SEVENTH FRAMEWORK PROGRAMME

ForgetIT

www.forgetit-project.eu

ForgetIT

Concise Preservation by Combining Managed Forgetting and Contextualized Remembering

Grant Agreement No. 600826

Deliverable D8.6

Work-package	WP8: The Preserve-or-Forget Reference Model and Framework
Deliverable	D8.6: The Preserve-or-Forget Framework – Final Release
Deliverable Leader	Francesco Gallo (EURIX)
Quality Assessor	Mark Greenwood (USFD)
Dissemination level	PU
Delivery date in Annex I	M36 (January 2016)
Actual delivery date	31 March 2016
Revisions	16
Status	Final Version
Keywords	Preserve-or-Forget Framework, Integrated Prototype and Components

Disclaimer

This document contains material, which is under copyright of individual or several ForgetIT consortium parties, and no copying or distributing, in any form or by any means, is allowed without the prior written agreement of the owner of the property rights.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the ForgetIT consortium as a whole, nor individual parties of the ForgetIT consortium warrant that the information contained in this document is suitable for use, nor that the use of the information is free from risk, and accepts no liability for loss or damage suffered by any person using this information.

This document reflects only the authors' view. The European Community is not liable for any use that may be made of the information contained herein.

© 2013-2016 Participants in the ForgetIT Project

Revision History

Version	Major changes	Authors
0.01	Document skeleton, ToC created	EURIX
0.02	Updated document structure, preliminary introduction, PoF architecture	EURIX
0.03	Added PoF Reference Model, sub-sections for each middleware component	EURIX
0.04	Described middleware implementation, REST APIs and CMIS, Preservation System, updated architecture	IBM, EURIX
0.05	Described preservation preparation and re-activation workflows, Digital Repository, updated appendices	EURIX, LUH, USFD, DFKI, IBM
0.06	Added information model, updated middleware REST APIs, preliminary sections about Semantic Desktop and TYPO3	EURIX, DFKI, dkd, LUH
0.07	Added ID Manager, Scheduler, Extractor, Collector/Archiver	LTU, CERTH, LUH, EURIX
0.08	Added Condensator and Contextualizer, updated previous components, introduction, preliminary executive summary	CERTH, USFD, EURIX
0.09	Completed middleware components, added prototype implementation	LTU, USFD, EURIX
0.10	Reviewed all sections, added software development, appendix for Forgettor APIs, added glossary	EURIX, LUH, DFKI, dkd
0.11	Completed all sections, updated references, first draft version circulated for comments	EURIX
0.12	Reviewed references and glossary, created executive summary	EURIX
0.13	First complete version for project wide review	EURIX
0.14	Implemented comments from all partners, version for internal QA	EURIX
0.15	Implemented comments from internal QA, final version	EURIX
1.00	Submitted version	EURIX

List of Authors

Partner Acronym	Authors
LUH	Andrea Ceroni, Tuan Tran
LTU	Ingemar Andersson
IBM	Doron Chen
DFKI	Heiko Maus, Andreas Lauer, Sven Schwarz
CERTH	Olga Papadopoulou, Evlampios Apostolidis, Alexandros Pournaras, Chrysa Collyda, Vasileios Mezaris
dkd	Johannes Goslar
USFD	Mark A. Greenwood
EURIX	Martina Fogliati, Francesco Gallo

Table of Contents

Executive Summary 8

1 Introduction 9

2 PoF Framework Architecture 12

3 PoF Reference Model 15

3.1 Implementation of the Reference Model 17

4 PoF Middleware 22

4.1 PoF Enterprise Service Bus 22

4.1.1 Message-Oriented Middleware 22

4.1.2 Enterprise Integration Patterns 24

4.1.3 Asynchronous Routing Engine 25

4.1.4 PoF ESB Implementation 26

4.2 Middleware Configuration 27

4.3 RESTful Service 28

4.4 CMIS Integration 28

5 PoF Middleware Integrated Components 31

5.1 ID Manager 31

5.2 Metadata Repository 34

5.3 Scheduler 35

5.4 Extractor 38

5.5 Condensator 41

5.6 Collector/Archiver 42

5.7 Forgettor 44

5.8 Contextualizer 48

5.9 Navigator 49

5.10 Context-aware Preservation Manager 50

6 Active Systems 53

6.1 Semantic Desktop 53

6.2 TYPO3 54

6.3 CMIS-based User Applications 55

7 Preservation System 57

7.1 Digital Repository 57

7.2 Preservation-aware Storage System 59

8 Third Prototype Implementation 61

9 Conclusion 67

9.1 Assessment of Performance Indicators 67

9.1.1 Evaluation of the PoF Framework 69

9.2 Lessons Learned 70

9.3 Vision for the Future 70

10 References 72

Glossary 76

A Middleware Configuration and Administration 77

B Preserve-or-Forget RESTful Service 86

C DSpace Installation and Configuration 89

C.1 Introduction 89

C.2 Installation Procedure 89

C.3 DSpace REST API 95

C.4 Administration and Users Permissions 99

C.5 Import and Export 100

C.6 Versioning and Other Features 102
C.7 AntiVirus in DSpace: ClamAV 103
C.8 Curation Tasks 104
C.9 Cloud Storage 106
C.10 Replication Suite 111
C.11 Customized Cloud Features: ownCloud 114

D Implementation of Reference Model Workflows 118
D.1 Preservation Preparation Workflow 118
D.2 Re-activation Workflow 123

E Experimental APIs of the Memory Buoyancy Assessor 125

Executive Summary

This document describes the Preserve-or-Forget (PoF) Framework, discussing the implementation of the prototype and the integrated components. In this deliverable we present the final release of the framework, developed during the third year of the project and based on the second release, described in deliverable D8.4.

The framework prototype is based on the architecture and integration plan defined in D8.1, integrates the components developed in the technical WPs and provides a foundation for application pilot development in WP9 and WP10. The PoF Framework is made up of the Active Systems (information management systems), the PoF Middleware (implementing core ForgetIT principles) and the Preservation System.

The reference workflows defined in the final version of the PoF Reference Model in D8.5 have been used for the development of the framework. Currently two workflows have been implemented and integrated, for preservation preparation and re-activation. The other workflows for the evolutionary part of the model have been partially implemented in the final framework release or in other components developed in the technical WPs. Compared to the second prototype, the final release also implements the PoF information model which was not available after the second year.

The PoF Middleware REST APIs, defined in D8.1, have been updated with respect to the second release. For data exchange between the Active Systems and the PoF Middleware we leverage the OASIS CMIS standard. The PoF Middleware has been implemented as a Message Oriented Middleware (MOM) and on top of the messaging layer we added a rule-based routing engine for workflow management. The implementation based on Apache ActiveMQ and Apache Camel is described. Further improvements to the workflow management are also outlined, for example those related to the use of Enterprise Integration Patterns (EIP) or to the integration of additional ESB components on top of the existing solution. For the middleware components identified in D8.1, either providing common tasks or implementing core ForgetIT functionality, we provide information about the status and their integration in the third release.

Concerning the Preservation System, we describe the two main components, the Digital Repository and the Preservation-aware Storage System, based on cloud technologies. Both systems implement the archive functionality for the preservation of ForgetIT content. The APIs exposed by the Preservation System are discussed and the implementation using DSpace and OpenStack Swift is described. We also describe how Storlets are involved in the current workflow.

Finally we provide additional information about the software development process and the collaborative tools, as well as preliminary considerations about the license for the core components of the PoF Framework. The software documentation for the PoF Middleware and the Preservation System APIs are available on the project web site.

We decided to include here also other parts from D8.4 which were not affected by the update during the third year development, in order to provide a self-contained document.

1 Introduction

The main topic of this document is the description of the final prototype implementation of the Preserve-or-Forget (PoF) Framework, which integrates the results achieved during the whole project lifetime and is based on the final release of the PoF Reference Model reported in deliverable D8.5 [Gallo et al., 2016]. This deliverable consists of the description of the prototype which is running in the ForgetIT testbed environment hosted by EURIX (see Section 6 in deliverable D8.1 [Gallo et al., 2013]).

The PoF Framework provides an integration framework for all available components and is based on the ForgetIT architecture described in deliverable D8.1, where an overview of the architecture layers and the main components are included. The framework is used to validate the basic workflows for the three core ForgetIT principles: *managed forgetting*, *contextualized remembering* and *synergetic preservation*. More specifically, the final prototype implements the relevant workflows of the functional part of the model, focusing on the Core and Remember & Forget Layers. The final release also implements the information model, including support for Situations, Collections and Items.

Continuing with the same approach adopted for the first two prototypes, the final prototype was built on top of the integrated components, keeping the original approach based on open and widely adopted technologies, with several improvements in term of flexibility and number of integrated components.

The development of the PoF framework is the joint effort of all project partners, performed in a collaborative way, sharing a code repository and tracking open issues to be discussed in periodic meetings by all interested partners.

The implementation of the final prototype leverages the outcomes of the other technical WPs: the analysis of workflow models for synergetic preservation, reported in deliverables D5.2 and D5.3; the definition of information packages created in the PoF Middleware and imported in the Preservation System, based on the results provided by WP5; the components developed by technical WPs and integrated in the prototype, described in detail in the last version of the corresponding deliverables, namely D3.4, D4.4, D5.4and D6.4 for the PoF Middleware components, D7.4 for the Preservation-aware Storage System, D8.4 for the Digital Repository and finally D9.5 and D10.4 for the Active Systems. Moreover, the outcomes of WP2 (see for example deliverables D2.2 and D2.4) contributed to the definition of the PoF Reference Model which inspired the current implementation, while the issues related to framework licensing and possible mechanisms to publish and disseminate ForgetIT software as open source were analyzed in collaboration with WP11.

The component descriptions in this document focus only on those aspects relevant for integration, such as APIs and I/O formats and protocols, while for component implementation details please refer to the relevant deliverables from the corresponding WPs.

The document is organized as in the following: a summary of the relevant information concerning the PoF Framework architecture is reported in Section 2; an overview of the PoF Reference Model, the relevant workflows implemented in the third prototype and the

implemented information model are described in Section 3; the implementation of the three main framework layers and their integration is discussed in dedicated Sections, for the PoF Middleware (Section 4 and Section 5), the Active Systems (Section 6) and the Preservation System (Section 7), respectively; in these Sections we also describe the internal components, the progress with respect to the second prototype and their final deployment; the prototype implementation, including a short description of the software development, documentation and licensing, is reported in Section 8; in Section 9 we describe the lessons learned and provide an assessment of the results against WP8 success indicators, which have been defined in the project proposal; finally we included a few Appendices containing details about the implementation or configuration of specific components: Appendix A provides implementation details for the configuration of the PoF Middleware, whose REST APIs are described in Appendix B; Appendix C provides further details about the installation and configuration of DSpace with some advanced topics that were investigated and developed during the third year, while Appendix D describes the implementation of the workflows; finally, Appendix E provides information about the experimental APIs for the Memory Buoyancy (MB) assessor.

Progress after second prototype

The third prototype includes several improvements with respect to the second release:

- the messaging layer infrastructure and the routing engine have been further developed, using the most updated versions of the core libraries and improving the definition of relevant workflows; the web console for managing the messaging infrastructure, monitoring the workflows and the queues, has been improved;
- the middleware software has been improved and makes use of additional enterprise pattern, making the software more robust and flexible;
- the PoF Middleware REST APIs have been updated; additional features of the CMIS standard [OASIS, 2013] are now used to support the preservation value and the context;
- the last versions of the middleware components have been integrated, while new components not available in the second release, such as the Context-aware Preservation Manager have been added;
- the Preservation System has been updated: a new version of the Digital Repository is used and the integration mechanism to preserve and re-activate content is now based on pure REST APIs; a new implementation of the cloud storage components with additional Storlets has been integrated;
- the integration of the Active Systems, based on CMIS standard, has been improved;
- the framework implements the information model, supporting situations, collections and items for the representation of content generated in the Active Systems.

The PoF Framework developed during the three project years implements the PoF Reference Model, which is now available, and better implements the core principles of the project, integrating the new results of the project. Further details are provided in Section 9, where we discuss the progress compared to the success indicators.

Target audience for this deliverable

This deliverable targets a technically oriented readership, which is interested in the technical aspects of the implementation of the PoF Framework, plans to adopt the framework or wants to use it as a blueprint for a similar project.

2 PoF Framework Architecture

The architecture of the PoF Framework, described in deliverable D8.1 [Gallo et al., 2013], is made up of three layers: *Active Systems*, *Preserve-or-Forget (PoF) Middleware* and *Preservation System*. The latest version of the PoF architecture is depicted in Figure 1.

The Active Systems represent user applications or any information management system. The Preservation System, which implements the PoF Framework archive, is composed by two sub-systems: a Digital Repository and a Preservation-aware Storage, which includes a Cloud Storage Service. The Preservation System provides both content management and typical archive features required for the synergetic preservation. The PoF Middleware is intended to enable seamless transition from Active Systems to the Preservation System (and vice versa) for the synergetic preservation, and to provide the necessary functionality supporting managed forgetting and contextualized remembering. The PoF Middleware provides the communication layer for all components developed in WP3-WP6, implementing the concept of Enterprise Service Bus (ESB) using a Message Oriented Middleware (MOM) (see Section 4). The middleware connects the user applications with the archive and provides the infrastructure to fetch content from the applications. Finally, the middleware manages the preservation preparation and re-activation workflows interacting with the Active System and the Preservation System.

Compared to previous versions, the updated component diagram in Figure 2 has been improved, mainly for what concerns the Preservation System composite structure, where the internal components have been reviewed according to the recent developments in WP7 and WP8. Additional details have been added for the PoF Middleware components: for example the Policy Engine developed by WP3 has been added as part of the Forgettor component. No other major changes have been applied to the PoF architecture compared to the version used by the first release of the framework.

The ForgetIT framework leverages the adoption of standard lightweight technologies for data exchange and communication between user application and middleware, the integration of the core components in the middleware using a message oriented approach with a rule engine for message routing and workflow management, the long-term preservation of content based on preservation-aware cloud-based storage where preservation tasks and other processing activities executed close to the data (directly in the storage).

The integration of Active Systems and Preservation System with the PoF Middleware is based on REST APIs, used to trigger preservation, re-activate content and monitor the running processes or the preservation status of specific resources. Bi-directional data exchange between Active Systems and PoF Middleware is based on Content Management Interoperability Services (CMIS) standard [OASIS, 2013]: the Active Systems publish the content to be preserved using a CMIS compliant repository and the re-activated content is provided by the PoF Middleware using another CMIS repository deployed in the middleware. Hence, any user application supporting CMIS can be seamlessly integrated with the PoF Framework. Further information about CMIS is in Section 4 and Section 6.

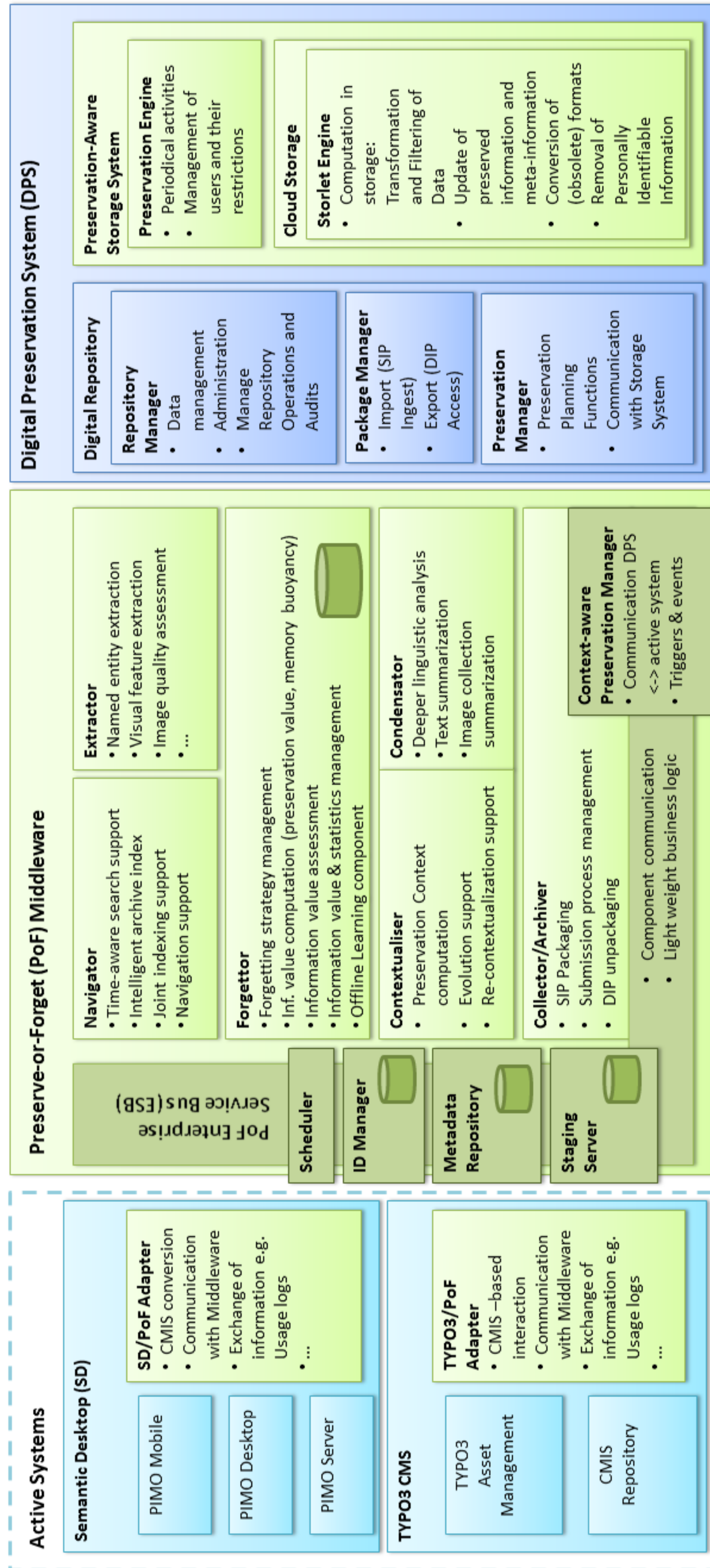

Figure 1: Architecture Diagram of the Preserve-or-Forget (PoF) Framework.

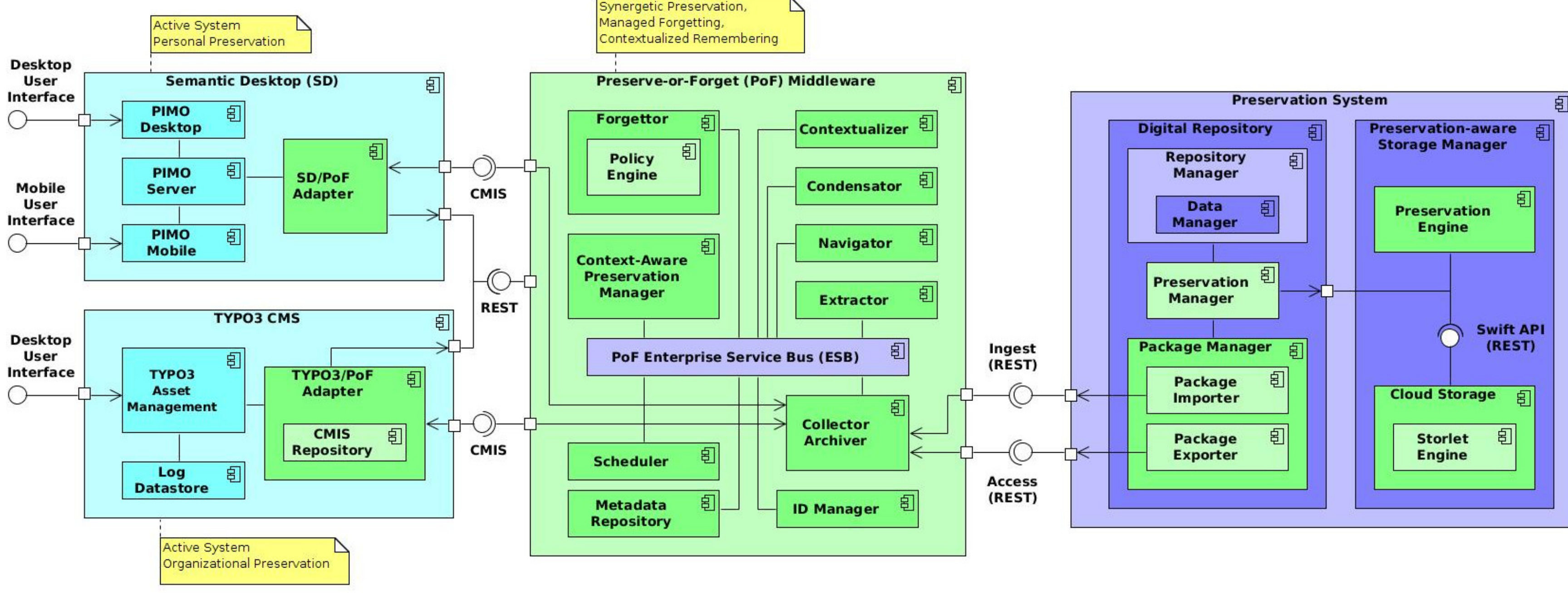

Figure 2: PoF Framework component diagram: the composite structure with internal components is shown.

3 PoF Reference Model

The PoF Reference Model, described in deliverable D8.5 [Gallo et al., 2016], served as conceptual guideline for the integration process of the PoF Framework and aims to encapsulate the core principles of the ForgetIT approach into a re-usable model.

In the following we summarize the main concepts for the functional and information part of the model, relevant for the prototype description.

A representation of the functional part of the model is depicted in Figure 3. The frame represents the domain of our model, where information and preservation systems are considered as part of a joint ecosystem, which stresses the smooth transitions and the synergetic interactions rather than the system borders. The functional part is made up of three layers. The Core Layer considers basic functionalities required for connecting the Active System and the Preservation System; building upon this layer, the Remember & Forget Layer introduces brain-inspired and forgetful aspects; finally, the Evolution Layer is responsible for all types of functionalities dealing with long-term change and evolution, such as implementing the contextualized remembering. For each layer we also show the functional entities and the representative workflows (double pointed arrows). The position of the workflows is associated to the layer they belong to, so for the outer arrows the Evolution layer. The precise positions are meant to show which part each of the evolution workflows is closer to, e.g. the Situation Change is closer to the Active System, the System Change can affect both Active and Preservation System, and the Setting Changes are mainly observed in the Preservation System.

The functional model workflows described in D8.5 are reported in the following: the Preservation Preparation, the Re-activation and the two related to system change, Active System Change and Preservation System Change. In a nutshell, they are responsible for transferring content to be preserved from the Active System to the Preservation System, to enable the Active System to retrieve and re-activate content previously transferred to the Preservation System and to manage changes in either the Active System or in the Preservation System.

The different steps of such workflows involve different PoF Framework components, mainly for what concerns the PoF Middleware. We provide a representation of the activation of the different components in each workflow in Figure 4 and Figure 5 for the Remember & Forget Layer and in Figure 6, Figure 7, Figure 8 and Figure 9 for the Evolution Layer.

It is worth noting that some components are involved only in one of the layers, e.g. the Remember & Forget Layer or the Evolution Layer (see Table 1) and that Figure 7 about the Setting Change workflow includes also the Active System and Preservation System components, while the others involve only the components within the PoF Middleware.

An overview of the Information Model is depicted in Figure 10. An explanation of the model entities is available in deliverable D8.5.

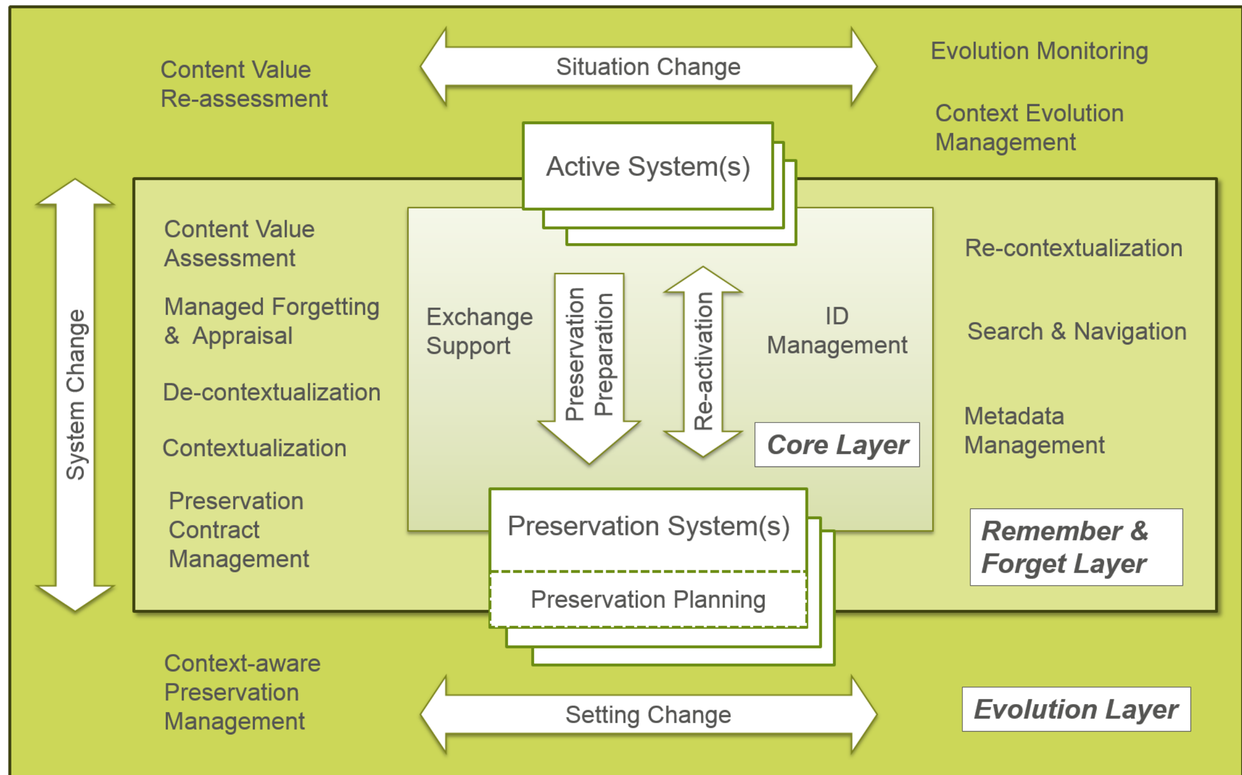

Figure 3: High-level functional view of the PoF Reference Model (from D8.5).

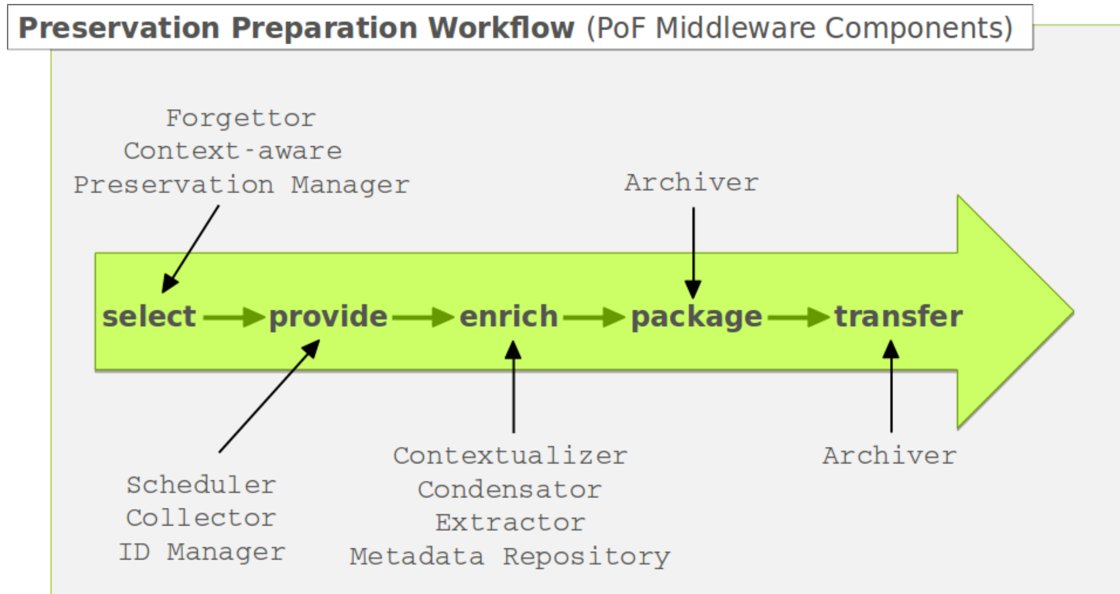

Figure 4: Mapping between the PoF Middleware Components and the Preservation Preparation Workflow.

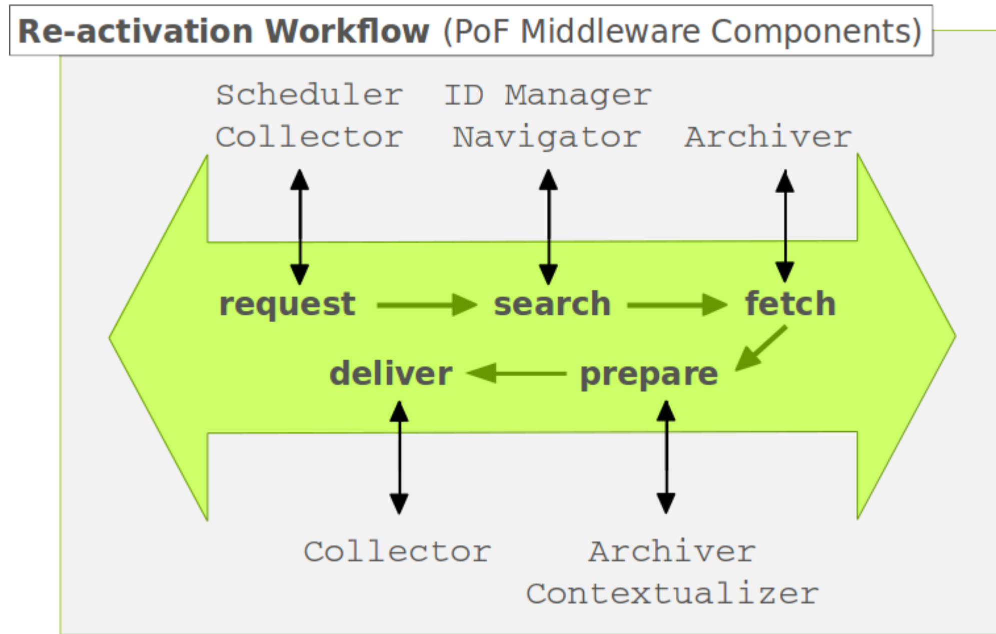

Figure 5: Mapping between the PoF Middleware Components and the Re-activation Workflow.

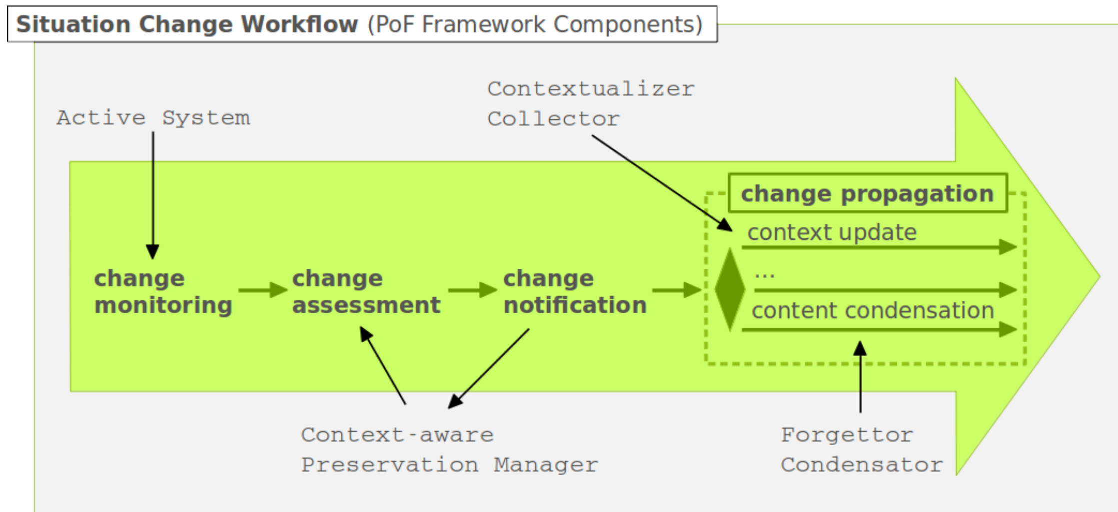

Figure 6: Mapping between the PoF Framework Components and the Situation Change Workflow.

3.1 Implementation of the Reference Model

The implementation of the framework is based on the functional and information models above. In the final release we improved the implementation of the Preservation Preparation and Re-activation workflow, while other parts of the remaining workflows were either implemented directly in the middleware or embedded in specific components, e.g. the

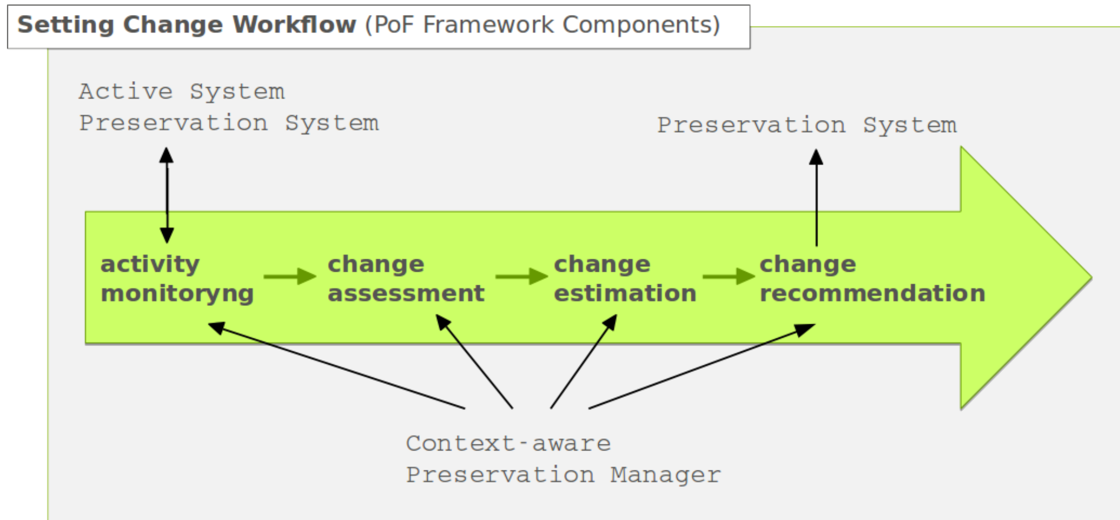

Figure 7: Mapping between the PoF Framework Components and the Setting Change Workflow.

