SEVENTH FRAMEWORK PROGRAMME

ForgetIT

www.forgetit-project.eu

ForgetIT

Concise Preservation by Combining Managed Forgetting and Contextualized Remembering

Grant Agreement No. 600826

Deliverable D8.4

Work-package	WP8: The Preserve-or-Forget Reference Model and Framework
Deliverable	D8.4: The Preserve-or-Forget Framework – Second Release
Deliverable Leader	Francesco Gallo (EURIX)
Quality Assessor	Johannes Goslar (dkd)
Dissemination level	PU
Delivery date in Annex I	M27
Actual delivery date	30 July 2015 (M30)
Revisions	13
Status	Final
Keywords	PoF Framework, second prototype, integrated 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.

© 2015 Participants in the ForgetIT Project

Revision History

Date	Version	Major changes	Authors
14-01-2015	0.01	Document skeleton	EURIX
20-01-2015	0.02	Updated document structure, preliminary introduction, PoF architecture	EURIX
29-01-2015	0.03	Added PoF Reference Model, subsections for each middleware component, appendix for scenarios	EURIX
24-02-2015	0.04	Described middleware implementation, REST APIs and CMIS, Preservation System, updated architecture	IBM, EURIX
13-03-2015	0.05	Described preservation preparation and re-activation workflows, Digital Repository, updated appendix	EURIX, LUH, USFD, DFKI, IBM
26-03-2015	0.06	Updated architecture, reference model, middleware REST APIs, completed scenarios for second release, preliminary sections about Semantic Desktop and TYPO3	EURIX, DFKI, dkd, LUH
11-05-2015	0.07	Added ID Manager, Scheduler, Extractor, Collector/Archiver	LTU, CERTH, LUH, EURIX
19-05-2015	0.08	Added Condensator and Contextualizer, updated previous components, Introduction, preliminary executive summary	CERTH, USFD, EURIX
28-05-2015	0.09	Completed middleware components, added prototype implementation	LTU, USFD, EURIX
19-06-2015	0.10	Reviewed all sections, added software development, appendix for Forgettor APIs, added glossary	EURIX, LUH, DFKI, dkd
27-06-2015	0.11	Completed all sections, updated references, first draft version circulated for comments	EURIX
10-07-2015	0.12	Implemented comments from all partners, updated version for internal QA	EURIX, dkd
30-07-2015	0.13	Implemented comments from internal QA, final version	EURIX, dkd

List of Authors

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

Table of Contents

Executive Summary **7**

1 Introduction **8**

2 PoF Framework Architecture **11**

3 PoF Reference Model **13**

4 PoF Middleware **17**

4.1 PoF Enterprise Service Bus 17

4.1.1 Message-Oriented Middleware 17

4.1.2 Enterprise Integration Patterns 19

4.1.3 Asynchronous Routing Engine 20

4.1.4 PoF ESB Implementation 21

4.2 Middleware Configuration 22

4.3 RESTful Service 23

4.4 CMIS Integration 24

4.5 Implementation of Reference Model Workflows 26

4.5.1 Preservation Preparation Workflow 26

4.5.2 Re-activation Workflow 31

5 PoF Middleware Integrated Components **33**

5.1 ID Manager 33

5.2 Metadata Repository 35

5.3 Scheduler 38

5.4 Extractor 39

5.5 Condensator 43

5.6 Collector/Archiver 44

5.7 Forgettor 46

5.8 Contextualizer . . . 50
5.9 Navigator . . . 51
5.10 Context-aware Preservation Manager . . . 52

6 Active Systems **54**
6.1 Semantic Desktop . . . 54
6.2 TYPO3 . . . 56
6.3 CMIS-based User Applications . . . 58

7 Preservation System **59**
7.1 Digital Repository . . . 59
7.2 Preservation-aware Storage System . . . 61

8 PoF Framework: Second Prototype Implementation **63**

9 Conclusions **69**
9.1 Summary . . . 69
9.2 Assessment of Performance Indicators . . . 69
9.2.1 Evaluation of the PoF Framework . . . 71
9.3 Next Steps . . . 71

10 References **73**

Glossary **76**

A Middleware Configuration and Administration **77**

B Scenarios for the Second Prototype Demonstrations **86**
B.1 Scenario 1: Incremental Photo Preservation . . . 86
B.2 Scenario 2: Automated Contextualization and Re-contextualization . . . 88
B.3 Scenario 3: Automated Generation of Multimedia Diary . . . 90

C Experimental APIs of the Memory Buoyancy Assessor **92**

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 second release of the framework, developed during the second year of project and based on the first release, described in deliverable D8.3. The second prototype has been presented at the second annual project review. The final release of the framework is expected at the end of the project and will be described in deliverable D8.5.

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 initial version of the PoF Reference Model described in D8.2 have been used for the development of the framework. Currently two workflows have been implemented, for preservation preparation and re-activation. The other workflows for the evolutionary part of the model will be implemented by the final framework release.

The PoF Middleware REST APIs, defined in D8.1, have been updated with respect to the first 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 for the second release 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 the workplan for 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.

1 Introduction

The main topic of this document is the description of the second prototype implementation of the Preserve-or-Forget (PoF) Framework, which integrates the results achieved during the first two years and is based on the first release of the PoF Reference Model reported in deliverable D8.2 [ForgetIT, 2015g]. 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 [ForgetIT, 2013d]).

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 second prototype implements the relevant workflows of the functional part of the model.

The first release of the PoF Framework, described in deliverable D8.3 [ForgetIT, 2014e], was based on the components developed within the project during the first year. The second prototype was built on top of the first 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 second 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 second 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 D3.3, D4.3, D5.3 and D6.3 for the PoF Middleware components, D7.3 for the Preservation-aware Storage System, D8.3 for the Digital Repository and finally D9.3 and D10.2 for the Active Systems. Moreover, the outcomes of WP2 (see for example deliverables D2.2 and D2.3) 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. All the aforementioned deliverables are available on the project web site and are reported in the References.

The component description in this document focuses 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.

In addition to the end-to-end preservation workflow, which was updated and improved with respect to the first release, for the implementation of the second release three ref-

erence scenarios were identified: *Incremental Photo Preservation*, *Automated Contextualization and Re-contextualization* and *Automated Generation of Multimedia Diary*. We briefly describe such scenarios in Appendix B, pointing to other deliverables for further implementation details whenever available.

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 and the relevant workflows implemented in the second prototype is 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 first prototype and the workplan for the final release; 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 future activities towards the final framework release and provide an assessment of the results against WP8 success indicators, which have been defined in the project proposal; Appendix A provides implementation details for the configuration of the PoF Middleware; Appendix B describes the three representative scenarios mentioned above, used to demonstrate the second framework: in this Section we also include application screenshots from the demonstrations which were shown at the second project review; finally, Appendix C provides information about the experimental APIs for the Memory Buoyancy (MB) assessor.

Progress after first prototype

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

- the messaging layer infrastructure and the routing engine have been further developed, providing more flexibility and support for the integration of middleware components and the definition of relevant workflows; a web console for managing the messaging infrastructure, monitoring the workflows and the queues, has been implemented;

- the middleware software has been improved and makes use of configuration files for the workflow definitions, dynamic creation of component instances and configuration of the parameters to access remote services; the coupling between the components and the dependency on implementation details have been reduced;

- the PoF Middleware REST APIs have been updated and now provide a more stable integration mechanism; additional features of the CMIS standard [OASIS, 2013] are now used: additional descriptive and technical metadata about the content to be preserved is now retrieved by the PoF Framework and archived in the Preservation System;

- the updated versions of the middleware components have been integrated, while new components not available in the first release have been added: examples of

such new components include the Condensator and the Navigator;

- the Preservation System has been updated, mainly for what concerns the cloud storage part; an updated version of the Digital Repository is used and the integration mechanism to preserve and re-activate content as been improved, as described in the following; a new implementation of the cloud storage components with additional Storlets is available: content metadata enrichment executed upon ingest has been implemented, metadata search for content stored in the cloud can be used;

- both Active Systems have been updated and the integration with the PoF Middleware has been improved; based on CMIS representation , the support for Semantic Desktop collections has been added to the preservation workflow; the Preservation Value (PV) for the resources in the Active Systems is fetched using CMIS and is used to take decisions about content to be retrieved by PoF Middleware and preserved: this is an initial step towards managed forgetting, to be further investigated for the third release implementation;

The results described above represent a good progress towards WP8 objectives: in a nutshell, the PoF Framework has been developed during the second year following the first version of 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 [ForgetIT, 2013d], is made up of three layers: *Active Systems*, *Preserve-or-Forget (PoF) Middleware* and *Preservation System*. The last version of the UML component diagram for the overall 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 1 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 for the first release 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.

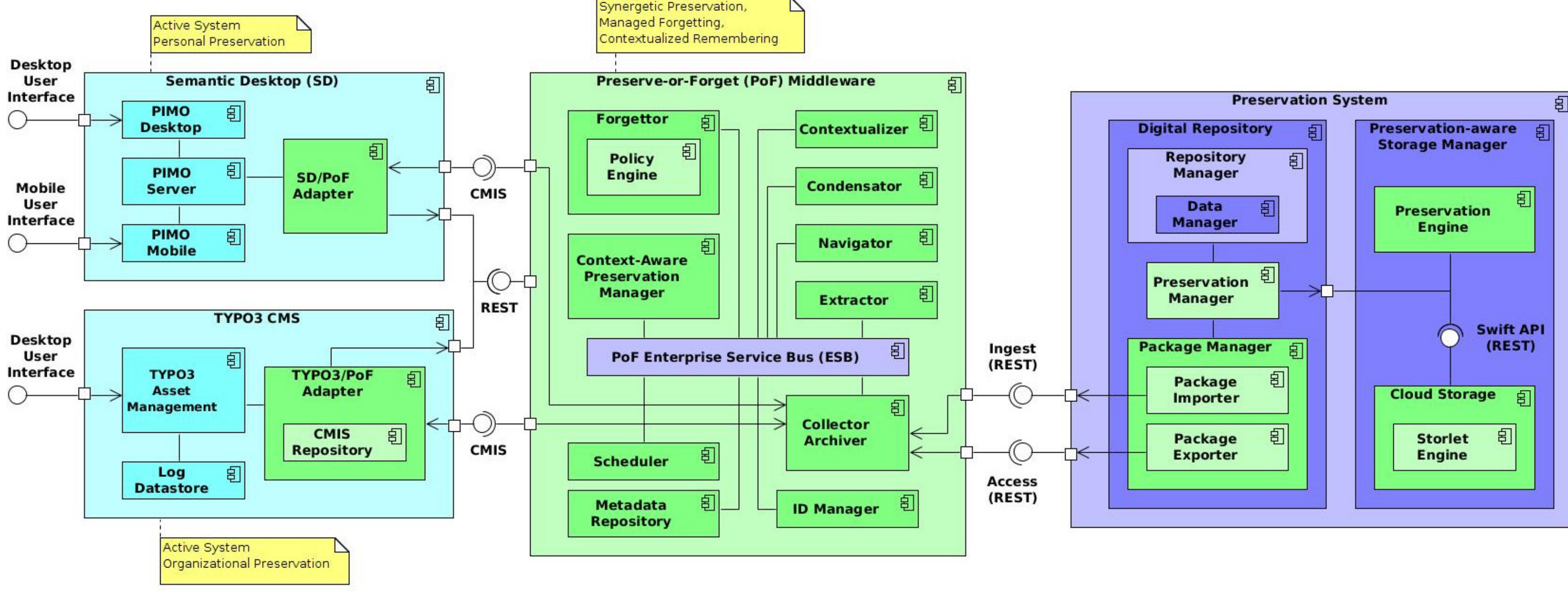

Figure 1: PoF Framework component diagram: the composite structure with internal components is shown.

3 PoF Reference Model

The PoF Reference Model serves 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. The forgetful, focused approach to digital preservation makes easier the adoption of preservation technology in the personal and organizational context. In the following we summarize the main concepts for the functional part of the model (see deliverable D8.2 [ForgetIT, 2015g]), relevant for the prototype description.

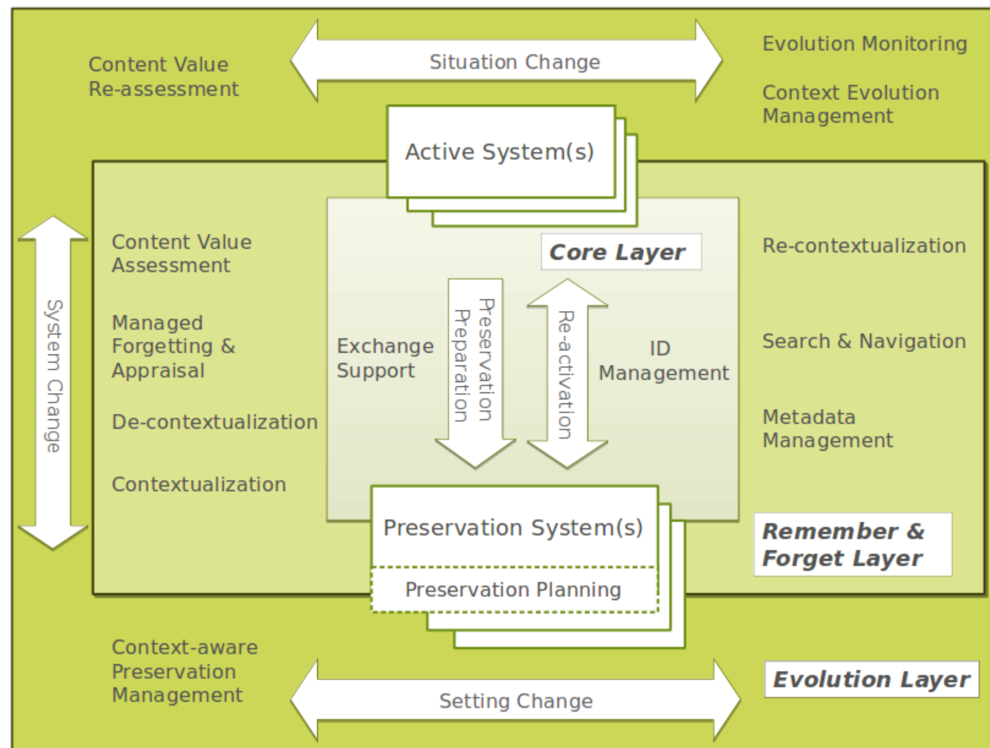

Figure 2: High-level functional view of the PoF Reference Model (from D8.2).

A representation of the functional part of the model is depicted in Figure 2. 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.

Among all workflows in D8.2, we mention here the Preservation Preparation and the Re-activation workflow: the former (Figure 3) is responsible for transferring content to be preserved from the Active System to the Preservation System, the latter (Figure 5) enables the Active System to retrieve and re-activate content previously transferred to the Preservation System. The different steps and functional entities are shown.

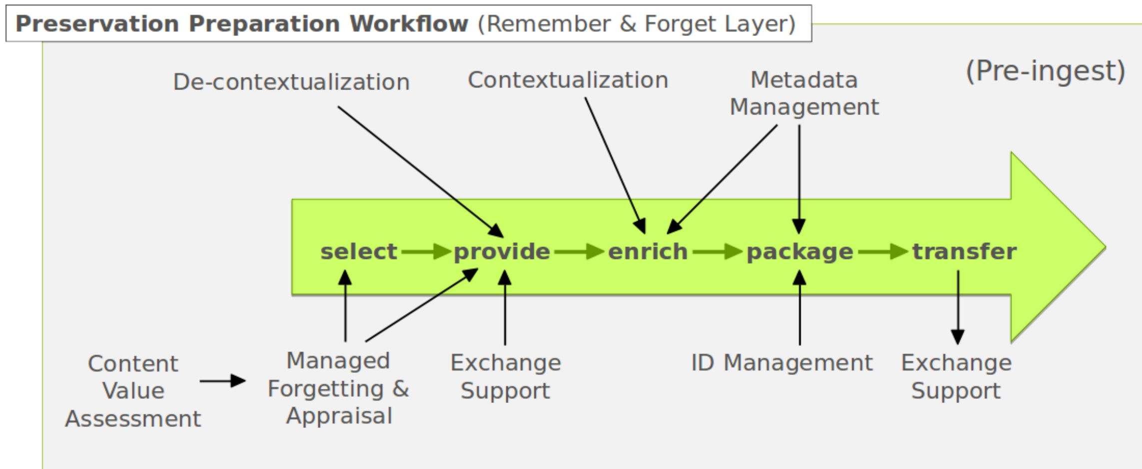

Figure 3: Preservation Preparation workflow in the Remember & Forget Layer (from D8.2).

