure 12. It may be a separate client to the repository, or the UI can be a part of the repository itself, and interfaces with the Plan Management API as described in section 22.

4.5. Workflow Compiler

The Platform's workflow compiler provides a general mechanism and tool for translating complex preservation workflows into applications that can execute in parallel over massive volumes of data on the Execution Platform. This tool will be a driver program, called PPL (for "Program for parallel Preservation Load"), for the optimized execution of parallel algorithms for digital preservation actions.

In order to enable scientists to easily scale workflows created with Taverna, the workflow translator automatically generates a Java file that can be uploaded and executed on Hadoop-based execution environments (section 2.1, i.e. SCAPE's Execution Platform).

The resulting parallel program consists of a linear list of MapReduce jobs produced from the input workflow, which can be arbitrarily complex. The required input for each individual job is either read locally (as provided by the Hadoop framework), or is fetched from other machines in the Hadoop distributed file system (HDFS), if required.

Figure 13 shows an overview of the PPL architecture. First, the SCUFL2 API[59] is used to read a given workflow, which has been authored using the Taverna workflow design application. The workflow is subsequently translated into a linear list in order to have the data output of each node available to successive nodes. In the following step, each node of the Taverna graph is translated into a MapReduce compliant code fragment using a corresponding template. Finally, the Java compiler generates a JAR, which can be deployed on the Hadoop cluster.

The use of templates allows easy extension of the PPL, e.g. to support plug-in activities. Templates are written as Java source code templates with placeholders. In general, three templates are associated with each Taverna activity: map, reduce, and run templates. The map and reduce templates specify the user defined functions (UDFs) map and reduce, while run defines how they are executed (job configuration, file formats, etc.). Furthermore, there needs to be an implementation of the abstract class taverna_to_hadoop.convert.activity_configs.**ActivityConfig**. The class should be called **<Activityname>Config** and it implements the translation of the template into Java source code for the creation of the Hadoop JAR. The PPL will automatically search for classes using the name for every activity it finds in the workflow.

Taverna activities allow for multiple in- and outputs. Thus, the Hadoop jobs resulting from the translation also need to read from and write to multiple HDFS paths. Hadoop supports reading multiple inputs for a job. In order to write to multiple locations from within a UDF, Hadoop supplies the org.apache.hadoop.mapreduce.lib.output.**MultipleOutputs** class.

At the moment, every activity is translated into its own Hadoop job, and the final JAR executes these jobs in the specified order. A future optimization might be the grouping of multiple activities into a single MapReduce job.

[59] https://github.com/mygrid/scufl2

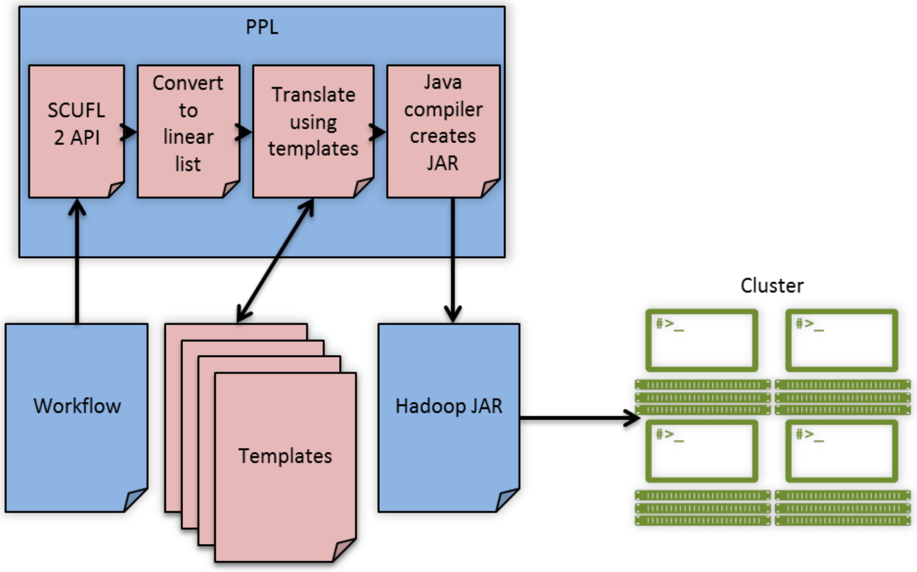


Figure 13 Overview of the PPL architecture

5. Conclusion

The architecture of the SCAPE Preservation Platform aims at a versatile design applicable to digital content from many domains and to different preservation and information management systems.

This document describes the system architecture from different viewpoints and provides insights on its design and implementation in the context of the SCAPE project. We introduce a design that structures the architecture in different system layers. The document subsequently describes the underlying technologies as well as the main system entities that make up the platform architecture. Furthermore, the services introduced by the SCAPE Platform are described and an overview of their HTTP interfaces has been given. The document provides also an overview on the applications that are being developed in the context of the SCAPE Platform Sub-project.

We note that the SCAPE Platform architecture does not prescribe a specific deployment or infrastructure provisioning model. The system may be set-up using a private or institutionally shared hardware infrastructure, or be hosted at an external data center. The architecture also supports virtualization and can be deployed using private or public cloud infrastructure.

Concluding, we would like to emphasise that this document summarizes the perception of the architecture of the SCAPE Preservation Platform after the second project year. We expect that things will be learned during the on-going implementation and integration work, which will lead to future improvements and refinements of the platform architecture, its components, and services. It is therefore planned that this document will be continuously improved and carried forward throughout coming project months.

