FP7-SME-1
Project no. 262289

HARMOSEARCH

Harmonised Semantic Meta-Search in Distributed Heterogeneous Databases

D2.2_final
Architectural Design

Due date of deliverable: 2011-03-31
Actual submission date: 2011-03-31

Start date of project: 2010-12-01 Duration: 24 month

Project funded by the European Commission within the Seventh Framework Programme

Dissemination Level		
PU	Public	**X**
PP	Restricted to other participants (including the Commission Services)	
RE	Restricted to a group specified by the Consortium (including the Commission Services)	
CO	Confidential, only for members of the Consortium (including the Commission Services)	

PROJECT ACRONYM: **HARMOSEARCH**

Project Title: Harmonised Semantic Meta-Search in Distributed Heterogeneous Databases

Grant Agreement: 262289

Starting date: December 2010 **Ending date:** November 2012

Deliverable Number: D2.2, Final

Title of the Deliverable: Architectural Design

Lead Beneficiary: X+O

Task/WP related to the Deliverable: WP 2, Task 2.4

Type (Internal or Restricted or Public): Public

Author(s): Adriano Venturini, Albert Rainer, Christoph Herzog, Claudio Prandoni, Alessandro Forti, Federico Galeazzi, Sabine Schneider, Thomas Motal

Partner(s) Contributing: eCTRL, TU-Wien, CPR

Contractual Date of Delivery to the CEC: March 31, 2011

Actual Date of Delivery to the CEC: March 31, 2011

PROJECT CO-ORDINATOR

Company name:	[X+O]
Name of representative:	Manfred Hackl
Address:	Hamburgerstrasse 10/7, A-1050 Vienna, Austria
Phone number:	+43-676-842755-100
Fax number:	+43-676-842755-599
E-mail:	manfred.hackl@xpluso.com
Project WEB site address:	www.harmosearch.org

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE OF THE DOCUMENT

This document defines the structure of the HarmoSearch system, that is, the main logical components, the main technologies and how the components will be developed to support the scenarios and the use cases identified in the Deliverable D 2.1. The document also considers how the existing Harmonise 2.0 technology should be updated to be able to integrate the new components developed during the project, such as the semantic registry, the query processor, and the mapping tool.

1.2 DEFINITIONS OF TERMS AND ABBREVIATIONS

Harmonise: name of the existing technological solution. The current version is Harmonise 2.0, which includes the Harmonise Ontology, Harmonise Service Centre and the Harmonise Portal.

Harmonise Platform: name identifying the whole set of Harmonise components

Metasearch: one of the major functions to be implemented in this project and the name of the component which will support it. It provides distributed search capabilities to the integrated data sources.

Semantic Registry: component to be developed within this project which will contain semantic profile information about the services available within the Harmonise networks.

Mapping tool: the mapping tool is a standalone application that supports a user with little technical knowledge in creating visually the necessary mapping definitions from the data model of a Harmonise participant to the one of Harmonise and vice-versa. It consists of a graphical User Interface to show and manipulate mappings, a pluggable set of algorithms to support automatic mappings, a generator to create mapping artefacts, and an infrastructure in order to manage a mapping project.

1.3 RELATIONSHIP WITH OTHER DOCUMENTS

Inputs for this document are the D2.1 Use Case Specification deliverable, which defines the functionalities that the system should support and which external systems should be integrated, and the architecture document of the existing Harmonise solution. This document poses the basis for the activities of the following work packages:

- Semantic concept and ontology (WP3), which defines query and domain model to be adopted in HarmoSearch;
- Query mapping (WP4), which defines and implements the components dedicated to the metasearch engine and distributed query processing;
- Semantic registry for metasearch (WP5) which defines and implements the components dedicated to storing and managing semantic information about the services participating in the network;

- Automatic mapping tool (WP6), which will support in a semi-automatic way the mapping definitions among Harmonise model and the participant content models.

1.4 STRUCTURE OF THE DOCUMENT

This document follows a typical multiple view architecture description approach. Each chapter is dedicated to a particular view describing the system according to a specific perspective:

- Chapter 2, Logical View, provides a structured view of the main logical component of the system, without considering a specific technology but providing an insight about which are the main logical modules which have been identified to support the defined use cases.
- Chapter 3, Activity View, provides a dynamic view, showing how the components identified in the logical view can support the major use cases identified in the Deliverable D2.1.
- Chapter 4, Software Infrastructure, provides an overview of the main technologies which will be used as foundation for the components which will compose the system.
- Chapter 5 Component View, defines the software components which will be developed, their roles and dependencies, and how they communicate through a set of well identified interfaces.
- Chapter 6, Development View, defines the development tools which will be adopted for the development and some issues common to all modules like error management, log management and testing process.
- Chapter 7, Physical View, shows how the system could be deployed on the network environment and discusses possible solutions according to the level of availability and workload which should be supported.

2 LOGICAL VIEW

This section provides an overview of the system from the logical point of view. The focus is on the main logical components of the system, their roles and how they are related, but without considering how they will be technically implemented, thus without considering the specific technologies that will be adopted. The goal is to identify the components which can support the use cases defined in the deliverable D2.1.

2.1 OVERALL DIAGRAM

The diagram in Figure 1 shows the components that have been identified to support the scenarios and use cases of the HarmoSearch system.

The system is structured in the following major subsystems:

- Core Components. The core of the system, the components dedicated to implement the business logic.
- Core Services. Core services implemented using the core components as identified in the deliverable D.2.1.
- External Services. Services which the platform will be able to provide by integrating services provided by third parties. They are the one identified in the deliverable D.2.1.
- External Components. Components provided by third parties which can be used by the platform to provide the external services.

The components in the diagram are coloured in red if they are new components. Existing components, which will be used by the new components, are shown in yellow.

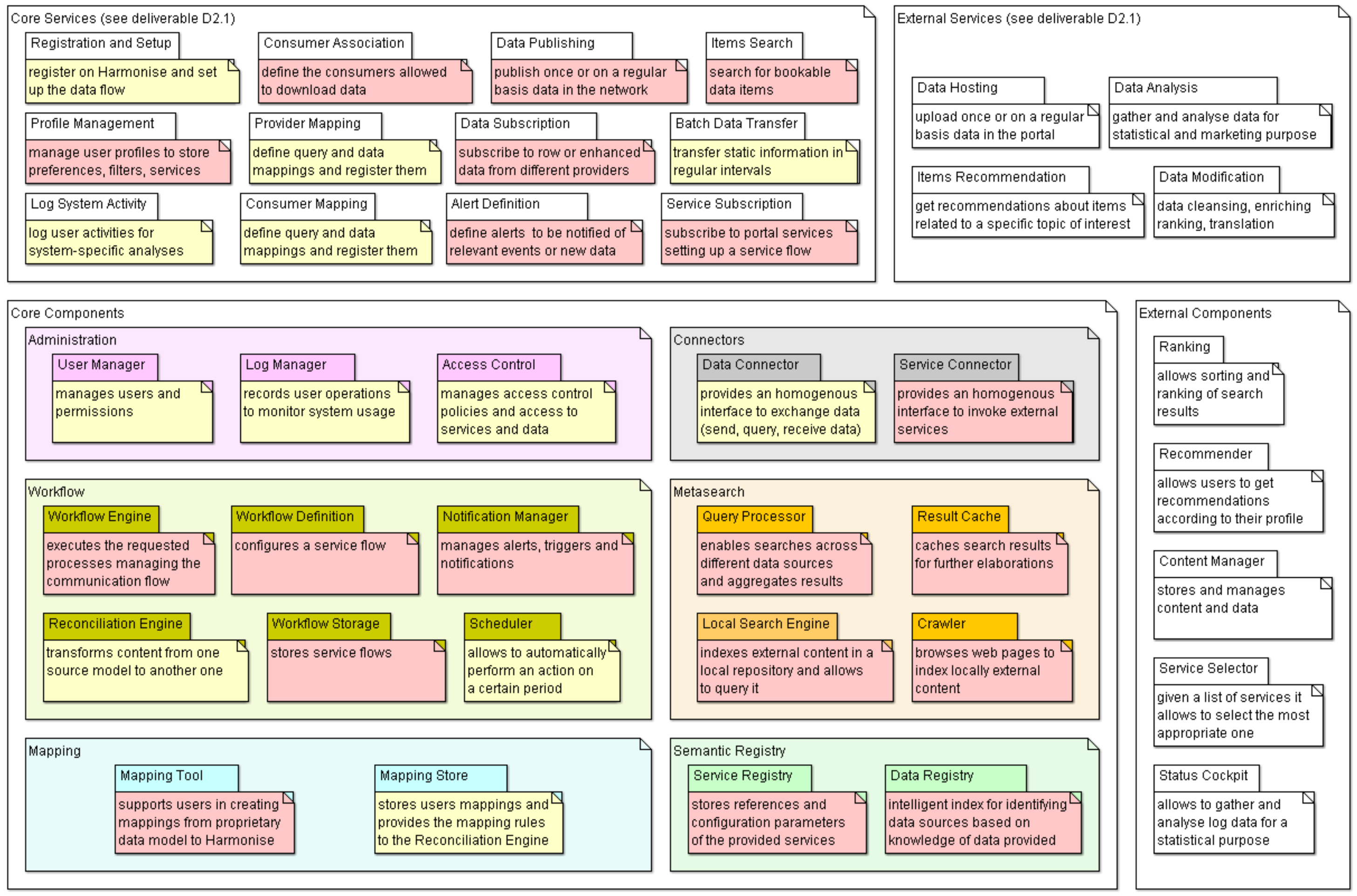

Core Services (see deliverable D2.1)
Registration and Setup
register on Harmonise and set up the data flow
Consumer Association
define the consumers allowed to download data
Data Publishing
publish once or on a regular basis data in the network
Items Search
search for bookable data items
Profile Management
manage user profiles to store preferences, filters, services
Provider Mapping
define query and data mappings and register them
Data Subscription
subscribe to row or enhanced data from different providers
Batch Data Transfer
transfer static information in regular intervals
Log System Activity
log user activities for system-specific analyses
Consumer Mapping
define query and data mappings and register them
Alert Definition
define alerts to be notified of relevant events or new data
Service Subscription
subscribe to portal services setting up a service flow
External Services (see deliverable D2.1)
Data Hosting
upload once or on a regular basis data in the portal
Data Analysis
gather and analyse data for statistical and marketing purpose
Items Recommendation
get recommendations about items related to a specific topic of interest
Data Modification
data cleansing, enriching ranking, translation
Core Components
Administration
User Manager
manages users and permissions
Log Manager
records user operations to monitor system usage
Access Control
manages access control policies and access to services and data
Connectors
Data Connector
provides an homogenous interface to exchange data (send, query, receive data)
Service Connector
provides an homogenous interface to invoke external services
Workflow
Workflow Engine
executes the requested processes managing the communication flow
Workflow Definition
configures a service flow
Notification Manager
manages alerts, triggers and notifications
Reconciliation Engine
transforms content from one source model to another one
Workflow Storage
stores service flows
Scheduler
allows to automatically perform an action on a certain period
Metasearch
Query Processor
enables searches across different data sources and aggregates results
Result Cache
caches search results for further elaborations
Local Search Engine
indexes external content in a local repository and allows to query it
Crawler
browses web pages to index locally external content
Mapping
Mapping Tool
supports users in creating mappings from proprietary data model to Harmonise
Mapping Store
stores users mappings and provides the mapping rules to the Reconciliation Engine
Semantic Registry
Service Registry
stores references and configuration parameters of the provided services
Data Registry
intelligent index for identifying data sources based on knowledge of data provided
External Components
Ranking
allows sorting and ranking of search results
Recommender
allows users to get recommendations according to their profile
Content Manager
stores and manages content and data
Service Selector
given a list of services it allows to select the most appropriate one
Status Cockpit
allows to gather and analyse log data for a statistical purpose

Figure 1 Logical View

The Core Services and External services components support the use cases and scenarios already described in the deliverable D.2.1. Core services are exactly the ones identified in D2.1, external services are grouped together: Data Analysis includes the Market Intelligence use cases ("Submit Ad Hoc Request", "Execute Interval Request", "Analyse Data" and "Manage Notifications"), Data Modification includes the Data modification use cases ("Data Cleansing", "Decision Support", "Data Enrichment" and "Data Translation"), Data Hosting includes "Data Download". "Negotiation" and "Payment" use cases are not covered in this document because they have been identified as out of the scope.

The rest of this section focuses on the core components, which define the major building blocks of the system. They have been grouped into six logical areas according to the functionality they provide: Administration Components (coloured in pink), Workflow Components (bright green), Mapping Components (light blue), Connector Components (grey), Metasearch Components (orange) and Semantic Registry Components (light green).

2.2 ADMINISTRATION

The *Administration* component subsumes functionality related to administrative issues regarding the definition and creation of user profiles and access policies and the observation and registration of relevant system activities. The Administration component is sub-divided into three sub-components: *User Manager*, *Access Control*, and *Log Manager*.

The main task of the *User Manager* component is to provide a general access point for registering organisations and users and for managing the profile of a Harmonise participant. Managing in this context means that the participant is able to add, edit, or remove certain information from his profile, like alerts, filters, subscriptions, data profiles, etc.

The *Access Control* component is used to define specific access control policies to restrict access to certain services. Access control policies operate on the data as well as on the user level. In other words, a service provider can specify *which* data should be available and *who* should have access to this data and under which conditions. The Access Control component is of interest to multiple scenarios. For example, restricting the access on certain data is of high importance for the *bookable item search* scenario, where Harmonise participants are able to query data items offered by different data providers.