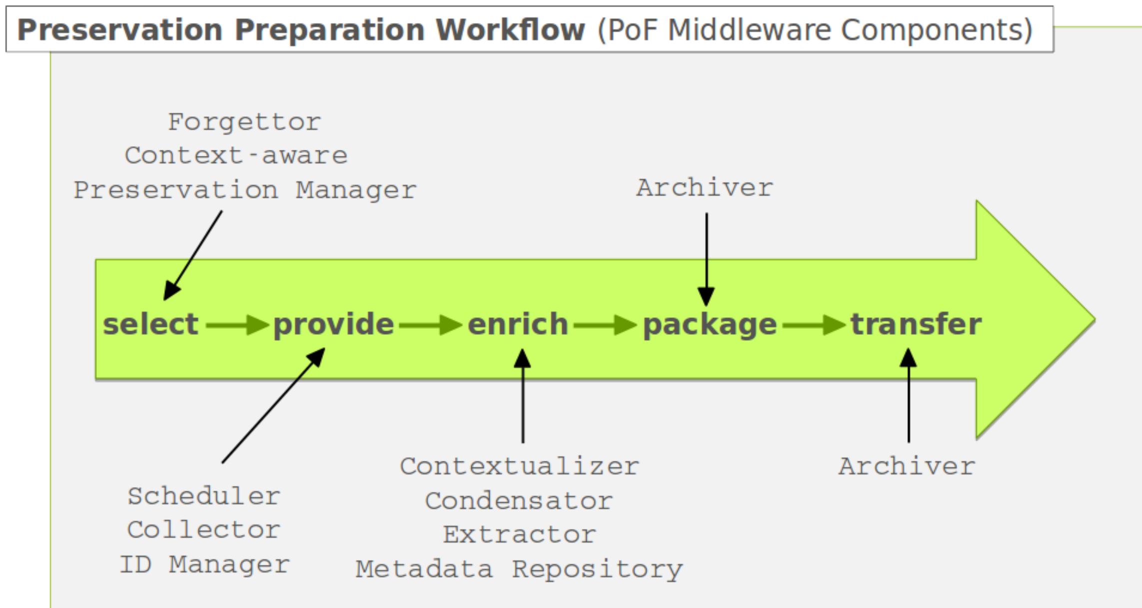

Figure 4: Mapping between the PoF Middleware components and the Preservation Preparation workflow (from D8.2).

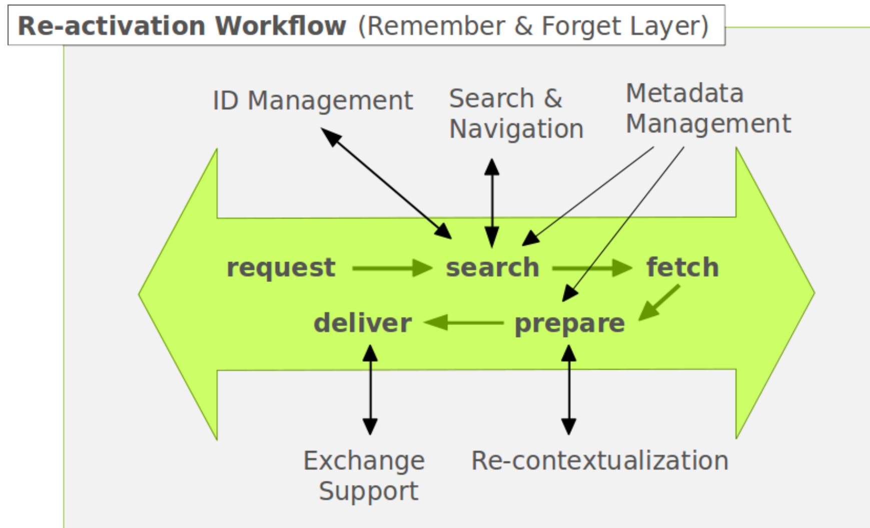

Figure 5: Re-activation workflow in the Remember & Forget Layer (from D8.2).

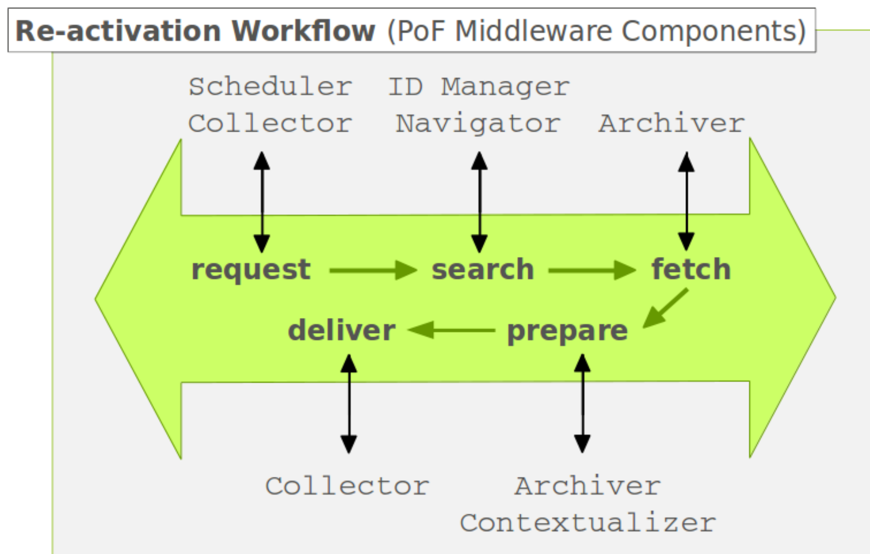

Figure 6: Mapping between the PoF Middleware Components and the Re-activation workflow (from D8.2).

Figure 4 and Figure 6 show how the PoF Middleware components are mapped to the two aforementioned workflows. The other framework components outside the middleware, such as the Active System and the Preservation System (including its internal components) are associated to the functional model workflows, mainly for the Evolution Layer, see Section 4.3 of deliverable D8.2 for further examples.

The mapping between the PoF Reference Model functional entities and the PoF Middleware components is reported in Table 1. It is worth noting that more than one component can participate in the implementation of a given functional entity of the model.

The `Scheduler` component is not explicitly mentioned in Table 1, since it mainly provides process management functionalities across the three model layers.

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`
Contextualization	*Remember & Forget*	`Contextualizer`, `Extractor`, `Condensator`
De-contextualization	*Remember & Forget*	`Contextualizer`
Re-contextualization	*Remember & Forget*	`Contextualizer`, `Archiver`
Search & Navigation	*Remember & Forget*	`Navigator`
Context Evolution Management	*Evolution*	`Context-aware Preservation Manager`, `Contextualizer`
Context-aware Preservation Management	*Evolution*	`Context-aware Preservation Manager`
Preservation Planning	*Evolution*	`Context-aware Preservation Manager`
Administration	*Evolution*	`Context-aware Preservation Manager`
Pre-ingest	*Evolution*	`Archiver`

Table 1: Mapping between PoF Reference Model functional entities and the PoF Middleware components (from D8.2).

The other model workflows which are not shown here will be implemented in the final framework release.

4 PoF Middleware

In this Section we describe the implementation of the PoF Middleware and the integration with Active Systems and 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 [ForgetIT, 2013c], D5.2 [ForgetIT, 2014c] and D8.1 [ForgetIT, 2013d]) and from the implementation point of view (see deliverable D8.3 [ForgetIT, 2014e]). 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 7: 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 7 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 senders (Producers or Publishers) know nothing about receivers (Consumers or Subscribers) and

Figure 7: Message based communication, taken from [Chappell, 2004].

receivers know nothing about senders, as depicted in Figure 7. 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 8 depicts an example of message exchange where the Consumer looses the connection to the MOM but the message does not get lost.

Figure 8: 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 9. 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.

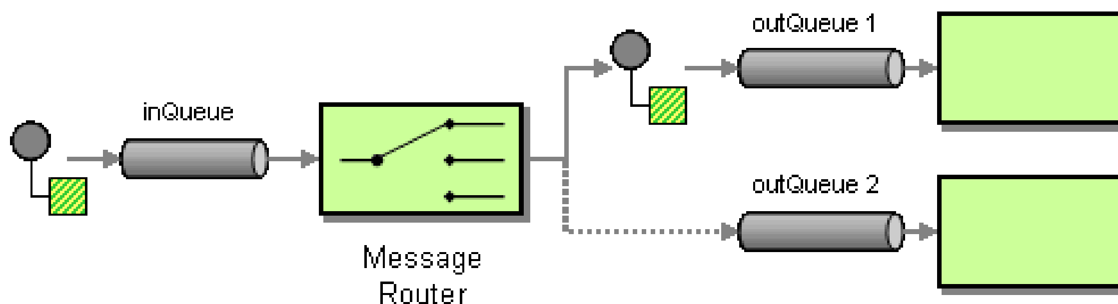

Figure 9: Message Router Enterprise Pattern.

Another pattern example, which was frequently used in the PoF Middleware implementation is the `Service Activator`, depicted in Figure 10. 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 invokation thorugh 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 be 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).

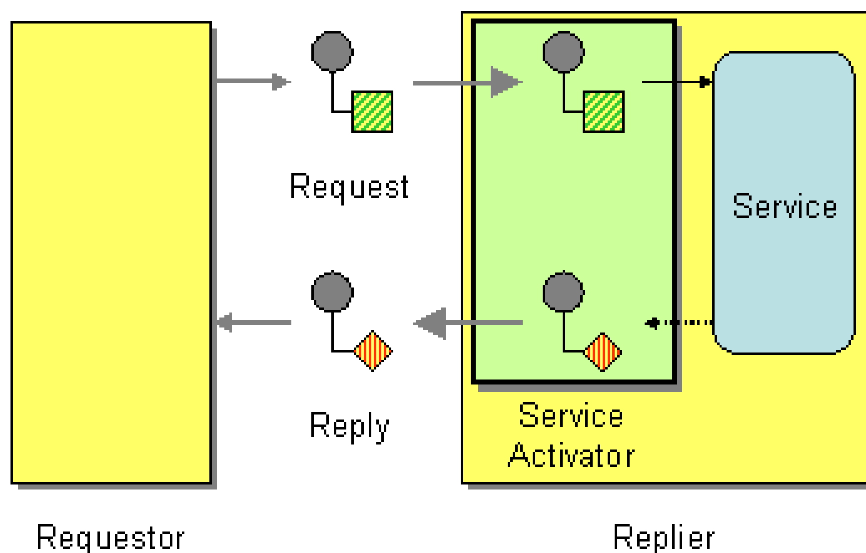

Figure 10: Service Activator Enterprise Pattern.

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

Some basic patterns have ben used very often in the middleware. Such patterns include for example `Request-Reply`, `Aggregator` or `Message Filter`, just to name 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 Apache ServiceMix, in particular we use 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.

For the second release we also improved the monitoring interface for the messaging system and the routing engine, using hawtio[2], 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 11. Additional screenshots of the hawtio console for the middleware instance running in the testbed are shown in Appendix A.

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

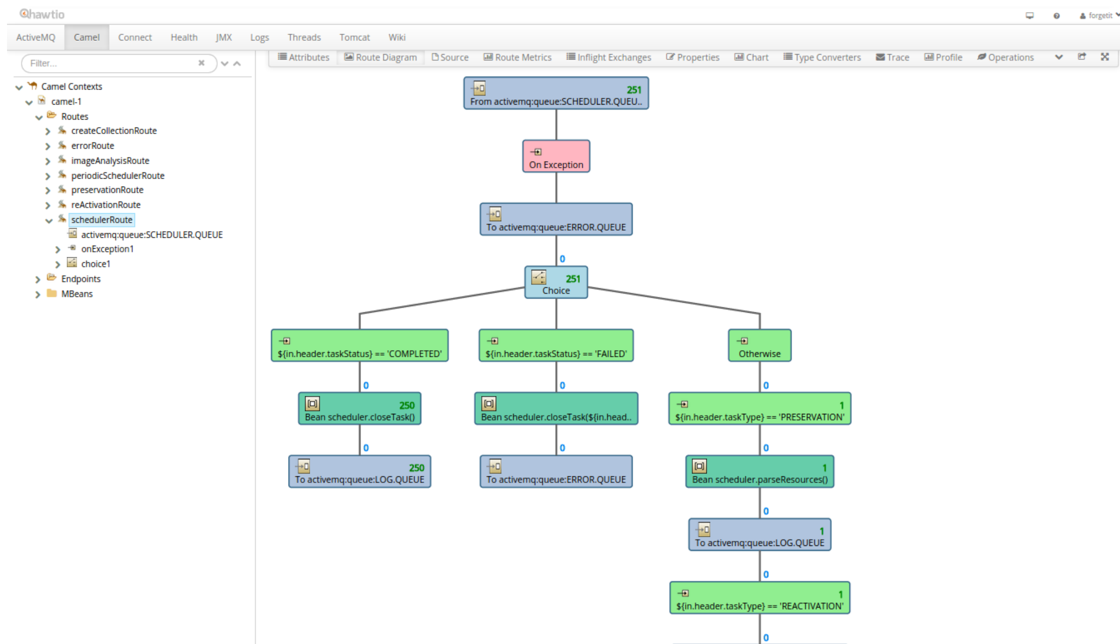

Figure 11: Message flow monitoring.

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 at start time 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[3], the reference implementation of JAX-RS specification for RESTful web services. In the following we list the available APIs with the expected parameters and the output format.

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

Table 2: Server information

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

Table 3: Server APIs List

POST	**/resource**	Triggers Preservation Preparation Workflow of single items or collections. Requires PV, CMIS Repository ID and CMIS Object ID.

Table 4: 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 5: Re-activation APIs

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

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

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 6: 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 7: Task Monitoring APIs

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 12). 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 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.

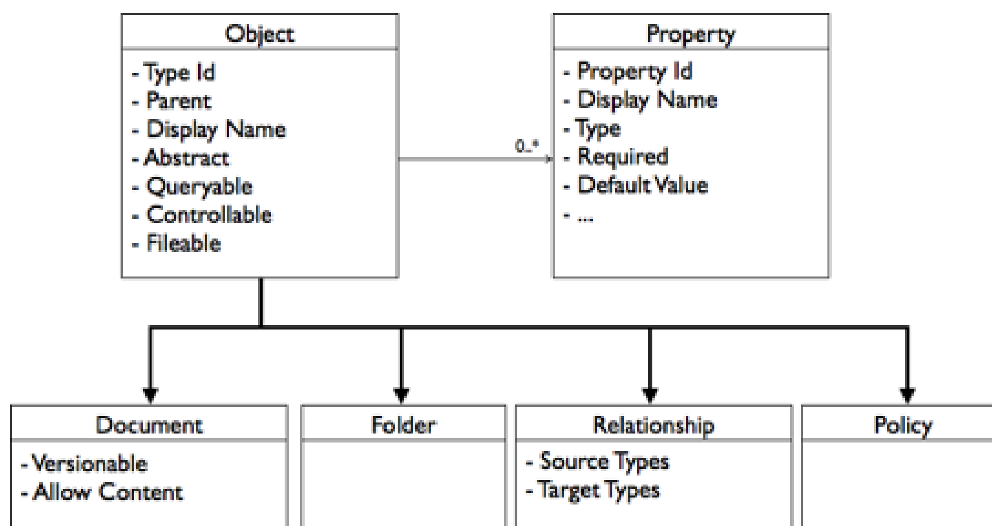

Figure 12: CMIS data model, taken from Alfresco web site.