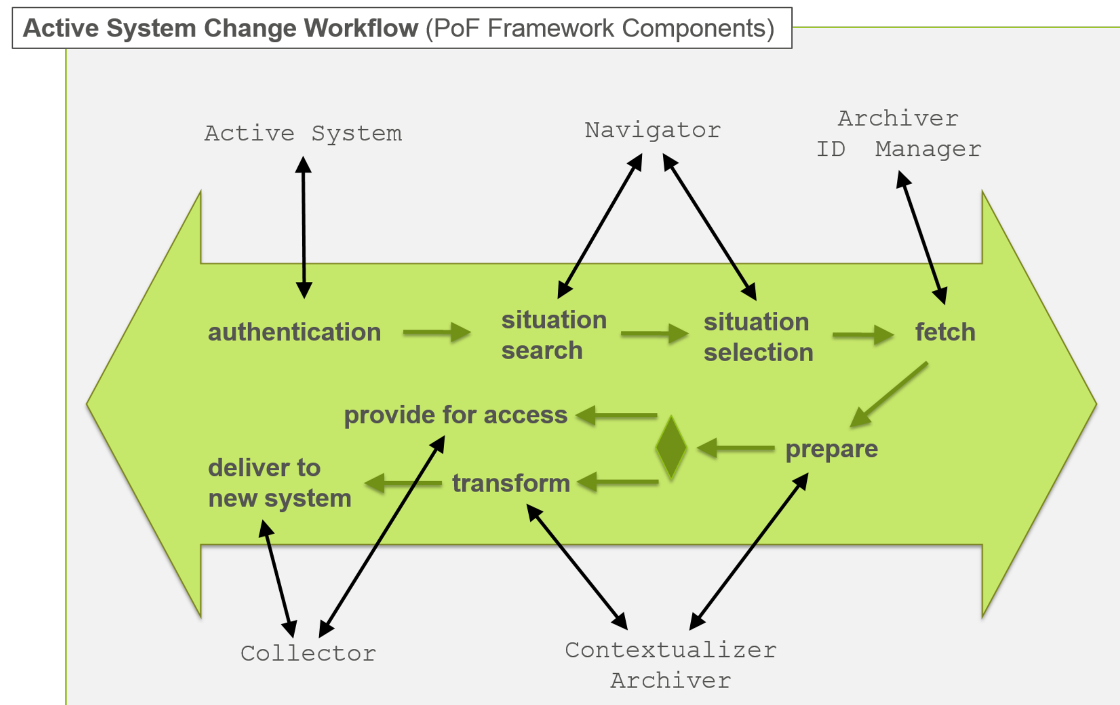

Figure 8: Mapping between the PoF Framework Components and the Active System Change Workflow.

Context-aware Preservation Manager or the Preservation-aware Storage System. The Preservation Preparation and Re-activation workflow have been implemented for both use cases.

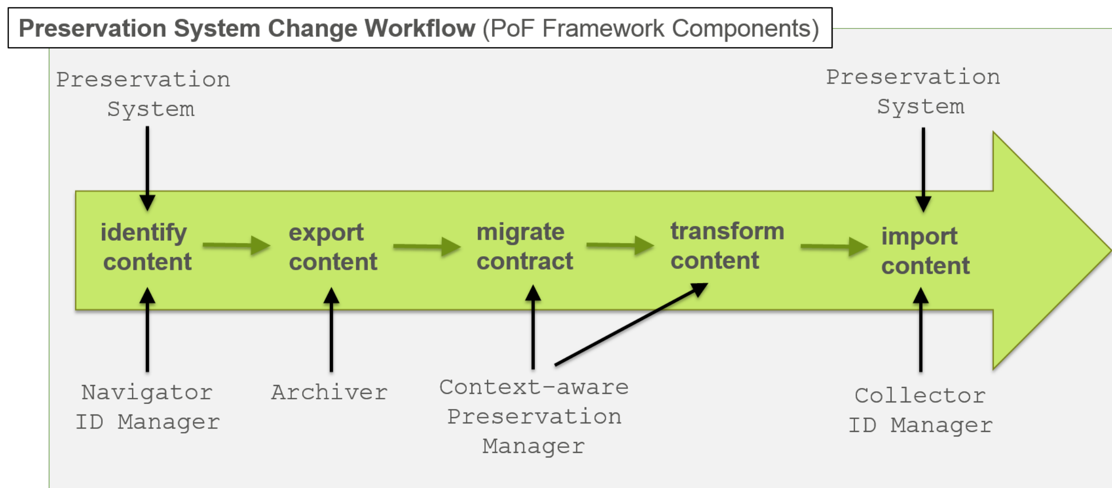

Figure 9: Mapping between the PoF Framework Components and the Preservation System Change Workflow.

The structure of the information model has been fully implemented in the final release. In particular, different user applications were integrated with the middleware by mapping the content in each application to Situations, Collections and Items using the CMIS standard, as discussed later. Moreover the Preservation Entity models the definition of the packages stored in the Preservation System, including different metadata types and the context. It is worth noting that the different IDs and the ID Mapping Table could be implemented in different ways, we used an approach based on Java Persistence with an object DB and a dedicated component, the ID Manager, to support the mapping.

Further details are provided in Appendix D, where we include some application screenshots from the prototype.

Functional Entity	Model Layers	PoF Middleware Components
ID Management	*Core*, *Remember & Forget*	`ID Manager`
Exchange Support	*Core*, *Remember & Forget*	`Collector`, `Archiver`, `Metadata Repository`
Content Value Assessment	*Remember & Forget*	`Forgettor`
Managed Forgetting & Appraisal	*Remember & Forget*	`Forgettor`
De-contextualization	*Remember & Forget*	`Contextualizer`
Contextualization	*Remember & Forget*	`Contextualizer`, `Extractor`, `Condensator`
Preservation Contract Management	*Remember & Forget*	`Context-aware Preservation Manager`
Re-contextualization	*Remember & Forget*	`Contextualizer`, `Archiver`
Search & Navigation	*Remember & Forget*	`Navigator`
Metadata Management	*Remember & Forget*	`Forgettor`, `Extractor`, `Condensator`, `Contextualizer`, `Metadata Repository`, `Collector`, `Archiver`
Content Value Re-assessment	*Remember & Forget*	`Forgettor`, `Contextualizer`
Context-aware Preservation Management	*Evolution*	`Context-aware Preservation Manager`
Evolution Monitoring	*Evolution*	`Context-aware Preservation Manager`
Context Evolution Management	*Evolution*	`Context-aware Preservation Manager`, `Contextualizer`

Table 1: Mapping between PoF Reference Model Functional Entities and the PoF Middleware Components (from D8.5).

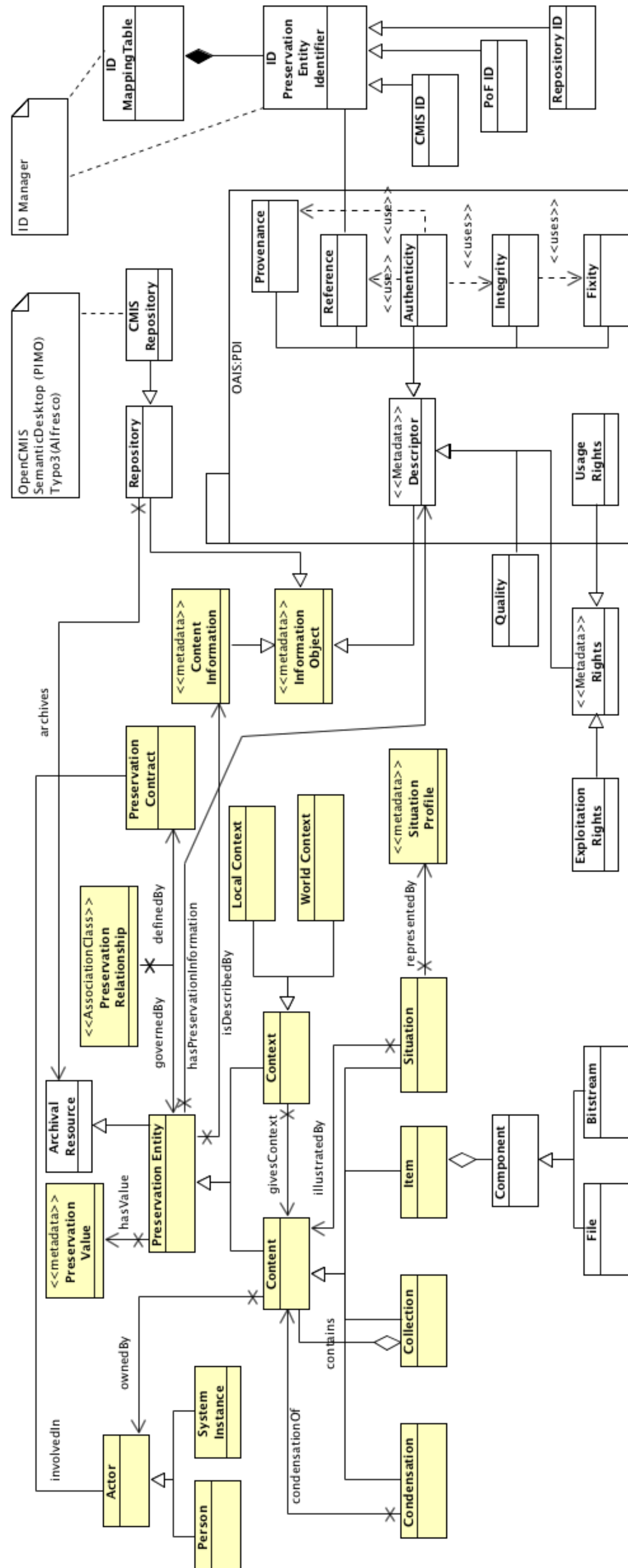

Figure 10: Information Model from an Implementation Perspective with All Components (User Perspective in Yellow)

4 PoF Middleware

In this Section we describe the implementation of the PoF Middleware and the integration with Active Systems and the Preservation System. We also describe the use of CMIS standard for data exchange between user applications and the PoF Framework. The components integrated in the PoF Middleware are described in Section 5.

4.1 PoF Enterprise Service Bus

The PoF Middleware has been designed using the Enterprise Service Bus (ESB) approach. The ESB is a well-established architecture design which has been adopted in many enterprise applications and systems over the past ten years and is still very popular in the implementation of both commercial and open source solutions. The role of the ESB in the middleware has been discussed in many previous deliverables, both from the architectural point of view (see for example deliverables D5.1 [Nilsson et al., 2013], D5.2 [Nilsson et al., 2014] and D8.1 [Gallo et al., 2013]) and from the implementation point of view (see deliverable D8.3 [Gallo et al., 2014]). In a nutshell, the role of the ESB in the PoF Middleware is mainly intended to provide a communication layer for all components, providing loose coupling and reducing the dependency between the components: using the ESB approach, the number of point-to-point connections among the components and the number of point of failures is reduced to a minimum (if not to zero) and the only requirement to *get on the bus* is to agree with the service *contract*, namely to integrate with the communication APIs exposed by the ESB and to support data exchange using a common exchange format. For a description of the ESB approach, see [Chappell, 2004].

4.1.1 Message-Oriented Middleware

In order to implement the PoF ESB, we adopted the Message Oriented Middleware (MOM) approach, where data and other information is received by or passed to the components connected to the ESB in the form of messages, as shown in Figure 11: this means that only a representation of the data is exchanged and this can be processed and modified locally by each component. A MOM lies between the applications acting as a message mediator between them by means of a communication channel that carries self-contained units of information which are the messages. The MOM mediates events and messages among distributed systems providing the required degree of decoupling. Figure 11 provides a view of this kind of architecture.

A MOM is intended mainly for communication in an loosely-coupled, reliable, scalable and secure manner amongst distributed applications or systems. Compared to situations where the information exchange takes place directly among the distributed applications (coupling), the MOM makes use of asynchronous messaging and the message

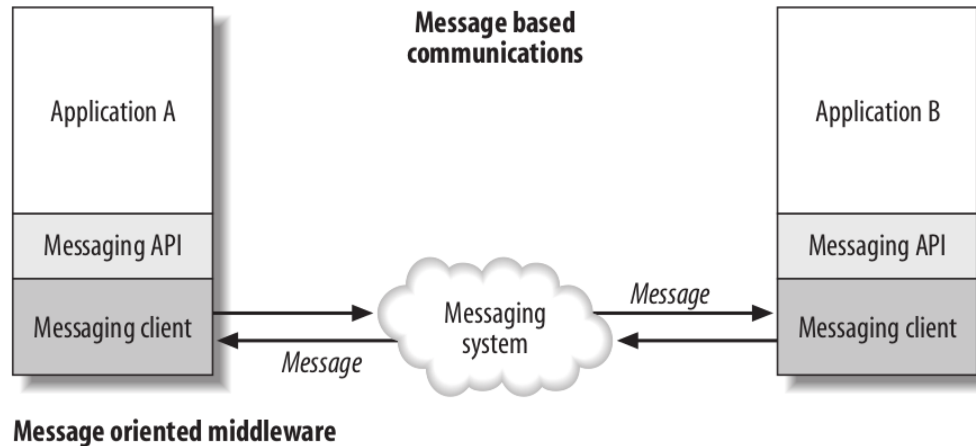

Figure 11: Message based communication, taken from [Chappell, 2004].

senders (Producers or Publishers) know nothing about receivers (Consumers or Subscribers) and receivers know nothing about senders, as depicted in Figure 11. MOM is a suitable solution for the management and the integration of the various components in the project, where several heterogeneous components are integrated in a middleware and asynchronous communication is a requirement. If the MOM provides a reliable and flexible communication infrastructure, we need to organize the data flow and task execution with messages in order to implement complex workflows.

The MOM also has the responsibility to ensure that the messages reach their intended destination and that they are not lost in case of network failure, therefore the messages have to be stored into a persistent memory and accessed when requested from the Consumer. This feature is referred to as *message persistence*. Figure 12 depicts an example of message exchange where the Consumer looses the connection to the MOM but the message does not get lost.

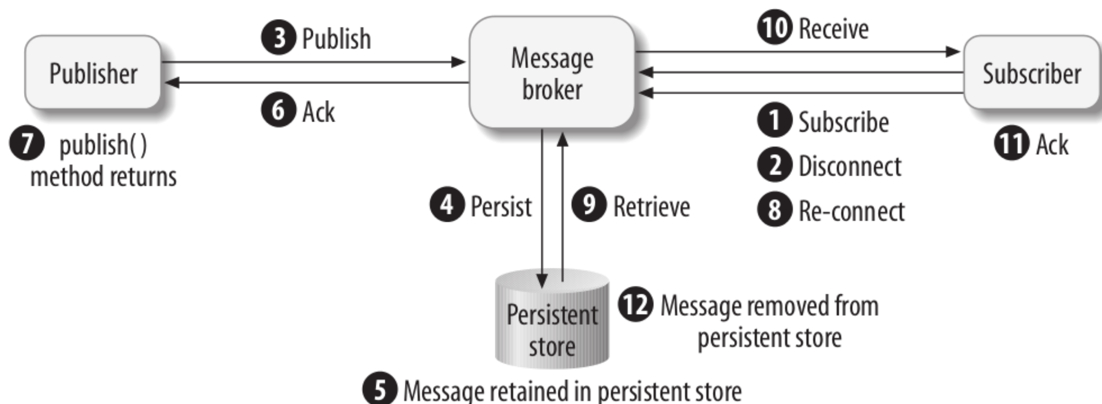

Figure 12: Example of message persistence, taken from [Chappell, 2004].

4.1.2 Enterprise Integration Patterns

For the implementation of the different workflows, we make use of Enterprise Integration Patterns (EIP), defined in the fundamental book by G. Hohpe [Hohpe and Woolf, 2003]. The EIP approach has been extensively adopted to design asynchronous messaging architectures used to build integration solutions and is used in several enterprise-class applications. The book describes 65 design patterns for the use of Enterprise Application Integration (EAI) and MOM in the form of a pattern language. They are accepted solutions to recurring problems within a given context. Patterns are abstract enough to apply to most integration technologies, but specific enough to provide hands-on guidance to designers and architects. Patterns also provide a vocabulary for developers to efficiently describe their solution. Patterns are not 'invented'; they are harvested from repeated use in practice. A coherent collection of relevant patterns that form an integration pattern language is available on the EIP web site[1].

An example of typical EIP is the `Message Router`, depicted in Figure 13. A `Message Router` pattern can be used to decouple a message source from the ultimate destination of the message, acting as a special filter which consumes a message from one message channel and republishes it to a different message channel depending on a set of conditions. The `Message Router` connects to multiple output channels and the components surrounding the `Message Router` are completely unaware of the existence of a `Message Router`. A key property of the `Message Router` is that it does not modify the message contents, being only concerned with the destination of the message. This pattern has been extensively used in the implementation of internal PoF Middleware components.

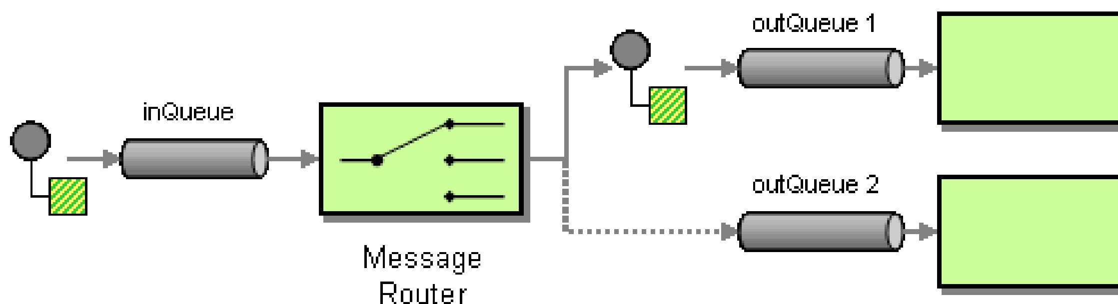

Figure 13: Message Router Enterprise Pattern.

Another pattern example, which was frequently used in the PoF Middleware implementation is the `Service Activator`, depicted in Figure 14. A `Service Activator` connects a message channel to a synchronous service, which is invoked whenever a message is received. The activator receives the message (asynchronously) and is capable to identify which service to invoke (synchronously) and what data to pass by processing the message and extracting information necessary to invoke the service, such as the query parameters. The activator can always invoke the same service (for example in the

[1] Enterprise Integration Patterns - `http://www.eaipatterns.com/`

middleware implementation we used configuration properties), or can use invoke a given service based on message content. The main purpose of the activator is to manage the messaging details and invoke the service like any other client (the service is not aware that it is invoked through messaging). In this way the service developers can assume that their service will always be invoked synchronously, without messaging, and the activator enables service invocation through messaging. After invoking the service, the aggregator blocks during service execution till request completion: when the service returns the result, the activator can return a message with such information, so the service invocation using an activator implements a regular `Request-Reply` behaviour. A Service Activator is also serving as another pattern, the `Messaging Gateway`, since it separates the messaging details from the service. The activator can implement two patterns: the `Polling Consumer` (it polls for a message, blocks while processing it and then polls for another, returning immediately if no message is available) or a `Event-Driven Consumer` (it is triggered by message delivery).

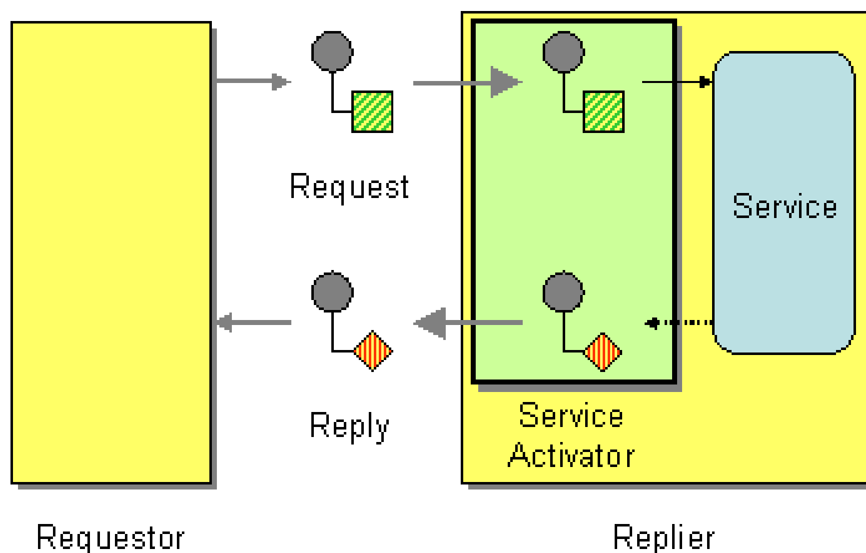

Figure 14: Service Activator Enterprise Pattern.

It is worth noticing that several patterns can be used in combination in order to achieve the required behaviour: for example when describing the `Service Activator` pattern, other patterns have been mentioned.

Some basic patterns have been used very often in the middleware. Such patterns include, for example, `Request-Reply`, `Aggregator` or `Message Filter`, to name just a few. The full list of EIPs is available in [Hohpe and Woolf, 2003].

4.1.3 Asynchronous Routing Engine

Message routers control how messages are routed among the services in a ESB application. Implementing a flexible and efficient message routing is crucial to fully exploit

the benefits of asynchronous messaging. Different kinds of routers are available, associated to the different patterns. For the PoF Middleware implementation we used an asynchronous routing engine supporting all the reference integration patterns to implement business logic within the middleware.

In the PoF Middleware, the Scheduler component makes use of the `Message Router` pattern described above to process the incoming messages and trigger specific workflows based on the message properties. We provide an example taken from the middleware source code in Appendix A, where the Scheduler message route is defined using Spring XML and Apache Camel (see next Section). Based on the value of different headers for the incoming message, a specific logic is implemented.

4.1.4 PoF ESB Implementation

For the implementation of the ESB we make use of two components provided by the Apache ServiceMix[2] suite: Apache ActiveMQ [Snyder et al., 2011], for implementing the messaging system (broker), and Apache Camel [Ibsen and Anstey, 2010], for implementing a rule-based routing engine running on top of the broker.

ActiveMQ is an open source, Java Message Service (JMS) 1.1 compliant MOM from the Apache Software Foundation that provides high-availability, performance, scalability, reliability and security for enterprise messaging. It also provides all the MOM functionalities allowing the user to implement and customize specific message producers and consumers that exchange information through queues and topics. ActiveMQ is commonly adopted in enterprise scenarios when an asynchronous message bus is needed (see for example [Henjes et al., 2007, DAI and ZHU, 2010] and other references available in the literature).

On top of the message broker implemented by ActiveMQ, a rule-based routing and mediation engine has been added, in order to implement the middleware workflows using one of the EIPs. The rule engine is provided by Apache Camel.

As will be described in Section 8, the package `eu.forgetit.middleware` of the PoF Middleware Java project contains the main classes for the implementation of the PoF Middleware.

The final release contains the latest versions of both components, which have been upgraded with respect to the second prototype: the new versions provide several improvements in terms of stability and configuration and required some effort to upgrade the middleware Java code accordingly.

For the final release we further improved the monitoring interface for the messaging system and the routing engine, using an updated version of hawtio[3], a web monitoring console based on HTML5 that integrates seamlessly with ActiveMQ and Camel: this graphical

[2] Apache ServiceMix - http://servicemix.apache.org
[3] hawtio - http://hawt.io

console replaces the old ActiveMQ GUI. The flow of messages in the different queues, updated in real time during workflow execution, is shown in Figure 15, where we expanded the route of the Scheduler component: the different conditions using the message patterns described above are shown with a graphical notation. Additional screenshots of the hawtio console for the middleware instance in the testbed are shown in Appendix A.

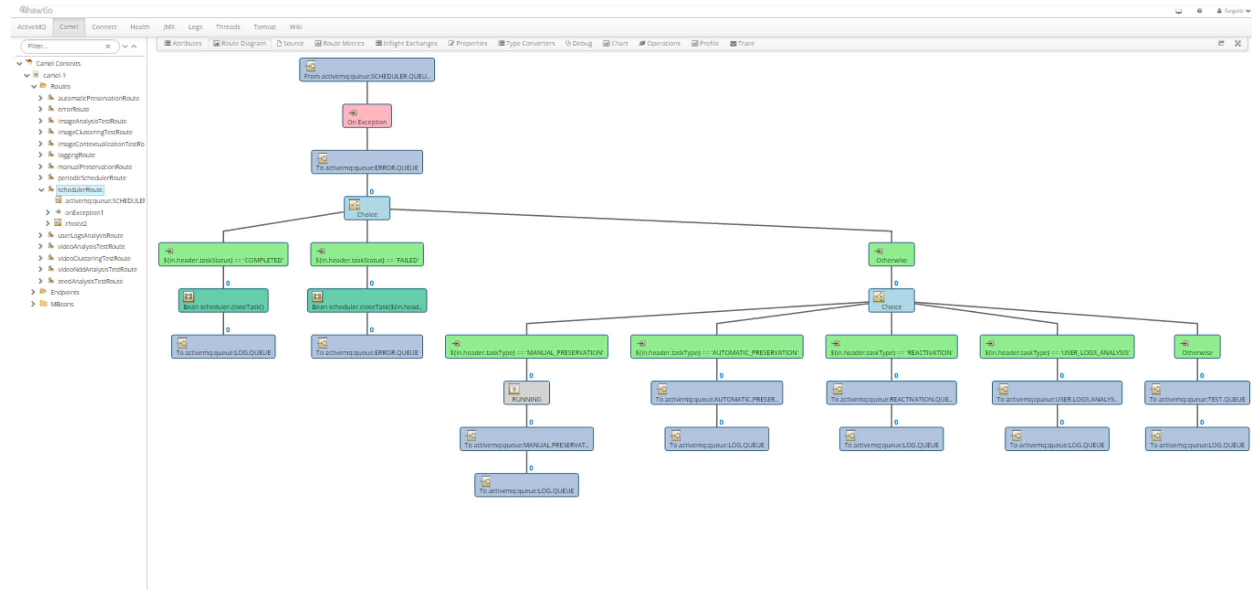

Figure 15: Message Flow Monitoring for the Scheduler Route.

4.2 Middleware Configuration

The configuration of the messaging system and of the routing engine makes use of Spring XML framework. Sample configuration files are described in Appendix A. The broker configuration is used to instantiate the connection when the PoF Middleware server running in Apache Tomcat is started. The queues and the topics are automatically created. Finally, all middleware components are defined as Spring beans, therefore their instances are created and maintained over time by the Spring framework.

The configuration of Apache Camel using Spring XML is straightforward. Sample configuration for the messaging broker and the route for two workflows (preservation preparation and re-activation) is reported in Appendix A. Each workflow is represented as a sequence of steps associated to specific Spring beans corresponding to the middleware components. The Spring XML representation is associated to different patterns and defines a language for implementing specific rules associated to the messages.

We also provide an excerpt of Java code taken from the Extractor in Appendix A: the method for image analysis used in the Apache Camel route defined above makes use of `Exchange` class, which is part of the Camel APIs and contains the message information (header and body). This approach has been used for all components in the middleware.

The message header is typically used to share high-level information required for flow control, while the message body contains the data. In the current implementation, we use JavaScript Object Notation (JSON) format to represent message content. After processing the message, extracting information and obtaining some results, the message body and header can be updated and then passed to the flow control wrapped in the `Exchange` object. Following the asynchronous message approach, the next destination of the message is unknown to the component class, the new message is sent to one of the instances of the next component in the flow using the route definition.

4.3 RESTful Service

REST APIs are published using Jersey[4], the reference implementation of JAX-RS specification for RESTful web services.

The REST APIs make use of JSON for information exchange and support CMIS identifiers for managing the resources.

The exposed APIs are reported in Appendix B, where we describe the available APIs with the expected parameters and the output format. The list of APIs exposed by the PoF Middleware RESTful web server is available as W3C WADL format.

4.4 CMIS Integration

Content Management Interoperability Services (CMIS) is an open standard that allows different content management systems to inter-operate over the web, defining an abstraction layer for controlling diverse document management systems and repositories using web protocols. CMIS defines a common data model, which encapsulates the core concepts found in most content management systems, covering typed files and folders with generic properties that can be set or read (see Figure 16). CMIS defines also protocol bindings that can be used by applications to manipulate content stored in a repository, using WSDL, SOAP and AtomPub. The CMIS specification provides an API that is programming language-agnostic. The Java-based library provided by Apache Chemistry [Müller et al., 2013] has been used in the Collector component implementation.

An Active System can interact with the PoF Framework through the REST APIs described above and exchanging data using CMIS standard [OASIS, 2013]. The Active System acts as a data-deliverer, so any information system that supports CMIS could act as an Active System in the PoF Framework. In Section 6 we describe the two main Active Systems under test in the project (Semantic Desktop and TYPO3 CMS), along with other prototype user applications which could be integrated with the framework.

CMIS supports items, documents and folders as basic types, including their relationships.

[4] Java Jersey - `https://jersey.java.net`

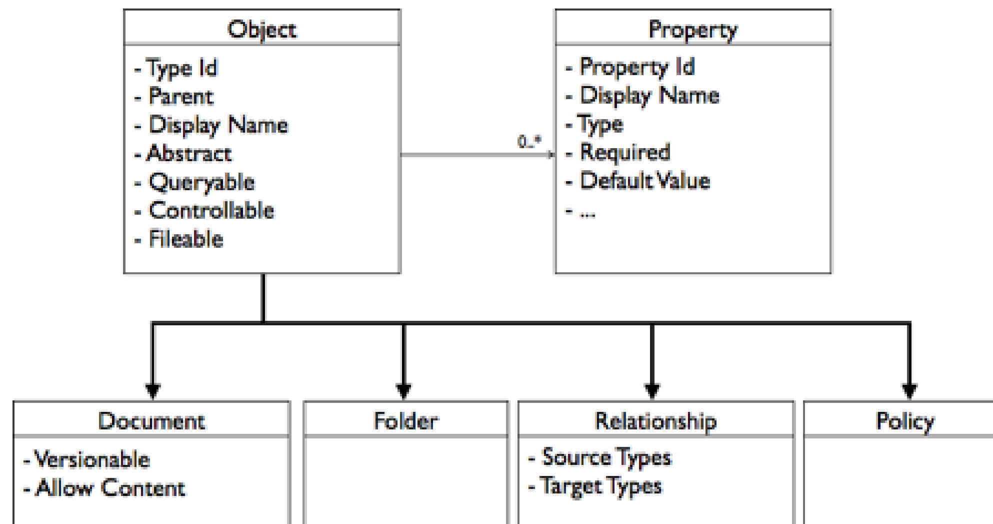

Figure 16: CMIS data model, taken from Alfresco web site.

Each CMIS Object type (`cmis:item`, `cmis:document`, `cmis:folder`) can be preserved in the PoF Framework without sending the actual details about how the data is stored in the Active System, since CMIS is used as a standard to exchange data between the Active System and the middleware.

Several implementations of a CMIS repository are available and have been used in the final framework release. For the Semantic Desktop the CMIS repository makes use of the OpenCMIS (Apache Chemistry) library [Müller et al., 2013], while TYPO3 CMS uses Alfresco CMIS [Bergljung, 2014].

The PoF Middleware accesses the content in the Active System CMIS repository using the CMIS ID which is provided when a preservation request is triggered using the REST APIs or when an external process is triggering such preservation. See the description of the Collector component in Section 5.6 for further details. The ID Manager component manages the mapping between the CMIS ID and the other identifiers in the framework (see Section 5.1).

The PoF Middleware can access these objects and pull required information. If the PoF Middleware needs to know how many relations to this item exist it asks for all associated `cmis:relation` entities. Relying on this CMIS standard enables the PoF to communicate with any component that supports CMIS, which is flexible and doesn't require any special ForgetIT implementations of the Active System to exchange data. Each document should contain information about what type of element the object is exactly.

The information provided using CMIS representation includes also the Preservation Value (PV) associated to the resource. We foresee two different approaches here: the calculation of the PV could be performed by the Active System itself (as done by the Personal Information MOdel (PIMO), for example) or could provide different evidences for such calculation. This case has been implemented within TYPO3.

Different strategies can be implemented by an Active System after a given content is preserved, for example it could be deleted from the Active System CMIS repository (but the Active System should be able to correctly identify it during re-activation).

When restoring an object from the Preservation System, the CMIS standard is used again, because the PoF Middleware provides its own CMIS repository based on OpenCMIS (Apache Chemistry) library: when requesting archived content, the PoF Middleware returns a CMIS ID which can be used to fetch the content from middleware CMIS repository (see REST APIs above). This can also enable new scenarios, when for example the Active System is no more available and the archived content must be retrieved.

The role of CMIS in the information model is highlighted in Figure 10, since it is one of the classes extending the Preservation Entity Identifier. In the model workflows, every time the ID Manager and Collector components are used, the CMIS standard is used to identify and retrieve the content. Moreover, during the Re-activation workflow, the middleware exposes the content through CMIS, as mentioned above: as a consequence, the way a user application can fetch content back from the middleware implies the use of CMIS.

One of the main improvements of the final release is the implementation of different CMIS repository adapters, for each application type, capable of converting the CMIS Object information to preserved content metadata and to map the original content in the user application to Collections and Items based on the CMIS Object type.

5 PoF Middleware Integrated Components

Compared to the previous prototypes, the final framework release integrates in the PoF Middleware the updated versions of existing components along with new ones. The individual components in the middleware are shown in Figure 1. In this section, for each middleware component we briefly describe the role in the overall framework, the contributing partners and reference deliverables, a short description of the integration mechanism and the deployment information.

Additional information about licensing the core components of the PoF Framework as open source is discussed in Section 8. In this Section we provide licensing information for each component separately.

5.1 ID Manager

Component Role The ID Manager mediates between the IDs used in the Preservation System components (Digital Repository and Preservation-aware Storage System) and the IDs used in the Active Systems. Such IDs are associated to the resources to be preserved and are used during Preservation Preparation and Re-activation workflows or for monitoring the preservation status of the resources. The mapping is maintained by the ID Manager using a unique ID which is generated and managed by the PoF Middleware internally. The information is stored in a internal object DB, shared with the Metadata Repository component (see Section 5.2).

WP and Deliverables The ID Manager is developed within WP8 (integration with the messaging layer and ID management), WP3 (scheduling of forgetting process) and WP5 (scheduling of archiving process). The previous release was described in deliverable D8.4 [Gallo et al., 2015a], the contributing partners are mainly EURIX and LUH.