The *Log Manager* component is in charge of logging relevant meta-information reflecting the behaviour of the Harmonise system. This is achieved by observing specific types of system activities that are either triggered or performed by Harmonise participants. The activities a Harmonise participant may perform are manifold. Thus, the monitoring mechanism of the *Log Manager* component is kept very general in order to allow a flexible observation of certain system activities. A good example for a user-driven activity is a search request among different service providers. In such a case, the query request will be analysed and the representative meta-data (e.g. which type of information has been queried among which partners by whom) will be extracted and stored. The *Log Manager* is of special interest to many components of the Harmonise system for example to the business intelligence scenarios (see D2.1) where the gathered system information will be used for analytical and statistical purposes. It provides a common interface to access and store log data.

2.3 WORKFLOW

The workflow component is in charge of orchestrating the processes described in the scenarios supported by HarmoSearch. A process is a controlled execution of specific services among the ones available as core services of the platform or as external integrated services. Example of processes are the "data transfer" process, which allows a data provider to push data to a given consumer, or the "metasearch" process, where a Harmonise participant can query distributed and heterogeneous data sources. More complex processes can be defined too, which chain together different services, like searching and getting ranked or translated results. Processes can be defined in a flexible way, by exploiting a specific process definition language. The Workflow component is sub-divided into six sub-components: *Workflow Engine*, *Workflow Definition*, *Workflow Storage*, *Notification Manager*, *Reconciliation Engine* and *Scheduler*.

The *Workflow Definition* component provides the necessary tools for supporting the process definition task. The defined processes are stored by the *Workflow Storage* component.

The *Workflow Engine* executes specific process instances. When another component needs to execute a certain process, it asks the Workflow Engine to start an instance of a given process definition. The *Workflow Engine* executes the steps described in the process definition, by calling the involved services. Among services which could be invoked, an important role is played by the *Reconciliation Engine*. The *Reconciliation Engine*'s role is to transform data and queries among different data formats, to enable the key feature of Harmonise which is to allow participants to participate into the network and exchange data without changing their data model. Thus the *Reconciliation Engine* is invoked by the *Workflow Engine* when a query or an instance data needs to be transformed.

The *Scheduler* is in charge of automatically executing specific workflow processes at defined interval times. It is highly configurable to allow defining when specific workflow processes should be automatically started.

Finally the *Notification Manager* monitors the system to discover when a specific process should be started. For example, it is used for supporting the publishing and subscription scenario, where specific data consumers want to be notified or to receive data they are interested in as soon as they are published to the network. In such case, the notification manager may trigger the execution of the process which allows getting data from a service provider and sending them to the subscriber.

2.4 METASEARCH

The aim of the metasearch component is to enable searches across different individual search components of heterogeneous websites and aggregate the results in a unified list. Partners will use this functionality to search heterogeneous data sources using a uniform interface and a consistent query language. The Metasearch component is sub-divided into four sub-components: *Query Processor*, *Result Cache*, *Local Search Engine* and *Crawler*.

The conducer of the metasearch process is the *Workflow Engine* (Workflow component): it receives the search query, controls access rights through the *Access Control* (Administration component), looks up the *Semantic Registry* (Semantic component) to find appropriate data providers and calls the *Query Processor* to perform the query and aggregate the results to be sent to the participant who performed the query.

Content available to be queried in the Harmonise network can reside in external repositories or can be stored internally on a local repository which indexes external content. The task of the *Query Processor* is therefore both to look up the internal repository through the *Local Search Engine*, and to forward the query to the external data providers, aggregating results coming both from the local db and from external data sources. Moreover, in order to allow searching data in a network of data bases, not only data but also queries need to be transformed. Thus, similar to translating data on the fly, the search or query string also needs to be transformed. The *Query Processor* has the responsibility to take care of this, making use, if necessary, of the *Reconciliation Engine* (Workflow component).

The *Crawler* is the component which is in charge of browsing web pages to fetch and index external content. Visited pages are indexed and stored in an internal repository using the *Local Search Engine*.

Finally, the *Result Cache* allows storing temporarily search results in order to be further elaborated: filtered, sorted, paginated, etc.

2.5 SEMANTIC REGISTRY

The semantic registry deals with describing on a metadata level what actual information or services are provided by the different Harmonise participants. Therefore, the semantic registry has two main sub-components, the *Data Registry* and the *Service Registry*.

The *Data Registry* is responsible for capturing information about the content provided by data providers. This is done by storing metadata describing the content and associating it to the respective data providers. There are several ways this data can be acquired or updated. First and most important is the active definition of the provided data, either when setting up the participant's account or later on when updating the meta-information. This is done manually through a user interface provided on the Harmonise platform. The process can be aided by analysing the provided mapping, which gives a basic indication of the provided content. The other important source of metadata information is a direct (automatic) update of the meta-information through a web-service call by the data provider. This is especially useful in order to notify other participants on the Harmonise network of new or different data being available. Apart from data providers, data consumers can also express their interest in certain data items in a similar way.

The main task of the *Data Registry* is to be able to find HarmoSearch participants associated to specific data. One use of this association is to identify topics of interest for HarmoSearch participants in a notification scenario. The main use case, however, is to identify data providers who have content that can possibly satisfy a given search query. Therefore, there exists a close association between the whole metasearch process and the *Data Registry*.

The *Service Registry* has the primary function to describe the external services offered by Harmonise participants. The service itself and its basic effects are described at the time of service registration. At this point the service provider makes the service and its technical implementation known to the Harmonise system. A user interface integrated into the Harmonise platform is used for registering the service. Furthermore, a specialised user interface allows Harmonise participants to discover these services and apply them to specific workflows (see 2.3, "Workflow").

2.6 MAPPING

The mapping component is responsible for the harmonisation of heterogeneous data and consists of two main parts - the *Mapping Tool* and the *Mapping Store*. The *Mapping Tool* is used at design time to create the necessary harmonisation artefacts and stores these artefacts at the *Mapping Store*. The *Reconciliation Engine* (Workflow component) then accesses the *Mapping Store* at run time in order to fetch suitable artefacts for a particular request.

The *Mapping Tool* is a stand-alone application and has no dependency to the Harmonise platform. In principle, it can be deployed in any harmonisation project. Mapping artefacts are created using a propose-critique-modify approach, i.e., a mixture of automatic matching and user interaction. The tool makes a matching proposal and presents this proposal to the user. The user then supervises the proposal and accepts or rejects it partially or completely, optionally asking for a new proposal. Alternatively, the user may manually manipulate proposals in cases the employed matching algorithms do not come up with a correct solution. Finally, from the defined matches the tool creates a mapping artefact that can then be uploaded to the *Mapping Store*.

The *Mapping Store* is in effect a general data storage component with predefined access functions that allow to store and retrieve mapping artefacts. In addition, it provides functions to manage the lifecycle and access rights of artefacts. Lifecycle management includes activation and deactivation of artefacts as well as version control, removal, and replacement. Access right management includes controlling read and write access for users, roles, and groups and is delegated to *the Access Control Component* from the Administration Component.

2.7 CONNECTOR

The *Connector* component is used to define a homogeneous interface for exchanging data between Harmonise participants and for invoking external services. The Harmonise participants can send or receive data to or from other Harmonise participants via mailboxes or through a service they can provide (e.g. a rest call or a web service). They may either simply query data from the Harmonise network or use an external service to get the results ranked or translated. In both cases it is necessary to provide a common interface. Thus, the *Connector* component is subdivided into two additional sub-components: the *Data Connector* and the *Service Connector*. Both act as a proxy to the system.

The main goal of the *Data Connector* is to abstract the participants, by providing a homogenous interface to all the partners of the network, allowing them to exchange data and to be queried by other participants.

The *Service Connector* is used to allow interactions with external services that are not core components of the Harmonise system, e.g. to rank or modify data or to select items to be recommended.

Kommentar [DF1]: External component are not described. Is that normal? Also, CIore services and External services are defined in the D2.1 document, but do not necessarily have the same name as the one used in the diagram. It is therefore not very easy to understand what they correspond to. THe number of items do not correspond either. If we cannot/do not want to undate D2.1, would could possibly map the services to the previous definitions.

Kommentar [CP2]: External components are not described because they are not part of the platform. They are provided by external service providers and the platform enables the possibility top lug them in in order to fulfill external services.
Concerning naming of services: core services are exactly the same as D2.1, external services are grouped together a bit, anyway I added one sentence explaining this below the figure.
Negotiation and Payment are not covered in this document because they have been identified as out of the scope.

2.8 DOMAIN MODEL

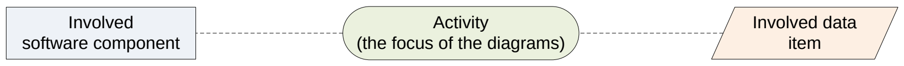
One typical section of the logical view in the architectural document is the domain model. The domain model will be described extensively in the deliverables D3.1, D3.2 and D3.3 of WP 3, thus we forward to them for its definition.

3 ACTIVITY DIAGRAMS

This section shows activity diagrams involving the components. The use cases described in D2.1 are used as starting points for defining the activity diagrams in this chapter and analyse how they are supported considering the identified components.

Note that the activities described in this chapter do not exhaustively cover everything that will be possible in the Harmonise system but are a summary of the identified use cases. Therefore, the selected diagrams should cover a large enough range of possible activities to identify all relevant components, their functions and important data items. All other possible activities in the Harmonise system are seen as variations or combinations of the ones presented here, configured through individual workflows (see section 5.3 ”workflow engine”).

In the activity diagrams the following graphical notation is used:


Involved software component
Activity (the focus of the diagrams)
Involved data item

3.1 HARMONISE REGISTRATION AND SETUP

Overview:
A Harmonise participants registers on the Harmonise platform and configures the data flow.

Use Cases which the diagram refer to:

MS-1 “Harmonise Registration and Setup”, IMPORT-1 “Harmonise Registration and Setup”.

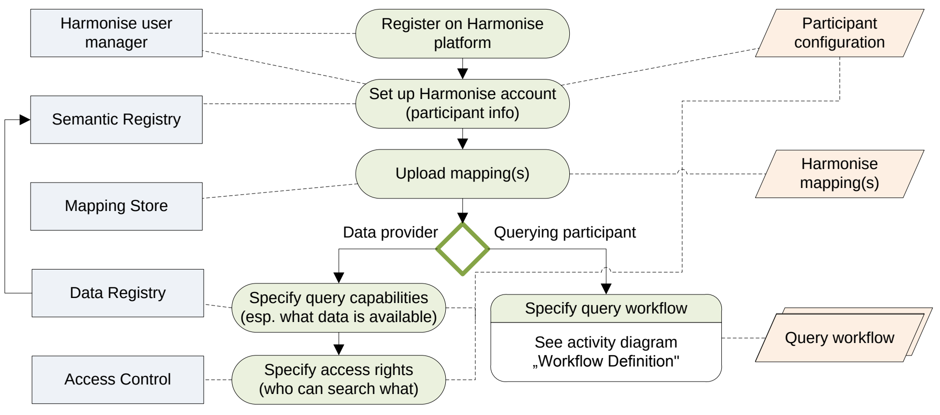
Activity Diagram:
Before being able to restrict access rights or queries to specific partners, these partners obviously have to be registered (if not fully configured) on the Harmonise platform.

The creation of mappings is outside the actual Harmonise system and therefore not considered in the activity diagrams. It is expected that the mappings have been created before registering.

Semantic Registry: is the major component concerned with registering all data providing or external services. It has two main parts:

Data Registry: deals with the description of what data a data provider can deliver. It is used to register and discover data providers and to find appropriate data providers for querying.

Service Registry: deals with the description of external services. These services take Harmonise data as input, apply operations (data enrichment, filtering, etc.) on the data and return the modified result set. The service registry deals with describing and discovering these services for being used in a query workflow.


Harmonise user manager
Register on Harmonise platform
Participant configuration
Semantic Registry
Set up Harmonise account (participant info)
Mapping Store
Upload mapping(s)
Harmonise mapping(s)
Data provider
Querying participant
Data Registry
Specify query capabilities (esp. what data is available)
Specify query workflow
See activity diagram „Workflow Definition"
Query workflow
Access Control
Specify access rights (who can search what)

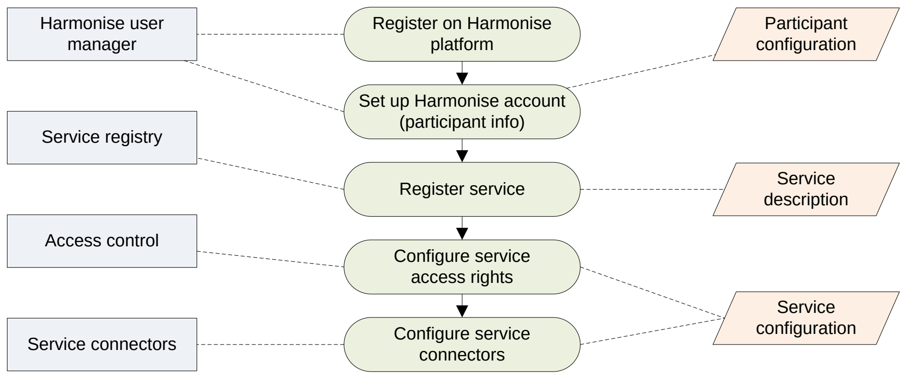
3.2 SERVICE REGISTRATION AND CONFIGURATION

Overview:
A Harmonise service provider registers his external service on the Harmonise platform and configures the service.

Use Cases which the diagram refer to:

MS-1 "Harmonise Registration and Setup", IMPORT-1 "Harmonise Registration and Setup".

Activity Diagram:
Before being able to restrict access rights to specific partners, these partners obviously have to be registered (if not fully configured) on the Harmonise platform.