Each CMIS Object (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 second framework release. For the Semantic Desktop the CMIS repository makes use of the OpenCMIS (Apache Chemistry) library, while TYPO3 CMS uses Alfresco.

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 foreach all cmis:relations. Relying on this CMIS standard enables the PoF to communicate with any component that supports CMIS, which is flexible and doesnt require any special ForgetIT implementations of the Active System to exchange data. Each document should contain an information about what type of element the object is exactly.

The information provided using CMIS representation includes also the 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 will be implemented for TYPO3 in the third release.

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. This scenario is under investigation for the final release.

4.5 Implementation of Reference Model Workflows

In this Section we describe the implementation of the two workflows defined in the PoF Reference Model which have been implemented in the second 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.

4.5.1 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 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 13, 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.

1. A preservation request is triggered by the Active System (see Figure 14): the CMIS ID of the selected resource is sent to the middleware REST endpoint (see Table 4). The CMIS resource can be a collection or a single item. The PV for the whole

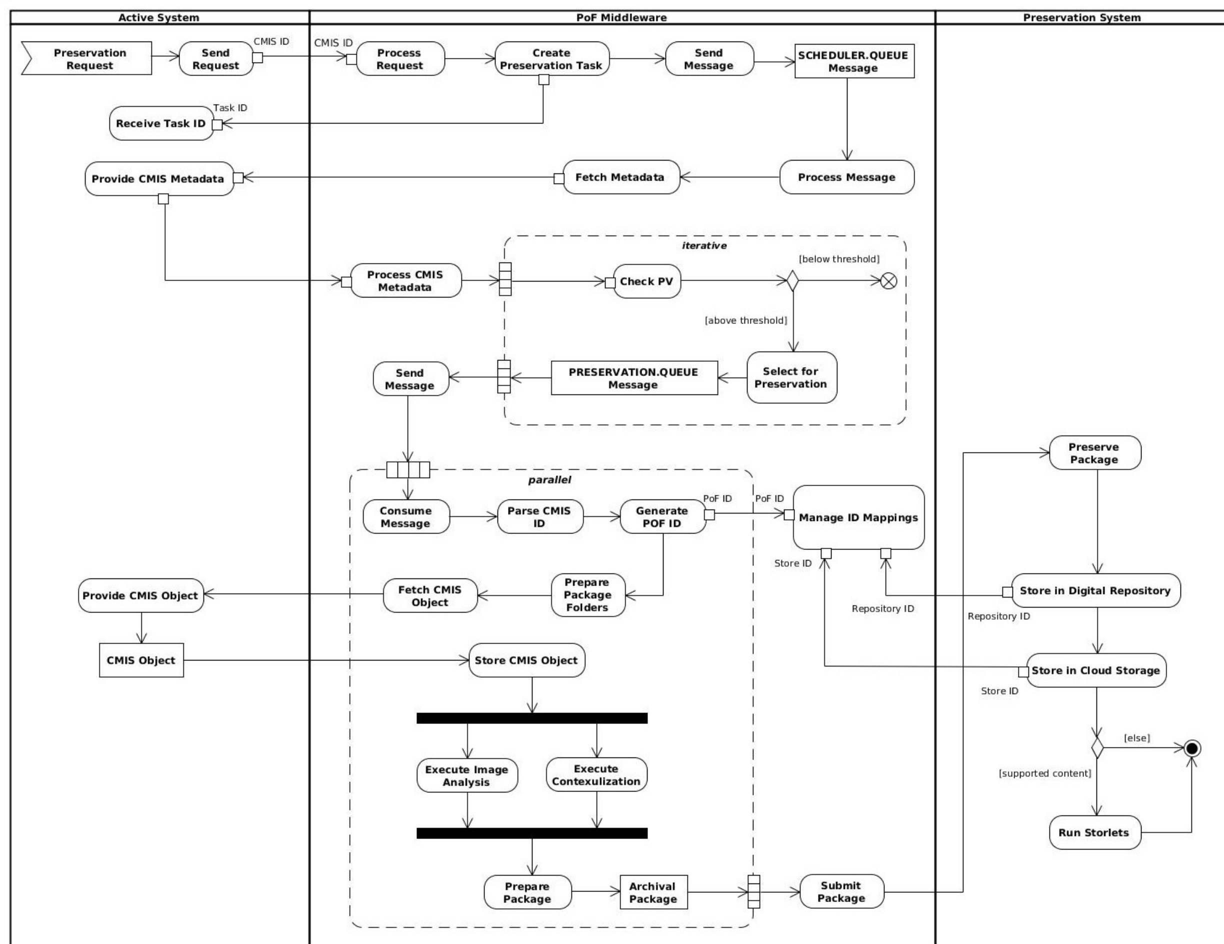

Figure 13: UML activity diagram for the Preservation Preparation workflow.

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 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

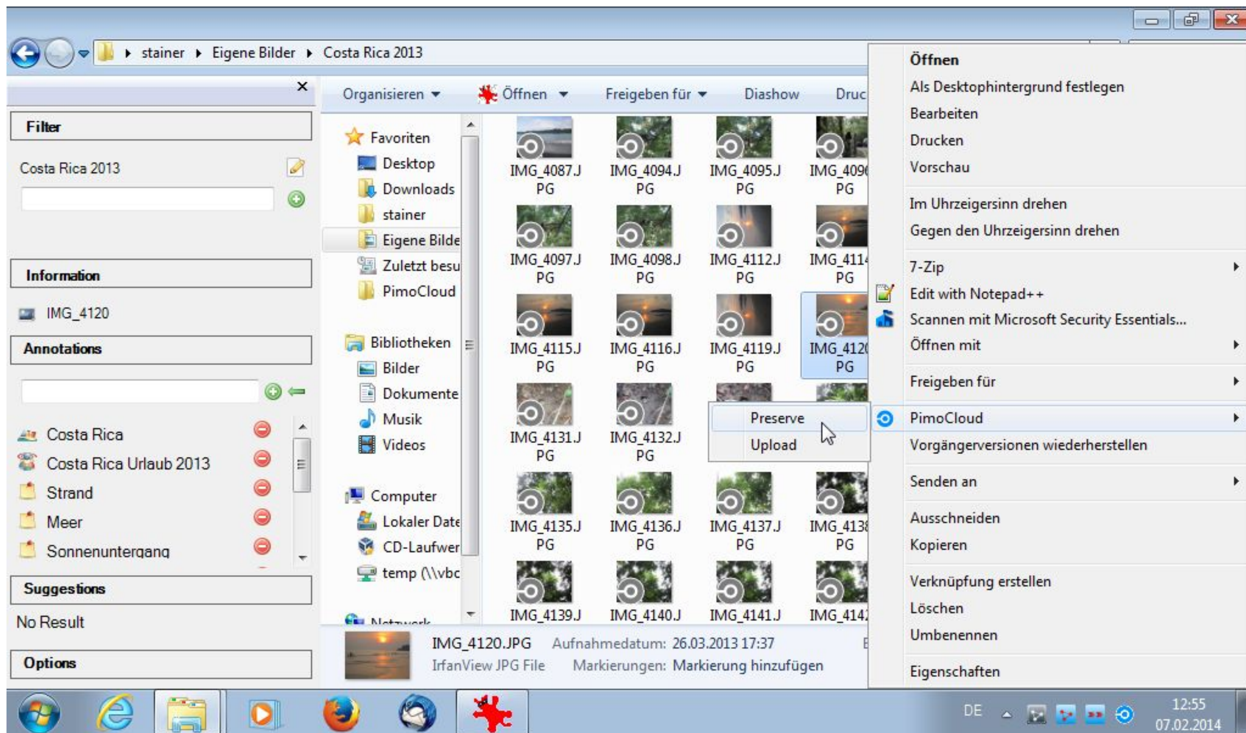

Figure 14: User interface of PIMO: selection of resource to be preserved.

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. Currently for the PV 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. 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.

6. 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.

7. 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.

8. After each resource is assigned a new unique ID, the Collector can fetch the files from the Active System CMIS repository and store them on 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 sub-folder 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.

9. 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.

10. 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).

11. 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.

12. 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.

13. 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 15: the resources and the associated metadata are shown.

DSpace Repository

Login

DSpace Home → ForgetIT Community → Testbed Collection → View Item

Show simple item record

dc.contributor	pimo	en_US
dc.contributor	Peter Stainer	en_US
dc.date.accessioned	2015-04-27T17:02:44Z	
dc.date.available	2015-04-27T17:02:44Z	
dc.date.created	Fri Apr 24 10:09:31 CEST 2015	en_US
dc.identifier	pimo:1429173727300:60	en_US
dc.identifier.uri	http://preservation-system:8080/xmlui/handle/600826/203	
dc.description.abstract	IMG_3299.jpg	en_US
dc.subject	none	en_US
dc.subject	none	en_US
dc.title	IMG_3299	en_US
dc.title.alternative	IMG_3299.jpg	en_US

Files in this item

Name: content/IMG_3299.jpg
Size: 1.368Mb
Format: JPEG image
View/Open

This item appears in the following Collection(s)

- Testbed Collection

Search DSpace

Go

Search DSpace
This Collection
Advanced Search

Browse

All of DSpace
Communities & Collections
By Issue Date
Authors
Titles
Subjects
This Collection
By Issue Date
Authors
Titles
Subjects

My Account

Login
Register

Figure 15: Preview of archived resource in DSpace.

14. 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 16).

15. 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).

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

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 16: 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.

4.5.2 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.[4]

[4] 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`

5 PoF Middleware Integrated Components

Compared to the first prototype described in deliverable D8.3 [ForgetIT, 2014e], the second 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. We also summarize the progress with respect to the first PoF Framework release and the workplan for the third release.

An evaluation of possible licensing mechanisms for the PoF Framework source code has been started in the ForgetIT consortium. A preliminary plan for 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 D8.3 [ForgetIT, 2014e], 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 Section 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 [obj, 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 up-

date 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 second 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. For the final release, the only foreseen improvement is associated to a refinement of mapping and better representation of the collections. The ID Manager code will be updated if a new version of the Java JDK will be used or if a new release of ObjectDB is available.

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 [obj, 2015].

License The component is released under Open Source license, the same used for the PoF Middleware, see Section 8. According to their website, the ObjectDB software is available under Open Source license 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 license.

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, 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 is developed within WP8. It was not 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 done 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.

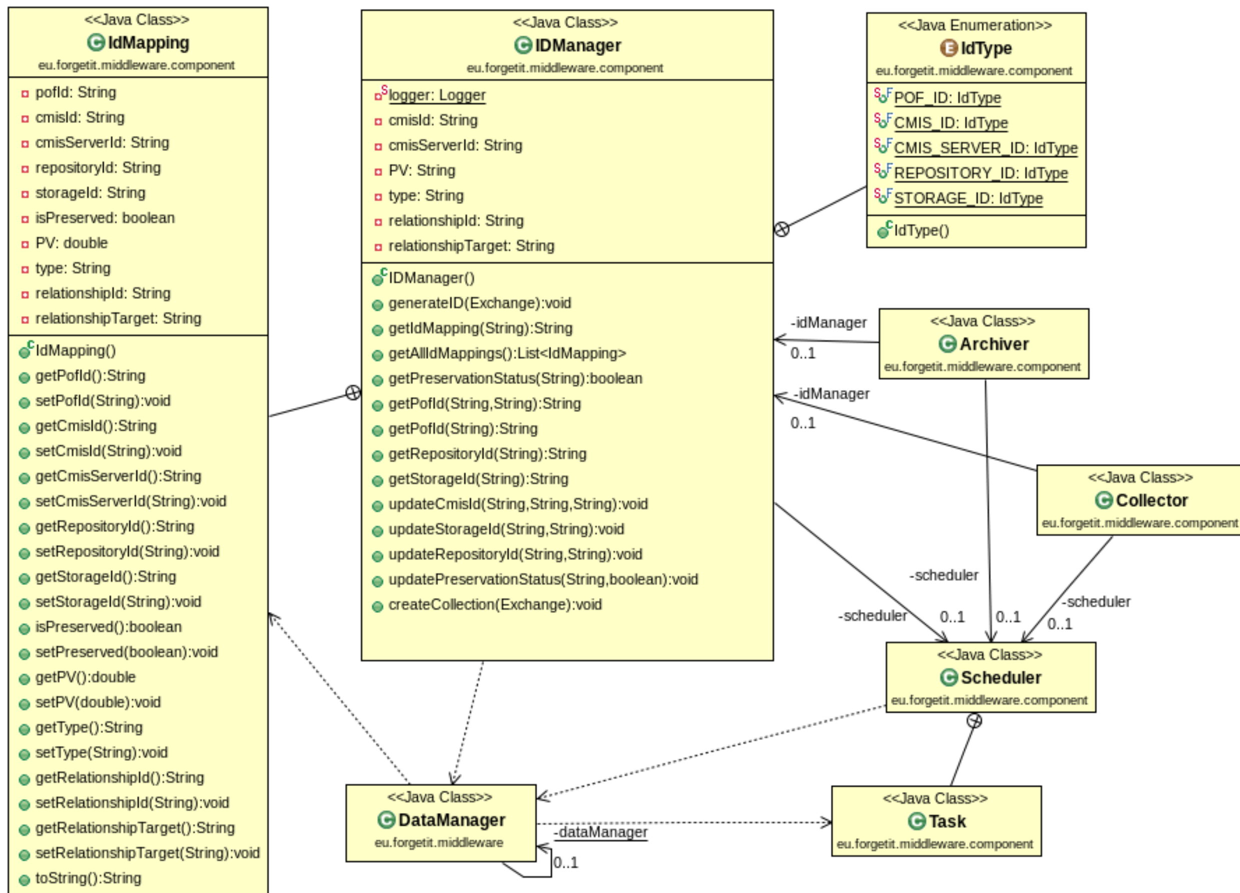

Figure 17: Class diagram for ID Manager component, with associated classes. Association with the Scheduler is related to process scheduling when creating new IDs.

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. This will be improved for the final release, in order to use a shared repository for all components and avoid replication of information.

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.