Integration and Deployment The ID Manager is written in Java and is included in the main PoF Middleware Java project (`eu.forgetit.middleware.component` package) available in the project SVN repository (see Section 8). The dependencies are managed with Maven. The APIs of the ID Manager are used by all internal components of the middleware: for example the Collector and Archiver components strongly depend on the ID Manager to properly collect resources from the Active System and to archive resources in the Preservation System. When the resource is a collection, an additional request is sent to the Scheduler, so the different resources in the collection are retrieved in parallel, based on preservation rules based on the concept of Preservation Value (PV) (see WP3 deliverables). For example, in the current implementation only resources with a PV above a given threshold are retrieved, the others are discarded. Based on the information provided by the ID Manager, the PoF Middleware can return information to the Active Systems concerning the preservation status of the resources. ID Manager APIs are also exposed to the other framework components outside the middleware, namely the Active Systems and the Preservation System, through the middleware REST APIs (see Sec-

tion 4.3). Using such APIs, the Active System can trigger and monitor the preservation of a given resource or can request content re-activation, by providing the CMIS ID (object ID and repository ID) of a given resource: this information is used by the ID Manager to create a new ID mapping during preservation preparation or to get the resource ID in the Preservation System to fetch the preserved content during re-activation. The Preservation System makes use of ID mapping through middleware REST APIs when a new resource is preserved, to update the ID mapping in the ID Manager internal DB. The ID Manager is instantiated using Spring XML (see Section 4.2), this is done automatically at middleware service start up. The connection with the broker to produce and consume messages is defined in the Apache Camel configuration, which defines the different routes and includes the ID Manager in the process.

API and I/O Formats The ID Manager provides APIs for creating new IDs and maintains the mapping among different IDs. Main methods include generation of new ID and retrieval of IDs from a internal repository. The APIs of the ID Manager and the associated classes in the middleware are shown in Figure 17. Currently the IDs used to identify a given resource are: `pofId` (middleware internal ID), `cmisId` and `cmisServerId` (CMIS Object ID and CMIS Repository ID), `repositoryId` (ID generated by the Digital Repository, DSpace in the current implementation) and `storageId` (ID generated by the cloud storage system, OpenStack Swift in the current implementation). As depicted in Figure 17, the ID Manager provides the methods to generate new unique IDs (using an internal seed) , to get the whole ID mapping or a specific ID associated to a given `pofId` and also to update ID mapping information (for example, when the resource is moved to cloud storage, a new `repositoryId` is added to the mapping). A Java inner class `IDMapping` and an enumerator `IdType` are used to represent such mapping. The `IDMapping` objects are Enterprise JavaBeans (EJB) instances, stored in a pure object database, ObjectDB [ObjectDB, 2015], where Create Read Update Delete (CRUD) operations are implemented using standard Java Persistence API. The ID mappings are managed by means of `get` and `set` methods in order to edit the internal properties corresponding to the given ID. The CRUD operations on the internal object DB are performed by the `DataManager` class (see Figure 18), which provides the persistence methods (based on Java Persistence API) to store `IDMapping` objects and is also used for persistence of `Task` objects. The object DB used by the ID Manager to store IDs is also used to implement part of the Metadata Repository functionalities, as described in Section 5.2. Different standards are available for identifiers, in the current implementation we make use of Universally Unique IDentifier (UUID) specification. The ID Manager is invoked internally during workflow execution, when assigning new IDs to the content processed in the middleware or when parsing a collection. The connection with the messaging layer is provided by the `Exchange` objects, which are passed to the broker using the messaging API and are managed by the routing engine (see Figure 17). As shown in Figure 17, the `Collector` and `Archiver` classes use the `IDManager` when the resources are fetched from the Active System (to create a new `IDMapping`) or when they are archived (to update the `IDMapping`). The `Collector` and `Archiver` methods are shown in detail in Figure 22.

Status and Workplan An updated version of the ID Manager has been developed for

the final prototype. This version provides all required features for ID management. All planned functionalities have been implemented for this component, the integration with the messaging layer has been completed and the current status is compliant to the integration plan described in D8.1. In particular, the final release provides a refinement of mapping and better representation of the Situation, Collections and Items based on the information model. The ID Manager code has been updated to the most recent versions of the Java JDK and ObjectDB.

Documentation and Reference Links The APIs and usage examples are available in the software documentation, see Section 8 . For Java Persistence API please refer to official Java documentation, for ObjectDB information can be found on the project web site [ObjectDB, 2015].

License The component is released as open source under GPL licence, the same used for the PoF Middleware, see Section 8. According to their website, the ObjectDB software is available under Open Source licence and used at no cost (including commercially) with the restriction of maximum ten entity classes and one million entity objects per database file, using it without these restrictions requires purchasing a licence.

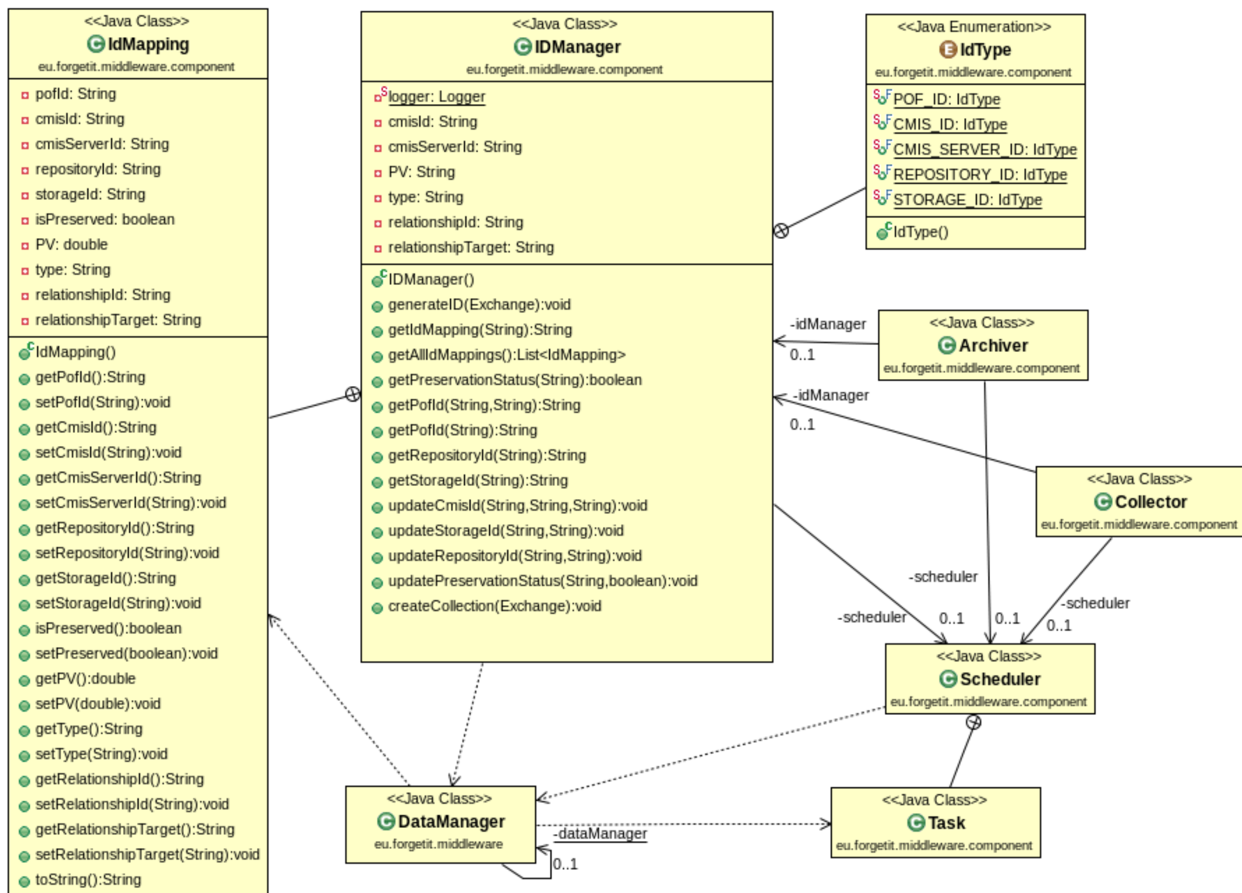

Figure 17: Class diagram for ID Manager component, with associated classes. Association with the Scheduler is related to process scheduling when creating new IDs.

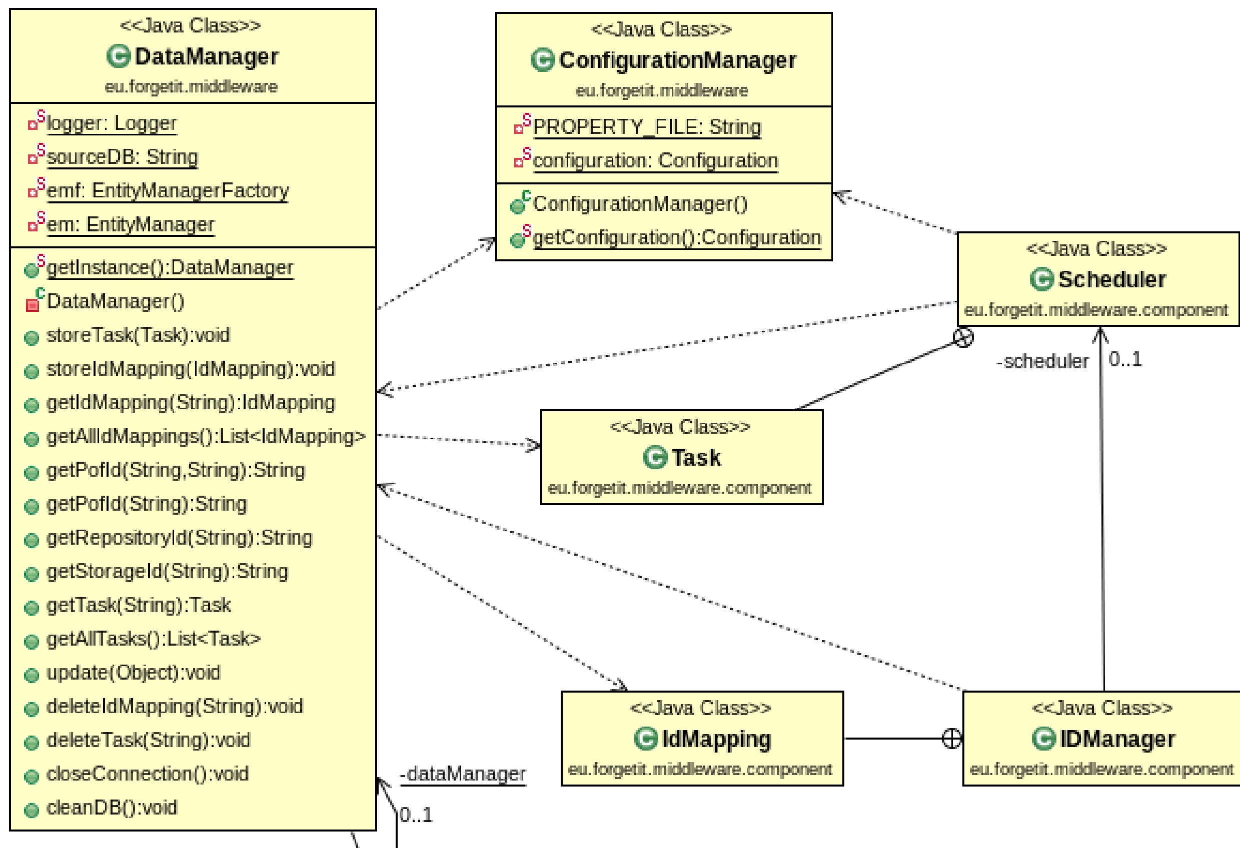

Figure 18: Class diagram for Data Manager: methods based on Java Persistence API are used for CRUD operations on the object DB. The Data Manager is part of the ID Manager and Metadata Repository implementation, but is used also by the Scheduler.

5.2 Metadata Repository

Component Role The Metadata Repository manages metadata extracted or computed for individual documents and collections and makes them available for other components. Examples of such metadata include descriptive metadata, relationship with other resources, extracted entities or features, context information, Memory Buoyancy (MB) and PV. The Metadata Repository relies on the fact that all resources are identified by an unique ID (see ID Manager in Section 5.1), which enables the retrieval of metadata stored in the repository for a given resource. It is worth noting that the Metadata Repository is not intended for persistence or long-term storage, since other components in the architecture are used for this purpose. The Metadata Repository is used to store temporary information during the execution of specific workflows in the middleware and as such it is shared among several components.

WP and Deliverables The Metadata Repository was developed within WP8 and was already included in the previous framework release. The main contributing partner is EURIX.

Integration and Deployment The implementation of the Metadata Repository was based on available technologies, since the functionalities provided by this component are limited and a lot of open source tools can be used. For the implementation we make use of ObjectDB, as used for the ID Manager (see Section 5.1), since the ID Manager is responsible for storing metadata information associated to CMIS objects along with the IDs. Some metadata associated to the resources (retrieved by the CMIS client provided by the Collector) are stored in such DB: the internal data structures for ID mapping contain information about the source CMIS repository (associated to a given Active System), the PV associated to the resource (provided by the Active System), the resource type (single resource or collection), the relationship with other resources (in case of collection), the preservation status (which is updated by the preservation workflow over time). The information about the resource type and its relationships are retrieved from CMIS object information before fetching the actual resources. Some components, such as the Collector and the Archiver, currently store the temporary metadata information also in their own internal DB or on the file system, while other components, such as the Contextualizer and the Extractor, just use the file system.

API and I/O Formats The information stored in ObjectDB only requires Java Persistence API and the data structure is based on EJB technology. The EJB objects are then mapped to other formats, such as XML or JSON, to fulfill specific requirements. The CRUD operations are performed by the `DataManager` class, shown in Figure 18.

Status and Workplan The current solution for the Metadata Repository provides all expected functionalities.

Documentation and Reference Links Methods and examples for the `DataManager` to store objects in the ObjectDB are available in the software documentation, see Section 8. For Java Persistence API please refer to official Java documentation, further information about ObjectDB can be found on the project web site [ObjectDB, 2015].

License The source code of this component is released as open source, as part of the PoF Middleware code. According to their website, the ObjectDB software is available as open source and used at no cost (including commercially) with the restriction of maximum ten entity classes and one million entity objects per database file, using it without these restrictions requires purchasing a licence.

5.3 Scheduler

Component Role The Scheduler is responsible for managing and organizing middleware activities, by receiving and dispatching requests for the different workflows and asynchronous processes and by interacting with the messaging infrastructure. The Scheduler triggers the different workflows, either by receiving input from other PoF Middleware components or by executing scheduled activities. The other middleware components interact with the Scheduler during the execution of complex processes. A typical example of such interactions is provided by the Collector: when retrieving information about the resources

in the Active System, the request for actual resource retrieval (or the retrieval of multiple resources in a collection) is sent to Scheduler, which creates the appropriate Tasks to be executed asynchronously. Another example is related to the re-activation of content archived in the Preservation System, which is scheduled by creating a specific Task. The Scheduler is also invoked through the middleware REST APIs: based on the request type, different Tasks are executed.

WP and Deliverables Component developed within WP8, since it is strongly related to the middleware messaging layer. The main contributing partner is EURIX. The first preliminary version was described in deliverable D8.3 [Gallo et al., 2014], a major improvement was achieved for the second prototype, when the Apache Camel routing engine was introduced (see D8.4 [Gallo et al., 2015a]) .

Integration and Deployment The Scheduler is written in Java and is included in the main PoF Middleware Java project (`eu.forgetit.middleware.component` package). In the first version a `WorkflowManager` class was used to bridge the gap between the web server and the messaging layer, preserving loose coupling, but in the second and third release this has been removed, since this functionality is now provided by Apache Camel, which acts as a rule-based routing engine for the messages in the broker and therefore is used for workflow definition and management (see Section 4). The `Scheduler` class is depicted in Figure 19, along with associated classes. The `Scheduler` uses the `ConfigurationManager` to get information about the middleware configuration (broker URL, queues, remote services, DB connection, etc.) and the `DataManager` to perform CRUD operations on the object DB described above and store information about Tasks. The `Task` class and two Java enumerators, `TaskType` and `TaskStatus`, are used. The `TaskStatus` contains the possible states for a Task, while `TaskType` is mapped to the different workflows. Currently the Scheduler supports the Preservation Preparation and Re-activation workflows defined in the PoF Reference Model (see Section 3). The `Task` class is a EJB with different properties, such as the Task identifier, type, start time and last completed step in the workflow. The `Task` body contains the results of the workflow, typically as a JSON object, and is mainly used for monitoring purposes. The information about each Task is stored in the object DB and is returned by the middleware through specific REST APIs, described in Section 4. The Task identifier can be used as a token for monitoring the progress of a given Task. The Scheduler is instantiated using Spring XML (see Section 4.2), this is done automatically at middleware service start up. The connection with the broker to produce and consume messages is defined in the Apache Camel configuration, which defines the different routes and includes the Scheduler in the process.

API and I/O Formats The Scheduler APIs allow the scheduling of processes based on time and events, to request status information and to delete scheduled events. A subset of these APIs has been already implemented and is shown in Figure 19. The Scheduler currently exposes APIs for scheduling Tasks based on requests received by the middleware REST web server or by scheduled activities defined within Apache Camel using Spring XML configuration. According to the request type, the Scheduler can trigger different workflows.

Status and Workplan Compared to the first release, the current version has been improved and now leverages the routing engine implemented by Apache Camel. Starting from the second release, the support for Task management has been added and a more flexible approach for triggering workflows and processes is used. The Scheduler provides public APIs for sending messages and for creating, deleting and updating Tasks. The workflow logic is no more hard-coded in the Scheduler code, since it is dynamically configured using Spring XML. The support for scheduled activities is important to implement missing workflows defined in the Evolution Layer of the PoF Reference Model. These workflows include periodic preservation tasks, monitoring of resources and associated PV and other time-dependent activities. It is worth noting that the current implementation already supports such periodic processes using Spring XML: a dummy periodic process has been defined to test the stability of the routing engine (see Section 4.2), this activity simply triggers the Scheduler and provides a control message. This mechanism is also used for all periodic tasks, such as automatic preservation, which periodically checks the status of resources.

Documentation and Reference Links The APIs and usage examples are available in the software documentation, see Section 8.

License The component is released as open source under GPL licence, the same used for the PoF Middleware, see Section 8.

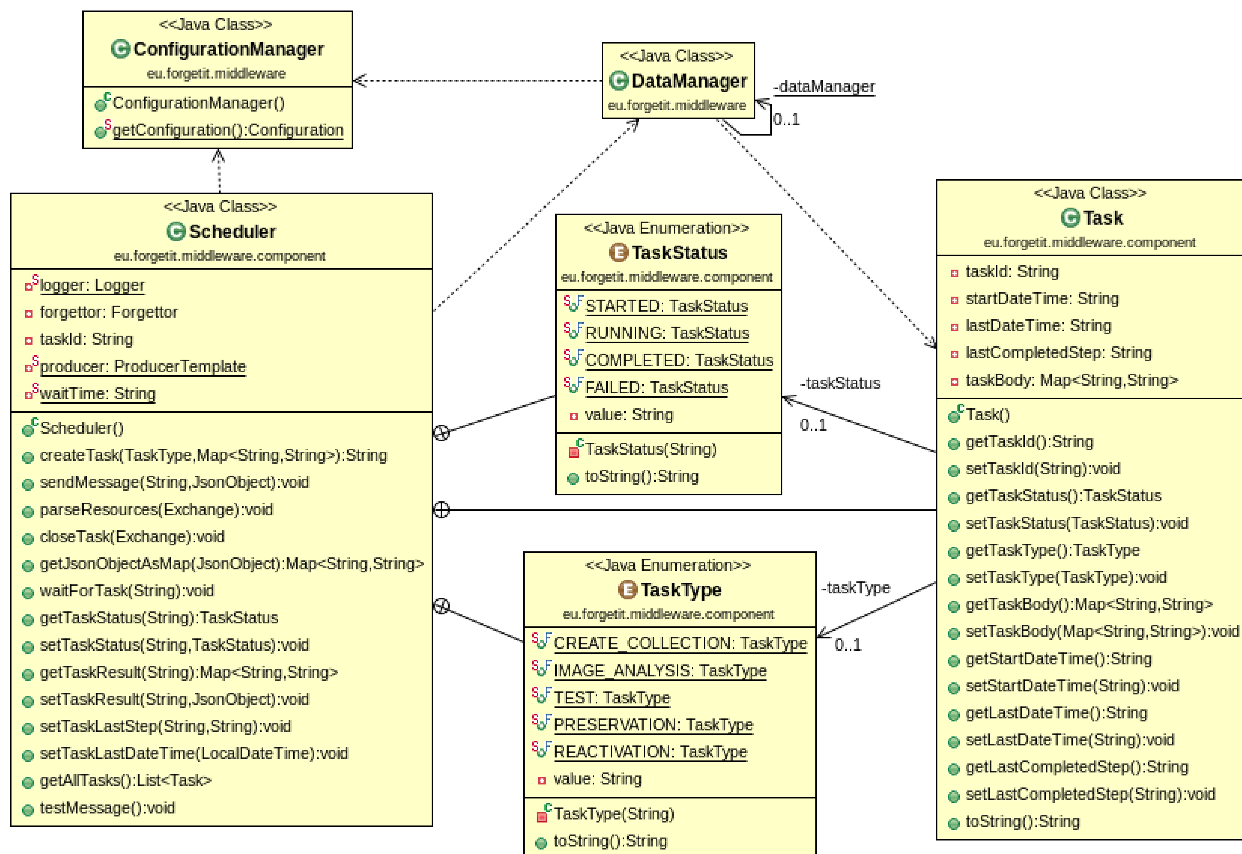

Figure 19: Class diagram for Scheduler component, with associated classes.

5.4 Extractor

Component Role The Extractor takes as input the original media items (e.g. a text, a video, an image collection or a collection of texts) and extracts information that is potentially useful not only for the subsequent execution of the Condensator (see Section 5.5), but also for other components or functionalities of the overall framework (e.g. search). Regarding visual information analysis methods, the Extractor implements: annotation with labels, near duplicate detection, image quality assessment (aesthetic and non-aesthetic) and face detection and clustering for images and videos. Text analysis methods cover basic linguistic processing, which is fed into the extraction of named entities and plays a role in the Condensator. Also, the Extractor implements a method that associates text and image items by sorting the images of an image collection based on their relevance with the given text.

WP and Deliverables The Extractor is developed within WP4, the contributing partners are CERTH, USFD, TT. The different technologies that are required for realizing the Extractor were reviewed in deliverable D4.1 [Papadopoulou et al., 2013]. Text analysis tools were initially introduced in D4.2 [Papadopoulou et al., 2014] and further developed in D4.3 [Solachidis et al., 2015], in which GATE [Cunningham et al., 2011] was also presented. Image analysis tools (image quality assessment, face detection, and feature extraction and concept-based image annotation) were initially presented in deliverable D4.2 [Papadopoulou et al., 2014]. Near duplicate detection, face clustering and multi-user time synchronization as well as updated versions of concept-based image annotation and face detection were presented in D4.3 [Solachidis et al., 2015]. Finally, in D4.4 [Solachidis et al., 2016], image quality assessment, image annotation, face detection and clustering and near duplicate detection have been updated and also extended to videos. In the same document, the text-image association method is also described.

Integration and Deployment All image and video analysis Extractor sub-components have been deployed as REST services running in CERTH servers. The text extraction components have been developed as GATE [The University of Sheffield, 2016] applications. GATE enables the rapid deployment and integration of GATE applications as web services. These can either be embedded directly into other Java applications and components or accessed as REST services using GATE WASP, as detailed in D4.3. Additional information about GATE is also available in [Cunningham et al., 2011]. The integration of the remote service providing Extractor functionalities is achieved using a `Service Activator` Enterprise Integration Patterns (EIP) (see Section 4.1.2): the service details are hidden to the other components. The Extractor implementation is made up of two main classes: the `Extractor` class exposes the image and video analysis methods and other methods to exchange messages with the broker, while the `ExtractorServiceConsumer` class provides the methods to interact with the REST service hosted by CERTH, which provides the actual image and video analysis methods. The `Extractor` class diagram is depicted in Figure 20. The `Extractor` method responsible for communicating with the messaging layer, consuming messages containing information about images and videos to be processed, makes use of `Exchange` class, part of Apache Camel API. The

`Extractor` class parses the message and sends the appropriate request to the CERTH service through the `ExtractorServiceConsumer` class. An excerpt of the `Extractor` code is shown in Listing 4. The `ExtractorServiceConsumer` class converts the received parameters into a REST request and then parses the response of the CERTH server, returning the information to the `Extractor` class. The main advantage of using a `Service Activator` pattern is the possibility to hide the details of the RESTful service (the response can change or the URL can be updated, for example) and also to deal with the issues related to web services, such as latency or unavailability of the service, just to name a few. The information about the CERTH service is stored in a configuration file and retrieved using the `ConfigurationManager` class. The execution of a particular image/video analysis method is supported by the use of a Java enumeration, `MethodType`. As shown in Figure 20, the progress of each image/video analysis task is managed by the Scheduler. The integration of the Extractor component in the middleware workflows is described in Section 4.

API and I/O Formats The REST APIs are documented in D4.4. The response of the web server is returned in XML format. For example, the image quality assessment takes as input an image (or a set of images) and returns its visual quality score by examining the presence of visual artifacts such as low contrast, noise, blur, etc., while the image annotation calculates the confidence scores for a set of concepts which indicate how much each concept is related to the image, taking as input an image (or a set of images) and returning for each image a vector that contains the confidence scores for all the concepts.

Status In the first framework release, the Extractor contained two image analysis subcomponents, for concept detection and image quality assessment. The second version had an updated concept detection method and two new methods: near duplicate image detection of an image collection and face detection. Furthermore, all implementations were written in C++ and were much faster than the previous ones. In the current version, the methods have been updated and also support video analysis. More specifically, image annotation, near duplicate detection, face detection and face clustering have been updated; and the aesthetic image quality assessment, as well as text and image association methods have been added. Regarding video analysis, the Extractor supports shot and scene segmentation, aesthetic and non-aesthetic quality assessment, annotation with labels, near duplicate detection and face detection and clustering. The text components are integrated via GATE WASP (see D4.3), allowing for an infinite variety of applications to be made available via the Extractor and integrated within the use case tools.

Documentation and reference links Additional information about the Extractor component and the RESTful web service hosted by CERTH can be found in deliverable D4.4 [Solachidis et al., 2016], which also provides some usage examples.

License CERTH libraries are Copyright ©2013-2016 CERTH, third-party libraries are available under open source (BSD) or as patented code in some countries. Some of the image analysis sub-components make internal use of third-party software and libraries, such as OpenCV (BSD license) and Liblinear (Copyright ©2007-2015 the LIBLINEAR

Project). GATE (and associated software) [The University of Sheffield, 2016] is available under an open source license, mostly GNU LGPL v3, although some code is covered by the GNU AGPL.

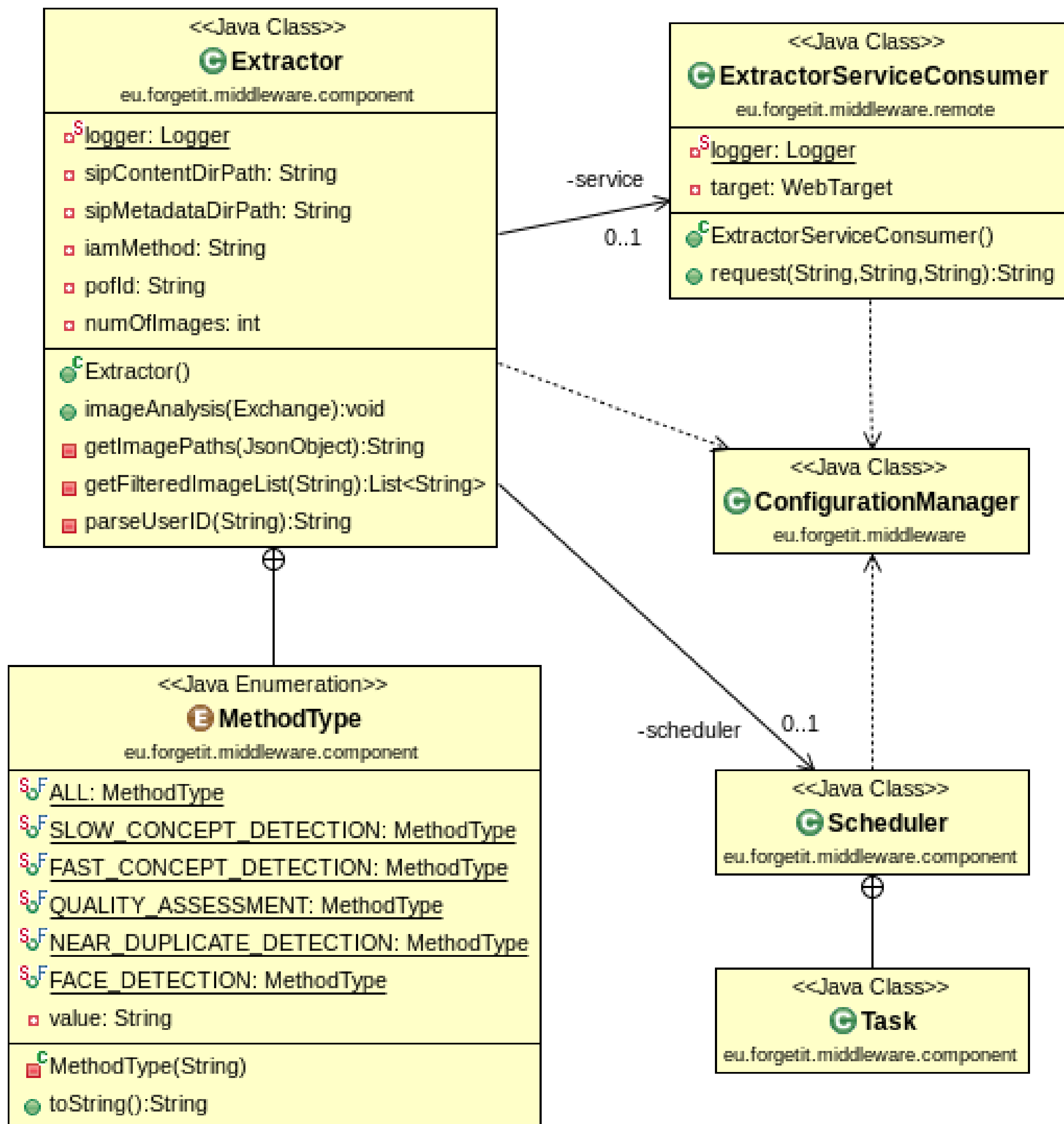

Figure 20: Class diagram for Extractor, with associated classes.

5.5 Condensator

Component Role The Condensator takes as input the output of the Extractor (see Section 5.4), in order to generate a condensed output of text and media items. Based on this input, the Condensator performs further text, image and video analysis tasks whose results are specific to the condensation process and thus of no need to other parts of the ForgetIT system. No feedback loop from the Condensator back to the Extractor is performed (thus, the Condensator can only be called after the Extractor has been executed for the processed data, and the Condensator results are not fed in any way back to the Extractor). The final output of the Condensator are the condensed media items or collections (subset of media items, condensed text etc.). Regarding images/videos, the current release contains a clustering method that is able, based on image/video features and metadata (capture time and GPS location) to cluster them into separate events and then to extract a representative image/video from each cluster in order to deliver a condensed version of the initial collection. For what concerns text, the Condensator provides single and multi-document summarization.

WP and Deliverables The Condensator is developed within WP4, the contributing partners are CERTH, USFD, TT. The different technologies that are required for realizing the Condensator were reviewed in deliverable D4.1 [Papadopoulou et al., 2013]. The text summarization methods and the image clustering methods are presented in deliverables D4.3 [Solachidis et al., 2015] and D4.4 [Solachidis et al., 2016].

Integration and Deployment The image and video clustering sub-components in the Condensator has been deployed as REST services running in CERTH servers. The text extraction components have been developed as GATE [The University of Sheffield, 2016] applications. These can either be embedded directly into other Java applications and components or are accessed as REST services using GATE WASP as detailed in D4.3. Similarly to the Extractor, the integration of the remote service providing Condensator functionalities is achieved using a `Service Activator` Enterprise Integration Patterns (EIP) (see Section 4.1.2). The Condensator implementation is made up of two main classes: the `Condensator` class exposes the image clustering methods and other methods to exchange messages with the broker, while the `CondensatorServiceConsumer` class provides the methods to interact with the REST service hosted by CERTH, which provides the actual image/video clustering methods. The `Condensator` class diagram is depicted in Figure 21. The `Condensator` method responsible for communicating with the messaging layer, consuming messages containing information about images/videos to be processed, makes use of `Exchange` class. As described for the Extractor, the `Condensator` class parses the message and sends the appropriate request to the CERTH service through the `CondensatorServiceConsumer` class, which converts the received parameters into a REST request and then parses the response of the CERTH server, returning the information to the `Condensator` class. The information about the CERTH service is stored in a configuration file and retrieved using the `ConfigurationManager` class. As shown in Figure 21, the progress of each Condensator task is managed by the Scheduler.

API and I/O Formats The REST APIs are documented in D4.4. The response of the web server is returned in XML format, as done for the Extractor.

Status In the first framework release, the Condensator contained an image clustering method that was using only image visual features. The second version has an updated clustering method that employs more image features and doesn't require the number of clusters as input. Furthermore, this implementation is in C++ and it is much faster than the previous one. The current version contains an updated clustering method for images and a new method for videos. The text components are integrated in the same way as integrated in the Extractor. Single document summarization was supported in the Year 2 version, while it has been extended to multi-document summarization in the current version.

Documentation and reference links Additional information about the Condensator component and the RESTful web service hosted by CERTH can be found in deliverables D4.3 [Solachidis et al., 2015] and D4.4 [Solachidis et al., 2016], which also provide some usage examples.

License CERTH libraries are Copyright ©2013-2016 CERTH, third-party libraries are available under open source (BSD) or as patented code in some countries. Some of the image analysis sub-components make internal use of third-party software and libraries, such as OpenCV (BSD license) and Liblinear (Copyright ©2007-2015 the LIBLINEAR Project). GATE (and associated software) [The University of Sheffield, 2016] is available under an open source license, mostly GNU LGPL v3, although some code is covered by the GNU AGPL.