Harmonise user manager
Register on Harmonise platform
Participant configuration
Set up Harmonise account (participant info)
Service registry
Register service
Service description
Access control
Configure service access rights
Service configuration
Service connectors
Configure service connectors

3.3 DATA PUBLISHING

Overview:

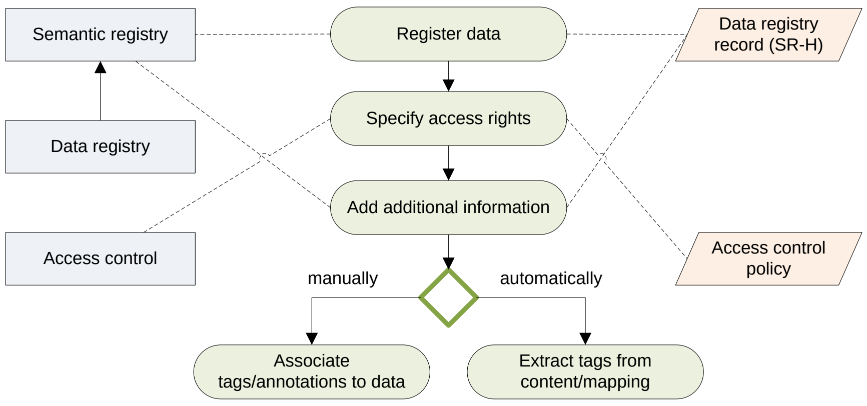
The goal of these use cases is to allow data providers to publish once or on a regular basis their data in the Harmonise network, storing some meta-data which are useful to describe what kind of data they have to offer. Data providers can then manually or automatically associate additional information to their data linking them together and creating data profiles to be used to facilitate subscription.

Use Cases which the diagram refer to:

PS-2 "Data Publishing", PS-3 "Data Enrichment", PS-4 "Consumer Association".

Activity Diagram:

Regarding data objects, format (SR-H) denotes the format of the Harmonise ontology for the Semantic Registry component. This activity diagram provides requirements for this ontology too.


Semantic registry
Register data
Data registry record (SR-H)
Data registry
Specify access rights
Add additional information
Access control
Access control policy
manually
automatically
Associate tags/annotations to data
Extract tags from content/mapping

3.4 DATA SUBSCRIPTION

Overview:

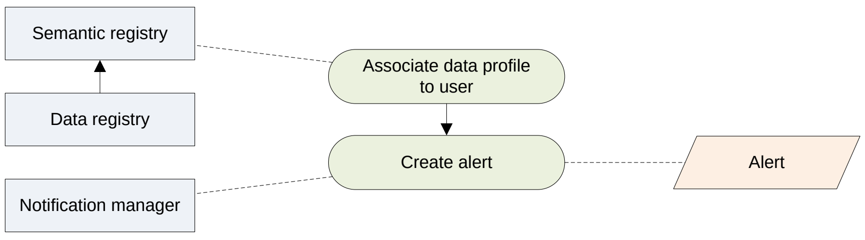
The goal of these use cases is to allow consumers to subscribe to data profiles in order to be notified if new data are available which may be of interest for them.

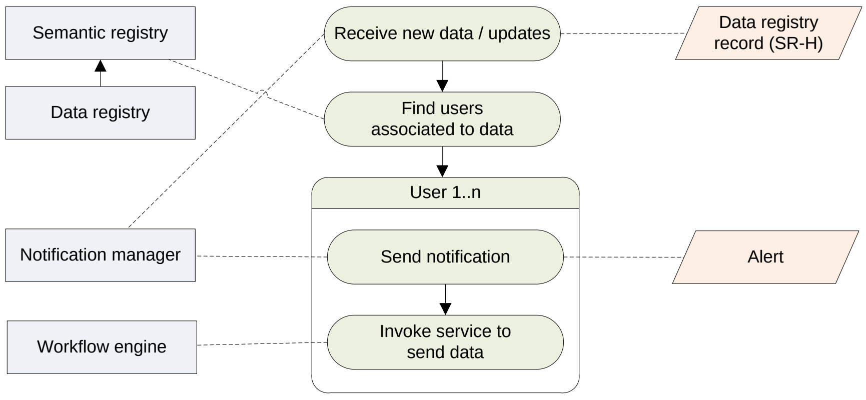
Use Cases which the diagram refer to:

PS-6 "Data Subscription", PS-7 "Alert Definition".

Activity Diagram:

Regarding data objects, format (SR-H) denotes the format of the Harmonise ontology for the Semantic Registry component. This activity diagram provides requirements for this ontology too.




Semantic registry
Data registry
Notification manager
Workflow engine
Receive new data / updates
Find users associated to data
User 1..n
Send notification
Invoke service to send data
Data registry record (SR-H)
Alert

3.5 WORKFLOW DEFINITION

Overview:

The goal of this use case is to allow users to set up a complex service or a service flow by chaining different services together. Example of services are booking, items recommendation, ranking, filtering, data translation, data modification, data hosting, statistical and market analyses, etc.

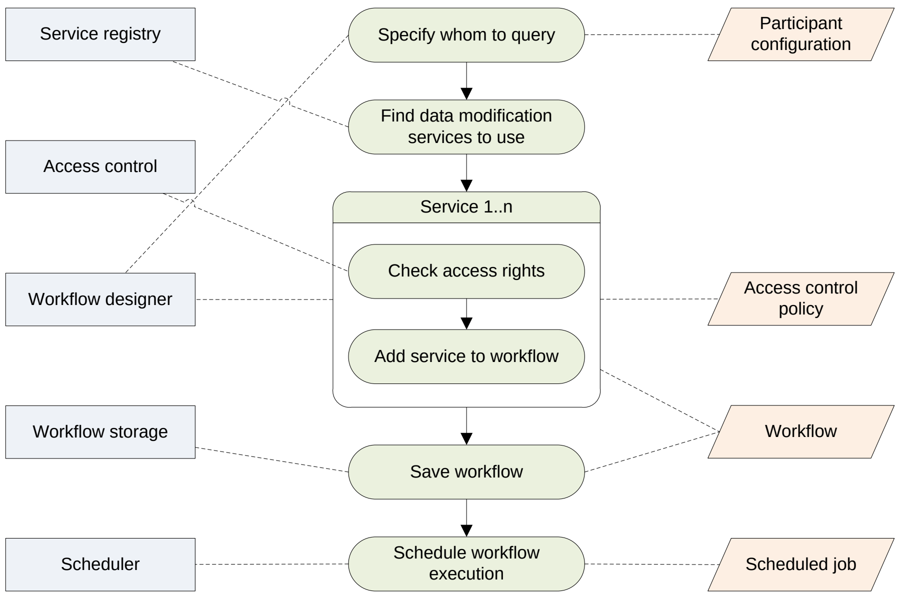
Use Cases which the diagram refer to:

MS-1 "Harmonise Registration and Setup", IMPORT-1 "Harmonise Registration and Setup".

Activity Diagram:

The activity diagram shows as an example of service flow the configuration of a query. A query flow can chain together different services, e.g. data modification services like filtering, ranking, cleansing, translating, etc.

Workflows can be scheduled to run when a given trigger occurs.



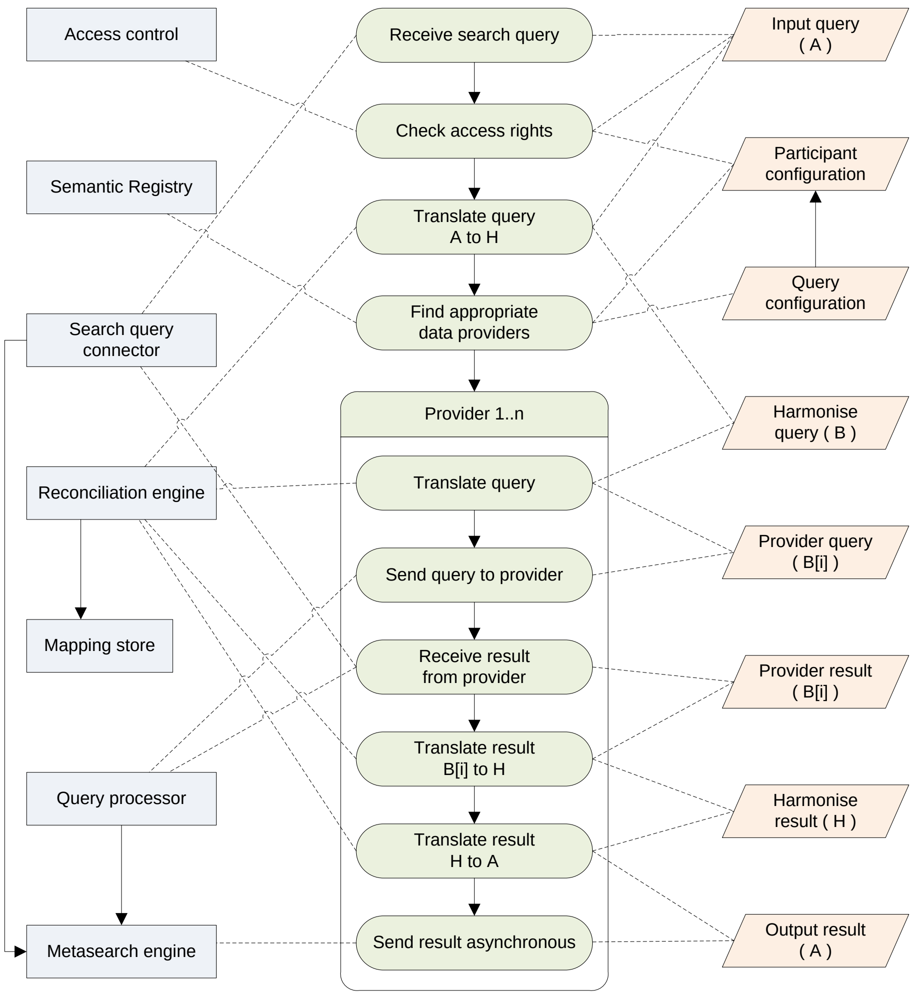
3.6 BOOKABLE ITEMS SEARCH

Overview:
The goal of the use case is to allow a harmonise participant (e.g., a tourism portal) to search for "live" (e.g., bookable) data items offered by specific data providers. The data is required to be up to date (e.g., availability information and price quotes). ontology. Please note that this type of search supports an asynchronous mechanism, assuming that some bookable data providers could provide results with different response time. This allows to send data to the receiver as data become available.

Use Cases which the diagram refer to:

MS-2 "Bookable Item Search".

Activity Diagram:
Regarding data objects, format (A) denotes the local format of the querying participant; (B[i]) the different formats of the data providers, namely the format of the i^{th} data provider; (H) the format of the Harmonise



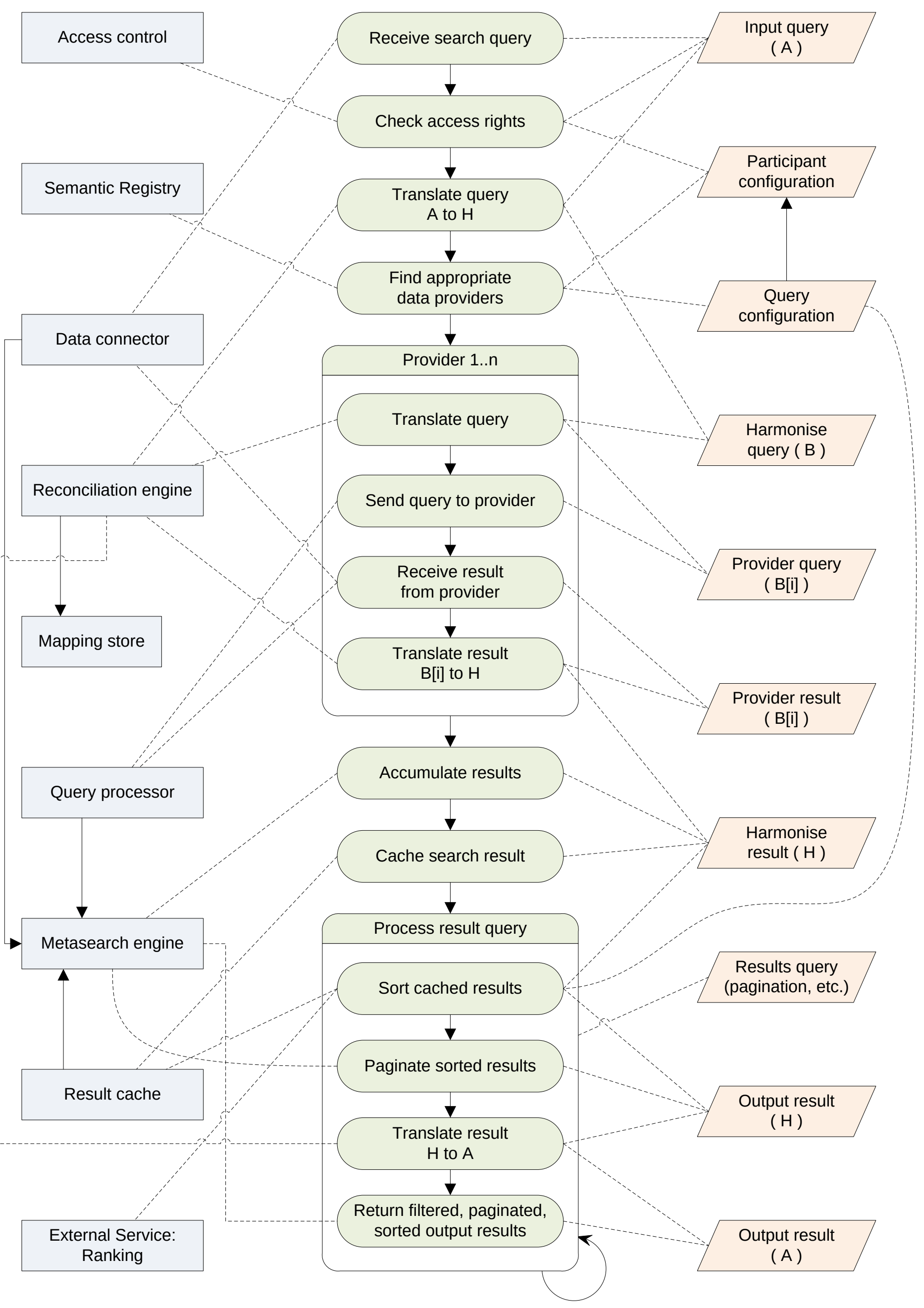
3.7 RANK AND PAGINATE RESULTS

Overview:
After a search has been conducted, the querying harmonise participant retrieves a specific view of the results, i.e., sorted and paginated (e.g., results 21-30). Possible other filters work likewise.