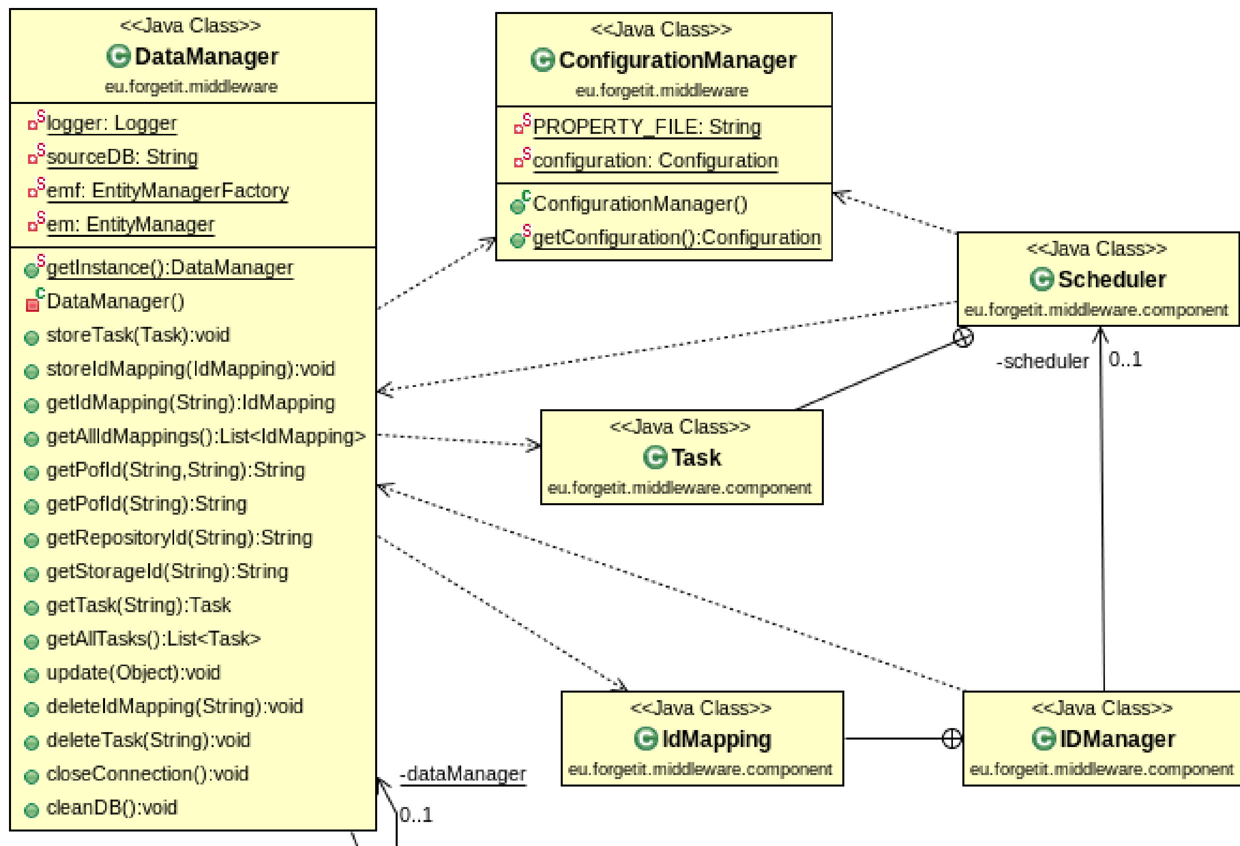

Figure 18: Class diagram for Data Manager: methods based on Java Persistence API are used for CRUD operations on the object DB. The Data Manager is part of the ID Manager and Metadata Repository implementation, but is used also by the Scheduler.

Status and Workplan The current solution for the Metadata Repository provides all expected functionalities. No major improvements are expected for the final release, the only planned activity is the adoption of a single DB for all middleware components, to avoid duplication.

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 [obj, 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 under Open Source license 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 license.

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 version was described in deliverable D8.3 [ForgetIT, 2014e].

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 previous version a `WorkflowManager` class was used to bridge the gap between the web server and the messaging layer, preserving loose coupling, but in the new 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), but all the other workflows defined in the model will be supported in the final release. 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. 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 can be dynamically configured using Spring XML. No major updates are expected for the third release for what concerns the Scheduler code. Possible improvements are expected in defining scheduled activities using Spring XML and Apache Camel and in Task management. 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, but for the final release this mechanism will be used to trigger specific preservation tasks.

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

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

5.4 Extractor

Component Role The Extractor takes as input the original media items (e.g. a text, an image collection, a collection of texts or a collection of images) 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 image analysis methods, the current release of the Extractor implements: concept detection, near duplicate detection, image quality assessment and face detection. Text analysis methods cover basic linguistic processing, which fed into the extraction of named entities and plays a role in the Condensator.

WP and Deliverables The Extractor is developed within WP4, the contributing partners

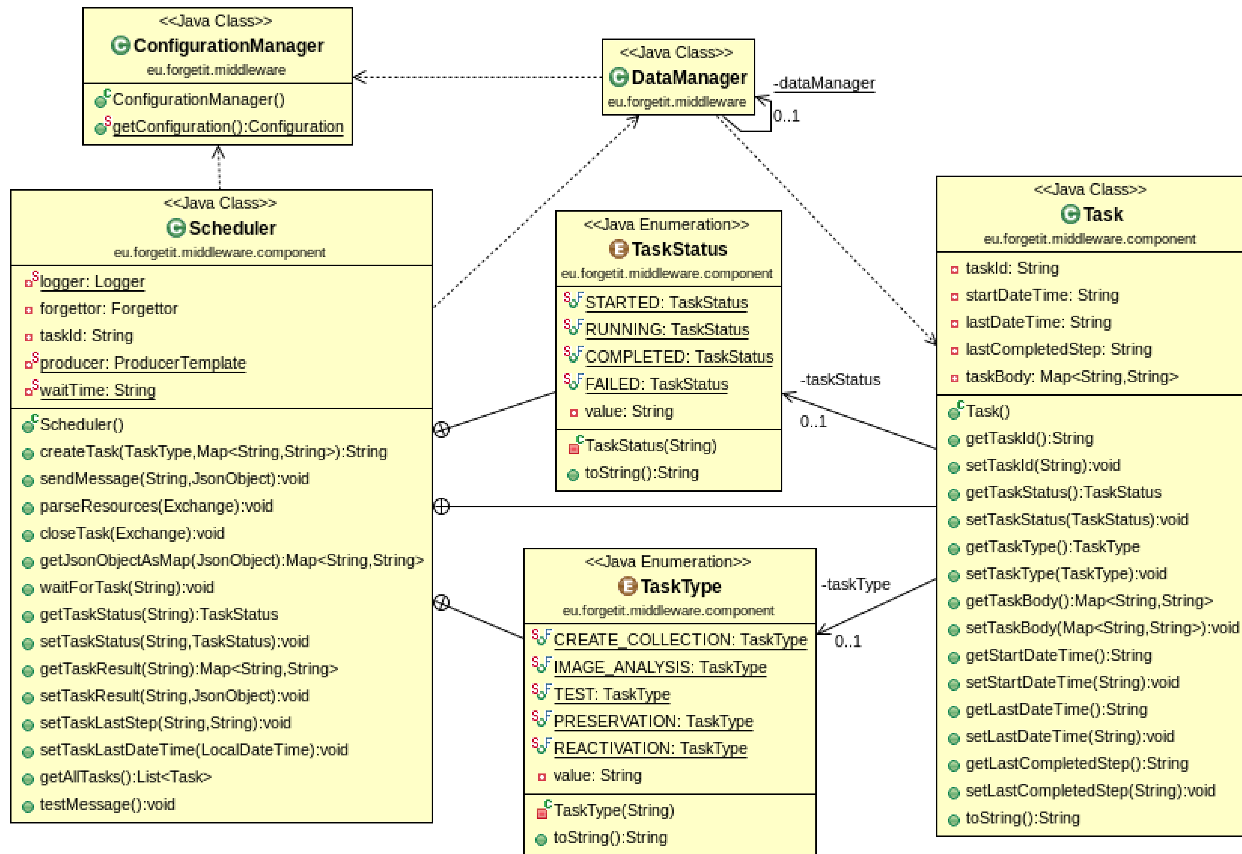

Figure 19: Class diagram for Scheduler component, with associated classes.

are CERTH, USFD, TT. The different technologies that are required for realizing the Extractor were reviewed in deliverable D4.1 [ForgetIT, 2013b]. Text analysis tools, as well as GATE [Cunningham et al., 2011], and image analysis tools (including an updated version of concept detection, face detection, and near duplicate detection) are presented in D4.3 [ForgetIT, 2015d]. Image quality assessment is presented in D4.2 [ForgetIT, 2015d].

Integration and Deployment All image analysis Extractor sub-components have been deployed as REST services running in CERTH servers. The text extraction components have been developed as GATE [gat,] 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 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 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 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 analysis method is supported by the use of a Java enumeration, `MethodType`. As shown in Figure 20, the progress of each image 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 D4.2 and D4.3. 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 concept detection 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 and Workplan In the first framework release, the Extractor contained two image analysis sub-components, for concept detection and image quality assessment. The current version has an updated concept detection method and two new methods: near duplicate image detection of an image collection and face detection. Furthermore, all implementations are written in C++ and are much faster than the previous ones. 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. Future releases will contain updated versions of the above sub-components as well as new components. Some new media analysis methods will be: face clustering, online concept training, video analysis, quality assessment, etc. It is also envisaged that a number of new text extraction applications will be requested via the use cases and will be integrated as required.

Documentation and reference links Additional information about the Extractor component and the RESTful web service hosted by CERTH can be found in deliverables D4.2 [ForgetIT, 2014b] and D4.3 [ForgetIT, 2015d], which also provide some usage examples.

License CERTH libraries are Copyright ©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 license) and Liblinear (Copyright ©2007-2015 the LIBLINEAR Project). GATE (and associated software) [gat,] is available under an open source license, mostly GNU LGPL v3, although some code is covered by the GNU AGPL.

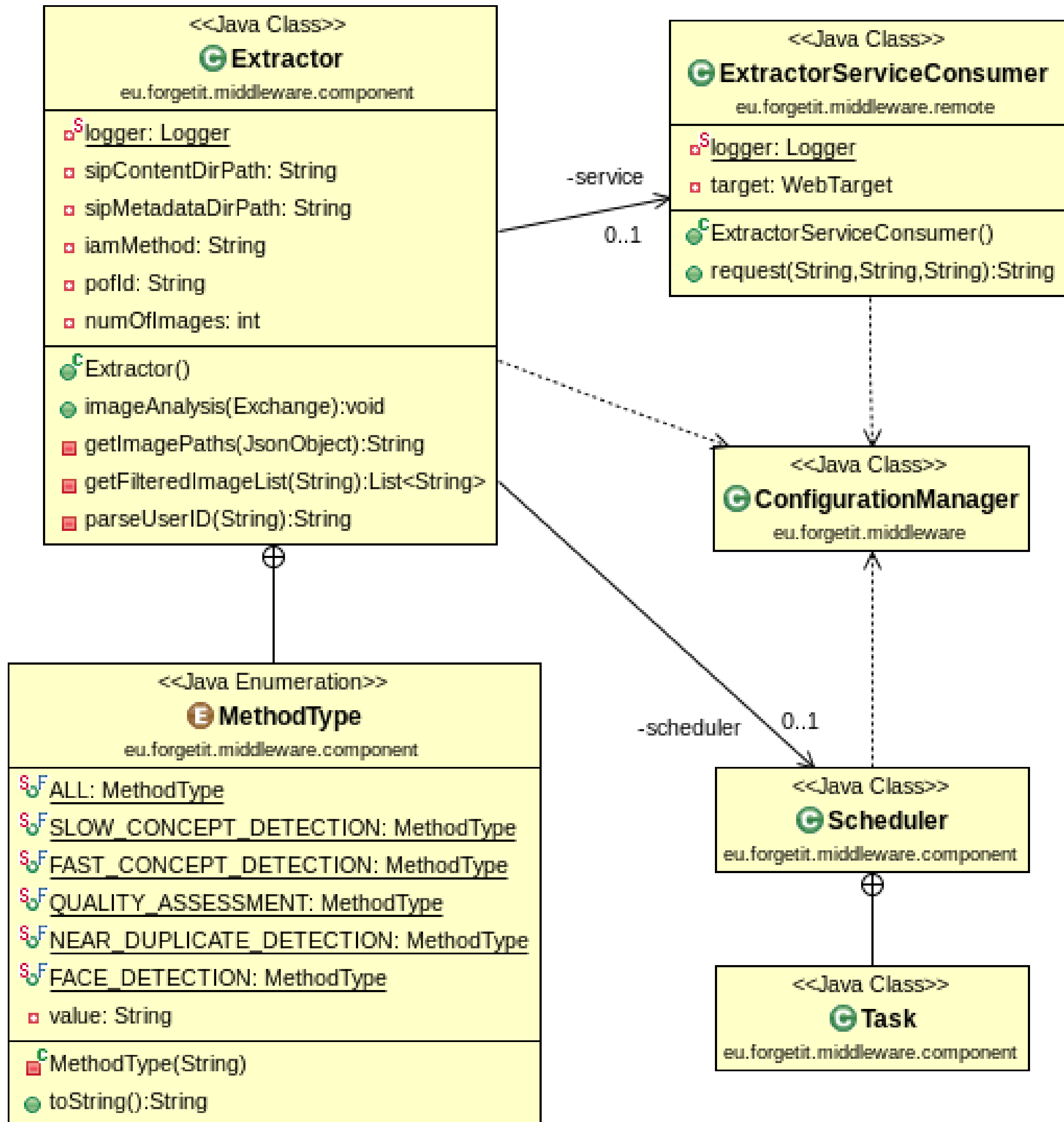

Figure 20: Class diagram for Extractor, with associated classes.

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 and image 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 image media, the current release contains a clustering method that is able, based on image features and image metadata (captured time and GPS location) to cluster the images into separate events. For what concerns text, the Condensator provides linguistic simplification and statistically-based single 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 [ForgetIT, 2013b]. The text summarization methods and the image clustering methods are presented in D4.3 [ForgetIT, 2015d].

Integration and Deployment The image 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 [gat,] 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 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 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 D4.2 and D4.3. The response of the web server is returned in XML format, as done for the Extractor.

Status and Workplan In the first framework release, the Condensator contained an image clustering method that was using only image visual features. The current 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 text components are integrated via GATE WASP (see D4.3), allowing for an infinite variety of applications to be made available via the Condensator and integrated within the use case tools. Future releases will contain updated versions of the sub-components above. In the case of media contextualization, the method will be updated by employing more cues (e.g. face clustering results) and it will be extended including not only image collection but also video collection summarization. It is envisaged that a number of new text extraction applications will be requested via the use cases and will be integrated as required. For example, it is already planned to include multi-document summarization as this is required in one of the WP9 use case scenarios.

Documentation and reference links Additional information about the Condensator component and the RESTful web service hosted by CERTH can be found in deliverables D4.2 [ForgetIT, 2014b] and D4.3 [ForgetIT, 2015d], which also provide some usage examples.

License CERTH libraries are Copyright ©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 license) and Liblinear (Copyright ©2007-2015 the LIBLINEAR Project). GATE (and associated software) [gat,] 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 and makes use of standardized metadata schemas adapted to receiving ingest functional entity in the Preservation System. The Collector/Archiver creates the acrshortsip structure based on the package structure defined in WP5. 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 checksums. The Collector/Archiver is also in use in the Re-activation workflow where digital content is brought back from

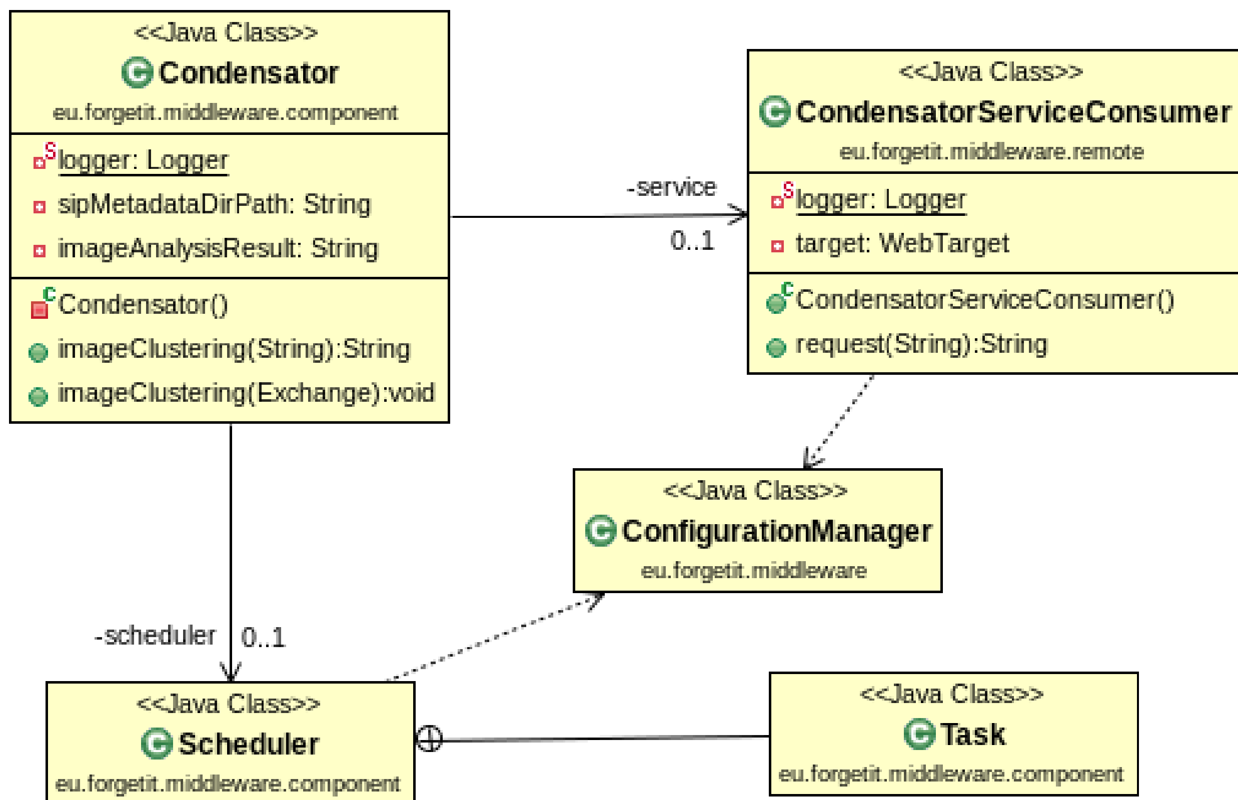

Figure 21: Class diagram for Condensator, with associated classes.