5.6 Collector/Archiver

Component Role The Collector/Archiver is the framework component which communicates and exchanges data with both the Active Systems (Collector) and the Preservation System (Archiver). In the Preservation Preparation workflow (see Section 3), the Collector/Archiver is responsible for automatically fetching digital content from Active Systems (Information Systems), assemble content and metadata to create a Submission Information Package (SIP), ready for transfer to receiving Preservation System. This component automatically fetches content, metadata, and physical/logical structure from Active System using the CMIS protocol. At the end of the Preservation Preparation process, the Collector/Archiver assembles extracted additional metadata and content to create a SIP based on the eARD specification[5] and makes use of standardized metadata schemas adapted to receiving ingest functional entity in the Preservation System. The Collector/Archiver creates the SIP structure based on the package structure defined in the Preservation Broker Contract (PBC). When the SIP is built, the results from all components (secondary products or transformed resources, as well as additional metadata files) are collected. This process also includes file identification and computation of fixity

[5] eARD - http://xml.ra.se/e-arkiv/eard.html

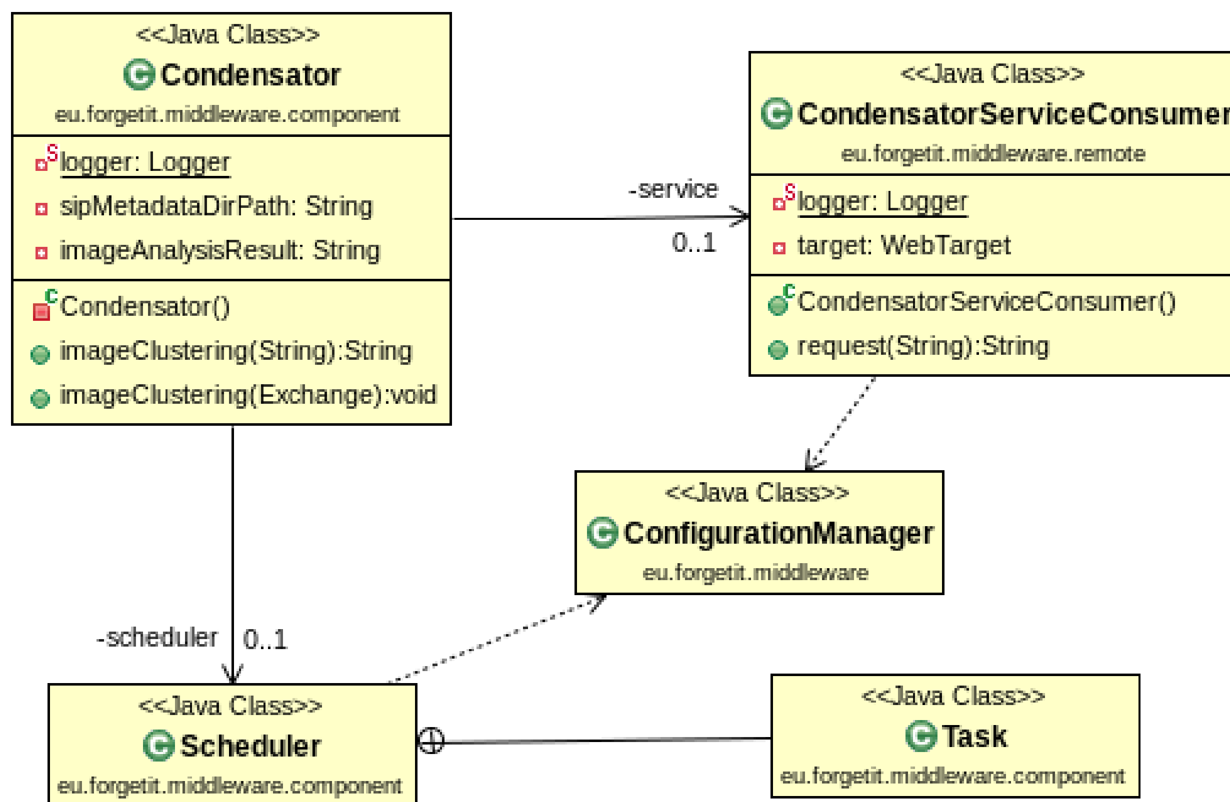

Figure 21: Class diagram for Condensator, with associated classes.

checksums. The Collector/Archiver is also in use in the Re-activation workflow where digital content is brought back from Preservation System to use in Active System. Resources in the Preservation System can be retrieved using the Collector/Archiver, which interacts with the ID Manager to get information about the resource IDs (see Section 5.1). In this process this component is responsible for uncompressing received DIP and restructuring it according to Active System needs, which should be defined in the Preservation Broker Contract. During the processing of Preservation Preparation and Re-activation workflows, the Collector/Archiver interacts with the PoF Middleware workflow manager (ESB) using the REST architectural style.

WP and Deliverables The Collector/Archiver component is developed within WP5, partners responsible are LTU and EURIX. The first version of Collector/Archiver has been described in D5.2 [Nilsson et al., 2014]. The second version, integrated in the second prototype, is described in deliverable D5.3 [Nilsson et al., 2015]. The current (third) version is described in deliverable D5.4 [Nilsson et al., 2016].

Integration and Deployment The Collector/Archiver is implemented by different software components. Two Java classes are deployed in the middleware Java code, `Collector` and `Archiver`. These classes provide methods for sending and consuming messages, using the `Exchange` class defined in Apache Camel APIs, and also methods for fetching content from the Active System and for importing content into the Preservation System. The core functionalities of the Collector/Archiver are deployed as a separate RESTful

web service running in the testbed environment, along with the middleware code. In order to interact with this service, a `Service Activator` pattern (see Section 4.1.2) is used and three classes have been defined: for the Collector functionality, the `CollectorServiceConsumer` communicates with the Collector REST APIs to trigger content and metadata retrieval from the Active System (see Figure 22); for the Archiver functionality, the `DigitalRepositoryServiceConsumer` class communicates with the Archiver REST APIs to package the content and with the Digital Repository REST APIs to ingest the SIPs and get information about archived content, while the `CloudStorageServiceConsumer` communicates with the Preservation-aware Storage System to store content in the SIP (see Figure 23). The Preservation System is described in Section 7, where further details about the Digital Repository and the Preservation-aware Storage System are provided. As depicted in Figure 22 and Figure 23, the Collector/Archiver uses the ID Manager to get information about IDs and to update the ID mappings, while the management of Collector/Archiver tasks is performed by the Scheduler. The activation of this component in the two main workflows is described in Section 4, where we also describe the use of Spring XML for workflow definition and for instantiating each component.

API and I/O Formats The Collector/Archiver exposes REST APIs which are documented in deliverable D5.4 [Nilsson et al., 2016].

Status and Workplan The third version of the Collector/Archiver has been improved and currently supports different metadata schemas, automatic fetching of additional contextual metadata and extraction of technical metadata, support for identification of file formats, extraction of file format identifiers based on PRONOM Persistent Unique Identifier (PUID), that provides the ability to integrate with the PRONOM format registry[6], support for management of physical and logical content structure from CMIS repository on the Active System, integration in the middleware with the implementation of REST interfaces for fetching, packaging and re-activation features, implementation of process logging to support functional validation and workflow redirection at errors and exceptions (using alternative workflows), and utilisation of the Preservation Broker Contract from the Context-aware Preservation Manager (CaPM) component (see Section 5.10) for configuring the Collector/Archiver.

Documentation and Reference Links Documentation of the component architecture and its interaction with internal and external components is in preparation, the description of the current version is available in deliverable D5.4 [Nilsson et al., 2016].

License The component is released as open source under GPL licence, the same used for the PoF Middleware, see Section 8.

5.7 Forgettor

Component Role The Forgettor is responsible for basic operations in the information value assessment. It takes information of the resources in the Active System, applying

[6]PRONOM - `http://www.nationalarchives.gov.uk/PRONOM`

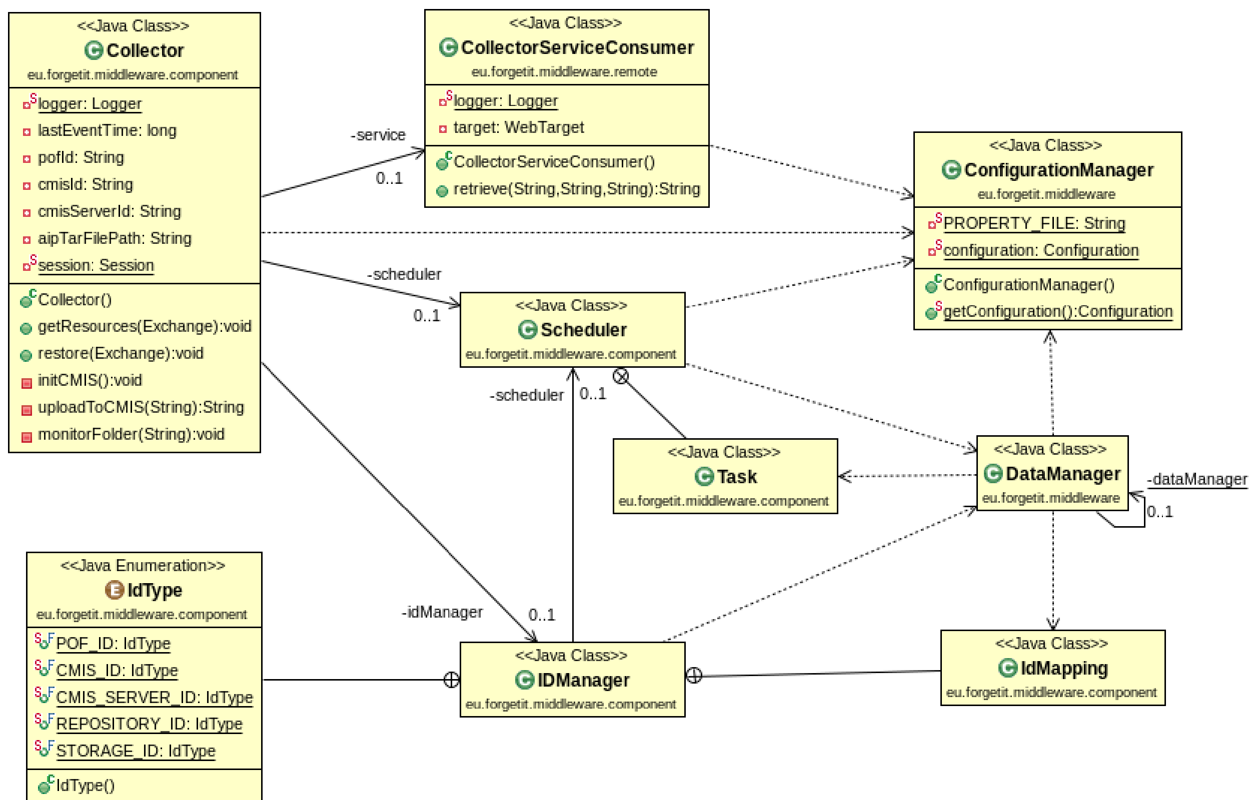

Figure 22: Class diagram for the Collector, with associated classes.

different methods in the managed forgetting framework, and provides outputs about the two values: MB and the PV for each resource. These values will then be used by other components (e.g. Collector, Archiver, or Contextualizer). This component consists of three major sub-components: MB assessor, PV assessor, and the Policy Engine. The MB assessor and PV assessor are responsible for computing the MB and PV in an automated or semi-automated fashion. These sub-components implement the novel managed forgetting methods that are developed within the WP3. The Policy Engine incorporates human preferences (individual or organizational) to the outputs given by the assessors, adapting them to specific scenarios defined by humans. The values of the Policy Engine will be used as the final output of the Forgettor that are exchanged to other components. Sub-components in the Forgettor are interfaced with other components via web services, web-based user interface, or standalone java packages and are called periodically as a background process.

WP and Deliverables This component is developed in the WP3, with contributions from LUH, DFKI, CERTH, TT. The foundations of the MB / PV assessors are described in deliverable D3.1 [Kanhabua et al., 2013]. The first prototype of the MB assessor is described in deliverable D3.2 [Kanhabua et al., 2014], and is followed up by a case study for the evaluation in deliverable D3.3 [Kanhabua et al., 2015], which discusses the Policy Engine design and implementation. Deliverable [Zhu et al., 2016] reports in details the PV assessors for both PIMO and PV active systems.

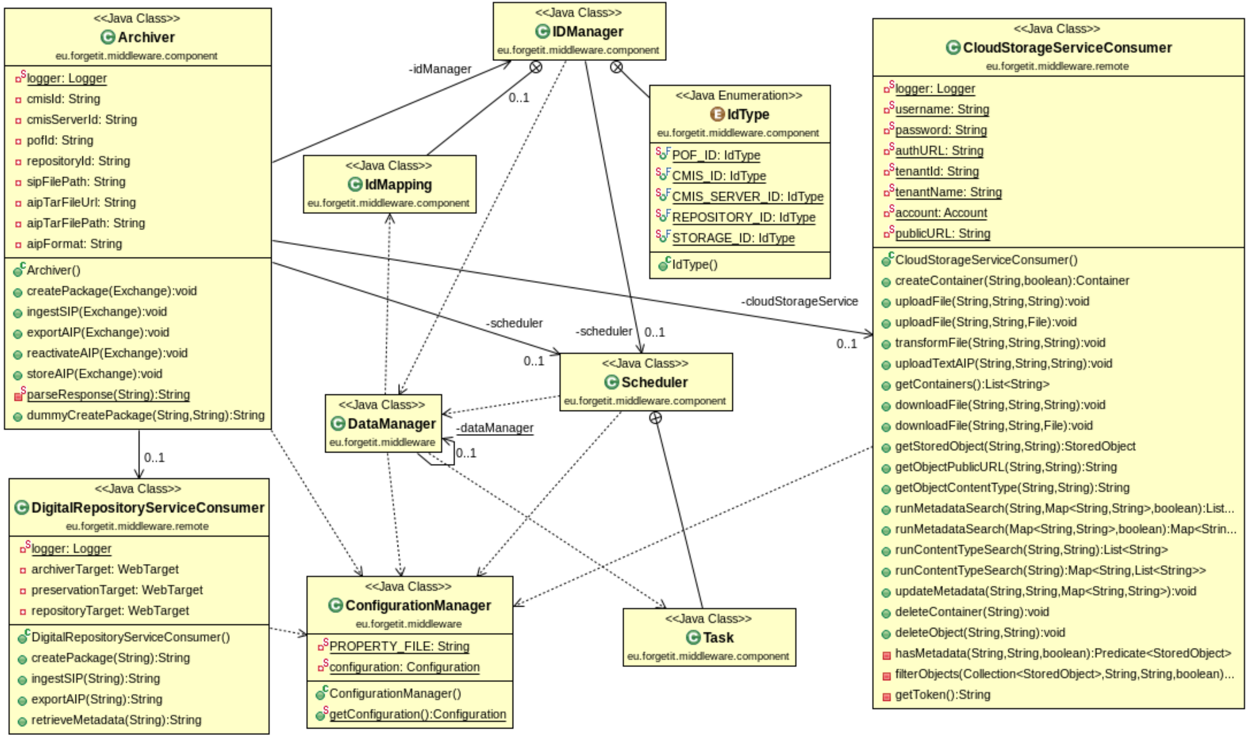

Figure 23: Class diagram for the Archiver, with associated classes.

Integration and Deployment The three sub-components are developed separately within WP3. Due to the difference in privacy constraints and architecture, the MB and PV assessors are deployed and integrated differently for the PIMO and TYPO3 active systems. For the PIMO, the assessors are developed as standalone Java packages in the active system, so as to preserve the privacy of resources in the experiment phase. The PV assessor for the PIMO system currently works with photo data, i.e. taking one collection of photos as input and returns the PV for each photo in the collection. The results are exchanged with the PoF framework via Web Services API. For the TYPO3, the two assessors are deployed directly in the middleware and constitutes part of the PoF framework. This is because TYPO3 data (FishShop case study, see D10.3) has less strict privacy requirements, making it ideal for prototyping the true integration of the assessors in the middleware. The method implemented by this sub-component has been described in detail in D3.4 [Zhu et al., 2016]. The Policy Engine is interfaced with web-based applications, as described in D3.3 Section 4 [Kanhabua et al., 2015].

API and I/O Formats The interfaces and input as well as results format for the Policy Engine are described in D3.3 Section 4. For the MB assessor, the experimental APIs are described in Appendix E.

Status and Workplan The Forgettor component is currently under development according to the plan in deliverable D8.1 [Gallo et al., 2013]. A new version will be integrated in the next release of the PoF Framework. Below reports the ongoing status of the development of the two sub-components.

Status of Policy Engine:

The current prototype of the Policy Engine consists of two main parts: The computational part facilitates the creation, management and exploration of policies, and the model part consists of domain expert-assisted policies, data models for specific scenarios. Current contributing partners of the two parts include L3S and DFKI. The data models and rules are provided by domain experts who operate the Active Systems, and embedded as POJO classes in the backend of the two applications. For the moment, information about digital documents are mirrored at the local database of the Forgettor to test the workflow. Next step would be exchanging the information via web services between the active systems and the Policy Engine, so as to preserve the privacy and support continuous integration.

Status of MB and PV Assessors:

The sub-components for assessing MB and PV values rely on the activity history of users in the information space (his Semantic Desktop or TYPO3 user log), as well as the ontological knowledge of the resource, including their structures and its relationships with other resources. In the middleware layer, the components are used as a service by the clients (PIMO) to numerically assess a resource, or as a local component (TYPO3). The computation can be triggered manually or periodically upon the arrival of the new logs.

The MB or PV assessors for the PIMO data have two parts. The first part serves as a background job that periodically gets triggered and estimates the resources' MB values. The results are then cached in a database. The second part, which is deployed directly to the middleware, is a web service repository that dispatches requests about MB values to the database and return results for respective context (time, persons who question, ...). For the TYPO3 data, these two parts of the assessors are directly deployed as local components in the middleware.

Documentation and reference links Additional information about the component can be found in deliverable D3.2 Section 2.2-2.3 [Kanhabua et al., 2014]. The data model and algorithms for the MB / PV assessor of TYPO3 data is described in D3.4 (Section 5.2) [Zhu et al., 2016]. The computational aspects of the PV assessor of the PIMO data is described in D3.4 (Section 7.2) [Zhu et al., 2016]. The Policy Engine is described in details in D3.3 (Section 4) [Kanhabua et al., 2015].

License: The policy engine and the advanced interface is developed using JBoss Drools Business Rule Management system[7] under the licence ASL 2. The basic interface is developed using Google Web Toolkit[8] under the Apache licence 2.0. The other components are available under GNU License GPL v3.0, Creative Commons License 3 and Apache License 2.0. Third party tools used in the MB / PV assessors are OpenCMIS API (Apache license 2.0) and Weka machine learning toolkit (GNU General Public License).

[7] Drools - http://www.jboss.org/drools

[8] Google Web Toolkit - http://www.gwtproject.org

5.8 Contextualizer

Component Role The Contextualiser takes as input the original media items (e.g. a text, an image, a collection of texts or a collection of images) and output from previous components (mainly the Extractor, see Section 5.4) and determines the wider *context* within which the item resides [Gorrell et al., 2015]. In conjunction with the Context-Aware Preservation Manager (see Section 5.10) the original item is then packaged for preservation along with the context information which enables the complete understanding of the item.

WP and Deliverables This component is developed within WP6, the contributing partners are USFD, LUH, CERTH, LTU, IBM, and DFKI. Deliverable D6.3 [Greenwood et al., 2015] describes the current status of the components and their integration within the PoF Middleware. Specifically three components for contextualization (two focused on text and one on images) are described alongside one component for the re-contextualization of text.

Integration and Deployment The prototype version of the contextualization via disambiguation component has been deployed as a REST service integrated into the PoF Middleware, based on GATE [The University of Sheffield, 2016]. The class diagram for the Contextualizer is shown in Figure 24, where the associated `ContextualizerService-Consumer` class is also shown. Also for the Contextualizer we make use of the `Service Activator` EIP (see above for further details).

API and I/O Formats A number of the contextualization components are fully integrated within the PoF Middleware and all are accessible to other consortium members directly in some form (usually as a RESTful service).

Status and Workplan The workplan for this component is focused on three main areas; use case integration, context evolution, and evaluation. Updated versions taking these three points into account will be documented and delivered as part of deliverable D6.4 [Greenwood et al., 2016] by the end of the project.

Documentation and Reference Links Additional information about the Contextualizer is available in deliverable D6.3 [Greenwood et al., 2015]. A demo version of one of the text based approaches to contextualization can also be accessed on GATE services web site[9].

License CERTH libraries are Copyright c 2013-2015 CERTH, third-party libraries are available under open source (BSD) or as patented code in some countries. Some of the image analysis sub-components make internal use of third-party software and libraries, such as OpenCV (BSD licence) and Liblinear (Copyright c 2007-2015 the LIBLINEAR Project). GATE (and associated software) is available under an open source licence; mostly GNU LGPL v3, although some code is covered by the GNU AGPL.

[9] GATE Contextualization Service - `http://services.gate.ac.uk/forgetit/contextualization/`

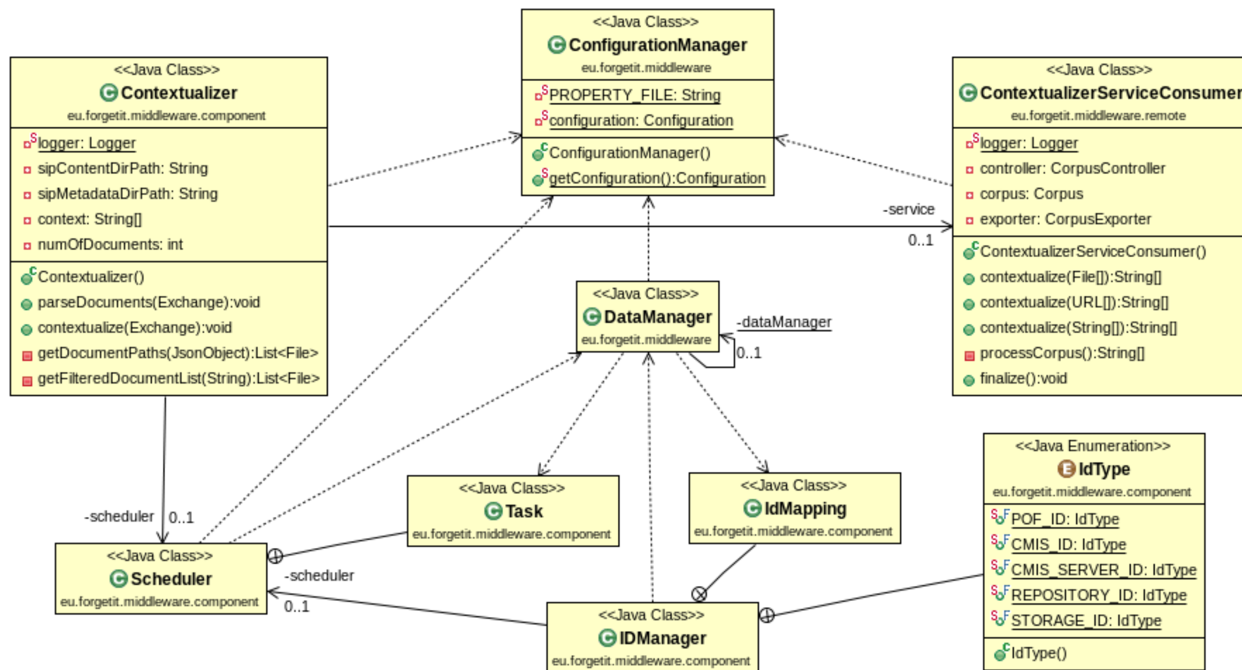

Figure 24: Class diagram for Contextualizer, with associated classes.

5.9 Navigator

Component Role The Navigator component provides the basic access to the preserved contextualized items. This allows access regardless of the presence of an Active System. This comprises metadata search which may take place within the archive (or object store) as well as search of the context information, the indexes for which are kept within the middleware for the purpose of efficient access. Different search approaches have been analysed in the project. In particular, we identified three different search types: Forgetful Search, Situation Search and Context Aware Search (see D6.4 Section 6 for further details). In the final version the main implemented functionality of the Navigator is the Situation Search, based on the capability to index metadata related to Situations and Collections in the middleware. Other search approaches have been investigated in WP3 and WP6.

WP and Deliverables The Navigator is a product of WP6 and WP8, a preliminary version was implemented in the second prototype (see D8.4 [Gallo et al., 2015a]).

Integration and Deployment The updated version of this component is now integrated within the PoF Middleware and is part of the middleware Java code.

API and I/O Formats The Navigator code available in the middleware provides APIs for the Situation Search, in order to index and search Situations and Collections using a Solr index with a dedicated core. The component is written in Java.

Status and Workplan A preliminary Navigator component was developed for the integration with the final prototype. It currently includes a component to index and search

Situations and Collections. Among the different search types mentioned above, the current implementation in the middleware only the Situation Search is supported, while for the other types preliminary software components have been developed in WP3 and WP6.

Documentation and Reference Links The APIs for the Situation Search are described in the code documentation, while for a general overview of Situation Search and other types please refer to D6.4 [Greenwood et al., 2016].

License The licence of the Java component implemented in the middleware is GPL. The implementation leverages Apache Solr which is available as open source under the Apache license.

5.10 Context-aware Preservation Manager

Component Role The Context-aware Preservation Manager (CaPM) is responsible for supporting the PoF Middleware activities by the creation of a submission agreement, a Preservation Broker Contract (PBC), which needs to be established between the producer information system (active system) and the digital preservation system (DPS). The PBC is an XML-file containing specifications and regulations in the form of structured information and rules that upholds the agreed-on structure and content of information packages and execution paths in the PoF middleware. The CaPM-PBC is always part of the execution of a preservation preparation workflow defined in D8.5 [Gallo et al., 2016] by receiving and dispatching requests from other PoF middleware components and by interacting with the PoF workflow infrastructure. A typical example of such interaction is provided by the Collector component: when retrieving objects from the active system it will always interact with the CaPM-PBC to identify from which active system it will fetch the objects by the retrieval of connection end-point information, if there are instructions regarding the fetch such as if physical/logical structure should be included, and if fetched objects is according to expectations regarding anticipated mime types, and that limit values for a single fetch is not exceeded. Another example is when the Archiver component creates the submission information package: the Archiver retrieves information about metadata standards to use. The Archiver also retrieves information from the PBC about the preservation organization, contact information, systems etc. used as provenance metadata. It also retrieves information about the package folder structure, definition of fixity algorithm, and the connection endpoint of the preservation system. Another example is related to the re-activation of content archived in the Preservation System, the CaPM-PBC provides information such as if there should be a migration at access and the migration path for a specific mime-type.

Besides being part of the preservation preparation and re-activation workflows the CaPM-PBC supports the scheduled transformation mechanism executed by the Preservation Aware Storage System [Chen et al., 2016] by providing agreed rules on the management of original objects and any copies. The CaPM-PBC may also contain information about agreed management of copies in the preservation system, as well as mechanisms regarding integrity checks, and how events in the preservation system should be communicated

back to the active system. This information is not intended to be executed on-the-fly in the preservation system, instead used as basis for manual configuration of mechanisms in the preservation system. The CaPM-PBC is invoked through a REST API based on the request path; different information from a specific PBC, identified by its ID, is retrieved. Another responsibility for the CaPM component is to support monitoring of DP activities through Activity Logging (AL) executed by other PoF middleware components. A typical example for the use of the CaPM-AL is to check status of the execution of activities in the preservation preparation workflow; the Archiver checks the logs created by the CaPM-AL to ensure that activities that should be executed according to the PBC has been carried out without errors. If an error is detected it sends a message to the PoF workflow manager. A REST API could also be invoked to request information from the CaPM-AL. Due to it's location in the middle of the interaction between active systems and preservation systems, the CaPM is also able to keep track of every object that passes through the PoF middleware. This functionality is referred to as Preservation Planning Support (PPS) and provides a bi-directional identification of systems involved, identified by the PBC, and the identification and logging of mime type versions that are part of an interaction. This information is useful as input to different preservation planning scenarios for detection of file format obsolescence and decision support in the choice of target file format in a migration scenario. This information is available by a REST API as a JSON output which can be imported by any preservation system and also provided via a web GUI through the CaPM-PPS.

WP and Deliverables This component is developed within WP5, the contributing partners are LTU, EURIX, IBM, DFKI and dkd. Deliverable D5.4 [Nilsson et al., 2016] describes its capabilities and placement in the PoF Middleware workflows.

Integration and Deployment The component is deployed in the PoF Middleware, and expose a number of REST services to be used by other components. The actual use of these services is not as high as it could be, mainly due to that CaPM has been developed after many other components since it was not in the original plan.

API and I/O Formats The CaPM component is written in Java and make use of available technologies. For the implementation of permanent store of data we make use of MySQL as RDBMS and a mix of JDBC and Persistence API for the communication with DB. The interaction with data stored in XML is done by the use of a java API for xml, JavaScript Object Notation (JSON) simple API, and Jersey RESTful API (JAX-RS). The logging mechanism is supported by the Apache log4j API. The web-GUI provided by the CaPM-PPS is implemented by the use of PrimeFaces and JSF API and the DB communication uses Java Persistence API. The classes part of the CaPM component is build to the `CaPM.jar` software package and the `capmmimeviewer.war`.

Status and Workplan The current solution for the Context-aware Preservation Manager provides expected functionalities within the project. The integration of it still needs improvements; which for example could mean different execution paths in the (PoF) Middleware based on information from the Preservation Broker Contract. Further development will take place outside the scope of this project.

Documentation and Reference Links A preliminary description of the Context-aware Preservation Manager is available in deliverable D8.1 [Gallo et al., 2013], where the role of the component in the framework is described, and in deliverable D8.2 [Gallo et al., 2015b], where the role of the component in the PoF Reference Model is explained. A preliminary design of the prototype is described in deliverable D5.3 [Nilsson et al., 2015], and deliverable D5.4 [Nilsson et al., 2016] holds documentation of the implemented version.

License The component is released under Open Source licence, the same used for the PoF Middleware, see Section 8.

6 Active Systems

In this Section we describe the current development of the two main user applications developed and tested in the project, the Semantic Desktop for the personal preservation and TYPO3 for the organizational preservation.

Both systems have been already described in detail in WP9 and WP10 deliverables, respectively. In this document we summarize the main achievements for the integration with the PoF Middleware.

Since the CMIS standard is crucial in our approach, we also describe how other user applications (e.g. developed in the other WPs for demonstration purposes) can be seamlessly integrated with the framework using CMIS.

6.1 Semantic Desktop

The Personal Preservation Pilots (see D9.3 [Maus et al., 2014], D9.4 [Maus et al., 2015], and D9.5 [Maus et al., 2016]) use the integration with the PoF Framework to provide services from ForgetIT. For full documentation of the Pilots and services used from ForgetIT, please refer to the mentioned deliverables.

In the following, the changes and enhancements in the **SD/PoF-Adapter** compared to the second release are described.

In the course of realizing Pilot II, the adapter to the PoF Middleware has been extended with following features.

The functionality of automatic preservation by the PoF, i.e., the PoF – namely the Forgettor – decides on what and when to preserve, required some extensions of the REST-API and the SD/PoF-Adapter.

First, as the Active System Semantic Desktop does the information value assessment both for MB and PV on the PIMO Server (see D9.4 for details), the PoF must be informed on the computed values for resources. Therefore, an interface method now allows to inform the PoF on updates of PV categories (gold, silver, bronze, wood, ash). This method allows for a bulk update of such values to reduce communication overhead.

Second, the PoF needs to be informed about the user's preferences on preservation. Therefore, an interface method allows to transfer and update the user's Preservation Broker Contract (PBC). This contract is described in D5.4 [Nilsson et al., 2016] and submitted in the method as a file. The current prototype does not (yet) cover all data for such a PBC, but restricts to key-value pair of PV category and Preservation Level (defaults are `premium`, `standard`, `basic`, `none`) and name and email-address of an alternative contact person. This data is entered by the user in a dedicated interface of the Semantic Desktop (see D9.4).

Finally, the technical handling of the context information accompanying a resource via the CMIS interface has been extended.

Due to the size restriction of CMIS attributes, context information from the PIMO (which will be archived as *Local Context*) is now transferred as separate `cmis:Document`. The document's ID is set in the `forgetit:context` attribute of the respective `cmis:Item`.

Technically, the context information export is an excerpt from the PIMO semantic graph describing the resource in the PIMO and its connection to other things such as topics for a document or persons attending an event. The format used for the exported excerpt is RDF/S using the PIMO Ontology RDF Schema and Turtle[10] as exchange format.

This interface was enhanced by handling collections of resources and using the additional context delivered by the SD/PoF Adapter.

6.2 TYPO3

The Organizational Preservation Pilot Application V1 (see D10.2 [Dobberkau et al., 2015a] and D10.4 [Dobberkau et al., 2016]) uses the CMIS standard to be able to provide content - originally restricted to use within TYPO3 only - to other systems, especially at this point the PoF Framework. Thus TYPO3 only acts as a data-deliverer, exemplifying the point that any system supporting CMIS could act as an Active System in the PoF Framework. TYPO3 is using an intermediary CMIS repository, as it is not providing a repository on its own. Relevant TYPO3 data structures are transformed to the CMIS standard as following: the page tree consists of `cmis:folder` objects and each content element on a page (text, image, video ...) is a `cmis:document`. Assets connected to these content elements are created as an CMIS object, connected with a `cmis:relationship`. Each of these *CMIS Objects* can be registered in the PoF Framework. The current communication consists of following parts:

Object Registration

After a new object was created, it is transferred to the CMIS repository. TYPO3 will receive the CMIS ID of this element. This identifier is registered in the PoF Framework, which is then able to access this object and pull required information. If the framework needs to know the item's relations, it can request all `cmis:relations`.

Meta-Data Enrichment

As any Active System could provide distinct meta information useful for PV and MB calculation, TYPO3 is exemplarily generating the following *meta information set* (exact values

[10] The *Terse RDF Triple Language* is a compact textual syntax for representing RDF, `http://www.w3.org/TR/turtle/`

to be changed) for each *website page*, split in two categories:
External Usage: *visits count*, *average length of a user's visit*, *bounce rate* and *incoming link count*
Internal Usage: *creation/modified data*, *status changes (visibility etc.)*, *edit history (editors, dates, ...)*, *external references* and *internal references*

The process to transfer this data to the PoF Framework is as following:

- Active System sends *add meta data* request to the PoF Framework containing the CMIS ID and the meta data
- PoF Framework asks the CMIS for the exact type of this CMIS ID
- PoF Framework knows *This object has this meta information* based on the algorithm generated by the PoF Framework before
- PoF Framework parses and saves the meta data
- PoF Framework calculates PV and MB, if more information is required, the CMIS repository can be queried

Semantic Enrichment

Within TYPO3 authors can semantically enrich their documents, the full description of the process is described in deliverable D10.3 [Dobberkau et al., 2015b]. The first step is the request for possible annotations from an annotation source, at the moment a *YODIE* endpoint [Greenwood et al., 2015] is used. Additionally the user adds annotations by hand. The annotations are stored inline in *RDFa Lite* format. Interested middleware components can easily extract these with the provided developed *GATE* plugin.

Archival and Restoration

For the moment only the manual archival is implemented in TYPO3. Before an object's deletion the PoF framework will receive an *archive* message, fetch the CMIS document (based on a CMIS ID) and put it into the archive. When this archival process is finished the document will be deleted from the CMIS repository.