Appendix A USE-CASES

The following section provides a general list of use-cases that are supported by the SCAPE Platform architecture. Use-cases are grouped into five categories pertaining component and workflow management, deployment on the execution platform, data staging, digital object management, and job execution.

Component and Workflow Creation, Registration, and Compilation

- User creates and validates a SCAPE component (i.e. a workflow) using the workflow modelling environment. This includes description of interfaces, (application) dependencies, and semantic information.
- User registers SCAPE component using SCAPE Component Catalogue.
- User retrieves components from SCAPE Component Catalogue and imports them into workflow modelling environment.
- User composes and configures specific preservation scenario using the workflow modelling environment.

Deployment of Tools and Components on the Cluster

- Cluster administrator updates (sequential) preservation tools on cluster using package manager and SCAPE package repository.
- User/Administrator deploys and registers a component on the cluster. (This involves the generation of a parallel application from a workflow using PPL compiler and verification that all dependencies are met. Registration is a prerequisite for instantiating the application using the Job Submission Service. The cluster maintains a very simple registry for deployed components. The SCAPE Component Catalogue maintains more detailed information.)
- List all deployed preservation tools.
- List all supported SCAPE component.

Data Staging

- User copies data on the cluster e.g. by manually using the HDFS shell or the DistCp tool. It is not planned to develop a SCAPE service for data staging.
- A user deletes data from the cluster or moves it to another storage media. Similar as above. R/W access can be controlled using home directories per user and access permissions on HDFS and/or HBase.

Digital Object Management

- User ingests digital objects into the repository using the Loader Application. Objects typically reference content, which might reside on HDFS, a NAS, or a data staging area.
- A component (i.e. a parallel application) processes content that is maintained by a digital object repository. This requires the Execution Platform to translate repository references to file references using the Connector API. (SCAPE Digital Objects are metadata constructs that only reference content. If the export of content to the cluster is required in order to execute a workflow, this has to be ensured before workflow execution in a separate step.)

- A component retrieves, processes and updates a large number of digital objects (e.g. for an analysis task). This might require a method to export digital object metadata from a SCAPE repository and /or to create an input file on HDFS.
- PW will monitor repository actions (e.g. ingestion, access of objects) using the Report API implemented by the repository. Detailed information about individual objects can be fetched via the repositories Connector API.

Job Execution

- A user creates a job description (component/workflow, resource requirements, wall clock time limit, job queue, QoS constraints, etc.).
- A user invokes a job on the cluster via the command-line. Will be supported using scripts that are aware of scape abstractions like SCAPE components, workflows, HDFS and repository resolvable references. Data might be supplied by any kind of references (object identifiers, HDFS/file references).
- A client application (like the DOR's Plan Management Interface) triggers a job execution via the Job Submission Service and monitors status. Data might be supplied by any kind of references (object identifiers, HDFS/file references).

Appendix B CLOUD DEPLOYMENT

The Austrian Institute of Technology (AIT) has deployed a test instance of the SCAPE Preservation Platform within a private cloud environment. The SCAPE infrastructure provides a Fully Automated Installation (FAI) server for configuring the cloud nodes. FAI is an automated installation framework that can be used to install Debian systems on a cluster. The service allows a system administrator to easily add new nodes to the system, which can be booted via a network card using PXE, a pre-boot execution environment most modern network cards support.

The cloud infrastructure, presently consisting of 20 nodes, has been set up using the Eucalyptus cloud software stack. Eucalyptus is a private cloud-computing platform that provides REST and SOAP interfaces which are compliant with Amazon's EC2, S3, and EBS services. The infrastructure's front-end hosts the Eucalyptus Cloud Controller, the Cluster Controller, and the Walrus storage service. The worker nodes in the cloud run the XEN hypervisor and a Debian distribution that includes a Xen Dom0 kernel.

The Execution Platform is based on an Apache Hadoop cluster running MapReduce and HDFS. Preservation tools are automatically installed on the individual nodes using Debian package management system. Using the cloud environment, a platform instance can be brought up dynamically in different configurations by specifying a particular virtual machine image and the desired size of the cluster. Data on each virtual cluster node can be stored persistently, as each cloud node is configured with a physical data partition that can be mounted to the file system of a virtual machine instance.

The platform nodes utilize this mechanism to establish a distributed file system that uses physical file system partitions underneath. Since data is already replicated by the Hadoop file system, it is not required to employ additional data redundancy mechanisms like RAID.

Appendix C GLOSSARY

API	Application Programming Interface
Cluster	A network of tightly coupled computer nodes used by the Platform exclusively for data storage and computation.
DAG	Directed Acyclic Graph
DOR	Digital Object Repository
METS	The Metadata Transmission and Encoding Standard
OAIS	The Open Archival Information System Reference Model
PREMIS	The PREMIS Data Dictionary for Preservation Metadata
PPL compiler	PPL (Program for Parallel Preservation Load) is a software program for conversion of a SCAPE component into a parallel application.
Parallel application	A parallelized SCAPE component that can be understood by the execution platform (e.g. a Taverna workflow transformed into one or more MapReduce Jobs)
Preservation tool	A packaged piece of software that can be deployed with a package manager on the cluster (e.g. for mime type detection)
RDF	The RDF specification provides a standard model for data interchange on the Web, which is maintained by the World Wide Web Consortium (W3C).
REST	A software architecture for distributed systems such as implemented by for the World Wide Web using the HTTP protocol.
SCAPE Component	A Taverna workflow that adheres to a specific component profile and includes semantic information as defined by the SCAPE ontology. SCAPE components are registered in the SCAPE components catalogue.