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. 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 [ForgetIT, 2014c]. The second version, integrated in the second prototype, is described in deliverable D5.3 [ForgetIT, 2015e].

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/Archive 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 `CollectorSer-`

viceConsumer 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 CloudStorageService-Consumer 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 D5.3.

Status and Workplan The second version of the Collector/Archiver has been improved with additional features: extended support for 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[5], support for management of physical and logical content structure from CMIS repository on the Active System, improved integration in the middleware with the implementation of REST interfaces for fetching, packaging and re-activation features. The Collector/Archiver is still under development, the current prototype supports the Preservation Preparation and Re-activation workflows in the PoF Middleware by REST interface. The work plan for the third framework release includes: implementation of process logging to support functional validation and workflow redirection at errors and exceptions (using alternative workflows) , integration with the Context-aware Preservation Manager (CaPM) component (see Section 5.10), restructuring of internal component architecture for increased reuse and integration capabilities.

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 prototype is available in deliverable D5.3 [ForgetIT, 2015e].

License The code of the Collector/Archiver is released by LTU as open source. The source code is currently available in the ForgetIT SVN repository.

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

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

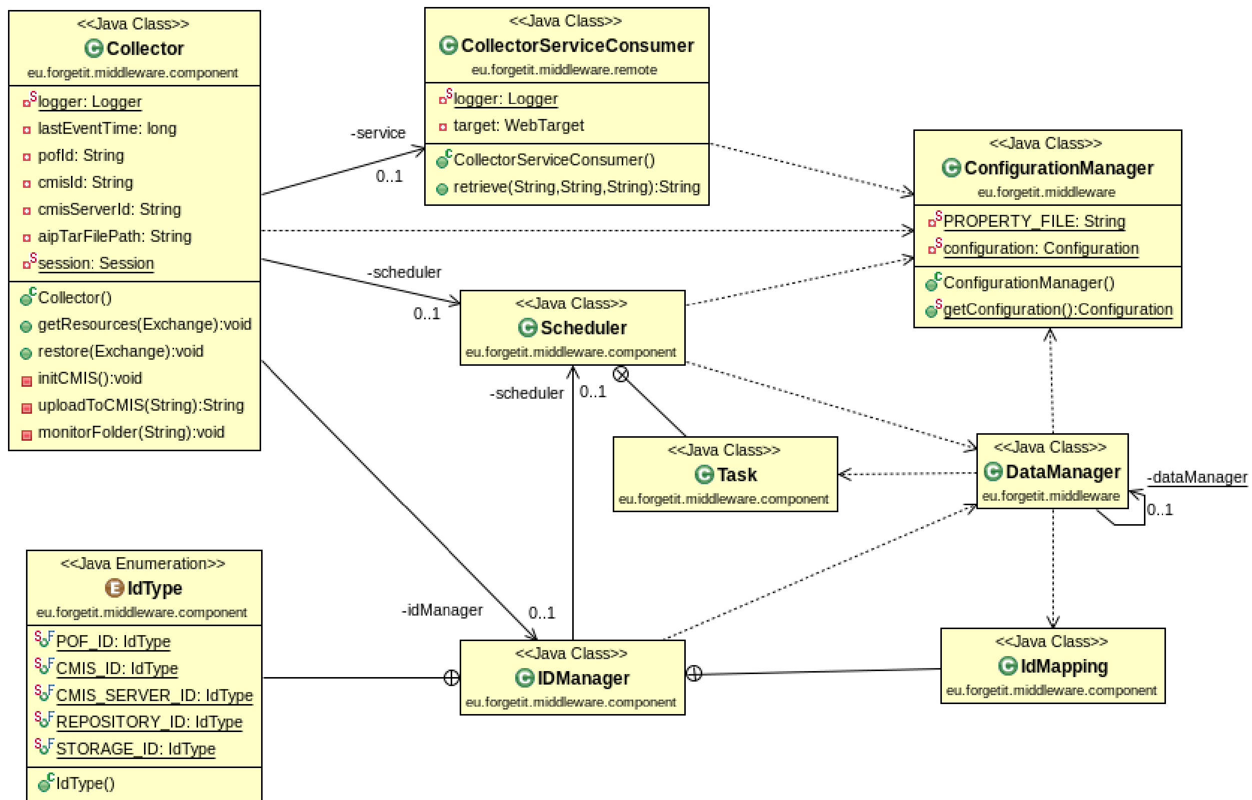

Figure 22: Class diagram for the Collector, with associated classes.

different methods in the managed forgetting framework, and provides outputs about the two values: Memory Buoyancy (MB) and the Preservation Value (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, 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 [ForgetIT, 2013a]. The first prototype of the MB assessor is described in deliverable D3.2 [ForgetIT, 2014a], and is followed up by a case study for the evaluation in deliverable D3.3 [ForgetIT, 2015c]. Deliverable D3.3 [ForgetIT, 2015c] discusses the Policy Engine design and implementation. It also describes the first prototype of the PV assessor that focuses on the photo preservation scenario.

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

Integration and Deployment The three sub-components are developed separately within WP3, and the comprehensive integration into the middleware is yet on plan for the Year 3. The MB assessor interfaces with the Active System via web service (see Appendix B), as well as standalone Java packages that are deployed in the PIMO system, so as to preserve the privacy of resources in the experiment phase. The PV assessor prototype for computation of photo PV takes one collection of photos as input and returns the PV for each photo in the collection. The method implemented by this sub-component has been described in detail in D3.3 [ForgetIT, 2015c]. It has also been exposed as web service to be used within Scenario 1 (see Appendix B), although it still has to be integrated in the ForgetIT middleware.

Policy Engine Integration: The Policy Engine is interfaced with web-based applications. There are two applications deployed:

- *Basic Policy Engine:* Targeting normal users who are the owners of the the digital documents to be preserved. It supports basic policy customization via question answering, exploring the information values of digital documents from the active systems (TYPO3, PIMO)[6]
- *Advanced Policy Engine:* Targeting advanced users who are the rule experts[7]. It supports management user interface, rule editor to specify rules (in machine-readable format .drl [ForgetIT, 2015c]).

[6] Basic Policy Engine - `http://forgetit.l3s.uni-hannover.de:8787/policy-basic`
[7] Advanced Policy Engine - `http://forgetit.l3s.uni-hannover.de:8787/policy-advanced`

Domain experts use the advanced interface to create and manage policies for different scenarios. Once finished, the policies are bundled as a Java package, compatible with Maven. Indeed, there is currently a local Maven repository used to maintain the policies in the Forgettor component. The normal users run the basic interface, which connects itself to the Maven repository and the database of digital documents of the Active Systems, and explore the information values as after applying the policies. User feedback towards the values are stored back in the database, which will be used later for adapting the Forgettor in the next rounds.

API and I/O Formats This is in progress and will be reported in the coming deliverables. The PV assessor for photo scenarios are described in Appendix B. For the MB assessor, the experimental APIs are described in Appendix C.

Status and Workplan The Forgettor component is currently under development according to the plan in deliverable D8.1 [ForgetIT, 2013d]. 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 backends 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 Assessor:

The MB assessor is responsible for estimating the MB values of a resource (Section 5.2, D3.1) in personal preservation contexts. The sub-component relies on the activity history of users in the information space (his Semantic Desktop), as well as the ontological knowledge of the resource, including its structures and its relationships with other resources. In the middleware layer, the MB Component is used as a service by the clients (PIMO or TYPO3 systems) to numerically assess a resource.

The MB assessor has 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, ...).

Documentation and reference links Additional information about the component can be found in deliverable D3.2 Section 2.2-2.3 [ForgetIT, 2014a].

License: The policy engine and the advanced interface is developed using JBoss Drools Business Rule Management system[8] under the license ASL 2. The basic interface is developed using Google Web Toolkit[9] under the Apache license 2.0. The other components are available under GNU License GPL v3.0, Creative Commons License 3 and Apache License 2.0.

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 [ForgetIT, 2015f] 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 based on GATE [gat,], integrated into the PoF Middleware. The class diagram for the Contextualizer is shown in Figure 24, where the associated `ContextualizerServiceConsumer` 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 D6.4 [ForgetIT, 2016a] by the end of the project.

Documentation and Reference Links Additional information about the Contextualizer is available in deliverable D6.3 [ForgetIT, 2015f]. A demo version of one of the text based approaches to contextualization can also be accessed on GATE services web site[10].

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

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

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

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 license) and Liblinear (Copyright c 2007-2015 the LIBLINEAR Project). GATE (and associated software) is available under an open source license; mostly GNU LGPL v3, although some code is covered by the GNU AGPL.

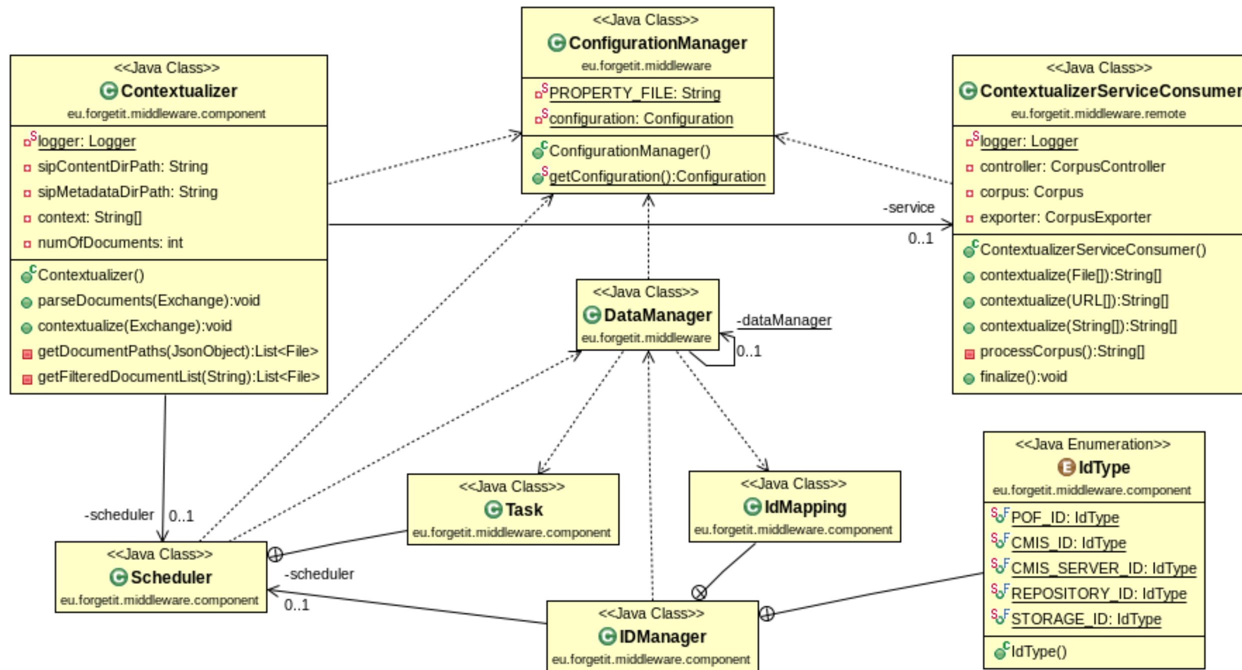

Figure 24: Class diagram for Contextualizer, with associated classes.

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.

WP and Deliverables The Navigator is a product of WP6 and WP8. As the current version is just a test bed it has not yet been documented in a deliverable, but it will be reported in D6.4 [ForgetIT, 2016a] and D8.6 [ForgetIT, 2016b] both of which are due by the end of the project.

Integration and Deployment The current version of this component is not integrated within the PoF Middleware as it was developed as a test bed for ideas and was never envisaged as a final component. This will clearly change by the time of the final release of the framework and components.

API and I/O Formats The Navigator component is still under development. Currently a web interface has been implemented as part of Scenario 2 (see Appendix B, the integration with the other components is under development

Status and Workplan An initial Navigator component was developed as part of the Scenario 2 demo for the year 2 review. This allowed for some testing of ideas and concepts but it is envisaged that the final component will be substantially different to provide a wider (not scenario orientated) access to the preserved items.

Documentation and Reference Links As the first full version of the component has not yet been developed there is no available documentation. Documentation will appear in D6.4 and D8.6 at the conclusion of the project.

License The license of the component will be determined once it has been developed based upon the final list of contributing partners and the libraries used.

5.10 Context-aware Preservation Manager

Component Role The main objective for the CaPM component is to support the Preservation Preparation and Re-activation workflows in the PoF Middleware solution by increasing the ability for seamless integration between Active System and Preservation System, to monitor change of ontologies by logging of logical and physical structures in use by active systems, to monitor the use of file format for detection and computation of format obsolescence issues, to support establishment of submission agreements that interconnect Active System (information system) submissions with expected content and metadata management procedures, to support matching of content submissions and re-activation purposes to needs of digital preservation services, and to support management of various physical package structures for seamless re-activation of content from Preservation System to information system.

WP and Deliverables This component is developed within WP5, the contributing partners are LTU, EURIX, IBM, DFKI and dkd. Deliverable D5.3 [ForgetIT, 2015e] describes some basic capabilities and placement in the PoF Middleware workflows.

Integration and Deployment Integration and deployment for this component have not been defined in detail yet. This component must be able to communicate with the other middleware components, but also to collect information about content in the Active Systems and in the Preservation System (mainly in the cloud storage component).

API and I/O Formats The actual prototype implementation is not available yet, at the moment of writing.

Status and Workplan The development of this component is still in progress: some preliminary ideas about the implementation have been discussed so far. The work plan for the third framework release is going to include the first implementation of this component with support for basic functionalities integrated in the PoF Middleware. The plan includes

implementation of logging and computation of content statistics, to support the use of submission agreements, and management of various logical and physical package structures in Re-activation workflows. The component will leverage the messaging infrastructure to asynchronously collect information about preservation processes or any other relevant event used for triggering the appropriate processes.

Documentation and Reference Links A preliminary description of the Context-aware Preservation Manager is available in deliverable D8.1 [ForgetIT, 2013d], where the role of the component in the framework is described, and in deliverable D8.2 [ForgetIT, 2015g], 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 [ForgetIT, 2015e].

License The code of the Context-aware Preservation Manager will be released by LTU as open source, the source code will be available in the ForgetIT SVN repository.