When a document should be restored from the archive a *restore* request is sent to the PoF Framework. The request contains the ID of the object in the archive and the destination CMIS repository, where the PoF Framework will restore the object in.

6.3 CMIS-based User Applications

The adoption of CMIS standard for the bi-directional data exchange between the Active Systems and the PoF Middleware enables the seamless integration of any user applica-

tion which supports CMIS for content publication. In the following we describe an example of such application which has been implemented for the second release.

Photo Summarization

A user application has been developed to support users in the selection of personal photos for preservation (see deliverable D9.3 [Maus et al., 2014]). This application offers different methods that users can exploit to automatically select valuable photos from their collections for subjecting them to special preservation activities without the requirement of an existing Semantic Desktop. The photos selected from a given collection, along with a set of metadata, are then stored into a publicly accessible CMIS server. This simplifies the data exchange with the ForgetIT middleware, which can retrieve the selected photos and preserve them into the ForgetIT archive.

7 Preservation System

According to the current architecture diagram (see Figure 1 in Section 2), the Preservation System is made up of two main components: the Digital Repository and the Preservation-aware Storage System. In the following Sections we briefly summarize the main changes with respect to the first release. Additional information about the implementation can be found in deliverables D8.1 [Gallo et al., 2013], D8.3 [Gallo et al., 2014] and D7.3 [Chen et al., 2015].

7.1 Digital Repository

The Digital Repository is implemented using the DSpace platform. For the second framework release we updated the DSpace software to the last stable version 5.5. DSpace code is available as open source on Sourceforge [dspace,] and GitHub [dspace, 2016], the documentation is available on the project web site[11]. A customized installation guide for DSpace was provided in D8.3 for the first prototype, an updated version is available in Appendix C.

In order to enable the interaction between the PoF Middleware and the DSpace repository, we make use of DSpace REST APIs for all CRUD operations concerning Communities, Collections and Items, while additional REST APIs have been implemented for specific tasks related to the access and the ingest processes which are not supporting by the current DSpace REST APIs. The ingest interface is used to trigger operations related to the SIP validation, its submission and the creation of the AIP. The access interface is used for the dissemination of the AIP. With the new REST APIs the ingest and access processes can be managed at the metadata and bitstream level, enabling the user to also update the existing information for preserved content, as described in the following.

DSpace internal data model is represented in Figure 25. Digital objects are organized into several layers such as Collections, Communities, Items, and Sites. This data model supports the package structure defined in WP5 and the definition of the PoF information model in D8.5 [Gallo et al., 2016]. Currently, there is a direct mapping between the DSpace collections and the ForgetIT collections described in Section 4 and Section 6 (the collection information is stored in the CMIS metadata, fetched by the Collector and included in the AIP package).

The latest DSpace release provides a implementation of REST APIs[12]. The previous versions only provided READ interfaces, whilst the new release includes CRUD operations: the creation of collections and items, the upload of resources (bitstreams) and metadata or the retrieval of digital items can be performed using the REST APIs only. As a consequence, it is now possible to register new items with only metadata and store the actual files only in the cloud storage. This preservation mechanism has been implemented for

[11] DSpace - http://www.dspace.org

[12] DSpace 5.x REST API: https://wiki.duraspace.org/display/DSDOC5x/REST+API

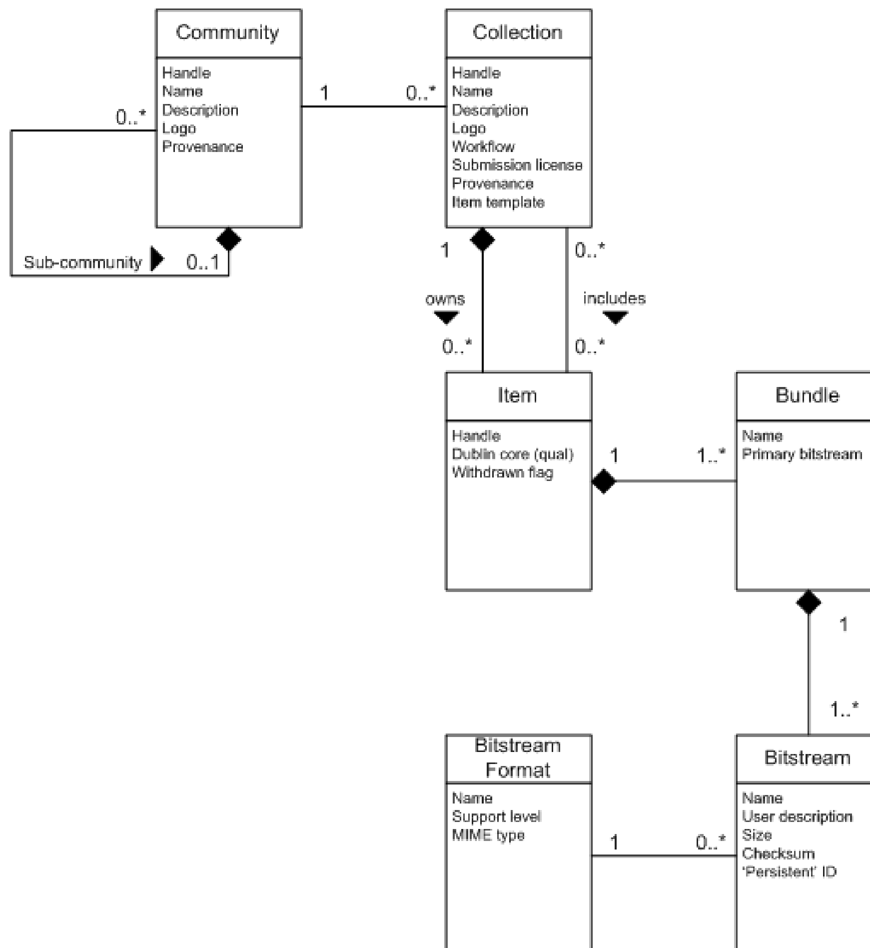

Figure 25: DSpace data model diagram.

the third release, in parallel with the development of the cloud storage (see next Section), with additional Storlets implementing preservation tasks.

DSpace is compliant to Open Archival Information System (OAIS) model (see deliverable D8.1): the main OAIS functionalities such as *Ingest*, *Access* or *Data Management* are implemented and the package exchange is inspired by the OAIS approach. DSpace implements the repository packages as Submission Information Package (SIP), Archival Information Package (AIP) and Dissemination Information Package (DIP) (see [CCSDS, 2012]). The relationship between the PoF Reference Model and the OAIS model is discussed in detail in deliverable D8.2 and will not be repeated here. From an implementation point of view, we adopted OAIS terminology for the packages to be compliant with DSpace.

The ingest and access endpoints exposed by the Preservation System are depicted in Figure 26. The code is written in Java and is deployed in the main PoF Middleware Java project, as part of the `eu.forgetit.preservation.server` package. The main class is `ServiceEndpoint`, which contains JAX-RS annotated methods which are published as REST APIs using Java Jersey. The server responses are provided in different

formats (XML, JSON, etc.). The other classes depicted in Figure 26 are used for other tasks required to interact with DSpace tools. The endpoint of the Preservation System is used by the Archiver component and is configured in the `DigitalRepositoryServiceConsumer` class of the Archiver (see Section 5.6).

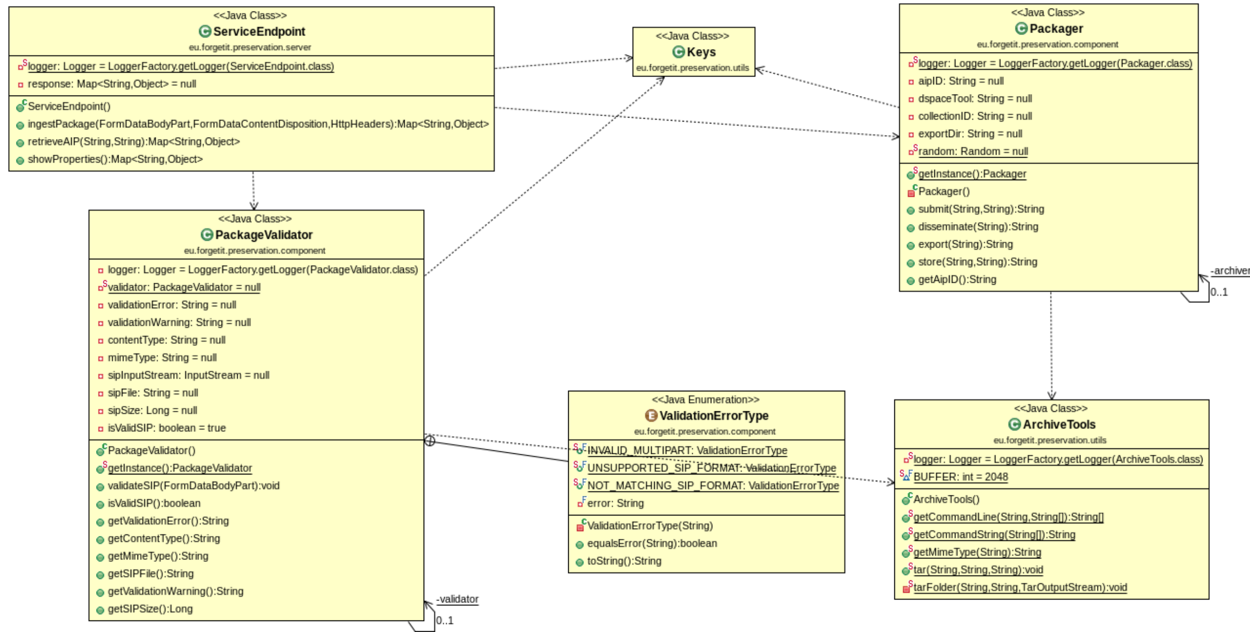

Figure 26: Class diagram for Preservation System endpoint, with associated classes.

7.2 Preservation-aware Storage System

The ForgetIT cloud-based Preservation-aware Storage system described in deliverable D7.3 [Chen et al., 2015], serves as the backend storage for the ForgetIT Digital Repository component. It is built on top of the OpenStack Swift object store, which is one of the top open-source cloud projects [Baker, 2014]. Our cloud storage has computational abilities, which come from a *storlet* engine that we have designed and implemented. The storage computational abilities allow offloading of preservation functionalities to the storage. Our cloud-storage also has metadata-search capabilities, which facilitate the retrieval of preserved information.

Storlets are typically used to transform the data, filter the data, or analyze the data, all in the object store. One of the main use cases for the use of storlets in an archival information system is that of format transformations. This is needed, for instance, when a file format has become obsolete.

The storlet engine works on streams, which means that storlets can begin streaming out the transformed data almost immediately, long before the entire transformation is complete. We demonstrate this through a storlet for the streaming of low-quality versions of videos, in order to create a video equivalent of photo thumbnails. In digital preservation it

is common to allow a user to get a low-quality viewable version of a resource, to help him decide whether he would like to directly access the resource preserved in the archive. The low-resolution storlet works on AIP packages, which are packaged as tar objects; these AIP packages contain both data and metadata files. The storlet extracts the data portion of the package, and streams out the transformation of that portion.

We have implemented and tested mechanisms of differentiating between the performances of different Preservation Levels. In one experiment we were able to improve the performance of Premium objects by 33.5 percent, at the expense of the Standard objects performance. We do this by giving different levels of replication to different Preservation Levels, and also tuning the "CPU shares" given to the different Preservation Levels.

We have turned the Storlets project into an open-source project, managed through the OpenStack Continuous-Integration platform (see deliverable D7.4 [Chen et al., 2016]). The Storlet engine is released under the Apache License version 2.0 to conform to the license used for the main OpenStack distribution. Using the OpenStack infrastructure, we have set up a set of tests which are run every time someone proposes a new contribution to the code; these tests make sure that the proposed new code adheres to our code-quality standards, and does not break the storlet-engine functionality. We are also working on cultivating a community around the Storlets project.

8 Third Prototype Implementation

Information about software development, deployment and testing was also provided for the previous prototypes in deliverables D8.3 [Gallo et al., 2014] and D8.4 [Gallo et al., 2015a].

In the following we summarize the relevant information for the development of the PoF Middleware and the RESTful server of the Digital Repository. For the implementation of the Active Systems, the middleware internal components and the Preservation-aware Storage System please refer to the related deliverables from other WPs.

Software Development

For what concerns the development of the PoF Middleware and Preservation System, two separate Java EE web projects have been created. Since the applications in the framework are distributed, we use the Java Enterprise Edition (EE) framework and the Eclipse Integrated Development Environment (IDE)[13] bundle for Java EE Developers. The version of Eclipse IDE used for the development is 5.1 (Mars), while for compilation we upgraded the code to use Oracle Java JDK 8. The source code is available on project SVN repository and Trac[14] is used as issue tracking system. Apache Maven[15] is used to compile and build the Java projects.

Software Projects and Packages

The Java packages of both projects are briefly described in Table 2 and Table 3, while the UML package diagrams are shown in Figure 27 and Figure 28. In both the Figures and the Tables we omitted test packages (used mainly for unit tests).

The UML packages in the model correspond exactly to the Java packages in both applications. Concerning the namespaces, the fully qualified names in the UML diagram based on UML specification are converted using Java naming convention, so for example the namespace for `middleware::component` sub-package corresponds in the Java code to `eu.forgetit.middleware.component`.

[13] Eclipse - http://www.eclipse.org
[14] The Trac Project - http://trac.edgewall.org
[15] Apache Maven - http://maven.apache.org

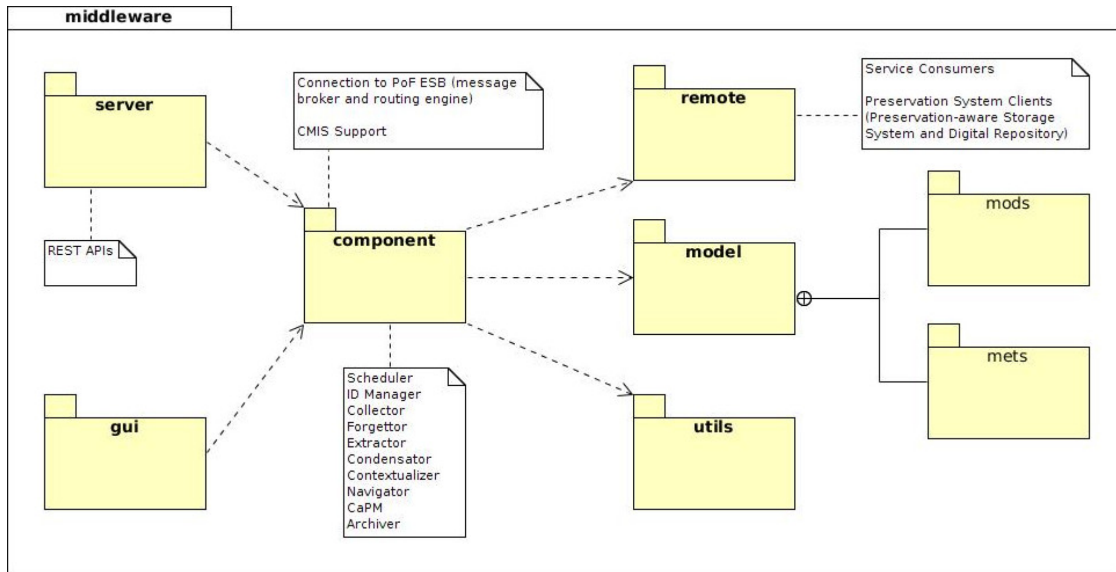

Figure 27: UML package diagram for the PoF Middleware.

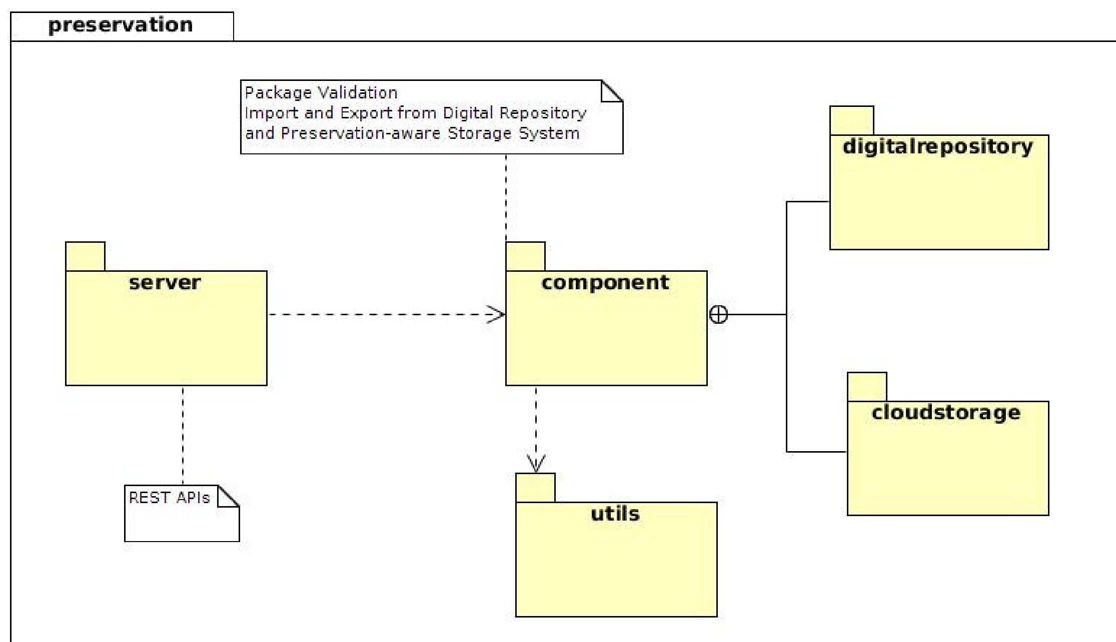

Figure 28: UML package diagram for the Preservation System.

Package	Description
eu.forgetit.middleware	Classes used by all other packages, to perform basic functions such as configuration and data management. ConfigurationManager manages properties for all components, for the broker and the workflows. DataManager provides APIs for the persistence of data in the middleware internal DB (e.g. IDs and tasks).
eu.forgetit.middleware.broker	Auxiliary classes associated to messaging component, e.g. to retrieve information shown in the GUI or for message logging. This package is no more responsible for the message exchange and routing, because this is implemented using Spring framework: all instances of message producers and consumers, the queues, the routes and the activation of the components are managed by ActiveMQ and Camel.
eu.forgetit.middleware.component	A class for each PoF Middleware component, implemented as EJB and exposing public methods accepting and producing Camel Exchange objects when consuming and producing messages. The classes implement also private methods for specific functionalities required for processing information extracted from messages and for executing specific tasks. These classes often instantiate service consumers in order to exchange information with external services providing specific functionalities.
eu.forgetit.middleware.gui	Auxiliary classes used by part of the middleware GUI, mainly for what concerns the status of resources, the ID mappings, the running tasks and the logging messages. The classes in this package will be updated to be used in the new middleware GUI based on hawtio.
eu.forgetit.middleware.model	Three main classes, Situation, Collection and Item, to represent situation, collections and single items to be preserved. The sub-packages contain auto-generated classes corresponding to specific metadata schemas, such as METS and MODS, used when preparing packages for preservation, since they are used in DSpace to represent archival objects and the associated descriptive metadata. Metadata classes have been automatically generated from the schema files (XSD) by means of JAXB.
eu.forgetit.middleware.remote	Classes implementing the Service Activator pattern, used by the components to interact with external (REST) services. Examples are ExtractorServiceConsumer (external service for image analysis) and ContextualizerServiceConsumer (external contextualization service), as well as service consumers for Digital Repository and cloud storage.
eu.forgetit.middleware.server	Classes implementing the middleware REST APIs (Java Jersey) and other support classes for specific tasks within the REST server, such as listeners or filters.
eu.forgetit.middleware.utils	Utilities used by the other packages, exposing mainly static methods: for example tools to create compressed folders or to obtain the MIME type of a given resource.

Table 2: Packages of the PoF Middleware project.

Package	Description
eu.forgetit.preservation	Classes used by all other packages, to perform basic functions such as configuration and data management. ConfigurationManager manages the properties for all components, such as the connection information about the Digital Repository and cloud storage REST services.
eu.forgetit.preservation.component	Two main classes, Packager and PackageValidator: the former is responsible for importing packages in the Digital Repository, while the latter performs extra package validation before ingest.
eu.forgetit.preservation.server	Classes implementing the Preservation System REST APIs (using Java Jersey). It also contains other support classes for specific tasks within the REST server, such as listeners or filters used during the REST requests. The Digital Repository uses the classes and methods in this package to process the ingest and access requests.
eu.forgetit.preservation.utils	Utilities used by the other packages, for example tools to manage different compressed archives or to validate the MIME type of a given file. The classes in these package mainly expose public static methods.

Table 3: Packages of the Preservation System projects.

Software Testing

In order to test the developed software for the PoF Middleware and the Preservation System, we performed unit tests using JUnit[16]. Using appropriate plug-ins, JUnit tests were executed within the Eclipse IDE during development, mainly for debugging purposes, while a list of tests is defined in the Maven project configuration and executed automatically during building. When generating the project artifacts (Java WAR files), we use the default Maven configuration to resolve and retrieve third-party dependencies, compile the source code, execute the unit tests and package the compiled code in a Java WAR file to be deployed in Apache Tomcat 8[17].

In order to test each component, we used dry run experiments. The components developed by other technical WPs have been tested by each partner separately, before releasing them for integration. The interaction between the PoF Middleware and the other components has been tested running different workflows. The end-to-end preservation workflow was tested incrementally, mainly by adding new steps as soon as the required input from previous steps was available.

[16] JUnit - http://junit.org
[17] Apache Tomcat - http://tomcat.apache.org

Software Deployment

For deployment we make use of virtualization: the different systems are running in the testbed environment as virtual machines (VMs). The virtualization infrastructure is based on KVM[18], a full virtualization solution for Linux.

The two Active Systems are deployed in dedicated VMs: for TYPO3 CMS a Linux server including also an instance of Alfresco (used for CMIS repository) is available; for the Semantic Desktop a Linux VM for the PIMO Server and a Windows VM for the PIMO Desktop are used (see Figure 1).

The two Java projects are deployed in separate instances of Tomcat 8, one running in the PoF Middleware virtual machine and one in execution within the Preservation System VM. For the PoF Middleware a Ubuntu Server VM is used to run the REST server, the broker and the routing engine, as well as all the components deployed within the middleware Apache Tomcat server.

For the Preservation System, a Ubuntu Server VM is used to run DSpace and the REST server, while a dedicated VM is used for the cloud storage, running the Storlet Engine and OpenStack Swift. Other VMs in the testbed provide additional services, such as a VPN server, a FTP server and a dedicated name server.

Software Documentation

The documentation of the source code is automatically generated using Doxygen[19] and will be available on the project web site at the following URL:

`http://www.forgetit-project.eu/en/project-results/code`

An internal task force was established within the project to publish the source code of the PoF Framework. This required an additional effort to clean up the source code, add APIs documentation, remove any dependency from the specific testbed configuration and, above all, to identify the core components for a minimal working system which could be released as open source and installed by interested users.

The pre-compiled binaries for the framework components (web applications, executables, libraries) as well as instructions for the installation and usage are provided by project partners. For detailed documentation about each component please refer to deliverables provided by the corresponding WP.

The backbone of the middleware is based on Apache Foundation software, such as ActiveMQ and Camel. The third-party dependencies used to compile the two Java projects are available on public Maven repositories and can be retrieved during compilation using project configuration in the `pom.xml` file. A few executables (e.g. `ffmpeg`) are used for specific tasks. The code has been developed and tested mainly for Linux (Ubuntu Server

[18]KVM - `http://www.linux-kvm.org`
[19]Doxygen - `http://www.doxygen.org`

14.04.4 LTS 64-bit), but since the code is written in Java it can be virtually executed in any operating system where a Java VM can be run. The executables written in other languages (e.g. C++) have to be replaced with the corresponding versions for that particular operating system (if the bundle is not available, they have a to be compiled from scratch).

Software Licensing

One of the goals of the ForgetIT project was to propose a new approach to digital preservation which can bridge the gap between information systems and preservation solutions. Many initiatives and individuals in the digital preservation domain believe that only an approach based on standards and open source technologies can produce valuable benefit for the different stakeholders, resulting in several open source preservation systems, which can be customized for specific requirements preventing vendor lock-in and the associated risk for the long term (see also deliverable D8.1 [Gallo et al., 2013], which contains an assessment of several open source digital preservation platforms).

Based on such ideas, the ForgetIT consortium agreed upon releasing under an open source license the core components of the PoF Framework.

The exact license type depends on the specific component. In parallel, project partners worked on improving the source code quality and documentation to a level adequate for dissemination. It is worth noting that the core libraries for the implementation of the PoF Middleware are available under the Apache license and that several components developed within the project have been already released as open source by the responsible partner, as described in the previous Sections. Any additional code developed to implement the PoF Middleware is available as open source, as well. The backbone of the PoF Middleware infrastructure is based on Apache ServiceMix components, which are available as open source.

Concerning the Preservation System, the Digital Repository is based on DSpace (available under the BSD license), while the licensing mechanism adopted by IBM for the Storlet Engine is also an open source license for the core part of the Storlet Engine (OpenStack Swift is already available as open source). Part of the effort for the cloud storage software is devoted to the proposal of including the Storlet Engine code in the OpenStack Swift mainstream development. A preliminary proposal has been submitted to the OpenStack community (see Section 7).

Concerning the Active Systems, TYPO3 is already available as open source, the licensing of additional customization is still under evaluation, while the Semantic Desktop will be available as open source.

9 Conclusion

The document provides a description of the final release of the PoF Framework. The updated prototype with all integrated components has been discussed. The software prototype reported in this document is the result of the effort performed by all partners during the whole project lifetime. Two main workflows (for Preservation Preparation and Re-activation) have driven the prototype implementation.

In the following sections, we briefly discuss the assessment of the results presented here, according to the WP8 performance indicators in the project proposal and then describe the lessons learned and the vision for the future.

9.1 Assessment of Performance Indicators

The expected WP8 outcomes, reported in the project proposal, are:

- the Preserve-or-Forget (PoF) Reference Model
- the PoF Framework

The third framework prototype refers to the second expected outcome, for which the following performance indicators have been identified in the project proposal:

1. *availability of interfaces and protocols exposed/published by software components to be integrated and delivered by technical work packages*,
2. *adequateness and effectiveness of the defined integration approach and strategy for the occurring integration tasks*,
3. *availability of infrastructure facilities for managing the development of the software framework (e.g. versioning system, software repository)*.

The prototype described here represents the final release of the PoF Framework. The results achieved so far are compatible with the expected progress and success indicators for WP8. Since the framework will be available as open source after the project end, future projects or adopters could provide further development and investigation to better support the whole reference model.

Indicator 1: APIs and protocols

The achievements for the first and second prototype already satisfied this indicator, similar considerations are reported here for the third prototype. The APIs and protocols of the components to be integrated have been tested, the third prototype integrates the components according to the integration plan in D8.1. The APIs published by the PoF Middleware and the Preservation System (Digital Repository and Preservation-aware Storage

System) are based on REST architectural style, hence different HTTP verbs are used to get and send data. The REST APIs have been implemented using Java reference software, to maximize integration with all external systems. Concerning the protocols, CMIS is used to retrieve resources and metadata from Active Systems. CMIS is a open standard protocol aimed to support interoperability and is widely adopted and supported. CMIS is also used to bring re-activated content back to use, since also the PoF Middleware publishes the re-activated content using its own CMIS repository. As a consequence, any user application supporting CMIS can be seamlessly integrated with the PoF Framework. Moreover, standard formats have been used for content packaging (XML-based formats such as METS, Dublin Core, PREMIS) and for communication with web services (XML or JSON).

Indicator 2: integration approach

The integration approach was established during the first year of project and is still valid. We leverage the best practices in Enterprise Application Integration (EAI), adopting well established concepts such as the Enterprise Service Bus (ESB) for the communication layer and Enterprise Integration Patterns (EIP) as industry level standard for complex integration patterns. Apache Camel, used for the message routing, implements all EIPs available in the literature, examples have been provided in the text and the benefits of such approach have also been discussed. The PoF Middleware is implemented as a Message Oriented Middleware (MOM), this approach has been further validated in the third year with the integration of additional components in the implementation of the new workflows defined in the PoF Reference Model.

A preliminary integration plan was summarized in Table 15 of deliverable D8.1, where we split the components in four categories and assigned an expected integration level for each framework release. For what concerns the Active Systems, the integration mechanism with the PoF is in place, according to the plan. Concerning the middleware shared components, compared to the second release, we completed the Metadata Repository and the Context-aware Preservation Manager implementation. Concerning the middleware core components and the Preservation System, the current status is compliant to the plan, now including an updated version of the Contextualizer and an integrated Condensator service, which was missing in the second prototype. Based on such considerations, we can estimate that all components are now fully integrated, although the Context-aware Preservation Manager, due to the complexity of the associated processes, would require further integration effort. The aforementioned integration plan was constantly discussed and monitored within the consortium by means of face-to-face meeting and periodic conference call, and appropriate actions have been taken to complete as much as possible the development and integration of all components on time for the final framework release.

Indicator 3: development and test infrastructure

The testbed environment was setup during the first year and is still accessible to all partners. The virtualization environment and the code versioning system (SVN) is maintained by EURIX. An issue tracking system (Trac) was used to keep track of all open issues identified during project meetings and periodic conference calls. It was used for ticketing, as well as to define milestones for the development (software releases, deadlines , etc.) and to share information about exceptions and errors. Each ticket was assigned to the appropriate partner, under the lead of EURIX as responsible for integration. Progress for each milestone and deadline can be monitored, taking into account open tickets. The approach adopted for software development is based on Agile methodology, using UML for sharing ideas and to describe software components, from preliminary sketches to complex modules. Only a minimal amount of additional documents were created and shared during the development phase, using the project wiki or other systems in the cloud (e.g. Google Drive) to prepare short technical notes and guidelines focusing on specific issues.

9.1.1 Evaluation of the PoF Framework

The evaluation of the framework during the third year mainly focused on the quality of the developed software and on the deployment process, analysing the effort required to install the prototype from scratch in a new environment, to customize or create new workflows and to extend the platform with additional components. As an example, an installation of the PoF Framework was performed at DFKI premises, using local resources and deploying many components in the new environment. This installation was pretty straightforward and no specific issues were pointed out.

Concerning the framework itself, the backbone of the middleware leverages established technologies for the broker implementation: the software components used for the messaging system and the workflow engine are provided by Apache ActiveMQ and Apache Camel, which are maintained and tested by a large community of developers and have been currently chosen also for enterprise grade solutions, such as JBoss FUSE middleware. Concerning the Preservation System, the Digital Repository is implemented by DSpace, which is also actively maintained and tested and, above all, is a solution adopted for many institutional repositories around the word (the number of acknowledged installations of DSpace from public institutions is currently a few thousands).

Moreover, many components have been tested separately within the corresponding WP, the evaluation of each component can be found in the corresponding WPs. Many components developed in the technical WPs leverage third party software tools that are usually actively maintained and tested by large open source communities.

For the tests of the two pilot applications and for the user studies, please refer to WP2, WP9 and WP10 deliverables.

9.2 Lessons Learned

The development of the PoF Framework was quite challenging, for several reasons:

- the number of components developed in the technical WPs was large, mainly for what concerns the middleware, and the architecture of the framework was quite complex;
- the integration effort involved almost all partners and required deep discussion and periodic checks of the status for the whole project lifetime;
- the development of the reference model, with the information and functional part and all the associated workflows, continued for the whole project lifetime, as a result of continuous investigation;
- the development of the components also was performed for all the three years, so additional integration effort was required for each framework release;
- the middleware had to integrate with existing information systems (e.g. PIMO and TYPO3) and preservation systems (DSpace and OpenStack Swift), which were in turn further developed during the project;
- the selection of candidate solutions for the middleware and archive implementation was conducted during the first year as part of the integration work.

The integration plan and the architecture of the whole framework were discussed in detail among all the project partners and this was crucial in order to deliver after the first year a working prototype implementing a basic end-to-end preservation workflow with components available at that time.

The use of best practices concerning software development and integration, as mentioned in this document, were helpful to reduce the issues related to integration and to provide a stable and flexible solution after the second year.

The full development of the reference model was completed at the very end of the project, according to the plan, but further investigation would be required to improve the implementation of the model, mainly for what concerns the evolution part of the model.

9.3 Vision for the Future

The source code of the PoF Framework will be available on the project web site as open source, so any potential adopter of ForgetIT outcomes can access it. The framework was intended as a preliminary implementation of the ForgetIT approach to digital preservation described in the reference model, so there is still room for future improvements.

The core technologies used for the implementation of the framework are well established and generally available as open source, so the integration of the PoF Framework with

existing solutions is simplified. The latest version of libraries and technologies were used for the implementation, so hopefully the maintenance of the code will not be an urgent issue on the short term.

10 References

[Baker, 2014] Baker, J. (2014). Survey Says: Openstack and Docker Top Cloud Projects. http://opensource.com/business/14/8/openstack-and-docker-top-cloud-projects. Retrieved March 2016.

[Bergljung, 2014] Bergljung, M. (2014). *Alfresco CMIS*. Packt Publishing.

[CCSDS, 2012] CCSDS (2012). Reference Model for an Open Archival Information System (OAIS) - recommended practice, ccsds 650.0-m-2 (magenta book) issue 2. also available as iso standard 14721:2012. `http://public.ccsds.org/publications/archive/650x0m2.pdf`.

[Chappell, 2004] Chappell, D. (2004). *Enterprise service bus*. O'Reilly Media, Inc.

[Chen et al., 2015] Chen, D., Loğoğlu, B., and Nilsson, J. (2015). ForgetIT Deliverable D7.3: Computational Storage Services - Second Release.

[Chen et al., 2016] Chen, D., Nilsson, J., Andersson, I., Gür, G., Greenwood, M., and Gallo, F. (2016). ForgetIT Deliverable D7.4: Computational Storage Services - Final Release.

[Cunningham et al., 2011] Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M. A., Saggion, H., Petrak, J., Li, Y., and Peters, W. (2011). *Text Processing with GATE (Version 6)*.

[DAI and ZHU, 2010] DAI, J. and ZHU, X.-M. (2010). Design and implementation of an asynchronous message bus based on activemq. *Computer Systems & Applications*, 8:062.

[Dobberkau et al., 2015a] Dobberkau, O., Due, C., Wegenast, G., Goslar, J., Doan, P., Egerer, S., Krasteva, V., Schaffstein, S., Wolters, M., Mayer-Schönberger, V., and Niederée, C. (2015a). ForgetIT Deliverable D10.2: Organizational Preservation Pilot Application V1.

[Dobberkau et al., 2015b] Dobberkau, O., Goslar, J., Gsedl, I., and Wolters, M. (2015b). ForgetIT Deliverable D10.3: Organizational Preservation Pilot Application V2.