Use Cases which the diagram refer to:

MS-2 “Bookable Item Search”.

Activity Diagram:



3.8 ITEM RECOMMENDATION

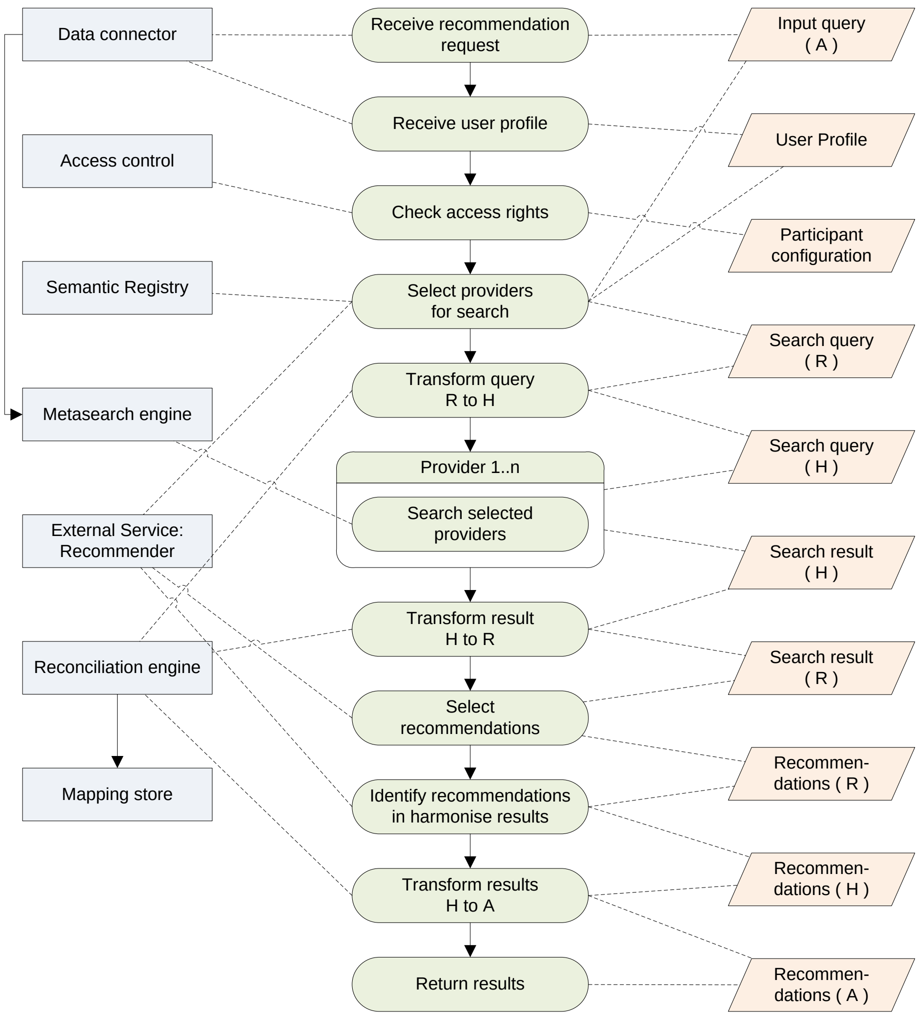
Overview:
A harmonise participants wants to get recommendations for a specific user about

items related to a specific topic of interest. The harmonise participant provides contextual information (for example geo-coordinates or specific theme of interest), some constraints and receives backs items best fitting the provided information.

Use Cases which the diagram refer to:

MS-3 “Item Recommendation”.

Activity Diagram:



3.9 BATCH TRANSFER OF STATIC DATA

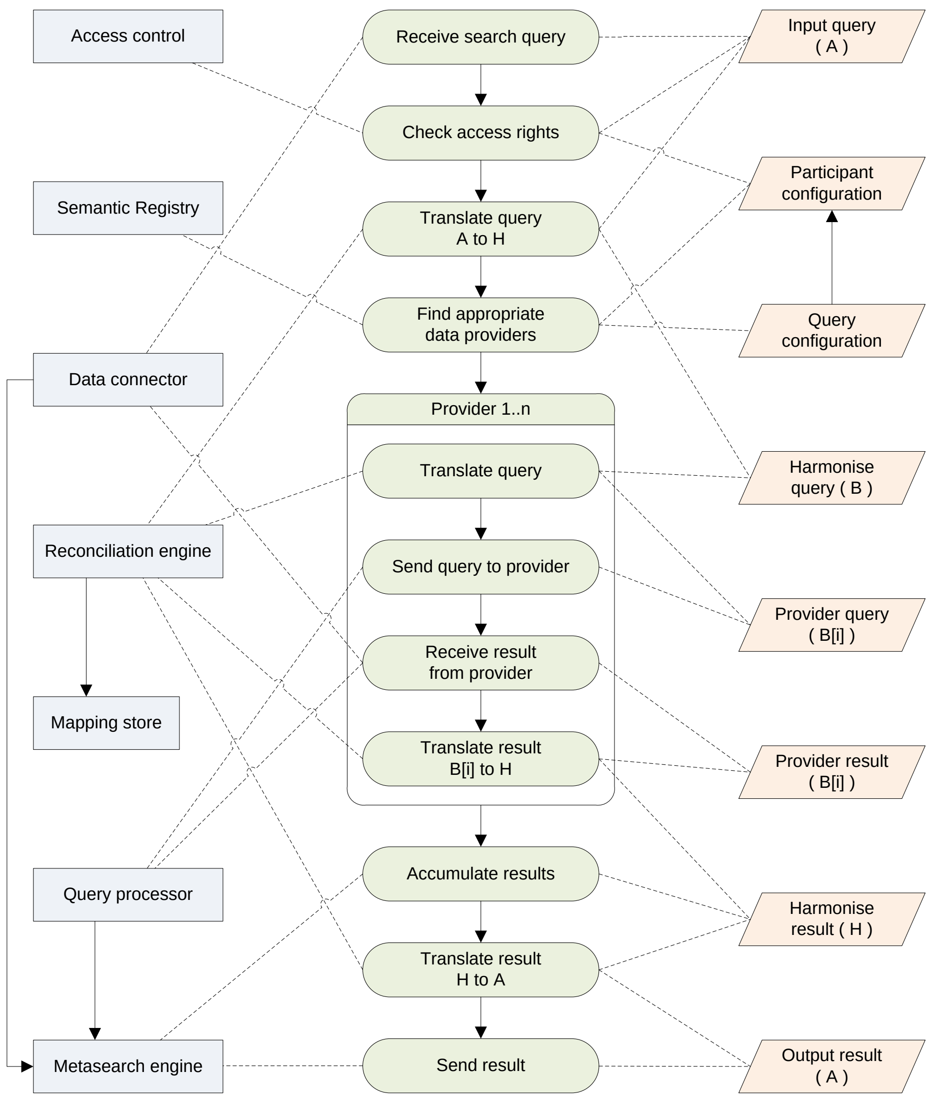
Overview:

The goal of the use case is to transfer static information (e.g. in case of accommodation: accommodation name, description, location, pictures, amenities, etc.) from data providers.

Use Cases which the diagram refer to:

IMPORT-2 “Batch transfer of static data”.

Activity Diagram:
Note that delta updates are simply specific queries causing specific results. Therefore this extension is not reflected in the activity diagram.



3.10 DATA HOSTING

Overview:

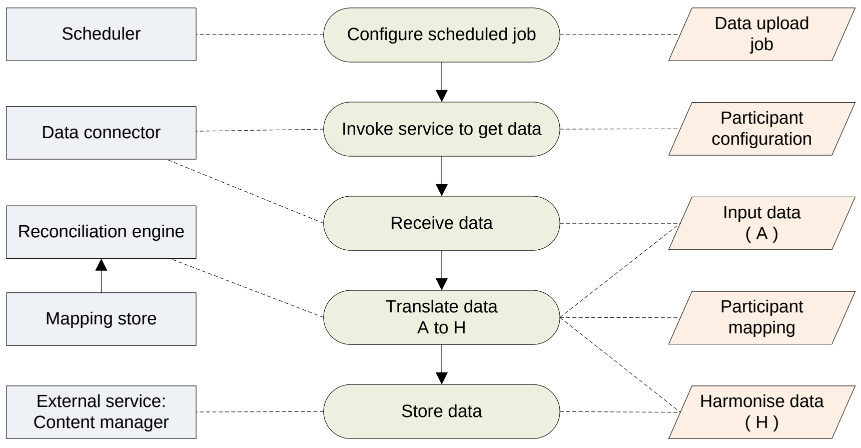
The goal of this use case is to allow data providers to upload once or on a regular basis their data in the Harmonise portal.

Use Cases which the diagram refer to:

PS-1 “Data Hosting”.

Activity Diagram:

Regarding data objects, format (A) denotes the local format of the data provider; (H) the format of the Harmonise ontology. The Content Manager could act also as a cache for other content providers.



3.11 DATA DOWNLOAD

Overview:

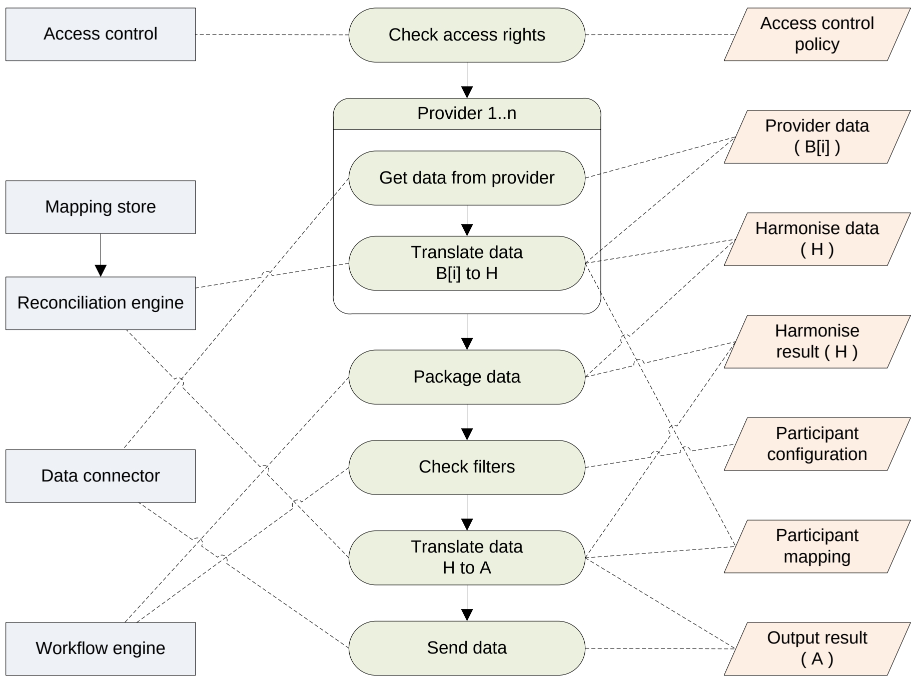
The goal of this use case is to allow consumers to download data pushed by data providers once or regularly.

Use Cases which the diagram refer to:

PS-5 “Data Download”.

Activity Diagram:

Regarding data objects, format (A) denotes the local format of the data consumer; (B[i]) the different formats of the data providers, namely the format of the ith data provider; (H) the format of the Harmonise ontology.



3.12 DATA MODIFICATION THROUGH EXTERNAL SERVICES

Overview:
After a search, the configured data modifying services for the search query are invoked. These data modification services process the search result in Harmonise format and finally return the processed results in the user's format.

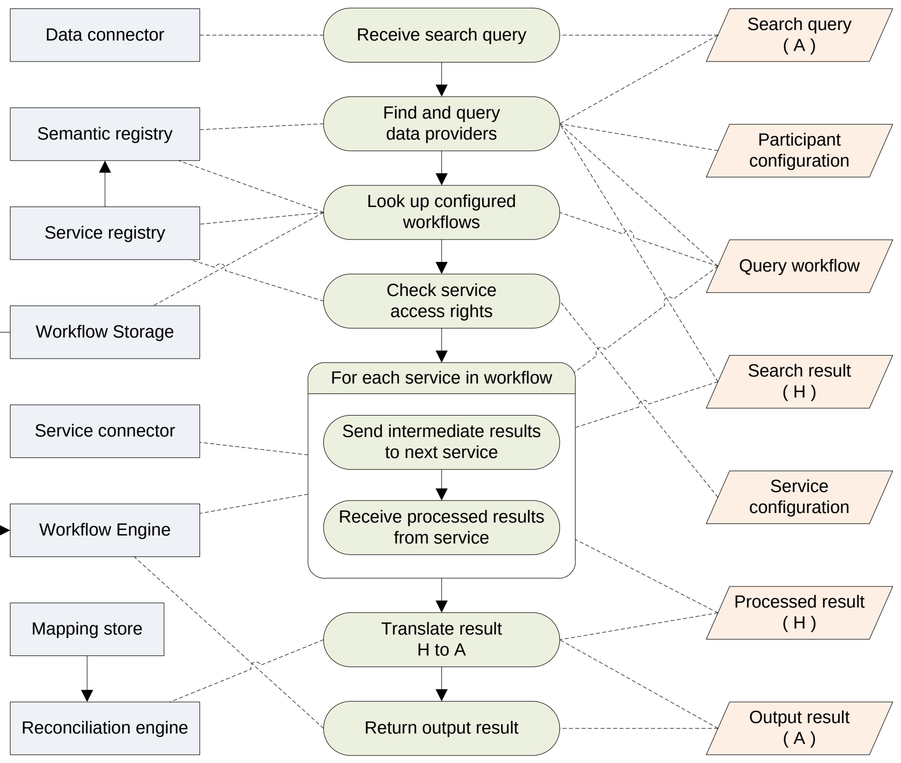
Note that this use case deals with the actual execution of a configured workflow (search and data modification services).