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 Pilot I (see deliverable D9.3 [ForgetIT, 2015h]) uses the integration with the PoF Framework to provide services from ForgetIT. To realize this, the following enhancements to the first release were accomplished in the pilot.

API and I/O Formats

Middleware services read contents of the PIMO using a CMIS compliant endpoint. This endpoint has been updated in several respects. Especially, the information model used here for transferring content to the PoF is one proposal for investigation for the future PoF Information Model.

Every concept of PIMO can now be a CMIS document (see Figure 12): thus, every concept can be fetched by the PoF Middleware allowing to perform preservation and restore operations for every single PIMO entity. This allows also to preserve and restore concepts without an explicit downloadable content such as persons or topics (in contrast to resources such as images or text files).

Additional CMIS properties and CMIS relationships represent membership relations between container concepts and therein contained concepts. Therefore, a new CMIS document type has been introduced: `forgetit:collection`. The specialty of the Semantic Desktop as Active System motivated not to choose the available low-level types of the CMIS model, namely `cmis:folder`:

The Semantic Desktop as Active System is based on a semantic graph knowledge representation as data structure (for details see deliverable D9.3 [ForgetIT, 2015h]), there are no folders. Furthermore, nodes of the graph can be connected virtually to anything (e.g., as an *is related*-relationship is allowed between nodes). However, to point out the adherence of resources to a collection, the `forgetit:collection` as CMIS document

type is introduced together with the `forgetit:containedIn` CMIS relationship. The relationship states the containment of a resource in a collection[11].

As now `forgetit:collection` is more generic than `cmis:folder` it allows to circumvent the folder metaphor and implementation to transfer content to PoF.

ForgetIT collections (see Section 4) are now implemented in the CMIS endpoint (see Figure 25). They are realized as a special view on the PIMO model which is computed during request time. This allows to store collection-like things in the PIMO such as projects or photo collections (represented in the PIMO as `pimo:LifeSituations`), as discussed below. Each collection can have contained documents which is represented using `forgetit:containedIn` connecting a container document (source) with a contained document (target).

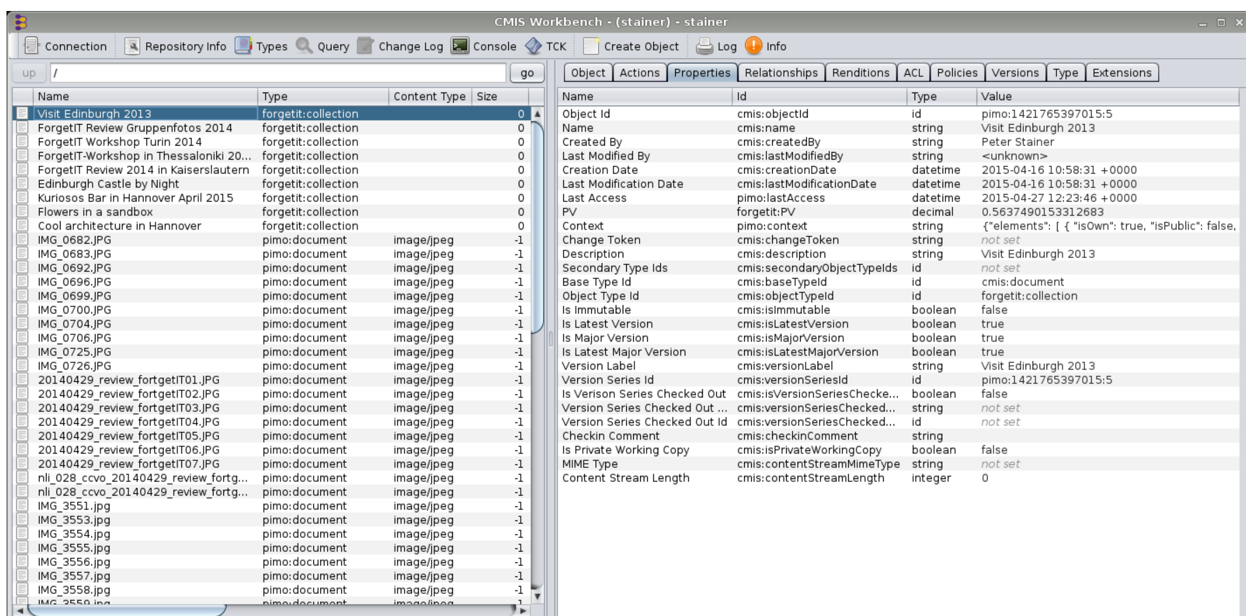

Figure 25: Browsing a photo collection stored in the Semantic Desktop's PIMO as `forgetit:collection` using a CMIS client.

The PVs calculated by the Semantic Desktop on the PIMO for demonstration purposes are provided by adding an additional document property `forgetit:PV` (this property is also used in those scenarios where the PoF Middleware does the PV calculation).

To provide extensive metadata for a resource to be preserved, the context of an entity is stored in the additional document property `forgetit:context`, which provides additional information about the resource to be preserved derived from the PIMO[12].

Additional changes include an improved feedback about running or completed middleware tasks such as succeeded or failed preservation which are reflected in the Semantic Desk-

[11] Using `cmis:relationship` is too general and would be up to an interpretation of the PoF that this could mean contained in a collection and not, e.g., *is annotated with* or similar.

[12] For the example of an image, these would be annotated concepts, its (semantically enriched) text written by the user, the collections it belongs to, etc.

top UI, e.g., as notifications appearing on the main page, or status information if an image is preserved or not. The use of PoF Middleware messaging capabilities avoids extensive polling for status information and leads to better overall performance of the system.

Finally, access to the PoF Middleware, e.g., for triggering restore operations, has been adapted to the changes made in the corresponding REST APIs.

License

Data exchange from the PIMO to the PoF Framework is done solely by means of the standardized CMIS interface, so no proprietary API has to be exposed for interacting with the Semantic Desktop.

pimo:LifeSituation

In the personal preservation use case, a customized CMIS Repository based on OpenCMIS library is provided by the Semantic Desktop.

The PIMO allows to organize photos using a dedicated PIMO Photo App (see deliverable D9.3 [ForgetIT, 2015h]). The Photo App uses the PIMO class `pimo:LifeSituation`[13] to represent such a photo collection. This allows to annotate the collection as a whole as well as individual photos (represented as `pimo:Image`) with with PIMO concepts, write a (semantically enriched) text (using seed from WP4), and ratings of photos.

Each `pimo:Image` additionally contains data derived from the photos' Exchangeable Image File Format (EXIF) data if available (location, date), concepts with confidence value detected by using image quality assessment from WP4, and finally, as a resource the photo itself (as so-called *grounding occurrence*; for instance, this could be a URL pointing to the file on a user's desktop and/or locations in the PIMOCloud).

6.2 TYPO3

The Organizational Preservation Pilot Application V1 (see D10.2 [ForgetIT, 2015a]) 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

[13] For a detailed discussion why it was called life situations (such as marriage, birth, holiday, etc.) and see the prototype PIMORE in deliverable D9.2 [ForgetIT, 2014f].

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 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 will be in deliverable D10.3 [ForgetIT, 2015b]. The first step is the request for possible annotations from an annotation source, at the moment a *YODIE* endpoint 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 application 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 [ForgetIT, 2015h]). 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), 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 [ForgetIT, 2013d], D8.3 [ForgetIT, 2014e] and D7.3 [ForgetIT, 2014d].

7.1 Digital Repository

The Digital Repository is implemented using DSpace platform. For the second framework release we updated the DSpace software to the last stable version 5.2. DSpace source code is available as open source on Sourceforge [dsp, b] and GitHub [dsp, a], the documentation is available on the project web site[14]. A customized installation guide for DSpace was provided in D8.3. In order to enable the interaction between the PoF Middleware and the DSpace repository, REST APIs for both the access and the ingest processes have been implemented. 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.

DSpace internal data model is represented in Figure 26. 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 preliminary definition of the PoF information model in D8.2. Currently, there is no direct mapping between the DSpace collections and the ForgetIT collections described in Section 4 and Section 6, because the work is still in progress (the collection information is stored in the CMIS metadata only).

The last DSpace release provides a implementation of REST APIs[15]. 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 scenario is currently under investigation and will be evaluated and implemented for the third release, in parallel with the development of the cloud storage (see next Section), with additional storlets implementing preservation tasks.

For the second release we kept the same approach used for the first prototype, using additional software to expose the Digital Repository REST APIs: this code, which will presumably be dropped for the final release, was necessary for previous DSpace releases,

[14] DSpace - http://www.dspace.org
[15] DSpace 5.x REST API: https://wiki.duraspace.org/display/DSDOC5x/REST+API

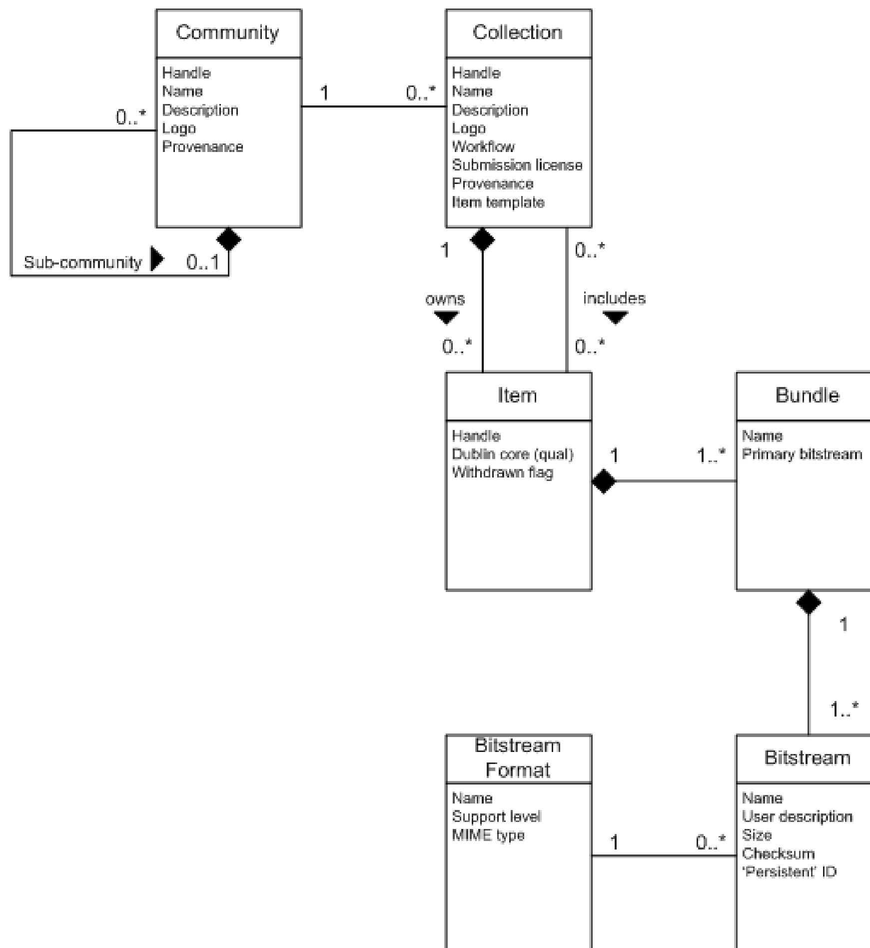

Figure 26: DSpace data model diagram.

since as explained above the DSpace REST APIs available with a vanilla installation only supported READ operations.

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 to 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 27. 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 27 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).

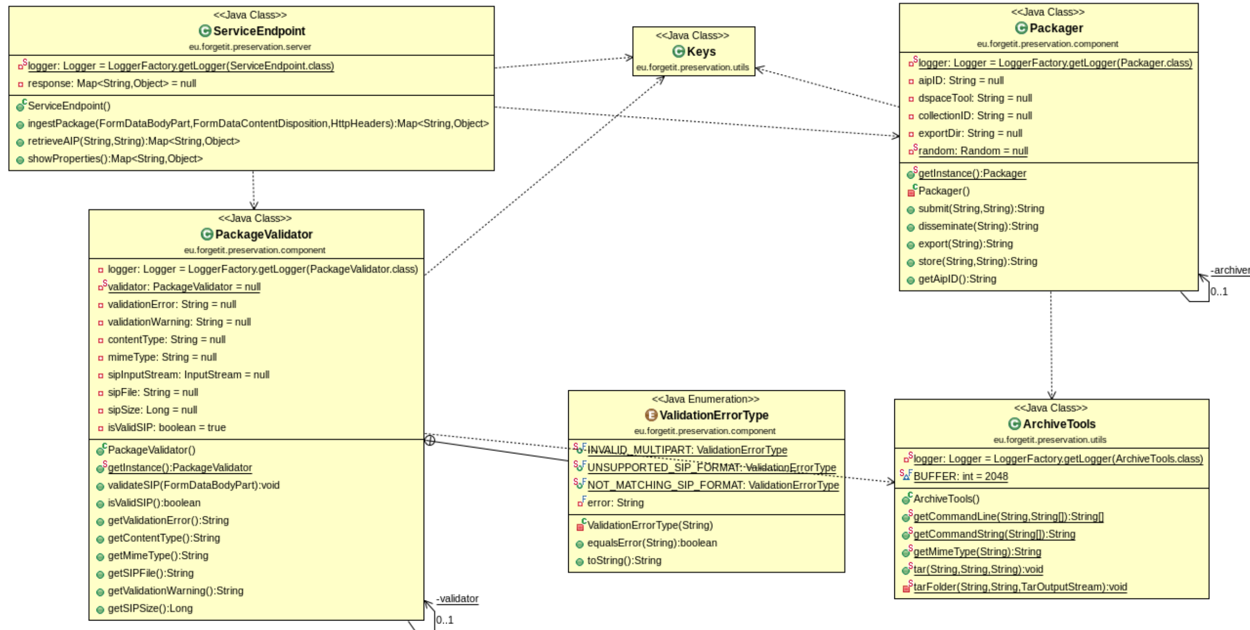

Figure 27: Class diagram for Preservation System endpoint, with associated classes.

7.2 Preservation-aware Storage System

The ForgetIT cloud-based Preservation-aware Storage system described in deliverable D7.3 [ForgetIT, 2014d], 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. Storlets can also be used to automatically add metadata fields to an object. In the integrated scenario presented during the second project review (see Appendix B.2), a storlet is run during the upload of a UK parliament debate package, to extract the date of the debate and the debate topics from the package, and inserts them into the object metadata. Once the upload is completed, the object metadata automatically becomes searchable. We demonstrated metadata searches based on the date and topics. We have also demonstrated the use of storlets to extracts only the relevant sections of the debate from the debate AIP package.

The Digital Repository communicates with the cloud storage through a REST interface (e.g. to upload or download an object). Storlet deployment is also done using the REST interface. Storlet deployment is essentially uploading a storlet jar file to a designated container in the Swift account. A storlet is a regular Swift object, but it must carry some metadata used by the storlet engine. 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 ForgetIT cloud storage stores not just the data of the preserved items it stores AIP packages which contain both the data and the associated metadata. This means that the transformation storlets, besides transforming the data, also need basic packaging abilities. With LTU we defined and implemented the basic packaging functionalities that the transformation storlets need to perform. These include computing the hash of the transformed image, and adding a timestamp for the transformation. The transformation activity also needs to be described and logged as part of the provenance of the object, since this is an important aspect of maintaining the authenticity of the content [Rothenberg, 2000].

With TT, we showed how a combination of open-source tools can be used to monitor the performance of a swift cluster. Monitoring of a swift cluster helps a swift administrator detect problems in the entire cluster or in specific nodes. It helps the administrator guarantee that the cluster meets the quality expectations of end users. We have also implemented and tested heuristics to dispatch storlet computations based on storage-node utilization, and showed that these heuristics lead to a better, and more consistent, user experience.