[Dobberkau et al., 2016] Dobberkau, O., Goslar, J., Gsedl, I., and Wolters, M. (2016). ForgetIT Deliverable D10.4: Organizational Preservation Report.

[dspace,] dspace. DSpace SourceForge Repository. `https://sourceforge.net/projects/dspace/files`.

[dspace, 2016] dspace (2016). DSpace GitHub Repository. `https://github.com/DSpace/DSpace`. Retrieved March 2016.

[Gallo et al., 2015a] Gallo, F., Ceroni, A., Tran, T., Chen, D., Andersson, I., Greenwood, M. A., Maus, H., Lauer, A., Schwarz, S., Papadopoulou, V. S. O., Apostolidis, E., Pournaras, A., Mezaris, V., and Goslar, J. (2015a). ForgetIT Deliverable D8.4: The Preserve-or-Forget Framework - Second Release.

[Gallo et al., 2013] Gallo, F., Kanhabua, N., Djafari-Naini, K., Niederée, C., Rad, P. A., Andersson, I., Lindqvist, G., Nilsson, J., Henis, E., Rabinovici-Cohen, S., Maus, H., Steinmann, F., Mezaris, V., Papadopoulou, O., Solachidis, V., Dobberkau, O., Doan, P., Greenwood, M., Allasia, W., and Pellegrino, J. (2013). ForgetIT Deliverable D8.1: Integration Plan and Architectural Approach.

[Gallo et al., 2015b] Gallo, F., Niederée, C., Andersson, I., Nilsson, J., Chen, D., Maus, H., Greenwood, M., and Logie, R. (2015b). ForgetIT Deliverable D8.2: The Preserve-or-Forget Reference Model - Initial Model.

[Gallo et al., 2016] Gallo, F., Niederée, C., Andersson, I., Nilsson, J., Chen, D., Maus, H., Greenwood, M. A., Logie, R., and Allasia, W. (2016). Deliverable D8.5: The Preserve-or-Forget Reference Model - Final Model.

[Gallo et al., 2014] Gallo, F., Niederée, C., Kanhabua, N., Chen, D., Maus, H., Solachidis, V., Damhuis, A., Greenwood, M. A., and Pellegrino, J. (2014). ForgetIT Deliverable D8.3: The Preserve-or-Forget Framework - First Release.

[Gorrell et al., 2015] Gorrell, G., Petrak, J., and Bontcheva, K. (2015). Using@ twitter conventions to improve# lod-based named entity disambiguation. In *The Semantic Web. Latest Advances and New Domains*, pages 171–186. Springer.

[Greenwood et al., 2015] Greenwood, M. A., Ceroni, A., Petrak, J., Gorrell, G., Mezaris, V., Papadopoulou, O., Solachidis, V., Eldesouky, B., and Maus, H. (2015). ForgetIT Deliverable D6.3: Contextualisation Tools - Second Release.

[Greenwood et al., 2016] Greenwood, M. A., Solachidis, V., Papadopoulou, O., Apostoliditis, K., Galanopoulos, D., Tastzoglou, D., Mezaris, V., Eldesouky, B., Maus, H., Tran, N. K., Hube, C., Niederée, C., Petrak, J., and Gorrell, G. (2016). ForgetIT Deliverable D6.4: Contextualisation Framework and Evaluation.

[Henjes et al., 2007] Henjes, R., Schlosser, D., Menth, M., and Himmler, V. (2007). Throughput performance of the activemq jms server. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 113–124. Springer.

[Hohpe and Woolf, 2003] Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Ibsen and Anstey, 2010] Ibsen, C. and Anstey, J. (2010). *Camel in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[Kanhabua et al., 2015] Kanhabua, N., Niederée, C., Ceroni, A., Naini, K. D., Kawase, R., Tran, T., Maus, H., and Schwarz, S. (2015). ForgetIT Deliverable D3.3: Strategies and Components for Managed Forgetting - Second Release.

[Kanhabua et al., 2013] Kanhabua, N., Niederée, C., Loggie, R., Tran, T., Djafari-Naini, K., Maus, H., and Schwarz, S. (2013). ForgetIT Deliverable D3.1: Report on Foundations of Managed Forgetting.

[Kanhabua et al., 2014] Kanhabua, N., Niederée, C., Tran, T., Nguyen, T. N., Djafari-Naini, K., Kawase, R., Schwarz, S., and Maus, H. (2014). ForgetIT Deliverable D3.2: Components for Managed Forgetting - First Release.

[Maus et al., 2014] Maus, H., Schwarz, S., Eldesouky, B., Jilek, C., Wolters, M., and Loğoğlu, B. (2014). ForgetIT Deliverable D9.3: Personal Preservation Pilot I: Concise Preserving Personal Desktop.

[Maus et al., 2015] Maus, H., Schwarz, S., Jilek, C., and Gallo, F. (2015). ForgetIT Deliverable D9.4: Personal Preservation Pilot II: Concise Preserving Mobile Information Assistant.

[Maus et al., 2016] Maus, H., Schwarz, S., Jilek, C., Wolters, M., Rhodes, S., Ceroni, A., and Gür, G. (2016). ForgetIT Deliverable D9.5: Personal Preservation Report.

[Müller et al., 2013] Müller, F., Brown, J., and Potts, J. (2013). *CMIS and Apache Chemistry in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[Nilsson et al., 2016] Nilsson, J., Andersson, I., and Gallo, F. (2016). ForgetIT Deliverable D5.4: Workflow Model and Prototype for Transition between Active System and AIS - Final release.

[Nilsson et al., 2015] Nilsson, J., Andersson, I., Lindqvist, G., and Westerlund, P. (2015). ForgetIT Deliverable D5.3: Workflow Model and Prototype for Transition between Active System and AIS - Second Release.

[Nilsson et al., 2014] Nilsson, J., Andersson, I., Rad, P. A., Lindqvist, G., Päivärinta, T., Rabinovici-Cohen, S., Maus, H., Dobberkau, O., Allasia, W., and Gallo, F. (2014). ForgetIT Deliverable D5.2: Workflow Model and Prototype for Transition between Active System and AIS - First Release.

[Nilsson et al., 2013] Nilsson, J., Päivärinta, T., Rad, P. A., Maus, H., and Dobberkau, O. (2013). ForgetIT Deliverable D5.1: Foundations of Synergetic Preservation.

[OASIS, 2013] OASIS (2013). Content Management Interoperability Services (CMIS) Version 1.1. OASIS Standard. `http://docs.oasis-open.org/cmis/CMIS/v1.1/CMIS-v1.1.html`. Retrieved March 2016.

[ObjectDB, 2015] ObjectDB (2015). Fast Object Database for Java. `http://www.objectdb.com`. Retrieved March 2016.

[Papadopoulou et al., 2013] Papadopoulou, O., Mezaris, V., Greenwood, M. A., and Logoglu, B. (2013). ForgetIT Deliverable D4.1: Information Analysis, Consolidation and Concentration for Preservation - State of the Art & Approach.

[Papadopoulou et al., 2014] Papadopoulou, O., Mezaris, V., Solachidis, V., Ioannidou, A., Eldesouky, B. B., Maus, H., and Greenwood, M. A. (2014). ForgetIT Deliverable D4.2: Information Analysis, Consolidation and Concentration Techniques, and Evaluation – First Release.

[Rabinovici-Cohen et al., 2014] Rabinovici-Cohen, S., Henis, E., and Ta-Shma, P. (2014). ForgetIT Deliverable D7.2: Computational Storage Services - First Release.

[Snyder et al., 2011] Snyder, B., Bosanac, D., and Davies, R. (2011). *ActiveMQ in Action*. Manning Publications Co., Greenwich, CT, USA.

[Solachidis et al., 2016] Solachidis, V., Apostolidis, E., Markatopoulou, F., Galanopoulos, D., Tzelepis, C., Arestis-Chartampilas, S., Pournaras, A., Tastzoglou, D., Mezaris, V., Chen, D., Harnik, D., Khaitzin, E., Eldesouky, B., Maus, H., Greenwood, M., and Tan, A. S. (2016). ForgetIT Deliverable D4.4: Information Analysis, Consolidation and Concentration Techniques, and Evaluation - Final Release.

[Solachidis et al., 2015] Solachidis, V., Papadopoulou, O., Apostolidis, K., Ioannidou, A., Mezaris, V., Greenwood, M. A., and Maus, H. (2015). ForgetIT Deliverable D4.3: Information Analysis, Consolidation and Concentration Techniques, and Evaluation - Second Release.

[The University of Sheffield, 2016] The University of Sheffield (2016). General Architecture for Text Engineering (GATE). `https://gate.ac.uk`. Retrieved March 2016.

[Zhu et al., 2016] Zhu, X., Niederée, C., Tran, T., Ceroni, A., Naini, K. D., Tran, N. K., , Maus, H., and Jilek, C. (2016). ForgetIT Deliverable D3.4: Strategies and Components for Managed Forgetting Final Release.

Glossary

AIP Archival Information Package. 57, 58, 60

CMIS Content Management Interoperability Services. 3, 8, 10, 12, 22, 28–30, 32, 35, 42, 44, 53–57, 65, 68, 87, 88, 118, 119, 121, 124

CRUD Create Read Update Delete. 32, 34–36, 57

DIP Dissemination Information Package. 43, 58

EAI Enterprise Application Integration. 24, 68

EIP Enterprise Integration Patterns. 8, 24–26, 38, 41, 48, 68

EJB Enterprise JavaBeans. 32, 35, 36, 63

ESB Enterprise Service Bus. 8, 12, 22, 25, 26, 43, 68

IDE Integrated Development Environment. 61, 64

JMS Java Message Service. 26

JSON JavaScript Object Notation. 28, 35, 36, 59, 68, 84, 86, 125

MB Memory Buoyancy. 10, 34, 45–47, 53–55, 125

MOM Message Oriented Middleware. 8, 12, 22–24, 26, 68

OAIS Open Archival Information System. 58

PIMO Personal Information MOdel. 29, 45–47, 53, 54, 65

PoF Preserve-or-Forget. 3, 8–13, 15, 16, 20, 22, 24, 26–31, 33, 35–37, 43, 44, 46, 48–55, 57, 58, 61, 63–68, 70, 78, 86, 89, 118, 123

PV Preservation Value. 29, 31, 34, 35, 37, 45–47, 53–55, 87, 118–120

SIP Submission Information Package. 42, 44, 57, 58

UML Unified Modeling Language. 61, 69

UUID Universally Unique IDentifier. 32

XML eXtensible Markup Language. 26, 27, 32, 35–37, 39, 42, 44, 59, 68, 77–80, 82, 86

A Middleware Configuration and Administration

In the following we provide additional examples about the actual configuration of the middleware, for what concerns the broker, the routing engine and the internal components. We also provide some screenshots from the new administrative web console implemented for the second release.

Scheduler Message Routing

An example taken from the middleware source code is shown in Listing 1, where the Scheduler message route is defined using Spring XML and Apache Camel. In the next paragraph we show other configured routes and the full Camel configuration file.

Based on the value of different headers for the incoming message, a specific logic is implemented: for example based on the request (Task type), the two main workflows are executed. The `from` element defines a message endpoint to consume messages from, while the `Message Router` pattern is implemented using the `choice` and `when` elements. The `bean` tag is used to invoke operations on specific Spring beans, which are Java classes instantiated at boot time. Finally, the `to` element defines a message destination. For the Scheduler route, these destinations are associated to other routes and can trigger other processes.

Listing 1: Scheduler route definition

```
<route id="schedulerRoute">

  <from uri="activemq:queue:SCHEDULER.QUEUE"/>

  <choice>

    <when>
      <simple>${in.header.taskStatus} == 'COMPLETED'</simple>
      <bean ref="scheduler" method="closeTask"/>
      <to uri="activemq:queue:LOG.QUEUE"/>
    </when>

    <when>
      <simple>${in.header.taskStatus} == 'FAILED'</simple>
      <bean ref="scheduler" method="closeTask(${in.header.taskId})"/>
      <to uri="activemq:queue:ERROR.QUEUE"/>
    </when>

    <otherwise>
      <when>
        <simple>${in.header.taskType} == 'PRESERVATION'</simple>
        <bean ref="scheduler" method="parseResources"/>
        <to uri="activemq:queue:LOG.QUEUE"/>
      </when>
      <when>
        <simple>${in.header.taskType} == 'REACTIVATION'</simple>
```

```
            <to uri="activemq:queue:REACTIVATION.QUEUE"/>
            <to uri="activemq:queue:LOG.QUEUE"/>
        </when>
    </otherwise>

  </choice>

</route>
```

Middleware Configuration

In the following we provide two sample configuration files for the messaging system and the routing engine in the PoF Middleware. Both examples make use of Spring XML framework.

A sample ActiveMQ configuration is shown in Listing 2. The broker configuration (name, ports, protocols) and the connection factory are provided, they are both instantiated at start time when the PoF Middleware server running in Apache Tomcat is started. The queues and the topics are defined providing just the name (with topics each message is sent to all subscribers, with queues each message is sent to a single consumer). Finally, all middleware components are defined as Spring beans, therefore their instances are created and maintained over time by the Spring framework.

Listing 2: ActiveMQ configuration with Spring XML

```
<broker id="broker" brokerName="pofBroker" useShutdownHook="false"
        useJmx="true" persistent="true" dataDirectory="activemq-data"
        xmlns="http://activemq.apache.org/schema/core">
  <transportConnectors>
    <transportConnector name="vm" uri="vm://pofBroker"/>
    <transportConnector name="tcp" uri="tcp://0.0.0.0:61616"/>
  </transportConnectors>
</broker>

<bean id="pooledConnectionFactory"
      class="org.apache.activemq.pool.PooledConnectionFactory"
      destroy-method="stop">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://pofBroker"/>
    </bean>
  </property>
</bean>

<bean id="scheduler.queue"
      class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg value="SCHEDULER.QUEUE"/>
</bean>

<bean id="preservation.queue"
      class="org.apache.activemq.command.ActiveMQQueue">
```

```
    <constructor-arg value="PRESERVATION.QUEUE"/>
</bean>
<bean id="create.collection.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="CREATE.COLLECTION.QUEUE"/>
</bean>
<bean id="image.analysis.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="IMAGE.ANALYSIS.QUEUE"/>
</bean>
<bean id="log.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="LOG.QUEUE"/>
</bean>
<bean id="test.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="TEST.QUEUE"/>
</bean>
<bean id="error.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="ERROR.QUEUE"/>
</bean>
<bean id="dead.end.queue"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="DEAD.END.QUEUE"/>
</bean>

<bean id="reactivation.notification.topic"
        class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="REACTIVATION.NOTIFICATION.TOPIC" />
</bean>
<bean id="preservation.notification.topic"
        class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="PRESERVATION.NOTIFICATION.TOPIC" />
</bean>

<bean id="scheduler" class="eu.forgetit.middleware.component.Scheduler" />
<bean id="idManager" class="eu.forgetit.middleware.component.IDManager" />
<bean id="collector" class="eu.forgetit.middleware.component.Collector" />
<bean id="extractor" class="eu.forgetit.middleware.component.Extractor" />
<bean id="contextualizer"
        class="eu.forgetit.middleware.component.Contextualizer" />
<bean id="archiver" class="eu.forgetit.middleware.component.Archiver" />
<bean id="condensator" class="eu.forgetit.middleware.component.Condensator" />
<bean id="forgettor" class="eu.forgetit.middleware.component.Forgettor" />
<bean id="logger" class="eu.forgetit.middleware.broker.MessageLogging" />
```

The configuration of Apache Camel using Spring XML is straightforward. An example of message route for the Scheduler is shown above. In Listing 3 we provide an excerpt of a sample configuration which defines the messaging broker and the route for two workflows: preservation preparation and re-activation. Each workflow is represented as a sequence of steps associated to specific Spring beans corresponding to the middleware components. During a given step, the method of the Java class defined in the configuration is

invoked. The Spring XML representation is associated to different patterns and defines a language for implementing specific rules associated to the messages.

Listing 3: Apache Camel configuration

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
   <property name="brokerURL" value="vm://pofBroker"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">

   <onException>
      <exception>eu.forgetit.middleware.WorkflowException</exception>
      <redeliveryPolicy maximumRedeliveries="2"/>
      <to uri="activemq:queue:ERROR.QUEUE"/>
   </onException>

   <route id="schedulerRoute">
     <!-- OMITTED, SEE ABOVE -->
   </route>

   <route id="preservationRoute">
      <from uri="activemq:queue:PRESERVATION.QUEUE"/>
      <setHeader headerName="taskStatus">
            <constant>RUNNING</constant>
      </setHeader>
         <bean ref="idManager" method="generateID"/>
         <bean ref="collector" method="getResources"/>
         <removeHeaders pattern="iamUserID"/>
         <setHeader headerName="iamType">
            <constant>ALL</constant>
         </setHeader>
         <bean ref="extractor" method="imageAnalysis"/>
         <bean ref="contextualizer" method="contextualize"/>
         <setHeader headerName="minClusteringImages">
            <constant>10</constant>
         </setHeader>
         <bean ref="condensator" method="imageClustering"/>
         <bean ref="archiver" method="createPackage"/>
         <bean ref="archiver" method="ingestSIP"/>
         <bean ref="archiver" method="exportAIP"/>
         <bean ref="archiver" method="storeAIP"/>
         <setHeader headerName="taskStatus">
            <constant>COMPLETED</constant>
         </setHeader>
         <multicast>
            <to uri="activemq:topic:PRESERVATION.NOTIFICATION.TOPIC"/>
            <to uri="activemq:queue:SCHEDULER.QUEUE"/>
         </multicast>

   </route>

   <route id="reActivationRoute">
      <from uri="activemq:queue:REACTIVATION.QUEUE"/>
```

```
      <setHeader headerName="taskStatus">
         <constant>RUNNING</constant>
      </setHeader>
      <bean ref="archiver" method="reactivateAIP"/>
      <bean ref="collector" method="restore"/>
      <setHeader headerName="taskStatus">
         <constant>COMPLETED</constant>
      </setHeader>
      <multicast>
         <to uri="activemq:topic:REACTIVATION.NOTIFICATION.TOPIC"/>
         <to uri="activemq:queue:SCHEDULER.QUEUE"/>
      </multicast>
   </route>

   <route id="periodicSchedulerRoute">
      <from uri="timer:pof?period=600s&amp;delay=180s"/>
      <transform>
         <simple>
            Scheduler Test Routing Message - ${date:now:yyyy-MM-dd HH:mm:ss}
         </simple>
      </transform>
      <setHeader headerName="taskStatus">
         <constant>COMPLETED</constant>
      </setHeader>
      <to uri="activemq:queue:SCHEDULER.QUEUE"/>
    </route>

    <route id="errorRoute">
       <from uri="activemq:queue:ERROR.QUEUE"/>
       <to uri="activemq:queue:SCHEDULER.QUEUE"/>
    </route>

 </camelContext>
```

The flow control makes use of message headers: setting the header of an incoming message to a given value, can influence the way the message is processed by the other components. The `multicast` element (in opposition to the `splitter`) and the `transform` element are used to implement other patterns (see [Hohpe and Woolf, 2003]). It is worth noting that the code exceptions and any error during the workflow execution are properly handled: the error messages are sent to the Scheduler to be processed and to a dedicated error queue used for monitoring.

Finally, a route executing periodic tasks is also shown: currently this is just used to send *heartbeat* messages, scheduled every 10 minutes, but for the future this mechanism could be used to implement periodic tasks associated to preservation or to monitor specific information associated to the content and trigger some pre-defined processes.

PoF Middleware Web Console

The monitoring interface for the messaging system and the routing engine is based on hawtio[20], a web monitoring console based on HTML5 that integrates seamlessly with ActiveMQ and Camel: this graphical console replaces the old ActiveMQ GUI and is multipurpose.

The flow of messages in the different queues, updated in real time during workflow execution, is shown in Figure 15 in Section 4.

Additional screenshots of the hawtio console for the middleware instance running in the testbed are shown in the following Figures: the status of queues and messages in the broker (Figure 29); the processes and threads running in the broker ((Figure 30); the routes defined in Camel using Spring XML, described before (Figure 31); .

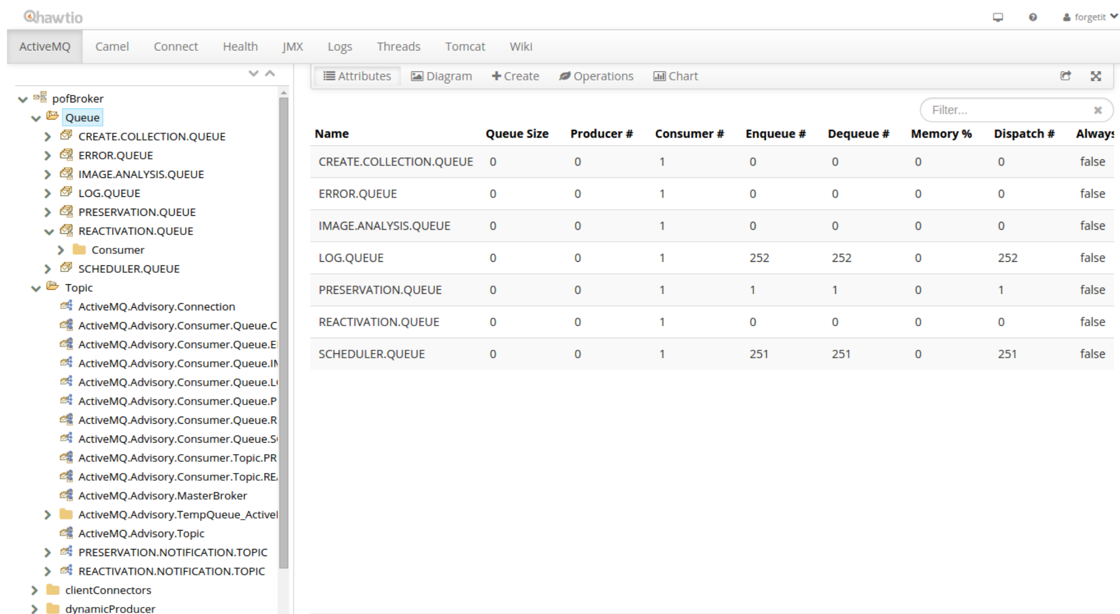

Figure 29: Message queues monitoring.

[20] hawtio - http://hawt.io

Total: 65 Runnable: 19 Timed waiting: 30 Waiting: 16

ID	State	Name	Waited Time	Blocked Time	Native	Suspended
85		http-nio-8080-Acceptor-0			(in native)	
5966		ActiveMQ BrokerService[pofBroker] Task-3083	166 ms			
5964		ActiveMQ InactivityMonitor Worker	30 seconds			
5962		ActiveMQ BrokerService[pofBroker] Task-3080	1 minute			
5961		ActiveMQ InactivityMonitor Worker	1 minute			
5904		ActiveMQ InactivityMonitor Worker	1 minute			
5815		ActiveMQ InactivityMonitor Worker	1 minute			
5726		ActiveMQ InactivityMonitor Worker	1 minute			
5719		ActiveMQ InactivityMonitor Worker	1 minute			
5586		ActiveMQ InactivityMonitor Worker	1 minute			
120		ActiveMQ Transport: tcp:///127.0.0.1:49990@61616			(in native)	
119		ActiveMQ Transport: tcp://localhost/127.0.0.1:61616@49990			(in native)	
113		ActiveMQ Transport: tcp:///127.0.0.1:49989@61616			(in native)	
112		ActiveMQ Transport: tcp://localhost/127.0.0.1:61616@49989			(in native)	
107		ConcurrentQueueStoreAndDispatch				

Figure 30: Process monitoring.

```
<route xmlns="http://camel.apache.org/schema/spring" id="schedulerRoute">
    <from uri="activemq:queue:SCHEDULER.QUEUE"/>
    <onException>
      <exception>eu.forgetit.middleware.WorkflowException</exception>
      <redeliveryPolicy maximumRedeliveries="2"/>
      <to uri="activemq:queue:ERROR.QUEUE"/>
    </onException>
    <choice>
      <when>
        <simple>${in.header.taskStatus} == 'COMPLETED'</simple>
        <bean ref="scheduler" method="closeTask"/>
        <to uri="activemq:queue:LOG.QUEUE"/>
      </when>
      <when>
        <simple>${in.header.taskStatus} == 'FAILED'</simple>
        <bean ref="scheduler" method="closeTask(${in.header.taskId})"/>
        <to uri="activemq:queue:ERROR.QUEUE"/>
      </when>
      <otherwise>
        <when>
          <simple>${in.header.taskType} == 'PRESERVATION'</simple>
          <bean ref="scheduler" method="parseResources"/>
          <to uri="activemq:queue:LOG.QUEUE"/>
        </when>
        <when>
          <simple>${in.header.taskType} == 'REACTIVATION'</simple>
          <to uri="activemq:queue:REACTIVATION.QUEUE"/>
          <to uri="activemq:queue:LOG.QUEUE"/>
        </when>
        <when>
          <simple>${in.header.taskType} == 'CREATE_COLLECTION'</simple>
          <to uri="activemq:queue:CREATE.COLLECTION.QUEUE"/>
          <to uri="activemq:queue:LOG.QUEUE"/>
        </when>
        <when>
          <simple>${in.header.taskType} == 'IMAGE_ANALYSIS'</simple>
          <to uri="activemq:queue:IMAGE.ANALYSIS.QUEUE"/>
          <to uri="activemq:queue:LOG.QUEUE"/>
        </when>
      </otherwise>
    </choice>
  </route>
```

Figure 31: Routes monitoring.

Extractor

In the following, an excerpt of Java code take from the Extractor component is shown: the method for image analysis used in the Apache Camel route defined above makes use of `Exchange` class, which is part of the Camel APIs and contains the message information (header and body). The message header is typically used to share high-level information required for flow control, while the message body contains the data. In the current implementation, we use JSON format to represent message content. After processing the message, extracting information and obtaining some results, the message body and header can be updated and then passed to the flow control wrapped in the `Exchange` object. Following the asynchronous message approach, the next destination of the message is unknown to the Extractor class, the new message is sent to one of the instances of the next component in the flow using the route definition (in the example above, it is the Contextualizer component).

Listing 4: Component methods for messages

```
package eu.forgetit.middleware.component;

...
import org.apache.camel.Exchange;
...
import eu.forgetit.middleware.component.Scheduler.TaskStatus;

public class Extractor {

...

  @BeanInject
  private Scheduler scheduler;

...

public void imageAnalysis(Exchange exchange){

...

  Map<String,Object> headers = MessageTools.getHeaders(exchange);

  String taskId = (String)headers.get("taskId");
  scheduler.setTaskStatus(taskId, TaskStatus.RUNNING);
  scheduler.setTaskLastStep(taskId,"IMAGE ANALYSIS");
  LocalDateTime lastDateTime = LocalDateTime.now();
  scheduler.setTaskLastDateTime(lastDateTime);

  exchange.getIn().setHeaders(headers);

  String iamType = (String)headers.get("iamType");

  ...

  JsonObject jsonBody = MessageTools.getBodyAsJSON(exchange);
```

```
  if(jsonBody!=null){

    // processing message body (JSON format)
    // new results are appended to the body

    exchange.getIn().setBody(jsonBody.toString());

  } else {

     headers.put("taskStatus", TaskStatus.FAILED.toString());
     exchange.getIn().setHeaders(headers);

  }

}
```

B Preserve-or-Forget RESTful Service

The PoF REST APIs are published using Jersey[21], the reference implementation of JAX-RS specification for RESTful web services. In the following we list some APIs with the expected parameters and the output format. The full list of REST APIs is available in the code documentation.

Server path	**/rest-api**
Supported response types	JSON and XML

Table 4: Server information

GET	**/rest-api/rest-api/application.wadl?detail=true**	Returns the list of REST APIs in WADL format, it is automatically updated by Jersey when starting up the server.

Table 5: Server APIs List

Other access APIs used for indexing and searching Situations have been omitted here. They are described in the code documentation.

The list of APIs exposed by the PoF Middleware RESTful web server is available as W3C WADL format.

[21] Java Jersey - `https://jersey.java.net`

POST	**/resource**	Triggers Preservation Preparation Workflow of single items or collections. Requires PV, CMIS Repository ID and CMIS Object ID.
POST	**/resources//{cmisServerId}**	Triggers Preservation Preparation Workflow for many resources (bulk request). Requires CMIS Repository ID and a list CMIS Object IDs with additional information in a JSON object.
POST	**/user-logs**	Triggers Fetching of User Logs for computing PV in the Middleware during Automatic Preservation. Requires CMIS Repository ID and CMIS Object ID.
POST	**/cmis-repository**	Register a new Active System with its associated CMIS repository. Requires CMIS Repository ID and other configuration parameters (e.g. repository URL).
GET	**/cmis-repository/{cmisServerId}**	Returns information about the registered CMIS repository. Requires CMIS Repository ID.
GET	**/resources/{cmisServerId}/last-update**	Used by the Active System to get information about the preserved resources and trigger automatic preservation. Requires CMIS Repository ID.

Table 6: Preservation APIs

GET	**/restore/{cmisServerId}/{cmisId}**	Triggers Re-activation Workflow for specified resource. Requires CMIS Repository ID and CMIS Object ID.
GET	**/restore?cmisServerId=...&cmisId=...**	Same as above but supporting Query Params.

Table 7: Re-activation APIs

GET	**/resource/cmisServerId/cmisId**	Returns information about preserved resource (different IDs,preservation status, metadata). Requires CMIS Repository ID and CMIS Object ID.
GET	**/resource?cmisServerId=...&cmisId=...**	Same as above but supporting Query Params.
GET	**/resources/cmisServerId**	Returns information about preserved resources for the specified CMIS Repository. Requires CMIS Repository ID.

Table 8: Access APIs

GET	**/tasks/taskId/status**	Returns information for the specified Task. Task ID is returned when submitting requests.
GET	**/tasks/taskId/result**	Returns results for the specified Task. Alternative method to message notifications. Task ID is returned when submitting requests.
GET	**/tasks**	Returns information for all Tasks. Only for administrative purposes.

Table 9: Task Monitoring APIs

C DSpace Installation and Configuration

C.1 Introduction

Information about DSpace can be found on the project web site[22]. The role of DSpace in the PoF Framework and the integration with the other components in the overall architecture is described in deliverable D8.1 [Gallo et al., 2013]. Additional information can be found in D7.2 [Rabinovici-Cohen et al., 2014] (integration with cloud storage) and in D5.2 [Nilsson et al., 2014] (synergetic preservation workflows). This guide is based on official DSpace documentation[23], tailored to Ubuntu Server 12.04 LTS, with additional configuration information for the PoF Framework. Other applications and libraries required to install and run DSpace are Apache Maven[24] and Apache Ant[25] for compiling and building DSpace sources, PostgreSQL[26] as internal DB used by DSpace and Apache Tomcat[27] for the runtime.

C.2 Installation Procedure

The following instructions have been tested with DSpace 4.1. If you are using a different version of DSpace, you should check the documentation available on DSpace web site. In order to install DSpace using the following instructions, you need a basic installation of Ubuntu Server 12.04 LTS. Please refer to Ubuntu documentation for the installation of the operating system. You can use a physical or virtual machine for installing Ubuntu, as done for example in the ForgetIT testbed. In the following we assume that you have installed Ubuntu and that you have access to the machine using either the `root` user or any user belonging to the `sudo` group. During the Ubuntu installation, it is advisable to include an OpenSSH server as additional software, mainly if you are installing DSpace in a virtual machine hosted by a remote server. Please note that in the instructions below, after the creation of the DSpace user, you need to start DSpace and apply any modifications to the DSpace configuration using this user only, who must also have writing permissions for all the directories used by DSpace.

Configuration of Ubuntu

From within a terminal, update the Ubuntu installation and reboot the machine:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo reboot now
```

[22] DSpace - `http://www.dspace.org`
[23] DSpace Guide - `https://wiki.duraspace.org/display/DSDOC/All+Documentation`
[24] Apache Maven - `http://maven.apache.org`
[25] Apache Ant - - `http://ant.apache.org`
[26] PostgreSQL - `http://www.postgresql.org`
[27] Apache Tomcat - `http://tomcat.apache.org`

Install the Java JDK 7[28] and Apache Maven with the following command:

```
$ sudo apt-get install openjdk-7-jdk maven
```

Check the Maven installation running:

```
$ mvn -version
```

You should get an output similar to the following:

```
Apache Maven 3.0.4
Maven home: /usr/share/maven
Java version: 1.7.0_51, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-7-openjdk-amd64
Default locale: it_IT, platform encoding: UTF-8
OS name: "linux", version: "3.8.0-35-generic", arch: "amd64",
family: "unix"
```

Since Maven on Ubuntu server will add version 6 of Java Runtime Environment as additional dependency, configure JRE in order to use version 7 (for Java compiler and other Java related utilities the used version should already be 7):

```
$ sudo update-alternatives --config java
```

and select version 7 when prompted (option 2 in the example below):

```
There are 2 choices for the alternative java (providing /usr/bin/java).

Selection    Path                                          Priority   Status
------------------------------------------------------------------------------
* 0          /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java   1061     auto mode
  1          /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java   1061     manual mode
  2          /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java   1051     manual mode

Press enter to keep the current choice[*], or type selection number: 2
```

Install Apache Web Server, including the proxy module (Apache WS will act as a proxy to forward the requests to the DSpace web applications running on Tomcat, while static content, such URLs for published AIP files, will be managed by Apache WS itself):

```
$ sudo apt-get install apache2 libapache2-mod-proxy-html
libxml2-dev
```

Install libraries for file compression (zip, p7zip-full, p7zip-rar): some of the previous libraries are not mandatory, but can be useful for testing DSpace locally, e.g. creating zip files for the SIP or to open exported AIPs.

Install tools for CIFS, to mount storage folder on external devices (e.g. NAS). NFS or iSCSI could be used, too.

```
$ sudo apt-get install cifs-utils
```

The automatic creation of new mount points must be added to `/etc/fstab` file, depending on your configuration.

[28]Open JDK - `http://openjdk.java.net`

Create “dspace” user

Create a new Linux user named `dspace` (with password `dspace`) and add it to the sudoers group with the following commands:

```
$ sudo adduser dspace
$ sudo adduser dspace sudo
```

From now on, switch to DSpace user `dspace`: you can either logout and then login again with user `dspace` or simply use the following command as `root`:

```
$ su dspace
```

Install and configure PostgreSQL 9.1

DSpace currently supports PostgreSQL and Oracle DB. PostgreSQL is the default choice and no additional drivers or adapters have to be added. Instructions below refer to the default installation with PostgreSQL. If you want to use Oracle DB please refer to the documentation available in the official DSpace documentation.