Use Cases which the diagram refer to:

DM-3 "Data Cleansing", DM-4 "Decision support", DM-5 "Data Enrichment", DM-6 "Data Translation".

Activity Diagram:
Regarding data objects, format (A) denotes the local format of the querying participant; (H) the format of the Harmonise ontology.

"Find and query data providers" can be any of the previously outlined search flows.



3.13 SUBMIT AD HOC REQUEST

Overview:

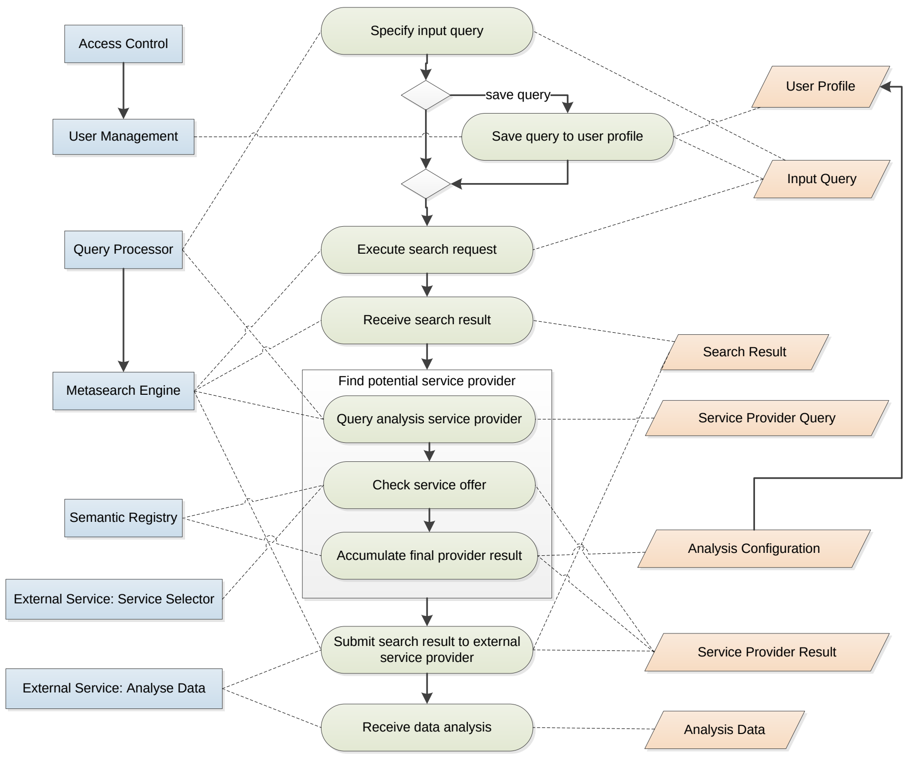
A Service Consumer/Service Provider is able to submit an ad-hoc search query collecting filtered data about accommodations and events used for analytical processing of market information conducted by a Market Analyser.

Use Cases which the diagram refer to:

MI-1 “Submit ad hoc request”.

Activity Diagram:

Regarding data objects, the input query corresponds to the initial search query specified by the user. The Analysis configuration corresponds to user specific requirements (e.g. data format (output), statistical methods, etc.).



This diagram represents a simple scenario, with no integration from multiple data sources. But it will be possible to compose more complex scenarios by chaining the activity diagrams described in the previous chapters (i.e. activity diagram at paragraph 3.7).

3.14 EXECUTE INTERVAL REQUEST

Overview:

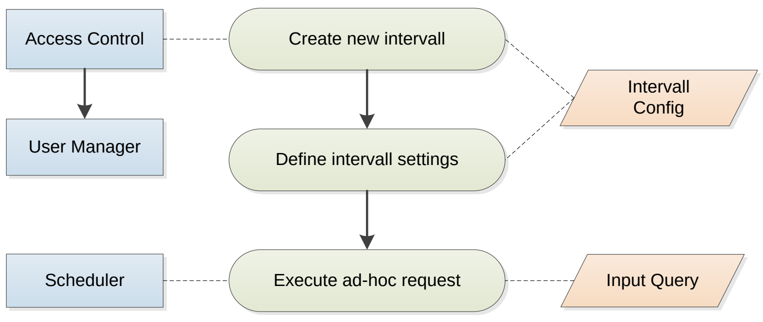
A Service Consumer/Service Provider is able to create a new or modify an existing search interval submitting a specific search query to the Harmonise system.

Use Cases which the diagram refer to:

MI-2 “Execute interval request”.

Activity Diagram:

An interval configuration includes all necessary information for executing an interval request (e.g. input query, data and time options, etc.).



3.15 ANALYSE DATA

Overview:

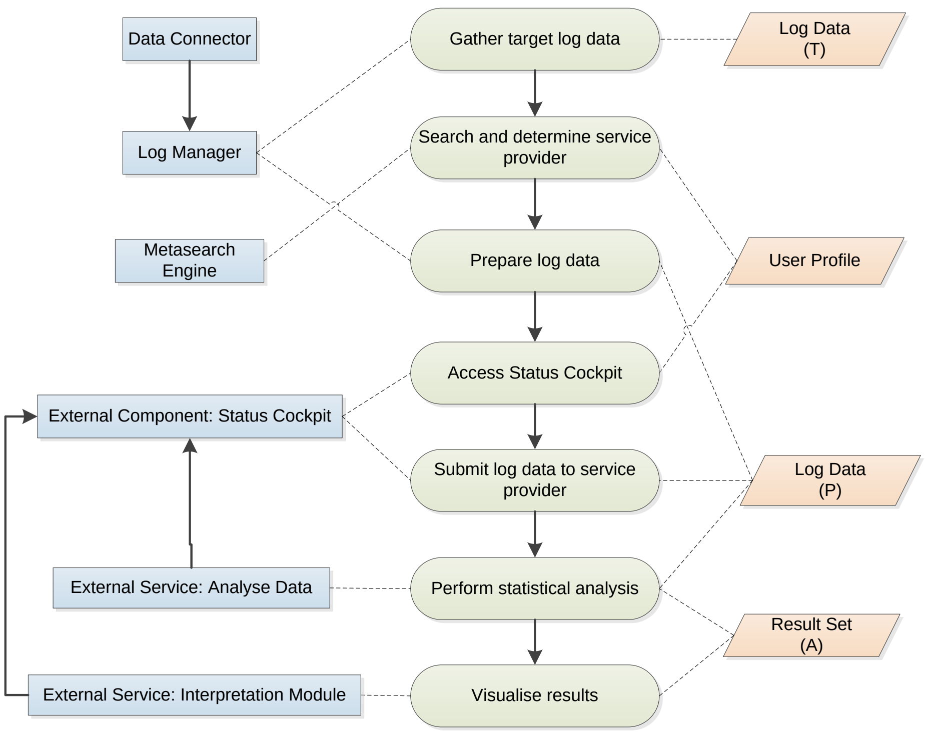
The Harmonise system enables a Harmonise Statistician to gather and analyse log data statistically.

Use Cases which the diagram refer to:

MI-3 “Analyse data”.

Activity Diagram:

Regarding data objects: Log data (T) refers to initial data as captured by the system with no specific format. Depending on the service provider who conducts the final analysis it may be necessary to transform the log-data to a specific format (Log Data (P)).


Data Connector
Log Manager
Metasearch Engine
External Component: Status Cockpit
External Service: Analyse Data
External Service: Interpretation Module
Gather target log data
Search and determine service provider
Prepare log data
Access Status Cockpit
Submit log data to service provider
Perform statistical analysis
Visualise results
Log Data (T)
User Profile
Log Data (P)
Result Set (A)

3.16 MANAGE NOTIFICATIONS

Overview:

The Harmonise Statistician/Harmonise Administrator should be able to monitor certain events and activities within the Harmonise system. Thus this activity diagram shows how the administrator can define and manage notification rules which allows members of the network to receive notifications about specific events when they happen.

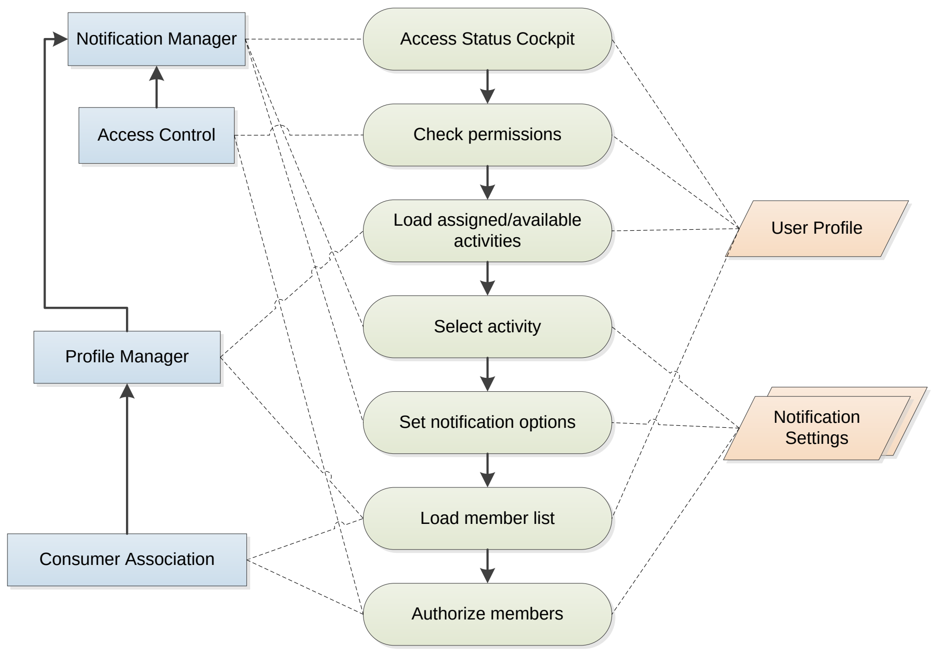
Use Cases which the diagram refer to:

MI-4 "Manage Notifications".

Activity Diagram:

The status cockpit in this activity description serves as the "overall" management instance managing the orchestration of the different tasks.

Regarding data objects, an activity profile corresponds with the activities a user wants to monitor, respectively informed in case of a specific event (e.g. data change, state change of an activity, etc.). A member profile allows the statistician/administrator to add an arbitrary number of participants which may be additionally informed in case of a specific event.



3.17 LOG SYSTEM ACTIVITY

Overview:

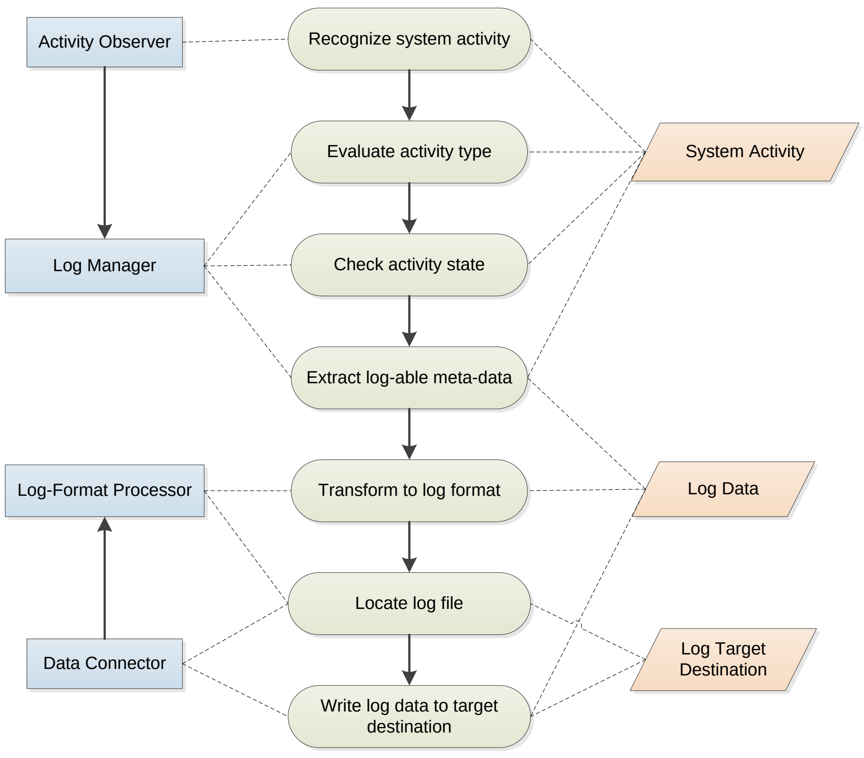
The Harmonise system shall be able to log certain activities conducted by Harmonise Participants. The collected data serves thereby as a basis for system-specific analyses.

Use Cases which the diagram refer to:

MI-5 "Log System Activity".

Activity Diagram:

System activity is the general description of observable operations executed by the system or a user. Candidates for such operations may be: search request, payment transaction, service request, etc.; In general one may distinguish long-term and short-term activities. Depending on the type an activity may have several states that can be adopted by the activity (e.g. pre-transaction phase or post-transaction phase); Depending on the type it may be useful to save the log-data (activity meta-data) in a specific format and to multiple/single target logs, in regard to context and further use (e.g. text mining, process minding); Log Target Destination refers to an abstract and general description of possible target log techniques such as databases or log-files, etc.