The storlet engine code has been open-sourced, and uploaded it to GitHub. We are also trying to work with the Swift community on making the storlet engine part of the Swift upstream. The license we have chosen for the storlet-engine code is Apache License version 2.0, to conform to the OpenStack Swift license.

We are currently working to add storlet support for Static Large Objects (SLOs). SLOs are a mechanism to overcome the 5GB Swift limit on objects, whereby a single object is divided to several segments. We are also working to finalize the integration of the preservation-aware storage system into the ForgetIT framework. This would entail, among other things, finalizing the mechanism through which the storlet engine would notify DSpace when AIP packages are transformed within the object store, for instance following a format transformation.

8 PoF Framework: Second Prototype Implementation

Information about software development, deployment and testing was already provided for the first prototype implementation in Section 8.1 of deliverable D8.3 [ForgetIT, 2013d].

In the following we provide additional information mainly 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.

The approach used for the second prototype was not changed for what concerns the software development and deployment, although the interaction among all partners was improved and the identification of the three scenarios described in Appendix B has driven the implementation effort. For each scenario a leading partner was identified and a dedicated team was established within the collaboration: each team worked on the assigned scenario almost independently, with periodic check points to verify the overall status and discuss common issues.

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)[16] bundle for Java EE Developers. The version of Eclipse IDE used for the development is 4.4 (Luna), while for compilation we upgraded the code to use Oracle Java JDK 8. The source code is available on project SVN repository and Trac[17]) is used as issue tracking system. Apache Maven[18] is used to compile and build the Java projects.

Software Projects and Packages

The Java packages of both projects are briefly described in Table 8 and Table 9, while the UML package diagrams are shown in Figure 28 and Figure 29. In both the Figures and the Tables we omitted test packages (used mainly for unitary 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`.

[16]Eclipse - http://www.eclipse.org
[17]The Trac Project - http://trac.edgewall.org
[18]Apache Maven - http://maven.apache.org

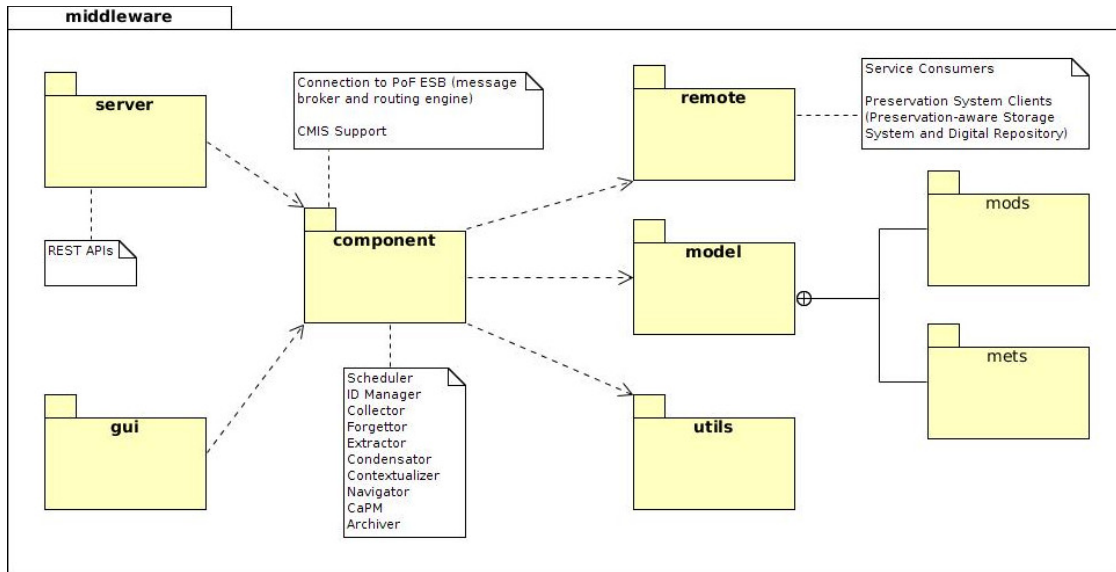

Figure 28: UML package diagram for the PoF Middleware.

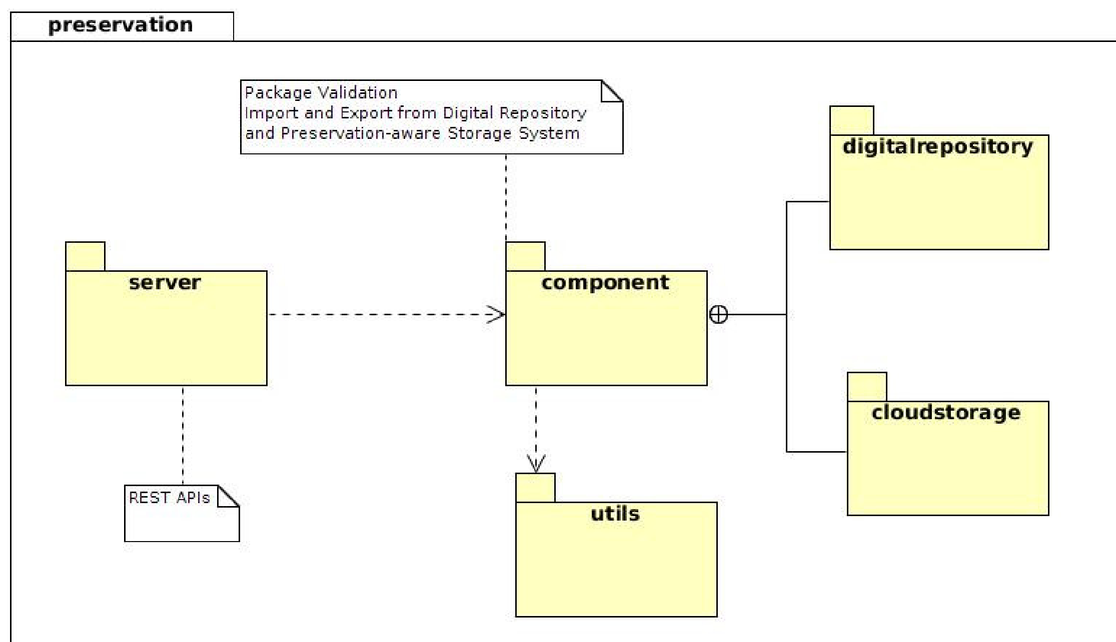

Figure 29: UML package diagram for the Preservation System.

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	Two main classes, Collection and Item, to represent collections and single items to be preserved. The sub-packages contain auto-generated classes corresponding to specific metadata schemas, such as METS and MODS, which are used when preparing packages for preservation, since such metadata formats 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 8: 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 9: 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 unitary tests using JUnit[19]. 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 unitary tests and package the compiled code in a Java WAR file to be deployed in Apache Tomcat 8[20].

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. For certain steps it was necessary to use the results of pre-processed data while the actual component was still under development and could not be integrated in the workflow. For additional details see also D8.3.

[19] JUnit - http://junit.org
[20] 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[21], 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 name server.

Software Documentation

The documentation of the source code is automatically generated using Doxygen[22] and will be available on the project web site at the following URL:

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

An internal task force has been established within the project to publish the source code of the PoF Framework on a public repository, presumably GitHub. This requires 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 can 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. When new versions of the framework components are released, they will be updated on the repository. For detailed documentation about each component please refer to WP8 deliverables and 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 based

[21] KVM - `http://www.linux-kvm.org`
[22] Doxygen - `http://www.doxygen.org`

on project configuration. A few executables (e.g. `ffmpeg`) are used for specific tasks. The code has been developed and tested mainly for Linux (Ubuntu Server 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 is 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 [ForgetIT, 2013d], 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 is still under discussion and will be defined for the final prototype release, described in deliverable D8.6 [ForgetIT, 2016b]. In parallel, project partners are working on improving the source code quality and documentation to a level adequate for dissemination on a public repository. This task is still in progress at the moment of writing. It is worth noting that the core libraries for the implementation of the PoF Middleware are already available under the Apache license and that several components developed within the project will be released as open source, as described in the previous Sections. Any additional code developed to implement the PoF Middleware will be 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 still under evaluation at the moment of writing, although an open source license for the core part of the Storlet Engine is foreseen (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.

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.

A task force has been established in the project, to evaluate candidate open source licenses, taking into account third party dependencies used by the different components.

9 Conclusions

9.1 Summary

The document provides a description of the second release of the PoF Framework which was demonstrated at the second annual project review. 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 second year of the project. Two workflows from the first release of the PoF Reference Model have driven the prototype implementation and several demos, associated to three main scenarios, have been presented.

In the following Sections we briefly discuss the assessment of the results presented here according to the WP8 performance indicators and then describe the plan for future work.

9.2 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 second 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 second release of the PoF Framework. The results achieved so far are compatible with the expected progress and success indicators for WP8, although further development is required for the final framework release, as described below.

Indicator 1: APIs and protocols

The achievements for the first prototype already satisfied this indicator, similar considerations are reported here for the second prototype. The APIs and protocols of the components to be integrated have been tested, the second 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 has been 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 second year with the integration of additional components in the implementation of the new workflows defined in the PoF Reference Model.

A preliminary integration plan is 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 lace, according to the plan, although additional workflows will be implemented in the third release, which require further integration effort. Concerning the middleware shared components, compared to the plan, we have almost completed the Metadata Repository implementation, while only the design and a early prototype is available for the Context-aware Preservation Manager. Concerning the middleware core components and the Preservation System, the current status is almost compliant to the plan: one exception to the plan is represented by the Contextualizer, which requires further development, while the Condensator service has been fully integrated. Based on such considerations, we can estimate that 9 of 14 components are now fully integrated and further development on such components will not affect their integration, while 4 components (Forgettor, Navigator, Contextualizer and Metadata Repository) have been only partially integrated, mainly due to their development status, and 1 component (Context-aware Preservation Manager) requires further development before integration. The aforementioned integration plan is constantly discussed and monitored withing the consortium by means of face-to-face meeting and periodic conference call, and pppropriate actions

have been taken to complete the development and integration of all components on time for the final framework release.

Indicator 3: development and test infrastructure

The testbed environment has been 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) is used to keep track of all open issues identified during project meetings and periodic conference calls. It is 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 is assigned to the appropriate partner. 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 documents is 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.

9.2.1 Evaluation of the PoF Framework

The evaluation of the framework has not been completed, yet. This is still under discussion within the consortium, to define a complete evaluation plan when all components will be developed and integrated in the framework and all workflows defined in the reference model will be implemented. This is planned for the third framework release and will be included in deliverable D8.6 [ForgetIT, 2016b].

Nevertheless, many components have already been tested individually within the corresponding WP. Many components leverage third party software tools that are actively maintained and tested by large open source communities. The backbone of the middleware infrastructure is based on Apache ServiceMix framework, which has been adopted in several open source and commercial products.

The two pilot applications will be tested again in the next months, a detailed plan is under preparation within WP2.

9.3 Next Steps

The workplan for the third framework release has been defined after the second year review, taking into account review recommendations. The primary focus for the third year is to fully support the PoF Reference Model (both the workflows for the functional part and the information model). The second release implements only two workflows defined in the

Core and Remember & Forget layers of the model: the missing workflows associated to the Evolution layer will be implemented, too. New releases of the existing components will be available for integration, but some effort will be used to integrate the new components such as the Context-aware Preservation Manager. No major updates are expected for the middleware infrastructure, but the Preservation System will be further developed: for example a better implementation of the preservation strategies for the long-term will be one of the main challenges. Finally, the evaluation of the final framework release will be performed, to validate the results of the project. A discussion among all partners to define the assessment criteria and methodology for all developed components has been started.

10 References

[dsp, a] DSpace GitHub Repository. https://github.com/DSpace/DSpace. Retrieved 31 July 2014.

[dsp, b] DSpace SourceForge Repository. https://sourceforge.net/projects/dspace/files. Retrieved 31 July 2014.

[gat,] General Architecture for Text Engineering (GATE). https://gate.ac.uk. Retrieved 30 June 2015.

[obj, 2015] (2015). ObjectDB - Fast Object Database for Java. http://www.objectdb.com. Retrieved 30 June 2015.

[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 30 June 2015.

[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. Retrieved 29 August 2014.

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

[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.

[ForgetIT, 2013a] ForgetIT (2013a). Deliverable D3.1: Report on Foundations of Managed Forgetting.

[ForgetIT, 2013b] ForgetIT (2013b). Deliverable D4.1: Information Analysis, Consolidation and Concentration for Preservation – State of the Art and Approach.

[ForgetIT, 2013c] ForgetIT (2013c). Deliverable D5.1: Foundations of Synergetic Preservation.

[ForgetIT, 2013d] ForgetIT (2013d). Deliverable D8.1: Integration Plan and Architectural Approach.

[ForgetIT, 2014a] ForgetIT (2014a). Deliverable D3.2: Components for Managed Forgetting – First Release.

[ForgetIT, 2014b] ForgetIT (2014b). Deliverable D4.2: Information Analysis, Consolidation and Concentration Techniques, and Evaluation – First Release.

[ForgetIT, 2014c] ForgetIT (2014c). Deliverable D5.2: Workflow Model and Prototype for Transition between Active System and AIS.

[ForgetIT, 2014d] ForgetIT (2014d). Deliverable D7.3: Computational Storage Services – Second Release.

[ForgetIT, 2014e] ForgetIT (2014e). Deliverable D8.3: Preserve-or-Forget Framework – First Release.

[ForgetIT, 2014f] ForgetIT (2014f). Deliverable D9.2: Use Cases & Mock-up Development.

[ForgetIT, 2015a] ForgetIT (2015a). Deliverable D10.2: Organizational Preservation Pilot Application V1.

[ForgetIT, 2015b] ForgetIT (2015b). Deliverable D10.3: Organizational Preservation Pilot Application V2.

[ForgetIT, 2015c] ForgetIT (2015c). Deliverable D3.3: Strategies and Components for Managed Forgetting – Second Release.

[ForgetIT, 2015d] ForgetIT (2015d). Deliverable D4.3: Information Analysis, Consolidation and Concentration Techniques, and Evaluation – Second Release.

[ForgetIT, 2015e] ForgetIT (2015e). Deliverable D5.3: Workflow Model and Prototype for Transition between Active System and AIS – Second Release.

[ForgetIT, 2015f] ForgetIT (2015f). Deliverable D6.3: Contextualisation Tools – Second Release.

[ForgetIT, 2015g] ForgetIT (2015g). Deliverable D8.2: Preserve-or-Forget Reference Model – Initial Model.

[ForgetIT, 2015h] ForgetIT (2015h). Deliverable D9.3: Personal Preservation Pilot I – Concise Preserving Personal Desktop.

[ForgetIT, 2016a] ForgetIT (2016a). Deliverable D6.4: Contextualisation framework and evaluation.

[ForgetIT, 2016b] ForgetIT (2016b). Deliverable D8.6: Preserve-or-Forget Framework – Final 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.

[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.

[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 30 June 2015.

[Rothenberg, 2000] Rothenberg, J. (2000). Preserving authentic digital information. In *Authenticity in a digital environment*, pages 51–68. CLIR.