Install PostgreSQL and check the installation with the following commands:

```
$ sudo apt-get install postgresql-9.1
$ psql - version
```

Edit PostgreSQL configuration files as described below:

- ***postgresql.conf*** in */etc/prostgresql/9.1/main/*: uncomment the line containing: *listen_addresses = 'localhost'* and edit the line to listen on all addresses, *listen_addresses = '*'*
- ***pg_hba.conf*** in */etc/prostgresql/9.1/main/*: add the lines *host dspace dspace 127.0.0.1 255.255.255.255 md5* and *host all all 127.0.0.1/24 trust*.

Restart PostgreSQL:

```
$ sudo service postgresql restart
```

Switch to user `root` and then switch to user `postgres`:

```
$ sudo su -
$ su postgres
```

Create a new user for PostgreSQL, with username `dspace` and password `dspace` (this user is different from the one created on Ubuntu and can be changed in the DSpace configuration):

```
$ createuser -U postgres -d -A -P dspace
```

Create a new PostgreSQL DB schema, named `dspace`, owned by the `dspace` PostgreSQL user created above:

```
$ createdb --owner=dspace --encoding=UNICODE dspace
```

Switch back to Ubuntu user `dspace` issuing twice the shell command `exit`.

Create a folder for copying source files of the required applications and for setup. The suggested configuration is to create a folder under `/opt` and to assign ownership to `dspace` user, as shown below:

```
$ sudo mkdir /opt/forgetit
$ sudo chown -R dspace:dspace /opt/forgetit
$ mkdir /opt/forgetit/applications
$ mkdir /opt/forgetit/setup
```

Install Apache Tomcat 7

Install Apache Tomcat 7 as user `dspace` created in the previous section. Download Tomcat 7, e.g. using the following command (check the link for the current version):

```
$ cd /opt/forgetit/applications
$ wget -O apache-tomcat-7.0.52.tar.gz
  http://apache.panu.it/tomcat/tomcat-7/v7.0.52/bin/apache-tomcat
  -7.0.52.tar.gz
```

or copy the .tar.gz file downloaded with another machine using `scp` command (you need an OpenSSH server running, see instructions above).

Uncompress the Tomcat tar.gz file into directory `/opt/forgetit/tomcat7` (in the following this directory will be referred to as `[Tomcat_Install_Dir]`) :

```
$ tar -xzvf apache-tomcat-7.0.52.tar.gz -C /opt/forgetit
$ mv apache-tomcat-7.0.52 tomcat7
```

Instructions to configure a service for Tomcat to start at boot are provided in the last section. Edit the Tomcat 7 configuration as described below:

- create file ***setenv.sh*** in *[Tomcat_Install_Dir]/bin* and set *JAVA_OPTS="-Xmx512M -Xms64M -XX:MaxPermSize=256M -Dfile.encoding=UTF-8"*
- Edit file ***server.xml*** in *[Tomcat_Install_Dir]/conf* adding a configuration option to the *Connector* element: *URIEncoding="UTF-8"*

Install DSpace 5.2

Download DSpace 5.2 from DSpace web site, e.g. using the following command (check the link for the current version):

```
$ cd /opt/forgetit/applications
$ wget -O dspace-5.2-src-release.zip http://sourceforge.net/projects/dspace/files/DSpace%20Stable/5.2/dspace-5.
```

As `dspace` user unpack the DSpace tar.gz file. The extracted folder (e.g. `dspace-4.1-src-release`) will be referenced to as `[dspace-source]` in the following. Create a directory to install DSpace, e.g. `/opt/forgetit/dspace-4.1`, referred to as (`[dspace-install]`):

```
$ cd /opt/forgetit/applications
$ tar -xzvf dspace-5.2-src-release.zip
$ mkdir /opt/forgetit/\emph{[dspace-install]}
$ cd /opt/forgetit/applications/\emph{[dspace-source]
```

Configure file ***build.properties***, editing the following properties:

- `dspace.install.dir` pointing to `[dspace-install]`
- `dspace.hostname = [HOSTNAME]` (set it to the hostname chosen during the installation, e.g. archive. Compare with hostname in file `/etc/hosts` or with environment variable `HOSTNAME`)
- `dspace.baseUrl` = http://`[HOSTNAME]`:8080
- `dspace.name =` DSpace for Preserve-or-Forget Framework (or any other name which is meaningful for you)
- `mail.server` = `YOUR_MAIL_SERVER`
- `mail.from.address` = `dspace-admin@forgetit-project.eu` (or change according to your configuration)
- `mail.feedback.recipient` = `CONTACT_USER_EMAIL`
- `mail.admin` = `DSPACE_ADMIN_EMAIL`
- uncomment the line `handle.canonical.prefix` = `$dspace.url/handle/` and comment the line `handle.canonical.prefix = http://hdl.handle.net/`, unless you want to subscribe to the handle service by CNRI
- `handle.prefix` = `ANY_VALUE` (use official prefix from handle service if available)

All properties above but the installation directory can be modified later, editing file `dspace.cfg`

Compile using Maven and install using Ant, with the following commands:

```
$ cd [dspace-source]
$ mvn package
$ cd [dspace-source]/dspace/target/dspace-[version]-build
$ ant fresh_install
```

DSpace is installed in the specified directory `[dspace-install]`.

Create an admin account for DSpace:

```
$ cd [dspace-install]
$ ./bin/dspace create-administrator
```

Deploy the created web applications (copy `[dspace-install]/webapps` content or create symlinks for all DSpace web applications in Tomcat webapps folder).

Start Tomcat 7 as `dspace` user (see Tomcat documentation for starting and stopping the server) and check the DSpace installation. Verify that DSpace is up and running at the following URLs (change `archive.forgetit-project.eu` with correct host):

- check DB connection with command: `$ ./bin/dspace test-database`
- check email settings: `$ ./bin/dspace test-email`
- `http://archive.forgetit-project.eu:8080/jspui` (JSP-based interface)
- `http://archive.forgetit-project.eu:8080/xmlui` (XML-based interface)
- sign in with administrator account created above on DSpace web interface
- try to create collections, new items, etc. (see instructions for Getting Started on DSpace web site)

To customize the home page of the XMLUI interface, edit file `/opt/forgetit/dspace-4.1/config/news-xmlui.xml`.

Optional configuration for Apache Web Server and Apache Tomcat

Apache WS can be configured to act as a reverse proxy for Tomcat (requests to DSpace are proxied by Apache WS and viceversa). Static content (e.g. AIP files exported from DSpace) is served by Apache WS.

Check that the proxy module is installed (see section about Ubuntu environment configuration) and enabled. Use command `sudo a2enmod proxy_http` and restart with `sudo service apache2 restart`).

Add the following directives to file `/etc/apache2/sites-available/default`, for each one of the applications in webapps to be proxied:

```
        ProxyPass /xmlui http://archive:8080/xmlui
        ProxyPassReverse /xmlui http://archive:8080/xmlui
```

According to the example above, the new URL of DSpace XML interface will be `http://archive/xmlui`.

Tomcat 7 can be configured to start at boot. Paste the example script below to a text file `tomcat7` and copy it to `/etc/init.d`, then execute command:

```
  $ update-rc.d <nomescript> defaults
```

The script must be executable (use `chmod` command: `$ sudo chmod ugo+rx tomcat7`). Note that in the provided example Tomcat is run as user `dspace`.

```
#! /bin/sh

### BEGIN INIT INFO
# Provides:              Tomcat7
# Required-Start:        $remote_fs $syslog
# Required-Stop:         $remote_fs $syslog
# Default-Start:         2 3 4 5
# Default-Stop:
# Short-Description:     Tomcat7
### END INIT INFO
```

```
set -e
. /lib/lsb/init-functions
TOMCAT_HOME=/opt/tomcat7
TOMCAT_USER=dspace

case "$1" in
  start)
        log_daemon_msg "Starting Tomcat7"
        su - $TOMCAT_USER -c "$TOMCAT_HOME/bin/startup.sh > /dev/null"
        log_end_msg 0
        ;;
  stop)
        log_daemon_msg "Stopping Tomcat7"
        su - $TOMCAT_USER -c "$TOMCAT_HOME/bin/shutdown.sh > /dev/null"
        log_end_msg 0
        ;;
  reload|force-reload)
        ;;
  restart)
        ;;
  *)
        log_action_msg "Usage: /etc/init.d/tomcat7 {start|stop|reload|force-
        reload|restart|try-restart|status}" || true
        exit 1
esac

exit 0
```

C.3 DSpace REST API

The REST API is an interface intended for administration of DSpace environment. From the DSpace 5.x version, it is possible to perform all CRUD actions; in the previous release only read features were allowed. The REST DSpace frees the administration from what GUI is used by the users: in this way it is possible to change the aspect of DSpace without modifying the system, so that the administrators don't perceive any changes in the environment management. All the actions that will be explained in this section are based on cURL syntax [29]. The main cURL requests used for DSpace are:

- GET for obtaining all the information about DSpace objects
- POST for modifying or adding Communities, Collections, Items, Metadata, Bitstreams and Access Policies
- OPTIONS for displaying all the actions allowed at a specified level
- DELETE for deleting the desired elements

As previously told, the GET request is used for obtaining the information about a DSpace object. The administrator can choose how to display these contents: they could be organized in a json script or in a xml one.

If the admin choose json:

[29] cURL - http://curl.haxx.se/docs/manpage.html

```
$ curl -s -H "Accept: application/json"
http://preservation-system:8080/rest/communities
```

If XML is chosen:

```
$ curl -s -H "Accept: application/xml"
http://preservation-system:8080/rest/communities
```

The GET word can be omitted: when a curl request is done, if there are no requests specified, it means that a GET is required.
From this point, it is assumed that the base URL to the ”REST” webapp will be $http : //preservation - system : 8080/rest$. The following parts of /rest/... are specific to what kind of objects the administrator wants to obtain or modify.

The first thing to do for accessing to reserved material is to login to the system: the request used in this case is POST.

The correct syntax is the following:

```
$ curl -X POST -H "Content-Type: application/json"
--data '{"email":"user@email.com","password": "userpassword"}'
http://preservation-system:8080/rest/login
```

A message is displayed, containing a code like $d9e103cl-8248-4b52-av5b-436c9edce4b2$. It contains the created username and password values. This will be used in the next steps for performing the administration actions.

The logout is an example of the use of this code:

```
$ curl -X POST -H "Content-Type: application/json" -H
"rest-dspace-token:d9e103cl-8248-4b52-av5b-436c9edce4b2"
http://preservation-system:8080/rest/logout
```

The previously mentioned code is used in this way: $"rest - dspace - token : d9e103cl - 8248 - 4b52 - av5b - 436c9edce4b2"$; The token will be fundamental for accessing or modifying reserved objects.

Now it is possible to start creating a first Community with the POST request:

```
$ curl -X POST -H "Content-Type: application/json" --data
'{"name":"TEST COMMUNITY","copyrightText":"NEW COMMUNITY (JSON)",
"introductoryText":NEW COMMUNITY (JSON)","shortDescription":
"NEW COMMUNITY (JSON)","shortDescription":"NEW COMMUNITY (JSON)",
```

```
"sidebarText":"NEW COMMUNITY (JSON)"}' -H "rest-dspace
token:d9e103cl-8248-4b52-av5b-436c9edce4b2" http://preservation-
system:8080/rest/communities
```

The $Content - Type$ field is used for specifying what kind of data will be used for defining the Community, so that the $--data$ parameter expects a json.

For creating a Collection:

```
$ curl -X POST -H "Content-Type: application/json"
--data '{"name":"TEST COLLECTION","copyrightText":
"NEW COLLECTION (JSON)","introductoryText":"NEW COLLECTION (JSON)",
"shortDescription":"NEW COLLECTION (JSON)","sidebarText":
"NEW COLLECTION (JSON)"}' -H "rest-dspace-token:
d9e103cl-8248-4b52-av5b-436c9edce4b2"
http://preservation-system/rest/communities/COMMUNITY_ID/collections
```

The administrator can create Items now. Differently from the GUI Items, in the REST API the administrator has to create the Item like a sort of container where, in a second step, he could put the Bitstreams.

```
$ curl -X POST -H "Content-Type:application/json"
--data '{"metadata":[{"key": "dc.contributor.author",
"value": "SURNAME, NAME"},{"key": "dc.description",
"language": "en_US","value": "DESCRIPTION"},
{"key": "dc.description.abstract","language": "en_US",
"value": "ABSTRACT"},{"key": "dc.title","language": "en_US",
"value": "TEST ITEM"}]}'
-H "rest-dspace-token:d9e103cl-8248-4b52-av5b-436c9edce4b2"
http://192.168.253.13/rest/collections/COLLECTION_ID/items
```

The Item is ready to be filled with documents (Bitstreams). For uploading documents the following requests has been used:

The first method works for plain text or .pdf documents:

```
$ curl -k -H "rest-dspace-token: d9e103cl-8248-4b52-
av5b-436c9edce4b2" -F upload=@"/path/to/document/
documentname.[ext]" -X POST http://preservation-
system:8080/rest/items/ITEM_ID/bitstreams?
name=TEST.[ext]&description=DESCRIPTION
```

The -k is used for maintaining active the connection even if it is insecure.

```
$ curl -k -H "rest-dspace-token: d9e103cl-8248-4b52-
av5b-436c9edce4b2" --data-binary @"/path/to/document
/documentname.jpg" -X POST "http://preservation-system:
8080/rest/items/65/bitstreams?name=TEST.jpg"
```

I used this last method especially for images, but it works with other kind of documents, too. The advantage of this second kind of request is the ease of use because of its brevity and the compatibility with more file extensions. It is possible to store more than one Bitstream in an Item and everyone can be of a different extension.

Once that objects are created, they can be modified and the admin can add some features to existing elements.

The POST request allows to add and modify elements, too. For example, if the administrator wants to modify the Metadata of an Item, the request is the following:

```
$ curl -X POST -H "rest-dspace-token: d9e103cl-8248-4b52-
av5b-436c9edce4b2" -H "Content-Type: application/json"
--data '[{"key":"dc.title","value":"TEST - MODIFIED",
"language":"en_US"},{"key":"dc.description","value":
"DESCRIPTION","language": "en_US"}]'
http://preservation-system:8080/rest/items/ITEM_ID/metadata
```

The administrator can decide if a user will be able to read or modify an Item. The Permission Policies, responsible for Bitstreams accessibility, can be managed even from the REST API. The admin can delete or add policies to Bitstreams. For objects added from DSpace REST API the users can read everything by defaults. If the administrator wants to deny the access to some data, he has to DELETE the relative policy:

```
curl -X DELETE -H "rest-dspace-token: d9e103cl-8248-4b52-
av5b-436c9edce4b2" http://preservation-system:
8080/rest/bitstreams/BITSTREAM_ID/policy/POLICY_ID
```

If the admin wants to give the permission of modifying Items to an user, a POST request has to be done for adding to the interested Bitstream a WRITE policy.

```
$ curl -X POST -H "rest-dspace-token: d9e103cl-8248-4b52-
av5b-436c9edce4b2" --data '{"id": POLICY_ID,"action": "WRITE",
"epersonId": -1, "groupId": 0, "resourceId": BITSTREAM_ID,
"resourceType": "bitstream","rpDescription": null,
"rpName": null,"rpType": "TYPE_INHERITED","startDate": null,
"endDate": null}'
http://preservation-system:8080/rest/bitstreams/BITSTREAM_ID/policy
```

These are the main features that needed to be explained. For the complete list of possible actions, see the official DSpace Guide `https://wiki.duraspace.org/display/DSDOC5x/REST+API`.

C.4 Administration and Users Permissions

This section is intended to guide an administrator of a DSpace instance during the setting of its environment. As seen in Section C.2, the administrator account has to be registered during the installation steps.
The settings here illustrated explain how to allow some users to share documents, with the possibility of adding and deleting every file shared in the common environment. The elements and the images here shown for explaining the process refer to the xml interface.
The first step is the creation of at least a Community and a Collection, in order to have a place for storing the Items. This is fundamental for going on with saving Items. Another important step is the users creation. There are two different scenarios for user creation:

- The user register his account by himself
- The user account is created by the administrator

There are no differences between these two methods: the final effect is the same. A communication is sent to the user e-mail, containing a link that addresses to a DSpace session, in order to confirm the account registration.

When a new account has been created, the administrator adds the user to a group. The groups are created by the administrator and their function is to include all users supposed to have the same permissions pattern. The only default group that exists at the moment of the environment creation is the Anonymous one: the administrator has to delete all the permissions assigned to this group in order to hide private contents of the Collection to foreign users. In fact, when a user registers his account to DSpace, it belongs by default to Anonymous group: in this way, only when the administrator includes the new account in a group the user will be able to access to Collections' contents.
The permissions given to a group are fundamental to decide what can or cannot do the users belonging to it. In DSpace the permissions are organized at different levels:

- Permissions at Community level: the users are able to read and write in a Community
- Permissions at Collection level: the users' available permissions are read, write, add, delete. The best way to set an account for this kind of environment is to give all these permissions. A user can create a new Collection or modify an old one but he can't delete one. The only one able to delete a Collection (or a Community) is the administrator.
- Permissions at Item level: the possible and advisable permissions for Items are read, write, obsolete (delete), add, delete (for both item and bitstream). In this way,

the users can modify, delete, add items created by every member of the community. The users with this permission pattern can add other files to Item's Bitstream, even in a second time, after the creation.

Figure 32: Permission pattern at Collection level

Another available permission for the users is the admin one: it is a lot different from the administrator account permission option. The only things that this setting allows are relative to metadata export and to the possibility to move an Item from a Collection to another.

C.5 Import and Export

Another DSpace feature is the the preservation of the contents' structure once they are exported and imported. The possibility of exporting or importing entire Communities in an environment is the key factor to preserve the desired contents without changing the structure of the documents organization. The administrator can both export and import Communities; the users with administration privilege can only make the export, as guarantee of the preservation and the general environmental maintenance.

Export

The Community/Collection is downloaded in a .zip file, containing, in case of a Community, some folders everyone corresponding to a Collection and every Collection folder containing the Items' sub-folders. When the user choose to export only a Collection the

organization is the same but there is only the Collection folder containing its Items. The content of the exported Items' folders is a series of files, everyone involved in the preservation of the contents:

- contents: this file contains the name of the documents that belong of the Item and the name of the licence file
- dublin_core.xml: this is the metadata file, containing all the information that allow the preservation of the structure for data organization
- handle: in this file is specified the id previously used for the Collection
- licence.txt
- Filename.[ext]: the document contained in the Item

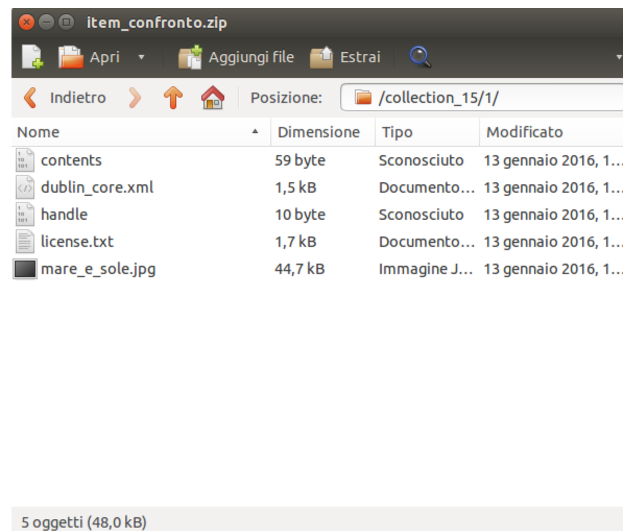

Figure 33: Export folder's contents

Import

The administrator can import a .zip file containing a Community or a Collection. First of all, if DSpace is completely empty, the admin has to create a Community and a Collection for giving an address for storing new Items. After that, it will be possible to proceed with the Import process. Every time an element is imported in a new DSpace environment, the dublin_core.xml file is automatically modified. Some information about the new destination are added and the old ones are anyway preserved.

Mostra i principali dati dell'item

dc.contributor.author	c, f	
dc.date.accessioned	2016-01-13T13:21:19Z	
dc.date.available	2016-01-13T13:21:19Z	
dc.date.issued	2016-01-20	
dc.identifier.uri	http://hdl.handle.net/600826/55	
dc.description.provenance	Submitted by martina fogliati (martyfog87@gmail.com) on 2016-01-13T13:21:19Z No. of bitstreams: 1 futuro.jpg: 59145 bytes, checksum: e05d091b701777a3203a02ef0b8da5bc (MD5)	en
dc.description.provenance	Made available in DSpace on 2016-01-13T13:21:19Z (GMT). No. of bitstreams: 1 futuro.jpg: 59145 bytes, checksum: e05d091b701777a3203a02ef0b8da5bc (MD5) Previous issue date: 2016-01-20	en
dc.title	casa	en_US
dc.type	Image	en_US

Figure 34: Metadata before the Import

Mostra i principali dati dell'item

dc.contributor.author	c, f	
dc.date.accessioned	2016-01-13T13:21:19Z	
dc.date.accessioned	2016-01-13T13:25:58Z	
dc.date.available	2016-01-13T13:21:19Z	
dc.date.available	2016-01-13T13:25:58Z	
dc.date.issued	2016-01-20	
dc.identifier.uri	http://hdl.handle.net/600826/55	
dc.description.provenance	Submitted by martina fogliati (martyfog87@gmail.com) on 2016-01-13T13:21:19Z No. of bitstreams: 1 futuro.jpg: 59145 bytes, checksum: e05d091b701777a3203a02ef0b8da5bc (MD5)	en
dc.description.provenance	Made available in DSpace on 2016-01-13T13:21:19Z (GMT). No. of bitstreams: 1 futuro.jpg: 59145 bytes, checksum: e05d091b701777a3203a02ef0b8da5bc (MD5) Previous issue date: 2016-01-20	en
dc.description.provenance	Made available in DSpace on 2016-01-13T13:25:58Z (GMT). No. of bitstreams: 2 futuro.jpg: 59145 bytes, checksum: e05d091b701777a3203a02ef0b8da5bc (MD5) license.txt: 1748 bytes, checksum: 8a4605be74aa9ea9d79846c1fba20a33 (MD5) Previous issue date: 2016-01-20	en
dc.title	casa	en_US
dc.type	Image	en_US

Figure 35: Metadata after the Import

C.6 Versioning and Other Features

The administrator can create a new version of an Item. After that, he has to give another time the permission to users for the Item because of the changes. The users will be able to choose what version of Item to use, selecting it from Show Version History. This is one of the most interesting DSpace feature because it allows to update the contents making possible to the users to see all the history of the Item contents without loosing the old information.

DSpace has a lot of functionalities that could be useful in case of a document that, for example, is not completely ready to be shown because of the need for some changes and updates. For this aim, a user or the administrator can choose to hide an Item with the Hide function: in this way, the element is not visible.

Questo item appare nelle seguenti collezioni

- collection two

Version History

Version	Item	Editor	Date	Summary
2	600826/55	martina fogliati	2016-01-13T14:52:00Z	1
1	600826/55.1*	martina fogliati	2016-01-13T13:21:19Z	

*Selected version

Figure 36: Available Versions of an Item's contents

C.7 AntiVirus in DSpace: ClamAV

DSpace is intended as an environment for sharing and preserving documents. For not losing the contents because of corrupted files or viruses, it could be fundamental to control for the presence of something wrong in items' metadata and bitstreams introduced by the users. For the security of the general environment, the presence of an antivirus is important. One of the most advisable for DSpace is ClamAV [30].
In this section, it is explained how to install and some way of use of ClamAV in Ubuntu. For details about other OS, see ClamAV.

ClamAV has to be installed in the machine where DSpace is running. DSpace 5.x contemplate the use of this antivirus by default.
The steps for installing ClamAV are the following:

```
$ sudo apt-get update
$ sudo apt-get install clamav clamav-daemon
```

After the installation step, it is better to upgrade the list of viruses that could affect metadata and bitstream items:

```
$ sudo freshclam
```

[30] ClamAV - http://www.clamav.net/

C.8 Curation Tasks

Now it is time to configure DSpace for curation tasks, the services provided by ClamAV for scanning the contents of the Items. This is the configuration for DSpace 5.x. [31]. For previous versions see [32]. It is from version 1.7 that DSpace supports curation tasks. The configuration property file for curation is $[dspace]/config/modules/curate.cfg$:

```
plugin.named.org.dspace.curate.CurationTask = \
org.dspace.ctask.general.NoOpCurationTask = noop, \
org.dspace.ctask.general.ProfileFormats = profileformats, \
org.dspace.ctask.general.RequiredMetadata = requiredmetadata, \
org.dspace.ctask.general.ClamScan = vscan, \
org.dspace.ctask.general.MicrosoftTranslator = translate, \
org.dspace.ctask.general.MetadataValueLinkChecker = checklinks
```

Curation Tasks from the Linux Shell

The curation tasks are made by default from the DSpace system. The control is made at the step 1 of the workflow: in this way, if a user tries to add a corrupted Item to a Collection, the dangerous content is blocked and a message is sent to the administrator, for awareness of the possible risk.
Anyway, it is possible to control the Item bitstream or metadata with some easy shell commands:

```
[dspace]/bin/dspace curate -t vscan -i 123456789/4 -v
```

The instruction $vscan$ is passed to -t for specifying the task that the administrator wants to perform; the id of the item that has to be scanned, $123456789/4$, is passed to the -i argument. The instruction -v is for a verbose response: in this way, a message is displayed after the curation task. The complete list of arguments is the following:

```
-t taskname: name of task to perform
-T filename: name of file containing list of tasknames
-e epersonID: (email address) will be superuser if unspecified
-i identifier: Id of object to curate. May be:
 (1) a handle (2) a workflow Id (3) 'all' to operate on the whole
 repository
-q queue: name of queue to process - -i and -q are mutually
   exclusive
-l limit: maximum number of objects in Context cache. If absent,
   unlimited objects may be added.
-s scope: declare a scope for database transactions. Scope must be:
 (1) 'open' (default value) (2) 'curation' (3) 'object'
-v emit verbose output
-r - emit reporting to standard out containing list of tasknames
```

[31] DSpace 5.x - https://wiki.duraspace.org/display/DSDOC5x/Curation+System
[32] DSpace - https://wiki.duraspace.org

Administrative User Interface

It is also possible to activate a button in the xml interface for an easier execution of the curation tasks. In DSpace 5.x is active by default. The script responsible for this feature is in $[dspace]/config/modules/curate.cfg$, the same file mentioned in C.7

```
ui.tasknames = \
     profileformats = Profile Bitstream Formats, \
     requiredmetadata = Check for Required Metadata
```

Figure 37: Curation Task button is now available in DSpace interface

A message is displayed after the curation task:

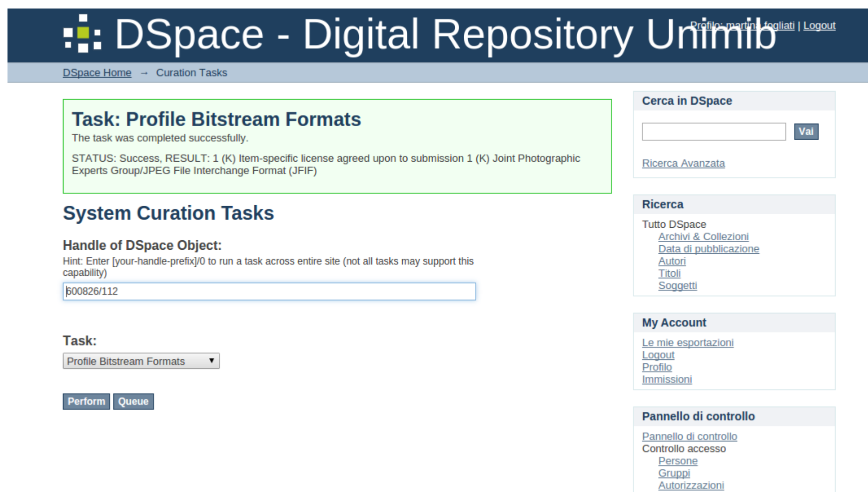

Figure 38: Message displayed after the curation task

C.9 Cloud Storage

In order to avoid unpleasant loss of DSpace contents, a backup of your environment could be a good practice. Traditionally DSpace supports a backup and restore method of database and bitstreams (the assetstores). It's from DSpace 1.7 release that a new an easier backup and restore method has been introduced. It consists of storing the objects in AIPs (Archival Information Packages) format, .zip files containing the bitstreams and a file named mets.xml containing all metadata information. This format maintains the environment hierarchy, respecting the dependencies between objects and allows at the same time to add a new object in a new DSpace instance, even without old dependencies maintenance, for example in case of insertion of a new Item in a preexisting Collection; the permissions and ePersons are maintained, too, in case of a complete site backup and restore.
Another advantage of this method is the ease of use. In case of traditional backup and restore mode, a deep knowledge of the DSpace's database and all its relations existing between the different objects is required. With AIPs, the only things to do are the download of the objects and restore or submit in a new DSpace instance. This is a very quick and easy to apply practice, so that for a good maintenance of DSpace, it allows a daily backup of new contents.

For making available these features, some configurations has to be done. For this guide, a DSpace 5.2 version has been used (for more details [33]). First of all, the pom.xml files above specified has to be modified in the DSpace source directory (for brevity, dspace-src):

dspace-src /dspace/modules/additions/pom.xml

dspace-src /dspace/pom.xml

dspace-src /dspace/modules/xmlui/pom.xml

The administrator has to add the following lines at the end of the ¡dependencies¿ sections, just before the closing tag ¡/dependencies¿

```
<dependencies>
    ...

    <!-- Adding this dependency will install the Replication Task Suite Addon -->
    <dependency>
        <groupId>org.dspace</groupId>
        <artifactId>dspace-replicate</artifactId>
        <version>1.3</version>
    </dependency>
</dependencies>
```

[33] https://wiki.duraspace.org/display/DSPACE/ReplicationTaskSuite

Now, move to the [dspace-src]/dspace/config/modules/ folder. Some lines has to be added to the file curate.cfg in the list ”Task Class Implementations” in order to enable and configure the replication tasks (plugin.named.org.dspace.curate.CurationTask):

```
plugin.named.org.dspace.curate.CurationTask = \
    ... (YOUR EXISTING TASKS) ... , \
    org.dspace.ctask.replicate.EstimateAIPSize = estaipsize, \
    org.dspace.ctask.replicate.ReadOdometer = readodometer, \
    org.dspace.ctask.replicate.TransmitAIP = transmitaip, \
    org.dspace.ctask.replicate.TransmitSingleAIP = transmitsingleaip, \
    org.dspace.ctask.replicate.VerifyAIP = verifyaip, \
    org.dspace.ctask.replicate.FetchAIP = fetchaip, \
    org.dspace.ctask.replicate.CompareWithAIP = auditaip, \
    org.dspace.ctask.replicate.RemoveAIP = removeaip, \
    org.dspace.ctask.replicate.METSRestoreFromAIP = restorefromaip, \
    org.dspace.ctask.replicate.METSRestoreFromAIP = replacewithaip, \
    org.dspace.ctask.replicate.METSRestoreFromAIP = restorekeepexisting, \
    org.dspace.ctask.replicate.METSRestoreFromAIP = restoresinglefromaip, \
    org.dspace.ctask.replicate.METSRestoreFromAIP = replacesinglewithaip
```

For giving a ”readable” name to the replication tasks, these lines are appended to the preexisting in the tasknames section:

```
ui.tasknames = \
    ... (YOUR EXISTING TASK NAMES) ... , \
    estaipsize = Estimate Storage Space for AIP(s), \
    readodometer = Read Odometer, \
    transmitaip = Transmit AIP(s) to Storage, \
    verifyaip = Verify AIP(s) exist in Storage, \
    fetchaip = Fetch AIP(s) from Storage, \
    auditaip = Audit against AIP(s), \
    removeaip = Remove AIP(s) from Storage, \
    restorefromaip = Restore Missing Object(s) from AIP(s), \
    replacewithaip = Replace Existing Object(s) with AIP(s), \
    restorekeepexisting = Restore Missing Object(s) but Keep Existing Objects,\
    restoresinglefromaip = Restore Single Object from AIP, \
    replacesinglewithaip = Replace Single Object with AIP
```

Optionally, it is possible to divide the curation tasks in two subgroups according to their purposes:

```
# Tasks may be organized into named groups which display together in UI
# drop-downs
ui.taskgroups = \
   general = General Purpose Tasks, \
```

```
   replicate = Replication Suite Tasks

# Group membership is defined using comma-separated lists of task names,
# one property per group
ui.taskgroup.general = profileformats, requiredmetadata, checklinks
ui.taskgroup.replicate = estaipsize, readodometer, transmitaip, verifyaip,
fetchaip, auditaip, removeaip, restorefromaip, replacewithaip,
restorekeepexisting, restoresinglefromaip, replacesinglewithaip
```

The AIPs are thought to be easily stored in a remote environment, for maintaining in a secure place all DSpace contents. By default, for saving objects in a cloud, the Duracloud is the one advised by default. Duracloud is intended for working in accordance with DSpace, in fact there are a lot of features integrated between these two environments that can be used for making easier the preservation process.
For configuring the work environment and automating the backup processes, the administrator has to make some configurations. Editing [dspace-src]/dspace/config/modules/duracloud.cfg file DSpace instance is connected with Duracloud, passing in this file the data for the connection with the cloud service.

```
# DuraCloud service location (just the hostname)
host = demo.duracloud.org

# DuraCloud service port (usually 443 for https)
port = 443
context = durastore

# DuraCloud user name
username = myduraclouduser
# DuraCloud password
password = passw0rd
```

The [dspace-src]/dspace/config/modules/replicate.cfg has to be edited. For enabling Duracloud as storage place the following settings are recommended:

```
# Replica store implementation class (specify one)
plugin.single.org.dspace.ctask.replicate.ObjectStore = \
    org.dspace.ctask.replicate.store.DuraCloudObjectStore
```

The space where to store AIP contents in Duracloud is named aip-store by default:

```
# The primary storage group / folder where AIPs are stored/retrieved when AIP
# based tasks are executed (e.g. "Transmit AIP", "Restore from AIP")
group.aip.name = aip-store
```

For the automation of synchronization process between DSpace contents and Duracloud ones, the administrator user needs to add to [dspace-src]/dspace/config/dspace.cfg some lines at the end of the list of ”event.consumer.”:

```
#### Event System Configuration ####

# ADD the "replicate" consumer to the end of the list of 'default.consumers'
#(This enables the consumer)
event.dispatcher.default.consumers = versioning, search, browse, discovery,
eperson, harvester, replicate

....

# Configure consumer to manage METS AIP content replication
event.consumer.replicate.class = org.dspace.ctask.replicate.METSReplicateConsumer
event.consumer.replicate.filters = Community|Collection|Item|Group|EPerson+All
```

Now in the previously edited replicate.cfg, the default configurations responsible for the replication tasks process are the following:

```
###  ReplicateConsumer settings ###
# ReplicateConsumer must be properly declared/configured in dspace.cfg
# All tasks defined will be queued, unless the '+p' suffix is appended, when
# they will be immediately performed. Exercise considerable caution when using
# +p, as lengthy tasks can adversely affect UI or other responsiveness.

# Replicate event consumer tasks upon install/add events.
# A comma separated list of valid task plugin names (with optional '+p' suffix)
# By default we transmit a new AIP when a new object is added
consumer.tasks.add = transmitsingleaip

# Replicate event consumer tasks upon modification events.
# A comma separated list of valid task plugin names (with optional '+p' suffix)
# By default we transmit an updated AIP when an object is modified
consumer.tasks.mod = transmitsingleaip

# Replicate event consumer tasks upon a delete/remove events.
# A comma separated list of valid task plugin names (with optional '+p' suffix)
# By default we write out a deletion catalog & move the deleted object's AIP
# to the "trash" group in storage (where it can be permanently deleted later)
consumer.tasks.del = catalog+p

# Replicate event consumer queue name - where all queued tasks are placed
# This queue appears under the curate.cfg file's 'taskqueue.dir'
# (default taskqueue location is [dspace]/ctqueues/)
consumer.queue = replication
```

It could be better to check if these settings are coherent with the ones here illustrated. After that, the automation of the synchronization process is assured.