Activity Observer
Recognize system activity
Evaluate activity type
System Activity
Log Manager
Check activity state
Extract log-able meta-data
Log-Format Processor
Transform to log format
Log Data
Locate log file
Data Connector
Log Target Destination
Write log data to target destination

4 SOFTWARE INFRASTRUCTURE

This chapter introduces the major infrastructural software components which may be used for the implementation of the modules needed to set up the Harmonise integrated platform.

Since this platform will be based on the existing Harmonise solution, some of the components will be reused as they are or adapted to the new HarmoSearch needs. For instance the web portal will be again based on Liferay Portal Server.

On the other hand, for the new HarmoSearch specific components (i.e. Metasearch, Mapping Tool, Semantic Registry, Workflow Engine), a state of the art of the technologies which have been taken in consideration as possible candidates to be used for the implementation is presented. The final choice will be taken in the specific design phase (see deliverables D3.1 “Ontology for the query model”, D3.2 “Ontology for the registry model”, D4.1 “Semantic query – Query language specification”, D5.1 “Registry Requirements analysis”).

The following figure presents an overview of the software infrastructure. As usual, completely new software components are displayed in red while existing ones are displayed in yellow.

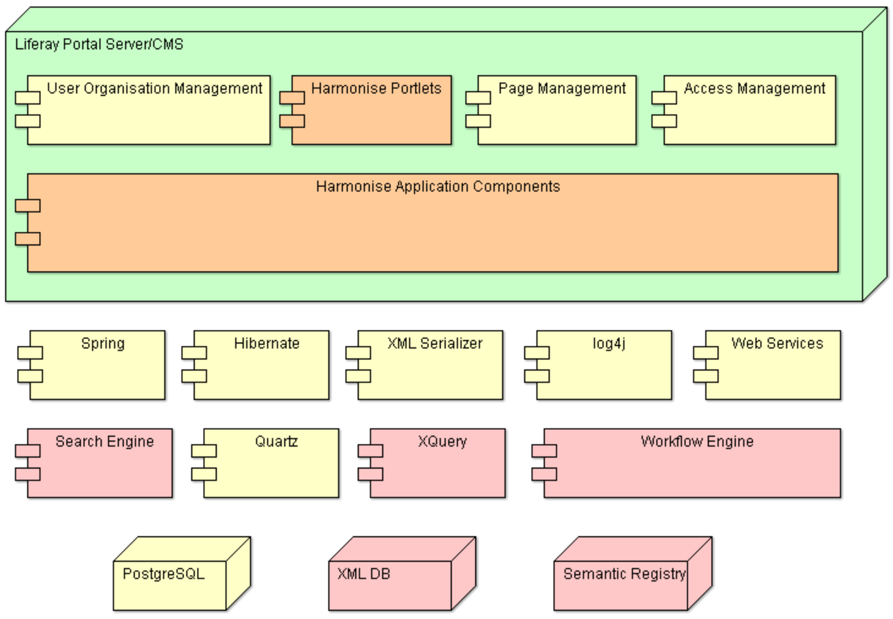


Figure 2 Infrastructural View

4.1 PORTAL SERVER

4.1.1 Liferay

Liferay is the leading open source portal service available today. The existing Harmonise system adopted Liferay as the basic infrastructure and integration platform as well as for providing its services through the web.

Liferay is characterized by the following features:

- **JSR 168 and JSR 286 compliant**. These are the portlet specification which have been standardized by a group of companies including Oracle, Bea, and Sun, whose goal is to define a common and interchangeable infrastructure for developing portal based solution. A portlet is a specific component of a web application which could have a Graphical User Interface and integrate on a web page by communicating with the other portlets. This standard allows defining web components which could be deployed on portal servers provided by different vendors.
- **User Management**. Liferay provides a rich way to manage users, organizations, communities, to define their roles and permissions. So it is exploited for supporting the HarmoSearch user management needs.
- **Theme Managemen**t. Specific user interface layouts can be built. This is useful to be able to support different communities and groups within HarmoSearch.
- **Social Components**, like wiki, forums, and blogs. This will be used to build the HarmoSearch knowledge base and share it among the members.

4.2 FRAMEWORK

4.2.1 Spring

The Spring Framework is an open source application framework for the Java platform.

The first version was written by Rod Johnson who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002. The framework was first released under the Apache 2.0 license in June 2003. The first milestone release, 1.0, was released in March 2004, with further milestone releases in September 2004 and March 2005. The Spring 1.2.6 framework won a Jolt productivity award and a JAX Innovation Award in 2006. Spring 2.0 was released in October 2006, and Spring 2.5 in November 2007. In December 2009 version 3.0 GA was released. The current version is 3.0.5.

The core features of the Spring Framework can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the Spring Framework does not impose any specific programming model, it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBean (EJB) model.

The Spring Framework comprises several modules that provide a range of services:

- Inversion of Control container: configuration of application components and lifecycle management of Java objects
- Aspect-oriented programming: enables implementation of cross-cutting routines
- Data access: working with relational database management systems on the Java platform using JDBC and object-relational mapping tools
- Transaction management: unifies several transaction management APIs and coordinates transactions for Java objects
- Model-view-controller: an HTTP and Servlet-based framework providing hooks for extension and customization
- Remote Access framework: configurative RPC-style export and import of Java objects over networks supporting RMI, CORBA and HTTP-based protocols including web services (SOAP)
- Convention-over-configuration: a rapid application development solution for Spring-based enterprise applications is offered in the Spring Roo module
- Batch processing: a framework for high-volume processing featuring reusable functions including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management
- Authentication and authorization: configurable security processes that support a range of standards, protocols, tools and practices via the Spring Security sub-project (formerly Acegi Security System for Spring).
- Remote Management: configurative exposure and management of Java objects for local or remote configuration via JMX
- Messaging: configurative registration of message listener objects for transparent message consumption from message queues via JMS, improvement of message sending over standard JMS APIs
- Testing: support classes for writing unit tests and integration tests

Central to the Spring Framework is its Inversion of Control container, which provides a consistent means of configuring and managing Java objects using callbacks. The container is responsible for managing object lifecycles: creating objects, calling initialization methods, and configuring objects by wiring them together.

Objects created by the container are also called Managed Objects or Beans. Typically, the container is configured by loading XML files containing Bean definitions which provide the information required to create the beans.

Objects can be obtained by means of Dependency lookup or Dependency injection. Dependency lookup is a pattern where a caller asks the container object for an object with a specific name or of a specific type. Dependency injection is a pattern where the container passes objects by name to other objects, via constructors, properties, or factory methods.

4.2.2 Hibernate

Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.

Hibernate is a free software that is distributed under the GNU Lesser General Public License.

Hibernate was started in 2001 by Gavin King as an alternative to using EJB2-style entity beans. Its mission back then was to simply offer better persistence capabilities than offered by EJB2 by simplifying the complexities and allowing for missing features. Early in 2003, the Hibernate development team began Hibernate2 releases which offered many significant improvements over the first release. JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers and worked with them in supporting Hibernate. As of 2010 the current version of Hibernate is Version 3.x. This version introduced new features like a new Interceptor/Callback architecture, user defined filters, and JDK 5.0 Annotations (Java's metadata feature). As of 2010 Hibernate 3 (version 3.5.0 and up) is a certified implementation of the Java Persistence API 2.0 specification via a wrapper for the Core module which provides conformity with the JSR 317 standard.

Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types). Hibernate also provides data query and retrieval facilities. Hibernate generates the SQL calls and attempts to relieve the developer from manual result set handling and object conversion and keep the application portable to all supported SQL databases with little performance overhead.

Mapping Java classes to database tables is accomplished through the configuration of an XML file or by using Java Annotations. When using an XML file, Hibernate can generate skeletal source code for the persistence classes. This is unnecessary when annotation is used. Hibernate can use the XML file or the annotation to maintain the database schema.

Facilities to arrange one-to-many and many-to-many relationships between classes are provided. In addition to managing association between objects, Hibernate can also manage reflexive associations where an object has a one-to-many relationship with other instances of its own type.

Hibernate supports the mapping of custom value types. This makes the following scenarios possible:

- Overriding the default SQL type that Hibernate chooses when mapping a column to a property.
- Mapping Java Enum to columns as if they were regular properties.
- Mapping a single property to multiple columns.

Hibernate provides transparent persistence for Plain Old Java Objects (POJOs). The only strict requirement for a persistent class is a no-argument constructor, not necessarily public. Proper behavior in some applications also requires special attention to the equals() and hashCode() methods.

Collections of data objects are typically stored in Java collection objects such as Set and List. Java generics, introduced in Java 5, are supported. Hibernate can be configured to lazy load associated collections. Lazy loading is the default as of Hibernate 3.

Related objects can be configured to cascade operations from one to the other. For example, a parent such as an Album object can be configured to cascade its save and/or delete operation to its child Track objects. This can reduce development time and ensure referential integrity. A dirty checking feature avoids unnecessary database write actions by performing SQL updates only on the modified fields of persistent objects.

Hibernate provides an SQL inspired language called Hibernate Query Language (HQL) which allows SQL-like queries to be written against Hibernate's data objects. Criteria Queries are provided as an object-oriented alternative to HQL.

Hibernate can be used both in standalone Java applications and in Java EE applications using servlets or EJB session beans. It can also be included as a feature in other programming languages. For example, Adobe integrated Hibernate into version 9 of ColdFusion (which runs on J2EE app servers) with an abstraction layer of new functions and syntax added into CFML.

4.2.3 XML Processing and Serialisation

XStream is a Java library to serialise objects to XML (or JSON) and back again. XStream uses reflection to discover the structure of the object graph to serialise at runtime, and doesn't require modifications to objects. It can serialise internal fields, including private and final, and supports non-public and inner classes.

XOM is a XML document object model for processing XML with Java that strives for correctness and simplicity.

dom4j is an open source Java library for working with XML, XPath and XSLT. It is compatible with DOM, SAX and JAXP standards.

4.3 WORKFLOW MANAGEMENT

4.3.1 BPMN 2.0

BPMN 2.0 is a standardized specification that defines a visualization and XML serialization of business processes, and can be extended (if necessary) to include more advanced features.

- It is a standard language, managed by the Object Management Group, and it is not vendor specific.
- It is based on XML, so it can be read/written almost on every platform and by every programming language.
- It has a graphical notation to represent the processes in a way similar to the UML sequence diagram.

One of the main goals of BPMN is to be intuitive for non-technical user and at the same time powerful in order to describe complex processes. Briefly, BPMN defines the following main categories of objects:

- Flow objects, which are the main describing elements within BPMN, and consist of three core elements:
    - Event: denotes something that happens and that influences the process. Events are graphically represented as circle. The most used events are:
        - Start event: the event that starts the process.
        - End event: the result of the process.
        - Intermediate event: represents something that happens between the start and end events.
    - Activity: describes the actions to be done. Activities are graphically represented as rounded corner rectangle. In the Activity category there are:
        - Task: a single unit of work, it is the elementary building block.
        - Sub-process: an object formed by tasks, events, gateways and sequence flows that hide business details.
        - Transaction: a particular kind of sub-process where all the activities included must be done as a whole or in case of failure none of them should be taken (so must be undone if already executed).
    - Gateway: determines a fork or merge of different paths depending on the conditions expressed. Gateways are graphically represented by a diamond.
- Connecting objects, which connect to each other different flow objects. They consist of three types:
    - Sequence flow: determines the order between the events, activities and gateways. It is represented as a solid arrow that connects the two objects involved in the sequence.
    - Message flow: specifies what messages flow across organisational boundaries. It is represented with a dashed line, an open circle at the start, and an open arrowhead at the end.
    - Association: associates an artifact or text to a flow object. It is represented with a dotted line.
- Swimlanes, which are a visual mechanism of organising and categorising activities, and in BPMN consist of two types:
    - Pool: represents the major participants in a process. A pool can be open (i.e., showing internal detail) when it is depicted as a large rectangle showing one or more lanes, or collapsed (i.e., hiding internal detail) when it is depicted as an empty rectangle stretching the width or height of the diagram.
    - Lane: organises and categorises activities within a pool according to function or role, and depicted as a rectangle stretching the width or height of the pool.
- Artifacts, which allow to bring some more information into the model/diagram. There are three pre-defined artifacts:
    - Data object: shows which data is required or produced in an activity.

- Group: group different activities. It is represented with a rounded-corner rectangle and dashed lines.
- Annotation: gives to the reader an understandable impression.

4.3.2 Activiti

Activiti is a workflow and Business Process Management (BPM) Platform for Java application. It's open-source, distributed under the Apache license, and it runs in any Java application, as a stand-alone one or on a server. Activiti is extremely lightweight but at same time provides powerful functionality such as the perfect integration with Spring.

The heart of the Activiti project is the Activiti Engine, a Java process engine that runs BPMN 2.0 process natively. The engine is easily configurable and it is simple to embed it in any Java application. Furthermore Activiti provides some useful utilities such as:

- Activiti Modeler: a web based editor that can be used to create BPMN 2.0 processes graphically;
- Activiti Probe: a web application that provides administration and monitoring capabilities to keep an Activiti Engine instance up and running;
- Activiti Explorer: a web application that provides access to the Activiti Engine runtime for all users of the system. It includes task management, viewing reports and process instance inspection.