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

Glossary

AIP Archival Information Package. 59–62

CMIS Content Management Interoperability Services. 3, 7, 9–11, 17, 23–29, 31, 32, 34–36, 44, 46, 54–59, 67, 71, 88

CRUD Create Read Update Delete. 34, 36–38, 59

DIP Dissemination Information Package. 45, 60

EAI Enterprise Application Integration. 19, 71

EIP Enterprise Integration Patterns. 7, 19–21, 40, 43, 50, 71

EJB Enterprise JavaBeans. 34, 36, 38, 65

ESB Enterprise Service Bus. 7, 11, 17, 20, 21, 45, 71

EXIF Exchangeable Image File Format. 56

IDE Integrated Development Environment. 63, 66

JMS Java Message Service. 21

JSON JavaScript Object Notation. 22, 23, 36, 38, 61, 71, 85, 93

MB Memory Buoyancy. 9, 35, 47–49, 57, 93

MOM Message Oriented Middleware. 7, 11, 17–19, 21, 71

OAIS Open Archival Information System. 60

PIMO Personal Information MOdel. 25, 48, 49, 54–56, 67, 91

PoF Preserve-or-Forget. 1, 3, 7–11, 13–17, 19, 21–26, 31, 33, 35, 37–39, 45, 46, 49–60, 63, 65–68, 70–72, 79, 89, 91

PV Preservation Value. 10, 23, 25, 26, 28, 33, 35, 36, 39, 47–49, 55, 57

SIP Submission Information Package. 44, 46, 59, 60

UML Unified Modeling Language. 11, 63, 72

UUID Universally Unique IDentifier. 34

XML eXtensible Markup Language. 21–23, 34, 36, 38, 39, 41, 43, 46, 61, 71, 78–81, 83

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[23], 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 11 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 30); the processes and threads running in the broker ((Figure 31); the routes defined in Camel using Spring XML, described before (Figure 32); .

Name	Queue Size	Producer #	Consumer #	Enqueue #	Dequeue #	Memory %	Dispatch #	Always
CREATE.COLLECTION.QUEUE	0	0	1	0	0	0	0	false
ERROR.QUEUE	0	0	1	0	0	0	0	false
IMAGE.ANALYSIS.QUEUE	0	0	1	0	0	0	0	false
LOG.QUEUE	0	0	1	252	252	0	252	false
PRESERVATION.QUEUE	0	0	1	1	1	0	1	false
REACTIVATION.QUEUE	0	0	1	0	0	0	0	false
SCHEDULER.QUEUE	0	0	1	251	251	0	251	false

Figure 30: Message queues monitoring.

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

hawtio

ActiveMQ Camel Connect Health JMX Logs Threads Tomcat Wiki

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

Object Monitor Usage Synchronizer Usage Thread Contention Monitoring

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 31: Process monitoring.

hawtio

ActiveMQ Camel Connect Health JMX Logs Threads Tomcat Wiki

Attributes Route Diagram Source Route Metrics Inflight Exchanges Properties Debug

Update

- Camel Contexts
  - camel-1
    - Routes
      - createCollectionRoute
      - errorRoute
      - imageAnalysisRoute
      - periodicSchedulerRoute
      - preservationRoute
      - reActivationRoute
      - schedulerRoute
        - activemq:queue:SCHEDULEF
        - onException1
        - choice1
    - Endpoints
    - MBeans

```
<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 32: 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 Scenarios for the Second Prototype Demonstrations

In the following Sections we present three representative scenarios that have been showcased during the second project review. With respect to the demonstrations after the first year (based on the first prototype), where the focus was on showing an integrated end-to-end preservation workflow to validate the ForgetIT approach and on demonstrating the main technologies available in the project, for the second release we identified some scenarios for both the personal and organizational preservation which could benefit from new components and an improved integration framework.

Scenarios 1 and 3 are part of the personal preservation use-case, while scenario 2 is associated to the organizational preservation use-case.

B.1 Scenario 1: Incremental Photo Preservation

Scenario 1 regards the problem of supporting users in selecting important photos for preservation from their own collections. We have developed a desktop application (see Figure 33) where the user can browse her own collections and apply automatic methods to make selections and create summaries for preservation. Differently from scenario 3 (see Section B.3), we do not assume the presence of any semantic metadata or annotation provided by the user. In order to keep personal preferences into account, the user can revise the selection done automatically before preserving the photos. Such feedback is used to update the selection model, which can adapt to the preferences of the user.

The main goals of the scenario are the following:

- Providing a structured browsing of photo collections, where the structure is determined via event clustering
- Suggesting sub sets of important photos for preservation and future revisiting

The high-level overview of scenario 1 is depicted in Figure 34. The desktop application has been developed by ARGELA, within the context of WP9. The input photo collection is sent to the Extractor developed by CERTH (Section 5.4), which processes the photos and extracts information such as clusters, quality, concepts, near-duplicates, and faces. This information is returned to the ARGELA application and sent to the components that the user has selected to perform a selection (Selector) or a summary (Summarizer). The main difference between these methods is that the Selector component is trained to meet user expectations, i.e. produces selections that are as close as possible to those that the user would have selected by themselves, while the Summarizer component produces a balanced summary of the collection by picking one or more representatives from each sub-event (cluster). The Selector (implemented by LUH within WP3) and the Summarizer (implemented by CERTH within WP4) are part of the Forgettor component (Section 5.7) and Condensator component (Section 5.5) respectively, and they have been described in more detail in deliverables D3.3 [ForgetIT, 2015c] and D4.3 [ForgetIT, 2015d].

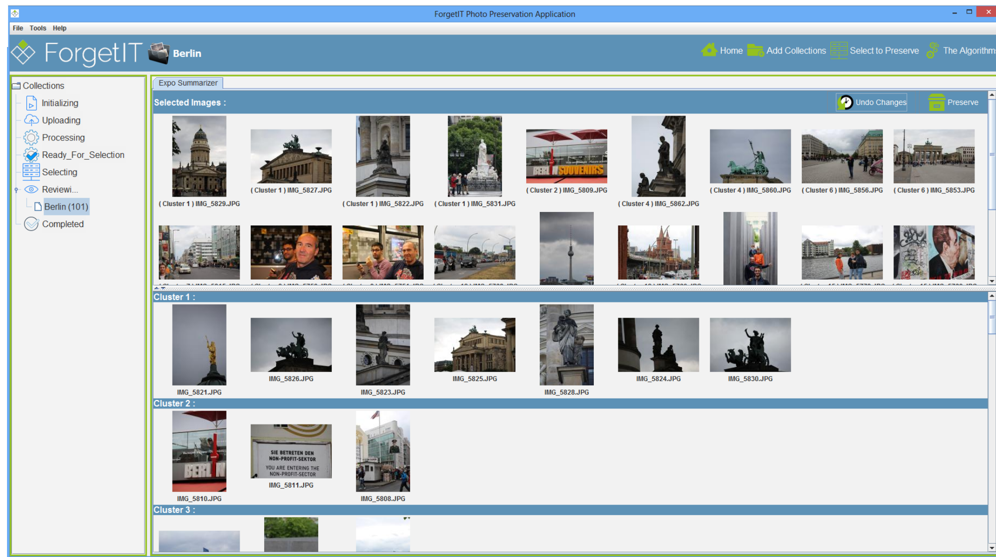

Figure 33: Screenshot of the desktop application developed within scenario 1.

Once the selection or summary has been created and sent to the desktop application, the user can revise it according to her preferences and finally store it into a publicly accessible CMIS server.

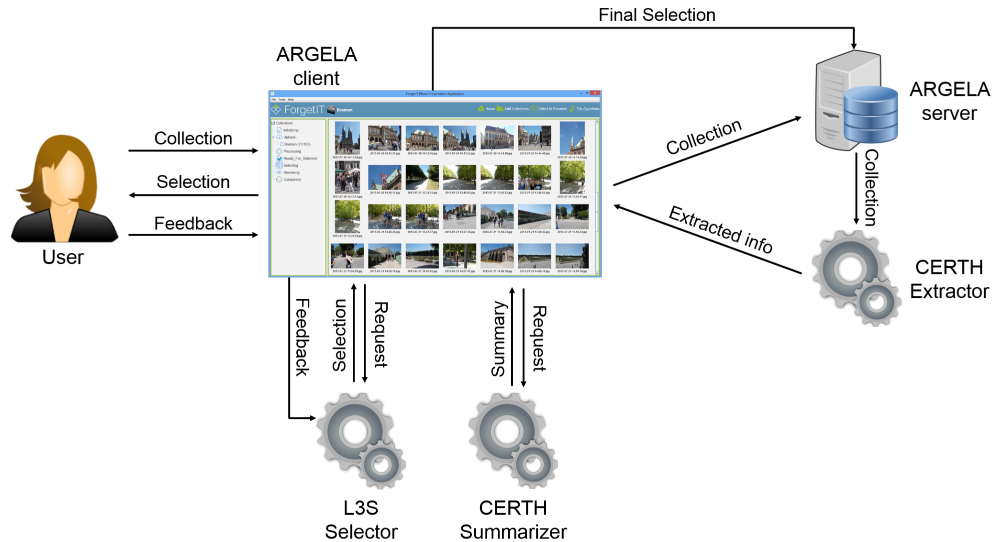

Figure 34: Workflow and components of scenario 1.

An overview of the different components and processes involved in the execution of this scenario are shown in Figure 34. The desktop application only requires Java to be installed and is available for download at the following link:

`http://forgetit.argela.com.tr/forgetIt_media/forgetItSecure.jnlp.`

B.2 Scenario 2: Automated Contextualization and Re-contextualization

This scenario is designed to highlight how contextualization and the evolution of context can be used to provide a rich search experience over both preserved and active content. In an attempt to show context evolution over an extended period of time we are using an organization setting, but one not linked directly to WP10.

We are using publicly available transcripts and descriptive metadata that covers the debates which take place within the UK Parliament[24]. This allows us access to data spanning a long period of time, in which there is continual evolution of political parties, constituency boundaries, ministerial positions, etc. The dataset for this scenario was preserved running the Preservation Preparation workflow described in Section 4. The UK Parliament debates were published using a CMIS repository (see Figure 35), so they could be accessed by the PoF Middleware components.

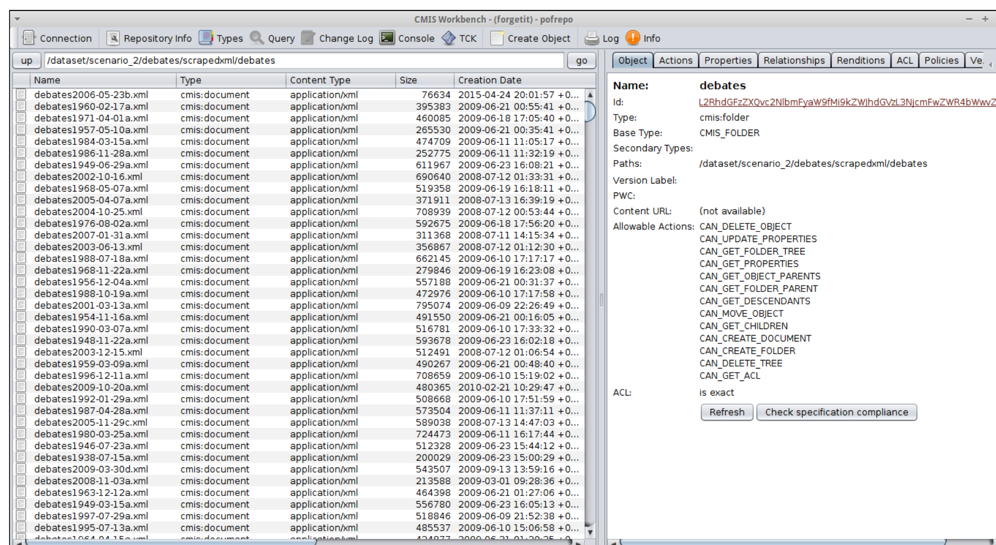

Figure 35: CMIS Repository used to publish UK Parliament dataset.

Figure 36 shows the main search UI for this scenario, where a topic based search can be performed. The dataset for this scenario has been processed in the PoF Middleware and preserved in the Preservation System. Upon ingest, a specific Storlet was executed to extract relevant information. The links in the left menu of the UI point to the preserved copies of the debates which can be retrieved from the cloud storage. The tag cloud for the world context is shown in Figure 37. This scenario is currently in a very early state of development to form an initial Navigator component (Section 5.9). It is envisaged that further development will take place to the demo alongside the work on the contextualization and context evolution components.

The current version of the demo is accessible online at the following link:

`http://services.gate.ac.uk/forgetit/search/.`

[24]ParlParse - `http://parser.theyworkforyou.com/`

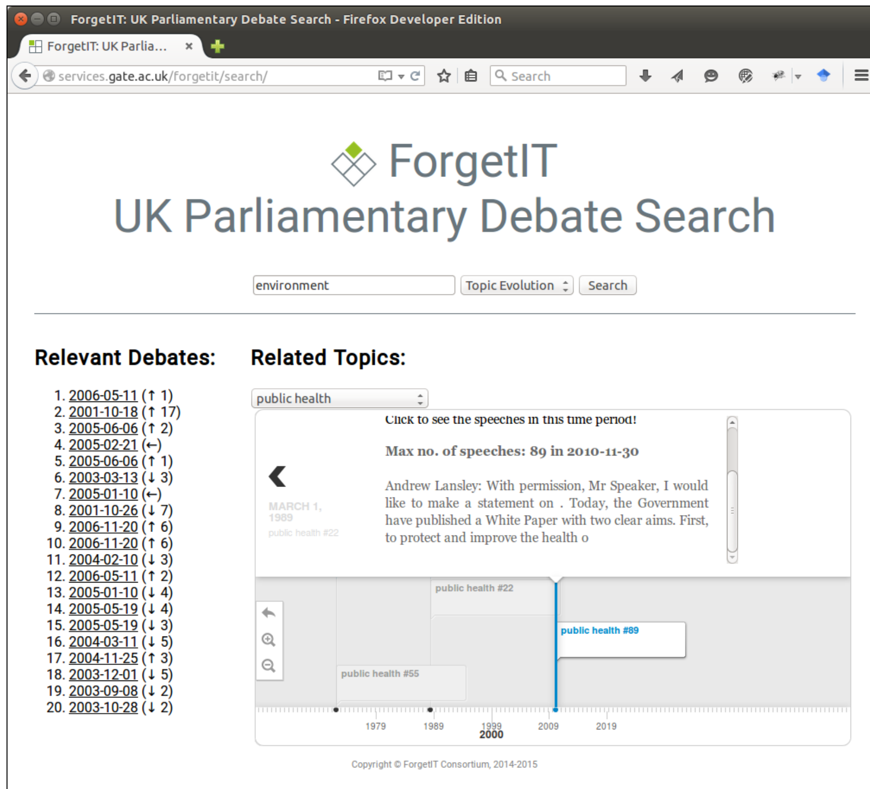

Figure 36: Main search interface for scenario 2.

Figure 37: World Context document view for scenario 2.

B.3 Scenario 3: Automated Generation of Multimedia Diary

In scenario 3 we show-cased the *PIMO Diary* (see Figure 38), a diary application built on top of the Semantic Desktop. This automatically generated diary enables reminiscence and keeps preserved content useful (and joyful to revisit). The following aspects were stressed:

- Contextualized Remembering with the PIMO as additional benefit for the Personal Information Management of a user;
- Resources are part of clusters resembling prominent activities in the chosen time period;
- Condensation of resources for a given time period and granularity;
- Rich context collected from the PIMO for resources (as local context for contextualization in the PoF Framework);
- Use of image quality assessment and image concept detection in the PIMO in the algorithm for selecting representatives for images in a diary entry.

The following components and methods were shown:

- Semantic Desktop & PIMO with the dedicated App PIMO Diary, developed in WP9;
- Image Quality Assessment & Classification, provided by the Extractor component (Section 5.4) developed within WP4;
- Condensation & Contextualization in PIMO, provided by the Condensator (Section 5.5) and Contextualizer (Section 5.5) components, developed within WP4 and WP6, respectively.

The demonstration was done using DFKI's Knowledge Management Department's PIMO including professional data as well as photos taken on business trips, most prominently on the ForgetIT workshops. It resembled therefore a business-related scenario covering also personal data, thus, still addressing the WP9 application scenario.

Explanations and videos on the PIMO Diary can be found in the Personal Preservation Pilot I online documentation at the following link:

`https://pimo.opendfki.de/wp9-pilot/pimodiary.html`.

Figure 38: The PIMO Diary shown in scenario 3: the diary of the presenter of 2014.

C 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[25]. 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.

[25] 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();
}
```