The last file to edit is [dspace-src]/dspace/config/modules/replicate-mets.cfg, used by replicate.cfg in case of AIP packager format (for details about other formats [34]). The contents are the following:

```
# Restore Task (hierarchical)
#   This task runs the recursive 'Default Restore Mode' (-r -a) option of
#   the AIP Backup & Restore tool.
restorefromaip.restoreMode = true
restorefromaip.recursiveMode = true
restorefromaip.createMetadataFields = true
restorefromaip.skipIfParentMissing = true

# Replace Task (hierarchical)
#   This task runs the recursive 'Force Replace Mode' (-r -f -a) option of
#   the AIP Backup & Restore tool.
replacewithaip.replaceMode = true
replacewithaip.recursiveMode = true
replacewithaip.createMetadataFields = true
replacewithaip.skipIfParentMissing = true

# Keep Existing Task (hierarchical)
#   This task runs the recursive 'Restore, Keep Existing' (-r -k -a) option of
#   the AIP Backup & Restore tool.
restorekeepexisting.keepExistingMode = true
restorekeepexisting.recursiveMode = true
restorekeepexisting.createMetadataFields = true
restorekeepexisting.skipIfParentMissing = true

# Restore Task (single object)
#   This task runs the 'Default Restore Mode' (-r) option of the AIP Backup &
#   Restore tool.
restoresinglefromaip.restoreMode = true
restoresinglefromaip.recursiveMode = false
restoresinglefromaip.createMetadataFields = true
restoresinglefromaip.skipIfParentMissing = false

# Replace Task (single object)
#   This task runs the 'Force Replace Mode' (-r -f) option of the AIP Backup &
#   Restore tool.
replacesinglewithaip.replaceMode = true
replacesinglewithaip.recursiveMode = false
replacesinglewithaip.createMetadataFields = true
replacesinglewithaip.skipIfParentMissing = false
```

[34] https://wiki.duraspace.org/display/DSPACE/ReplicationTaskSuite

These are the basic settings for the work environment that I wantend to highlight. The complete replicate.cfg, replicate-mets.cfg and curate.cfg are available at the following link: `https://github.com/DSpace/dspace-replicate/tree/master/config/modules`.

Once that pom.xml has been modified and the other files has been created, it is possible to apply the changes. Move to [dspace-src]/dspace/ folder and run

```
mvn clean package
```

After that, move to [dspace-src]/dspace/target/dspace-[version]-build/ folder and run

```
ant update
```

At this point, everything is ready for the backup and restore features.

C.10 Replication Suite

The replication suites can be performed from command line running the ./dspace packager command from the [dspace]/bin/ install folder. For specific actions the admin has to add some options. For downloading DSpace content in AIP format the -d command is added to the packager command. It is possible to download a single object at a time or a hierarchy of objects, adding -a information.

A single object is downloaded, an Item for example:

```
[dspace]/bin/dspace packager -d -t AIP -e admin@dspace_account_mail.edu -i
1234/567 item-aip.zip
```

Where the mail address is the one used by the administrator for DSpace registration, 1234/567 is the Item handle and item-aip.zip is the chosen name for the downloaded AIP. When the whole hierarchy is required, adding -a, all the children objects will be downloaded:

```
[dspace]/bin/dspace packager -d -a -t AIP -e admin@dspace_account_mail.edu -i
1234/56 collection-aip.zip
```

It is possible to download the whole site and its dependencies, using the site handle, 1234/0.
When the administrator wants to restore old contents or adds new ones on a DSpace instance, there are two ways to proceed. The first and easier is the ingestion: an object is added to an existing environment using the submit option (-s) in the command line and a new object is created

```
[dspace]/bin/dspace packager -s -t AIP -e admin@dspace_account_mail.edu
-p 1234/890 /path/to/the/ingesting/aip.zip
```

The -p option needs the new parent of the object as parameter and a new handle will be assigned to the submitted object by default. When a whole hierarchy is submitted, the -a option is added:

```
[dspace]/bin/dspace packager -s -a -t AIP -e admin@dspace_account_mail.edu
-p 1234/0 /path/to/the/ingesting/community-aip.zip
```

This last example illustrates how to add a Community to an existing site. The handle assigned as parent to the object is the site one.
The other way to insert objects in a DSpace instance is the restore mode. The restore features attempt to maintain the preceding structure of an object, reducing the reported changes to a minimum. Some different options are available for managing the restore process. The basic option is the -r one: it restores an object and its hierarchy, maintaining the handles and, if preexisting, changing the information contained in the AIP as less as possible. Adding the -k command, the restore is done and, if an object already exists, the restore process skips over it (and all children objects), and continue to restore all other non-existing elements.

```
[dspace]/bin/dspace packager -r -a -t AIP -e admin@dspace_account_mail.edu
aip1234.zip
```

It is possible to see that, differently from -s option, this feature doesn't require the parent object option -p. If possible, it retrieves it from the metadata contained in the mets.xml file by DSpace. This feature works fine if the aip1234.zip hierarchy doesn't exist in the DSpace instance. If some children objects already exist, an error will be displayed. In this situation the administrator has to use the more powerful Restore Keep Existing Mode (-r -k) or the Force Replace Mode (-r -f).

The -r -k options attempt to skip over objects that already exist. This set of options will report to the user if some objects already exist. It could be useful when a part of a hierarchy has been lost, so for restoring the missing objects the user can run the following command:

```
[dspace]/bin/dspace packager -r -a -k -t AIP -e admin@dspace_account_mail.edu
aip1234.zip
```

The -r -f options, force restore mode, operate overwriting objects found to already exist, deleting the old contents and substituting with the new ones. It is useful when the admin wants to restore an object maintaining the links with existing children objects. This

method could be dangerous because it deletes the old contents for inserting new ones. For example, when attempting to restore an entire site on a clean DSpace instance it is a powerful way; In case of problems about dependencies between objects, the force restore recreates them.

```
[dspace]/bin/dspace packager -r -a -f -t AIP -e admin@dspace_account_mail.edu
-i 1234/0 -o skipIfParentMissing=true /full/path/to/your/site-aip.zip
```

In this way, the administrator can obtain the old work environment maintaining all the dependencies. The -o option is for adding other requests to the command line; in this case the administrator has chosen skipIfParentMissing set to true, in order to ignore the errors involving the missing parents. When a recursive ingestion is performed, it doesn't cause problems. Missing parent objects are then ingested, so that it will automatically restore the Item mapping that caused the error.

An interesting feature about all the restore modes seen in this section is the maintenance of previously created groups and ePersons. In fact, when the admin recreates the DSpace site, the mail that he uses is the one for registering to the old DSpace instance. All privileges are maintained, so that the environment will be the same that the administrator wanted to preserve.
These are some useful features of command line packager. For more details, see the DSpace official guide [35].

The before mentioned commands can be executed from the xml GUI's section Curation Tasks, as could be understood from the configuration paragraph.

In Figure 40, it is possible to see that the curation tasks are divided in two groups: general tasks and replication suite tasks. The latter are the ones involved in backup and restore functionalities.

These are the available options in the GUI:

- Estimate Storage Space for AIP(s) determines how many space is available for AIPs in the storage place
- Read Odometer gives some information about DSpace contents (dimension of existing objects, dimension of uploaded and downloaded contents)
- Transmit AIP(s) to Storage downloads the AIPs in the storage place (local folder or cloud)
- Verify AIP(s) exist in Storage for detecting if a specified object already exists in the storage place

[35] https://wiki.duraspace.org/display/DSDOC5x/AIP+Backup+and+Restore

System Curation Tasks

Handle of DSpace Object:

Hint: Enter [your-handle-prefix]/0 to run a task across entire site (not all tasks may support this capability)

Choose from the following groups:

General Purpose Tasks

General Purpose Tasks

Replication Suite Tasks

Task:

Profile Bitstream Formats

Perform Queue

Figure 39: Curation Tasks divided according to their functionalities

- Fetch AIPs from Storage takes from the storage place the missing children objects of the specified ones. This corresponds to the submission mode explained in command line section
- Audit against AIP(s) checks if there are differences between the stored objects and the ones in DSpace
- Remove AIP(s) from storage eliminates objects from the storage space
- Restore Missing Object(s) from AIPs restore the missing objects in DSpace from the specified AIP's handle.
- Replace Existing Object(s) with AIP(s) substitutes the existing DSpace contents with the stored ones, like with -r -f options.
- Restore Missing Object(s) but Keep Existing Objects checks for the missing objects in DSpace and restore these ones from the AIPs. This feature corresponds to the -r -k one.
- Restore Single Object from AIP and Replace Single Object with AIP correspond to the before mentioned ones, but specific for single objects management.

C.11 Customized Cloud Features: ownCloud

DSpace works fine with Duracloud by default, but what to do in case of a different cloud service? In this section I try to explain how did I set a different scenario, using ownCloud.

System Curation Tasks

Handle of DSpace Object:
Hint: Enter [your-handle-prefix]/0 to run a task across entire site (not all tasks may support this capability)

Choose from the following groups:
Replication Suite Tasks

Task:
Estimate Storage Space for AIP(s)
Estimate Storage Space for AIP(s)
Read Odometer
Transmit AIP(s) to Storage
Verify AIP(s) exist in Storage
Fetch AIP(s) from Storage
Audit against AIP(s)
Remove AIP(s) from Storage
Restore Missing Object(s) from AIP(s)
Replace Existing Object(s) with AIP(s)
Restore Missing Object(s) but Keep Existing Objects
Restore Single Object from AIP
Replace Single Object with AIP

Figure 40: All the replication tasks available in the curation tasks section of the xml interface

ownCloud Installation

First of all the administrator has to install the ownCloud client service. Here is a basic installation of this service in order to show that it is easy and quick to start storing DSpace content in ownCloud.

```
$ sudo apt-get install owncloud-client
```

Then, it is useful to install its command line tool:

```
$ sudo apt-get install owncloudcmd
```

This tool allows to use the ownCloud features from command line.

ownCloud Configuration

For starting to store AIPs in ownCloud, the user has to synchronize the local folder with the remote storage place provided by ownCloud. It is possible to do it creating an hidden folder that we could call .owncloud and to create inside it a configuration file, owncloud.cfg, with the following contents:

```
[General]

[ownCloud]
url=http://owncloud-site/owncloud/remote.php/webdav/
http_user=
authType=http
user=name_registered_in_owncloud
```

In this way the access to ownCloud is granted.

The user can load contents in ownCloud manually, passing in command line the following lines:

```
$ owncloudcmd --confdir /path/to/configuration/file/owncloud.cfg
/path/to/AIP/folder/ owncloud://"username":"password"@"owncloud-site-
address/owncloud"
```

The argument passed to –confdir option is the location of the configuration file just edited; the second folder pointed is the one containing the AIPs that we want to synchronize with ownCloud.
The user has to perform two steps manually: download the AIPs from DSpace and store them in the cloud. It could be a little frustrating if this process has to be replicated once every hour of the day! It is possible to automate these processes making two cron jobs. In a Linux machine it is possible to write a cron table in this way:

```
$ crontab -e
```

The following content has to be written in the cron table:

```
0 */1 * * *  /bin/dspace packager -u -d -a -t AIP -e admin@dspace_account_mail.edu
-i 1234/0 /path/to/AIP/folder/sitewide-aip.zip


0 */1 * * * owncloudcmd --confdir /path/to/configuration/file/owncloud.cfg
```

```
/path/to/AIP/folder/ owncloud://"username":"password"@"owncloud-site-
address/owncloud"
```

The 0 */1 * * * indicates that the jobs are run once every hour of the day.

D Implementation of Reference Model Workflows

In this Appendix we describe the implementation of the two workflows defined in the PoF Reference Model which have been implemented in the third prototype, as discussed in Section 3: Preservation Preparation and Re-activation. For each workflow we present a sequence of steps, with the help of some application screenshots.

D.1 Preservation Preparation Workflow

The Preservation Preparation workflow includes several tasks from the selection of the content to be preserved up to the transfer of such content to the archive. The steps of the workflow in relationship with the framework components are depicted in Figure 4 in Section 3.

In the following we describe the implementation of each step in the current prototype. The *select* step is the first task in the workflow, which could be triggered by the Forgettor (e.g based on PV calculation or on other evidences provided by the Active System) or by the Context-aware Preservation Manager (taking into account specific preservation rules). These two components are still under development (see Section 5) and are not fully integrated in the middleware, so this process is not fully automated. For demonstration purposes, the user triggers the preservation sending a request to the PoF Middleware (the calculated PV can be used to guide the selection of the content to preserve). All the other steps in the workflow have been fully implemented.

A simplified representation of the Preservation Preparation workflow is shown in Figure 41, where the details about the involved components and flow branches in case of errors or exceptions have been omitted for the sake of clarity. The internal details of the routing engine behaviour have been omitted as well: the messaging system and the routing engine logic have been simplified using iterative or parallel expansion regions. It is worth noting that the set of steps below refer more specifically to the image selection scenario (for example the Extractor related step is referring to image analysis), so additional or modified steps have been implemented for the other scenarios.

1. A preservation request is triggered by the Active System (see Figure 42): the CMIS ID of the selected resource is sent to the middleware REST endpoint (see Table 6). The CMIS resource can be a collection or a single item. The PV for the whole collection or for the single item is also sent to the middleware. This task corresponds to the *select* step in the workflow.

2. The next step in the workflow, *provide*, is implemented by different components. The request sent to the middleware REST server is processed by the Scheduler, which instantiates a new Task (with `TaskType` equal to PRESERVATION). The Task is stored in the object DB used by the ID Manager and Metadata Repository. The Task ID is returned to the user, this ID can be used to monitor the progress of the request

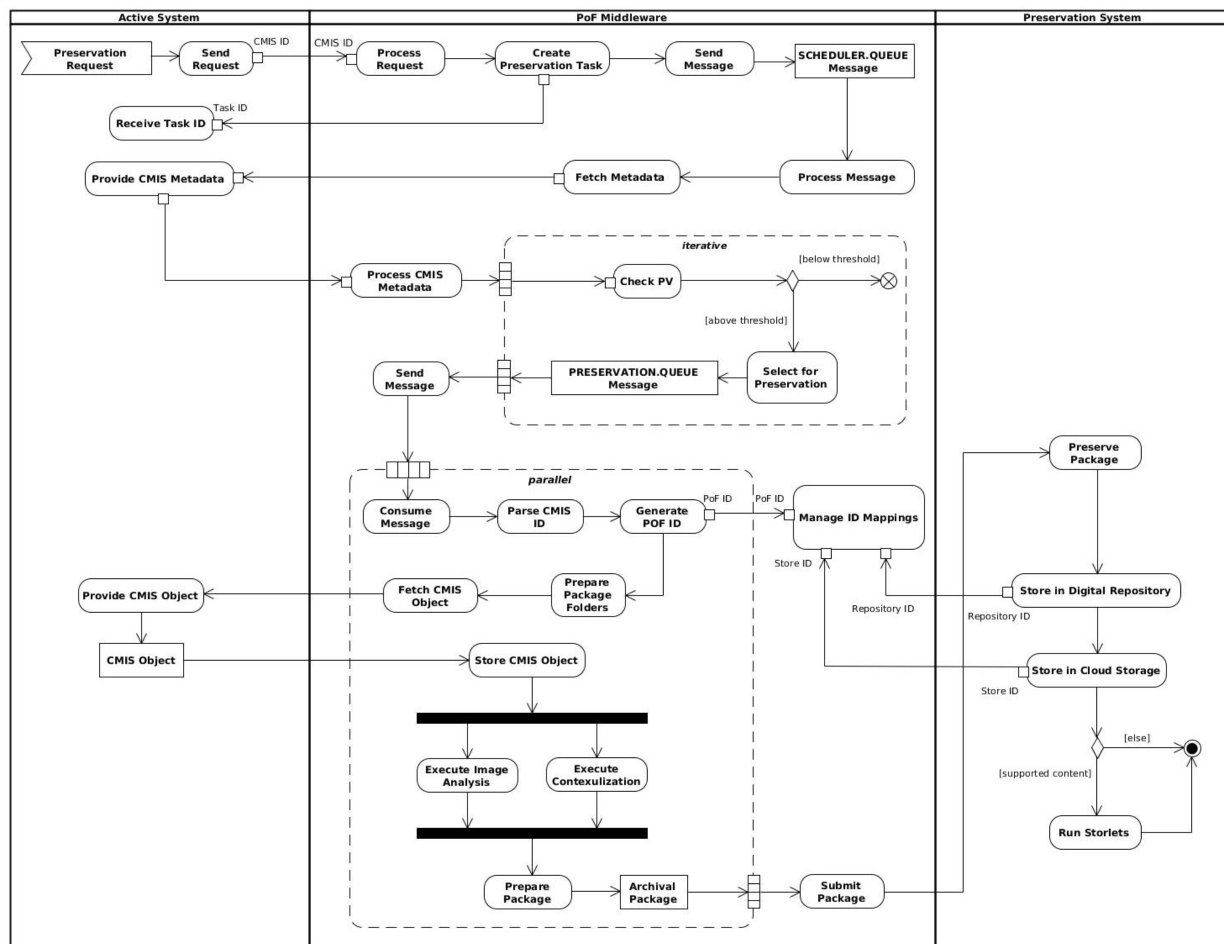

Figure 41: UML activity diagram for the Preservation Preparation workflow.

and to get the results when completed.

3. The Scheduler prepares a new message wrapping the received information about the resource and sends it to the SCHEDULER.QUEUE. The message header contains the information about the Task type. The flow control is now managed by the routing engine, which takes care of dispatching the message to the appropriate components.

4. The CMIS ID provided by the Active System (and stored in the message body) is used to fetch information about the content to be preserved: based on CMIS object attributes, the Collector checks whether the content to be preserved is a single resource or a collection (e.g. a `pimo:LifeSituation`).

5. If the content is a single resource, a single message is sent to PRESERVATION.QUEUE. If the content is associated to a collection, the Collector retrieves the information about each resource in the collection (using the CMIS relationship attribute) and for each resource the CMIS ID and the corresponding PV are retrieved.

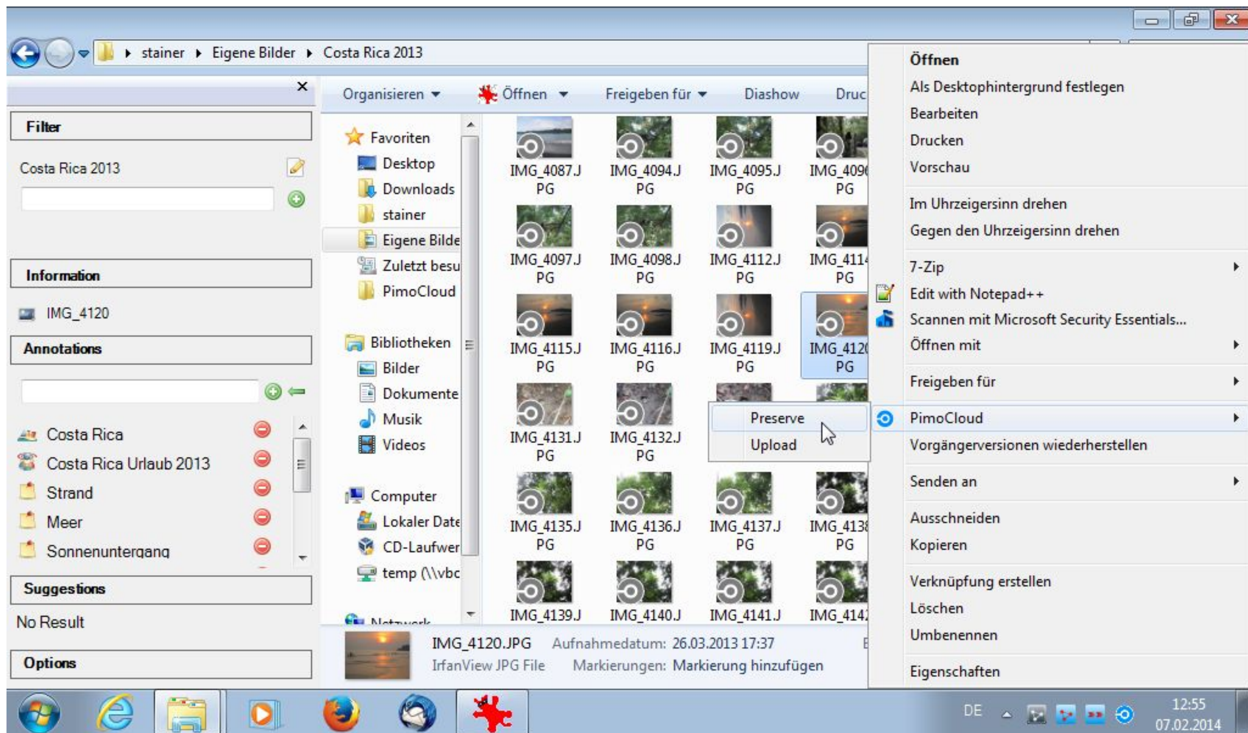

Figure 42: User interface of PIMO: manual selection of resource to be preserved.

6. The following preservation decision depends on the availability of a *Preservation Broker Contract* where the respective policies are defined.
   In case no contract is available (e.g., for prototypes in early stages of integration into the PoF) we choose a threshold equal to 0.8: only resources in the collection with a high PV are considered eligible for preservation. The PV threshold is configured in the Forgettor code deployed in the middleware.
   In case a contract defines the PV categories (`gold`, `silver`, `bronze`, `wood`, `ash`) to be preserved (possibly including the Preservation Levels (e.g., `premium`, `standard`, `basic`, `none`, as done in the Personal Preservation Pilot in D9.4 [Maus et al., 2015])), those resources are fetched where the Preservation Level is not `none`. (Depending on the policies for the other levels, also a distinction in time periods of fetching or updating of resources could be possible, such as premium every time changes are there, and basic only every other year).

7. For each selected resource a separate message is sent to the PRESERVATION.QUEUE. It is worth noticing that also the collection itself is preserved: the package representing the collection has no resources inside, but just a list of resources in the collection and some global descriptions referring to the whole collection.

8. Note: in the following we describe the other steps in the workflow from the point of view of a single resource; when dealing with collection each step is executed in parallel for all resources in the collection and for the collection object itself. Currently each resource is stored in the Preservation System as a separate package and the collection package is used to preserve information about the aggregation (e.g. the

collection represents a photo collection for a business trip). The only step which is referring to the whole collection is associated to the Condensator, because the clustering algorithm is executed on the set of images in the collection.

9. The messages sent to the PRESERVATION.QUEUE are consumed under the control of the routing engine. The next step is the ID generation: the ID Manager gets the information about the CMIS ID parsing the message body, generates a new unique ID (`pofId`) and stores the mapping between the two in its internal object DB. From this step onward, all the other tasks use the `pofId` taken from the body of the received messages. This task completes the *provide* step in the workflow.

10. After each resource is assigned a new unique ID, the Collector can fetch the files from the Active System CMIS repository and store them in the middleware: for each resource a folder named according to the unique ID is created. This folder is the temporary folder for package preparation and is used during the next steps: a subfolder for the content and one for the metadata is created. The Collector fetches all descriptive metadata associated to each CMIS object and stores such metadata in its internal DB. Due to the asynchronous nature of the messaging layer, multiple resources can be retrieved in parallel and the results are stored in the corresponding folder with the unique name. This task completes the *provide* step in the workflow.

11. The *enrich* step in the workflow is also associated to different components. After the retrieval process for a given resource has been completed, the Collector returns a message to the routing engine containing information about the package folder for that resource. The routing engine sends this information to the next component in the flow, the Extractor.

12. The request sent to the Extractor contains information about content types for each package. In the current implementation the Extractor executes image analysis for all the images in the package, if any, otherwise it is skipped. The Extractor component running in the middleware prepares a public URL for the images and sends this information to the remote image analysis service running at CERTH. The image analysis type is an additional parameter which can be configured in the workflow (in the provided routing engine sample configuration all implemented image analysis methods are executed).

13. The next step involves the Contextualizer, which processes messages for text resources. The contextualization result is added to the temporary package folder, as part of the metadata. A context referring to the whole collection could be stored in the collection package.

14. An optional step is defined in the workflow, for collections of images: a clustering algorithm (provided by the Condensator) can be executed. Currently if the number of selected images from the original collection is equal or greater than 10, the Condensator is executed and the results are stored within the collection package, since they are related to the whole collection and not to each resource separately. The intermediate products (metadata files, temporary results, etc) are stored in the

middleware internal object DB or on the file system. This task completes the *enrich* step in the workflow.

15. The last two steps in the workflow, *package* and *transfer*, are assigned to a single component, the Archiver. After all processing steps have been completed, the Archiver receives a request message to prepare the package and submit it to the Preservation System. The package is sent to the archive using its REST APIs. An example of archived content in DSpace is shown in Figure 43: the resources and the associated metadata are shown.

Figure 43: Preview of archived resource in DSpace.

16. The package submission is made up of two steps: first the package is imported into DSpace and then it is copied in the cloud storage. Two additional IDs are assigned to the package: a `repositoryId` (from DSpace) and a `storageId` (from cloud storage). Both are added to the ID mapping for that resource and stored in the object DB by the ID Manager (see Figure 44).

17. Different Storlets are executed in the Preservation-aware Storage System upon content ingest: for example a Metadata Enrichment Storlet is executed on text content (the extracted metadata are indexed and are used by the metadata search functionality exposed by the cloud storage).

18. The steps above are executed for each resource. The status of the resource is updated: resource is shown as *preserved* in the Active System. In case of collections, the tasks above are executed for each resource in parallel and the preservation status is updated only when the resources in collection and the collection object itself

Home | Middleware | Preservation System

DSpace XML UI

Web GUI for DSpace administration

DSpace OAI-PMH UI

OAI-PMH Data Provider

CMIS Repository	CMIS Object Type	PV	PoF Middleware	Digital Repository	Cloud Storage
mw	cmis:document	1.0	0fbe8297-766c-4bc8-9b87-42a3b81bc913	600826/178	pof-documents:package-0fbe8297-766c-4bc8-9b87-42a3b81bc913.tar
mw	cmis:document	1.0	5bd9974b-b611-4190-b8c0-67b310cb7ccd	600826/180	pof-documents:package-5bd9974b-b611-4190-b8c0-67b310cb7ccd.tar
mw	cmis:document	1.0	91a62b5e-5f14-451d-b0cc-d19df095bc03	600826/177	pof-documents:package-91a62b5e-5f14-451d-b0cc-d19df095bc03.tar
pimo11	pimo:document	1.0	c87c3ddd-867f-4c36-b685-5ca99670edb1	600826/175	pof-images:package-c87c3ddd-867f-4c36-b685-5ca99670edb1.tar
pimo11	pimo:document	0.8582130074501038	ce5df4f2-1792-4451-a72a-427bb89c2208	600826/176	pof-images:package-ce5df4f2-1792-4451-a72a-427bb89c2208.tar
mw	cmis:document	1.0	dbce3f58-4067-49b5-a711-892662efe06d	600826/179	pof-documents:package-dbce3f58-4067-49b5-a711-892662efe06d.tar

Figure 44: Web interface of the PoF Middleware, the different IDs associated to the same content are shown, as well as the preservation status and the associated PV.

have been correctly transferred to the Preservation System. This task completes the *transfer* step in the workflow.

D.2 Re-activation Workflow

After the content has been successfully preserved, a request to restore one or more resources can be triggered, as described below. This is associated to the Re-activation workflow defined in the model. Currently almost all the steps have been implemented: the only task which is still under development is the re-contextualization. Compared to the first release, the re-activated content is now retrieved from the cloud storage, where it has been actively preserved and possibly transformed.

1. The first step in the workflow is the `request`: the Active System can send a request to the PoF Middleware using the REST APIs, in order to restore or update the preserved content locally.

2. Similarly to the Preservation Preparation workflow, the request sent to the middleware REST server is is processed by the Scheduler, which instantiates a new Task (with `TaskType` equal to REACTIVATION). The Task is stored in the object DB used by the ID Manager and Metadata Repository. The Task ID is returned to the user, this ID can be used to monitor the progress of the request and to get the results when completed.

3. The Scheduler prepares a new message wrapping the received information about

the resource and sends it to the SCHEDULER.QUEUE. The message header contains the information about the Task type. The flow control is now managed by the routing engine, which takes care of dispatching the message to the appropriate components. Additional information about the preserved content could be retrieved by the Collector. This task completes the `request` step in the workflow.

4. The next step in the workflow is the `search`: the ID Manager receives a message whose body contains the CMIS ID of the preserved content to be reactivated (single resource or collection). Using the ID mappings stored in the ID Manager object DB, the repository and storage ID are retrieved. The Navigator can provide additional search features, but currently it is not used because the content ID is provided. After correct identification of the content, the `search` step completes.

5. During the `retrieve` step, the Archiver receives a message with the content information and retrieves it from the Preservation System sending a request to the archive REST service. Compared to the original resource, the content is returned as a package (including both resources and metadata, including the context). This task is associated to the `prepare` step in the workflow. In case of collections, the resources are retrieved separately and several packages are returned to the user. This is under development, to combine multiple archived packages in a single dissemination package. This task completes the `package` step in the workflow (the re-contextualization is still under development).

6. The last step in the workflow is the `deliver`: the content is published by the Collector on the CMIS repository implemented in the middleware and can be accessed by the Active System using the CMIS ID on the middleware repository. In order to get this CMIS ID, the Active System can use two different mechanisms: using the task monitoring mechanism, the task ID provided by the middleware at the beginning of the workflow can be used to monitor the status of the re-activation process and when the task is completed the CMIS ID is available in the task results (which can be retrieved using the middleware REST API); alternatively, the Active System can register with the messaging system and obtain a dedicated queue where such notifications are published: a message consumer running in the Active System can retrieve a message containing the CMIS ID. The task monitoring is currently used by TYPO3 CMS, while for the Semantic Desktop the message queue is the preferred mechanism.

7. The Active System can retrieve the content from the middleware CMIS repository and the re-activation workflow is then completed.[36]

[36] A video showing preserve and restore can be seen in the Personal Preservation Pilot I at `https://pimo.opendfki.de/wp9-pilot/preservation_sd.html`

E Experimental APIs of the Memory Buoyancy Assessor

In the following we describe the main APIs and I/O formats for the MB Assessor component, which is part of the Forgettor (Section 5.7). All the services reported below have been deployed on a test RESTful server running at LUH premises, hosting the Forgettor Server[37]. Sample code for the MB Assessor client, written in Java, is reported in Listing 5.

Querying MB Values of PIMO resources

This service allows the client to query the estimated MB values, and get a numerical value from 0 to 1 in plain-text as a result (or NaN if the values are not yet estimated, or the resource is not registered in the system).

1. REST service type: GET.
2. URI Input: `http://forgetit.l3s.uni-hannover.de:8092/pimo/mb/query?u=<userID>&r=<resourceID>&t=<timestampinUNIXepochs>`.
3. URI output: (plain-text) MB score in [0,1] or NaN.
4. Query example: `http://forgetit.l3s.uni-hannover.de:8092/pimo/mb/query?u=pimo:1327593979868:1&r=pimo:1381327141334:56&t=1384506130`.

Register PIMO resources

In order to compute the MB scores using the background sub-component, the resources must be registered; this service allows the client to send the list of resource IDs to register for the computation.

1. REST service type: POST.
2. URI Input: `http://forgetit.l3s.uni-hannover.de:8092/pimo/res/register`.
3. URI output: a JSON response object that containing
   - the response status code:
     - CREATED: the resources have been successfully registered.
     - NOT_MODIFIED: the resources are already registered, or the attempt makes no changes in the system.
     - INTERNAL_SERVER_ERROR: server failed to register, internal error.
     - PARTIAL_CONTENT (for bulk registration): only a sub set of resources are registered.
   - the list of IDs for successfully registered resources.

[37] Forgettor Server - `http://forgetit.l3s.uni-hannover.de:8092/application.wadl`

Listing 5: Sample code for MB Assessor client.

```
import javax.ws.rs.client.AsyncInvoker;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.InvocationCallback;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import org.glassfish.jersey.client.ClientConfig;
import eu.forgetit.l3s.services.schema.MBRequest;
import eu.forgetit.l3s.services.schema.MBVEntity;
import eu.forgetit.l3s.services.schema.MBVList;
...

// Define the entry point of the web service domain
ClientConfig clientConfig = new ClientConfig();
client = ClientBuilder.newClient(clientConfig);
WebTarget target = client.target("http://forgetit.l3s.uni-hannover.de:8092");
...

// Define an asynchronous REST request
final AsyncInvoker asyncInvoker = target.path("/pimo/mb/bulk-query").
request(MediaType.APPLICATION_JSON).async();

// Define a request object which contains collection ID (account), epoch value of demanded
// calculation timestamp, and a list of resource IDs

MBRequest req = new MBRequest();
req.setAccount("pimo:1327593979868:1");
req.setTime(1386686731);

List<String> res = new ArrayList<>(4);
res.add("pimo:1381327141334:56"); // the CMIS ID used in the PoF Middleware
res.add("pimo:1374842706949:3");
res.add("pimo:1385386608202:18");
res.add("pimo:1381327141334:55");
res.add("pimo:1365627012409:41");

req.setResources(res);

// Send the asynchronous request to the service
Entity<MBRequest> reqEntity = Entity.entity(req, MediaType.APPLICATION_JSON);

MBVList futureResp = null;

try {
    futureResp = asyncInvoker.post(reqEntity, new
    InvocationCallback<MBVList>() {

        @Override
        public void completed(MBVList response) {
            System.out.println("Response entity '" + response + "' received.");
            for (MBVEntity mbve : response.getValues()) {
              System.out.println(mbve.toCompiledString());
            }
        }

        @Override
        public void failed(Throwable throwable) {
            System.out.println("Invocation failed.");
            throwable.printStackTrace();
        }

    }).get();

} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```