4.3.3 jBPM

jBPM is a workflow engine written in Java that can execute processes described in many languages (e.g. BPMN, BPEL or its own process definition language jPDL). It is released under the LGPL license by the JBoss Community.

jBPM is a flexible Business Process Management (BPM) Suite. It makes the bridge between business analysts and developers. It offers process management features in a way that both business users and developers like it.

jBPM takes process descriptions as input. A process is composed of activities that are connected with transitions. Processes represent an execution flow. The graphical diagram of a process is used as the basis for the communication between non-technical users and developers. Each execution of a process definition is called a process instance. jBPM manages the process instances. Some activities, like sending an email or executing a script, are automatic. Other activities involve waiting for an external occurrence, such as a person completing a task or an application calling back with the results of a request. jBPM keeps track of the state of the process executions during those wait periods.

jBPM is based on a generic process engine, which is the foundation to support multiple process languages natively.

jBPM5 is the latest community version of the jBPM project. It is based on the BPMN 2.0 specification as the language for expressing business processes and supports the entire life cycle of the business process (from authoring through execution to monitoring and management).

The current jBPM5 snapshot offers open-source business process execution and management, including

- embeddable, lightweight Java process engine, supporting native BPMN 2.0 execution
- BPMN 2.0 process modelling in Eclipse (developers) and the web (business users)
- process collaboration, monitoring and management through the Guvnor repository and the web console
- human interaction using an independent WS-HT task service
- tight, powerful integration with business rules and event processing

4.4 SEARCH ENGINE

4.4.1 Lucene

Apache Lucene(TM) is a high-performance, full-featured text search engine library written entirely in Java. It is supported by the Apache Software Foundation and is released under the Apache Software License.

Lucene was originally written by Doug Cutting. It was initially available for download from its home at the SourceForge web site. It joined the Apache Software Foundation's Jakarta family of open source Java products in September 2001 and became its own top-level Apache project in February 2005. Until recently, it included a number of sub-projects, such as Lucene Java, Droids, Lucene.Net, Lucy, Mahout, Solr, Nutch, Open Relevance Project, PyLucene and Tika. Solr has been merged into the Lucene project itself and Mahout, Nutch and Tika have been moved to be independent top-level projects.

Lucene has been ported to other programming languages including Delphi, Perl, C#, C++, Python, Ruby and PHP.

While suitable for any application which requires full text indexing and searching capability, Lucene has been widely recognized for its utility in the implementation of Internet search engines and local, single-site searching.

At the core of Lucene's logical architecture is the idea of a document containing fields of text. This flexibility allows Lucene's API to be independent of the file format. Text from PDFs, HTML, Microsoft Word, and OpenDocument documents, as well as many others can all be indexed as long as their textual information can be extracted.

Lucene itself is just an indexing and search library and does not contain crawling and HTML parsing functionality. However, several projects extend Lucene's capability:

- Apache Nutch provides web crawling and HTML parsing
- Apache Solr – a fully-featured search server
- Compass – a Java Search Engine Framework
- ElasticSearch – A Distributed, Highly Available, RESTful Search Engine

4.4.2 Solr

Solr is an open source enterprise search platform from the Apache Lucene project. Its major features include powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document (e.g., Word, PDF) handling. Solr is highly scalable, providing distributed search and index replication, and it powers the search and navigation features of many of the world's largest internet sites.

Solr is written in Java and runs as a standalone full-text search server within a servlet container such as Apache Tomcat. Solr uses the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it easy to use from virtually any programming language. Solr's powerful external configuration allows it to be tailored to almost any type of application without Java coding, and it has an extensive plugin architecture when more advanced customization is required.

Apache Lucene and Apache Solr are both produced by the same ASF development team since the project merge in 2010. It is common to refer to the technology or products as Lucene/Solr or Solr/Lucene.

In 2004, Solr was created by Yonik Seeley at CNET Networks as an in-house project to add search capability for the company website. Yonik Seeley along with Grant Ingersoll and Erik Hatcher went on to launch Lucid Imagination a company providing commercial support, consulting and training for Apache Solr search technologies. In January 2006, CNET Networks decided to openly publish the source code by donating it to the Apache Software Foundation under the Lucene top-level project.[2] Like any new project at Apache Software Foundation it entered an incubation period which helped solve organizational, legal, and financial issues. In January 2007, Solr graduated from incubation status and grew steadily with accumulated features thereby attracting a robust community of users, contributors, and committers. Although quite new as a public project, it is already used for several high-traffic websites. In September 2008, Solr 1.3 was released with many enhancements including distributed search capabilities and performance enhancements among many others. November 2009 saw the release of Solr 1.4 This version introduces enhancements in indexing, searching and faceting along with many other improvements such as Rich Document processing (PDF, Word, HTML), Search Results clustering based on Carrot2 and also improved database integration. The release also features many additional plug-ins. In March 2010, the Lucene and Solr projects merged. Separate downloads will continue, but the products are now jointly developed by a single set of committers.

4.4.3 Nutch

Nutch is open source web-search software. It builds on Lucene and Solr for the search and index component, adding web-specifics, such as a crawler, a link-graph database, parsers for HTML and other document formats, etc.

Nutch originated with Doug Cutting, creator of both Lucene and Hadoop, and Mike Cafarella. In June, 2003, a successful 100-million-page demonstration system was developed. To meet the multimachine processing needs of the crawl and index tasks, the Nutch project has also implemented a MapReduce facility and a distributed file system. The two facilities have been spun out into their own subproject, called Hadoop. In January, 2005, Nutch joined the Apache Incubator, from which it graduated to become a subproject of Lucene in June of that same year. Since April, 2010, Nutch has been considered an independent, top level project of the Apache Software Foundation.

Nutch is coded entirely in the Java programming language, but data is written in language-independent formats. It has a highly modular architecture, allowing developers to create plug-ins for media-type parsing, data retrieval, querying and clustering.

The fetcher ("robot" or "web crawler") has been written from scratch specifically for this project.

Nutch can run on a single machine, but gains a lot of its strength from running in a Hadoop cluster

The system can be enhanced (e.g. other document formats can be parsed) using a plug-in mechanism.

Nutch installations typically operate at one of three scales: local file system, intranet, or whole web, so that it is possible to configure Nutch in order to satisfy different purposes. Although Nutch and Lucene could meet the same needs, indeed Nutch is a better fit for sites where you don't have direct access to the underlying data, or it comes from disparate sources.

The architecture of Nutch divides in two parts: the crawler and the searcher. The crawler fetches pages and turns them into an inverted index, which the searcher uses to answer users' search queries. Between these two parts is the index, that is, roughly speaking, the only one contact point.

- The crawler system builds and maintains several types of data structures, including the web database, a set of segments, and the index.
    - The database, or WebDB, is a specialized persistent data structure to store two types of entities: pages and links. A page represents a page on the Web, and is indexed by its URL and the MD5 hash of its contents. A link represents a link from one web page (the source) to another (the target).
    - A segment is a collection of pages fetched and indexed by the crawler in a single run.
    - The index is the inverted index of all of the pages the system has retrieved, and is created by merging all of the individual segment indexes.
- The search phase is quite straightforward: once installed Nutch web application, using whatever servlet container, it is enough to tell Nutch where to find the indexes and the segments generated by the crawler. Then, it is possible to connect to the Nutch home page and to start searching.

Nutch uses Lucene for its indexing, so all of the Lucene tools and APIs are available to interact with the generated index.

4.5 SCHEDULER

4.5.1 Quartz

Quartz is a full-featured, open source job scheduling service that can be integrated with, or used alongside virtually any Java EE or Java SE application. Quartz can be used to create simple or complex schedules for jobs, whose tasks are defined as standard Java components. The Quartz Scheduler includes many enterprise-class features, such as JTA transactions and clustering.

Jobs are scheduled to run when a given Trigger occurs. Triggers can be created with nearly any combination of the following directives:

- at a certain time of day (to the millisecond)
- on certain days of the week
- on certain days of the month
- on certain days of the year
- not on certain days listed within a registered Calendar (such as business holidays)
- repeated a specific number of times
- repeated until a specific time/date
- repeated indefinitely
- repeated with a delay interval

Jobs can be any Java class that implements the simple Job interface. Job class instances can be instantiated by Quartz, or by the application's framework. When a Trigger occurs, the scheduler notifies zero or more Java objects implementing the JobListener and TriggerListener interfaces (listeners can be simple Java objects, or EJBs, or JMS publishers, etc.). These listeners are also notified after the Job has executed. As Jobs are completed, they return a JobCompletionCode which informs the scheduler of success or failure. The JobCompletionCode can also instruct the scheduler of any actions it should take based on the success/fail code - such as immediate re-execution of the Job.

The design of Quartz includes a JobStore interface that can be implemented to provide various mechanisms for the storage of jobs. With the use of the included JDBCJobStore, all Jobs and Triggers configured as "non-volatile" are stored in a relational database via JDBC. With the use of the included RAMJobStore, all Jobs and Triggers are stored in RAM and therefore do not persist between program executions - but this has the advantage of not requiring an external database.

4.6 SEMANTIC REGISTRY

The semantic registry has the task of storing metadata about the content different data providers have available. This includes reasoning on data items and on the available mappings as well as reasoning on user configurations and created workflows in order to find suitable data providers for a given query by a given user.

Furthermore, not only the definition of available data but also the definition of interests for specific content items with respect to configured workflows and access rights must be supported. Finally, also the description and discovery of third party services to be included in the workflows must be possible.

These specific requirements cannot be directly fulfilled by existing registry implementations. However, an existing system will be used as a basis on top of which the extensions for the Harmosearch project are implemented.

The detailed requirements and state-of-the-art analysis for the semantic registry is done in deliverable D5.1 (Registry Requirements analysis Report). Currently there are two highly probably candidates for a base technology, the *OMAR* ebXML registry and the *FUSION Semantic Registry*.

4.6.1 OMAR

The Object, Metadata and Artifacts Registry (OMAR) is an implementation of the ebXML registry specification, supporting XML based business interactions. It provides a set of services which enables the sharing of content and metadata between different participants. It allows managing any content type and the standardised metadata that describe it.

OMAR offers several features that make it a promising candidate of a base technology for the Harmosearch Semantic Registry. Among these is a role based access control, facilities for the cataloguing XML content as well as content based event-notation. OMAR offers Java user interfaces as well as API access for all relevant user actions.

OMAR is built in Java as a web application running on an application server (Apache Tomcat is recommended). It needs a relational database system to operate. Besides Derby and HSQLDB which are shipped with the application, also PostgreSQL and Oracle databases have been tested. Compatibility with other database management systems needs to be checked.

OMAR is distributed as open-source software under the very liberal “freebxml License”, which makes no restrictions on deriving and selling software based on the OMAR registry.

4.6.2 FUSION

The *FUSION Semantic Registry* is a semantically-enhanced service registry. It is based on the UDDI[1] specification but adds machine understandable semantics for specifying and discovering services. Therefore, unlike its UDDI base, the FUSION Semantic Registry supports fully automated, and therefore effective, service discovery.

It was developed in the context of the IST Research Project "FUSION", funded by the European Commission in the 6th Framework Programme. , led by SAP AG and a

[1] Universal Description, Discovery and Integration; A standard for registering and locating web services. *http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm*

consortium consisting of 14 partners from five European countries (Germany, Poland, Greece, Hungary, Bulgaria).

The FUSION registry uses of SAWSDL[2] annotating the service interface descriptions. Furthermore, it makes use of OWL-DL[3] for describing service capabilities and reasoning.

FUSION is implemented in Java and runs as a standalone web application on a standard web application server (e.g., Apache Tomcat), using a UDDI compliant server (e.g., the open-source JUDDI[4] server implementation). The FUSION registry itself is released as open source software under the GPL v3.0 software license.

4.7 QUERY LANGUAGE

4.7.1 XQuery

XML Query Language – or simply XQuery – is a functional programming and query language for XML databases or collections of XML data. Its first version was developed by the XML Query working group of the W3C, with the syntax based on XSLT, SQL and C. For its data model and the function library it uses XPath and XML schema. XQuery became a W3C recommendation, and is Turing-complete.

XQuery is useful to extract single parts of large XML data collections and therefore to access XML files like databases (whilst XSLT should be used if complete XML documents have to be transformed).

In order to realize the (semi-) automatic mapping from and to the various components in the european tourism market, XQuery can be used to specify these mappings in structured and well-defined ways. Its application and usage in HarmoSearch can thus be observed in two different fields: sending a request to all other participants and getting responses back from them. When getting responses, XQuery can be used for mapping structures and values, while for sending requests (or queries) to other participants, the mapping or mediation of the query languages is a key aspect.

The mapping rules can be specified as rather simple XML documents, which define the mappings from internal data fields to the one in the central Harmonise schema (and vice-versa). This means, for each locally defined field name, the corresponding name of the field in the Harmonise data structure has to be specified, such that the content of the required fields can then be obtained. Additionally, with the same specification it is also possible to map the local fields to the ones of the central model for outgoing responses.

In addition to the structure, values might also have to be mapped or transformed between the various XML schemas. The problem, when using only the relatively

[2] Semantic Annotations for WSDL and XML Schema. *http://www.w3.org/TR/sawsdl/*

[3] OWL Web Ontology Language. *http://www.w3.org/TR/owl-features/*

[4] An open source Java implementation of the UDDI specification. *http://juddi.apache.org/*

simple structure mapping described previously, is that not only the field names vary among the participants, but additionally in some cases the data is not even represented in the same way. Therefore, reference lists have to be mapped too. These lists contain instructions to extract or compute the fields required for the Harmonise model from the local XML schema.

Another aspect to be considered is that the various participants might use different query languages, from which the need for a mediator or some semantic web mediation architecture arises. This leads to the conclusion that the search queries have to be mapped in some appropriate way. One possible attempt to face this problem would be to use query by example. In this case, the query is not represented by text like in SQL, but by a table structure. Thus, it is often also called a graphical query language. The reason why this could be of interest in the HarmoSearch case is that in this case the query language could be standardized for the whole system, independent of the kind of database each participant is actually using. In this context, XQuery could be used to represent the tabular structure, i.e. in fact some XML structure which could also be interpreted as being a table – containing only the fields or logical expressions that are of interest for a specific query. A query consisting of a concatenation of AND-conditions could be subdivided in the single expressions, which would then be the various elements in the XQuery statement. In this way, how to create queries and interpret them could be unified, while each participant could still keep its own query language.

4.8 LOGGING

4.8.1 log4j

Log4j is an open-source Java-based logging API that is widely used among the open-source community. It has been developed under the Jakarta Apache project and has been released in 1999 under the Apache License. Since then, several open-source and industrial projects used log4j as their logging API of choice. Popular and well known open-source projects including log4j are the Hibernate and JBoss initiative. Due to its success, log4j can be assumed as the de-facto standard for logging Java applications. Over time, log4j has been ported to other programming languages such as C/C++/C#, Perl, PHP, and Python. In addition log4j influenced the further development of related logging APIs. Its principles and core concepts served as a basis for logging frameworks such as the Java Logging API, SLF4j, or Logback.

The main goal of the log4j API is to allow software developers to investigate and record the behaviour of the underlying application by means of log data. Thereby, log statements are directly written and included within the source code and assigned to different levels of priority. To provide an adequate, flexible, and clean solution log4j is built upon three major components: loggers, appenders, and layouts.

Loggers serve as centres for capturing relevant log data. They observe and evaluate log statements that might occur during the execution of an application and generate appropriate log requests. Log requests are the output holding the concrete log data. They are forwarded to appropriate log destinations in order to store the relevant system information. Log statements are always assigned to a certain priority level in order to indicate their level of importance. For example, a severe exception causing the application to quit has a higher relevance than a user notification or warning. By default log4j supports 5 different priority levels (listed ascending starting with the lowest level): *DEBUG*, *INFO*, *WARN*, *ERROR*, and *FATAL*. According to the priority level of the log statement the logger generates a log request subsuming the relevant log information. In addition a logger may also be assigned to a priority level. As a consequence the logger will only accept log statements with a priority level equal to or greater than its own.

The appender component represents an interface to a specific log destination where incoming log requests are stored. Possible log destinations are usually log files or databases. Again, log4j supports many different destinations per default. The most important built-in appenders are: the *JDBC appender*, the *SMTP appender*, and the *socket appender*. In some cases it makes sense to save log requests depending on their priority level to destinations – e.g. notifications to a log file and fatal errors to a database for further analysis. Thus, log4j supports the concept of appender additivity, which allows loggers to have multiple appenders. In such a case the logger decides – based on the priority level of the incoming log statement – which appender will be used to store the resulting log request.

The third component of the log4j API is the layout component. Layouts are used to customize the output of a log request. This includes both the textual representation (e.g. HTML, XML strings, etc.), as well as the content that should be stored. The latter comprises the functionality to constrain the data to a certain degree of detail. For example, in some scenarios it makes sense to enrich the log information by additional data such as execution time, source class, priority level, the line number from where the log message originated, etc. However, other cases require reducing the level of detail by excluding certain information from the log request in order to form an adequate and process-able result, which may serve as an input for further analysis or debugging issues. Useful layouts that are shipped with log4j are for example the DataLayout, the HTMLLayout, the XMLLayout and the PatternLayout.

Log4j can be configured programmatically or by external configuration files. The latter is of special interest to large and complex projects. In such a case it makes sense to outsource logging configuration to an external file, since it reduces the initialization effort to a minimum. As a consequence the logging overhead in the source code can be minimized and the code can be kept nice and clean. Log4j configuration files are implemented either as XML files or Java property files. Both can be created and built by a text editor of choice and are not bound to specific development tools or libraries. The configuration is kept very flexible and allows the developer to fully customize the logging mechanism by – e.g. specifying the priority level or type of an appender; number and type of appenders assigned to a specific logger, etc.

4.9 WEB SERVICES

4.9.1 JAX-WS

The Java API for XML Web Services (JAX-WS) is a Java programming language API for creating web services. It is part of the Java EE platform from Sun Microsystems. Like the other Java EE APIs, JAX-WS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints. It is part of the Java Web Services Development Pack.

The Reference Implementation of JAX-WS is developed as an open source project and is part of project GlassFish, an open source Java EE application server. It is called JAX-WS RI (For Reference Implementation) and is said to be production quality implementation (contrary to the former Reference Implementation being a proof of concept). This Reference Implementation is now part of the Metro distribution.

JAX-WS also is one of the foundations of WSIT.

JAX-WS 2.0 replaced the JAX-RPC API in Java Platform, Enterprise Edition 5. The name change reflected the move away from RPC-style and toward document-style web services.

4.9.2 Spring WS

Spring Web Services (Spring-WS) is a product of the Spring community focused on creating document-driven Web services. Spring Web Services aims to facilitate contract-first SOAP service development, allowing for the creation of flexible web services using one of the many ways to manipulate XML payloads.

The key features of Spring Web services are:

- Powerful mappings. Incoming XML requests can be distributed to any object, depending on message payload, SOAP Action header, or an XPath expression.
- XML API support. Incoming XML messages can be handled not only with standard JAXP APIs such as DOM, SAX, and StAX, but also JDOM, dom4j, XOM, or even marshalling technologies.
- Flexible XML Marshalling. The Object/XML Mapping module in the Spring Web Services distribution supports JAXB 1 and 2, Castor, XMLBeans, JiBX, and XStream. And because it is a separate module, it can be used in non-Web services code as well.
- Spring-WS uses Spring application contexts for all configuration, which should help Spring developers get up-to-speed nice and quickly. Also, the architecture of Spring-WS resembles that of Spring-MVC.
- WS-Security support. WS-Security allows signing SOAP messages, encrypting and decrypting them, or authenticating against them. The WS-Security implementation of Spring Web Services provides integration with Acegi Security too.
- Built by Maven.
- Apache license.

4.10 REPOSITORIES

4.10.1 PostgreSQL

PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS). It is released under an MIT-style license and is thus free and open source software. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation.

An enterprise class database, PostgreSQL boasts sophisticated features such as Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, asynchronous replication, nested transactions (savepoints), online/hot backups, a sophisticated query planner/optimizer, and write ahead logging for fault tolerance. It supports international character sets, multibyte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. It is highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate.

PostgreSQL prides itself in standards compliance. Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard. It has full support for sub-queries (including sub-selects in the FROM clause), read-committed and serializable transaction isolation levels. While PostgreSQL has a fully relational system catalog which itself supports multiple schemas per database, its catalogue is also accessible through the Information Schema as defined in the SQL standard.

Data integrity features include (compound) primary keys, foreign keys with restricting and cascading updates/deletes, check constraints, unique constraints, and not null constraints.

It also has a host of extensions and advanced features. Among the conveniences are auto-increment columns through sequences, and LIMIT/OFFSET allowing the return of partial result sets. PostgreSQL supports compound, unique, partial, and functional indexes which can use any of its B-tree, R-tree, hash, or GiST storage methods. Other advanced features include table inheritance, a rules systems, and database events.

PostgreSQL runs stored procedures in more than a dozen programming languages, including Java, Perl, Python, Ruby, Tcl, C/C++, and its own PL/pgSQL, which is similar to Oracle's PL/SQL. Included with its standard function library are hundreds of built-in functions that range from basic math and string operations to cryptography and Oracle compatibility. Triggers and stored procedures can be written in C and loaded into the database as a library, allowing great flexibility in extending its capabilities. Similarly, PostgreSQL includes a framework that allows developers to define and create their own custom data types along with supporting functions and operators that define their behaviour. As a result, a host of advanced data types have been created that range from geometric and spatial primitives to network addresses to even ISBN/ISSN (International Standard Book Number/International Standard Serial Number) data types, all of which can be optionally added to the system.

Just as there are many procedure languages supported by PostgreSQL, there are also many library interfaces as well, allowing various languages both compiled and interpreted to interface with PostgreSQL. There are interfaces for Java (JDBC), ODBC, Perl, Python, Ruby, C, C++, PHP, Lisp, Scheme, and Qt just to name a few.

4.10.2 Exist

eXist-db is an open source database management system built using XML technology. It stores XML data according to the XML data model and features efficient, index-based XQuery processing.

eXist allows software developers to persist XML data without writing extensive middleware. eXist follows and extends many W3C XML standards such as XQuery. eXist also supports REST interfaces for interfacing with AJAX-type web forms. Applications such as XForms may save their data by using just a few lines of code. The WebDAV interface to eXist allows users to "drag and drop" XML files directly into the eXist database. Because eXist automatically indexes documents using a keyword indexing system it is very easy to create high-performance document search systems with eXist.

eXist was created in 2000 by Wolfgang Meier who still is the lead developer as of 2010. In September 2006, it reached version 1.0 and 1.1 (new numbering scheme). Current maintenance activities are on the 1.4.x versions and new developments are on the 1.5dev version that will be released as 1.6.0.

eXist-db supports many (web) technology standards making it an excellent platform for developing web based applications:

- XQuery 1.0 / XPath 2.0 / XSLT 1.0 (using Apache Xalan) or XSLT 2.0 (optional with Saxon)
- HTTP interfaces: REST, WebDAV, SOAP, XMLRPC, Atom Publishing Protocol
- XML database specific: XMLDB, XUpdate, XQuery update extensions (to be aligned with the new XQuery Update Facility 1.0)

The 1.4 version adds a new full text index based on Apache Lucene, a lightweight URL rewriting and MVC framework as well as support for XProc. Most important, the XQuery engine has seen a major redesign, resulting in improved performance.

eXist-db is highly compliant with the XQuery standard (current XQTS score is 99.4%). The query engine is extensible and features a large collection of XQuery Function Modules.

eXist-db provides a powerful environment for the development of web applications based on XQuery and related standards. Entire web applications can be written in XQuery, using XSLT, XHTML, CSS and Javascript (for AJAX functionality). XQuery server pages can be executed from the filesystem or stored in the database.

4.10.3 Oracle Berkeley DB

Berkeley DB (BDB) is a computer software library that provides a high-performance embedded database for key/value data. Berkeley DB is a programmatic software library written in C with API bindings for C++, PHP, Java, Perl, Python, Ruby, Tcl, Smalltalk, and most other programming languages. BDB stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key. Berkeley DB is not a relational database. BDB can support thousands of simultaneous threads of control or concurrent processes manipulating databases as large as 256 terabytes, on a wide variety of operating systems including most Unix-like and Windows systems, and real-time operating systems. Berkeley DB is also used as the common name for three distinct products; Oracle Berkeley DB, Berkeley DB Java Edition, and Berkeley DB XML. The first is the traditional Berkeley DB, written in C. Berkeley DB Java Edition (JE) is a pure Java database. Its design resembles that of Berkeley DB without replicating it exactly, and has a feature set that includes many of those found in the traditional Berkeley DB and others that are specific to the Java Edition. Since it is written in pure Java, no native code is required. It has a log structured storage architecture, which gives it different performance and concurrency characteristics. Three APIs are available—a Direct Persistence Layer which is "Plain Old Java Objects" (POJO); one which is based on the Java Collections Framework (an object persistence approach); and one based on the traditional Berkeley DB API. The Berkeley DB Java Edition High Availability option (Replication) is available. Note that traditional Berkeley DB also supports a Java API, but it does so via JNI and thus requires an installed native library. The Berkeley DB XML database specializes in the storage of XML documents, supporting XQuery via XQilla. It is implemented as an additional layer on top of (a legacy version of) Berkeley DB and the Xerces library. DB XML is written in C++ and supports multiple language bindings, including C++, Java (via JNI), Perl and Python.

Berkeley DB originated at the University of California, Berkeley as part of the transition (1986 to 1994) from 4.3BSD to 4.4BSD and of the effort to remove AT&T-encumbered code. The first code, due to Seltzer and Yigit, attempted to create a disk hash table that performed better than any of the existing Dbm libraries. In 1996 Netscape requested that the authors of Berkeley DB improve and extend the library, then at version 1.86, to suit Netscape's requirements for an LDAP server and for use in the Netscape browser. That request led to the creation of Sleepycat Software. This company was acquired by Oracle Corporation in February 2006, which continues to develop and sell Berkeley DB.

Since its initial release, Berkeley DB has gone through various versions. Each major release cycle has introduced a single new major feature generally layering on top of the earlier features to add functionality to the product. The 1.x releases focused on managing key/value data storage and are referred to as "Data Store" (DS). The 2.x releases added a locking system enabling concurrent access to data. This is what is known as "Concurrent Data Store" (CDS). The 3.x releases added a logging system for transactions and recovery, called "Transactional Data Store" (TDS). The 4.x releases added the ability to replicate log records and create a distributed highly available single-master multi-replica database. This is called the "High Availability" (HA) feature set. Berkeley DB's evolution has sometimes led to minor API changes or log format changes, but very rarely have database formats changed. Berkeley DB HA supports online upgrades from one version to the next by maintaining the ability to read and apply the prior release's log records.

The FreeBSD and OpenBSD operating system continue to use Berkeley DB 1.8x for compatibility reasons; Linux-based operating systems commonly include several versions to accommodate for applications still using older interfaces/files.

Berkeley DB is redistributed under the Sleepycat Public License, which is an OSI-approved open source license as well as an FSF-approved free software licence. The product ships with complete source code, build script, test suite, and documentation. The code quality and general utility along with the licensing terms have led to its use in a multitude of free and open source software. Those who do not wish to abide by the terms of the Sleepycat Public License have the option of purchasing another proprietary license for redistribution from Oracle Corporation. This technique is called dual licensing.

Berkeley DB includes compatibility interfaces for some historic Unix database libraries: dbm, ndbm and hsearch (a System V library for creating in-memory hash tables).

Berkeley DB has an architecture notably simpler than that of other database systems like Microsoft SQL Server and Oracle. For example, like SQLite, it does not provide support for network access — programs access the database using in-process API calls. Oracle added support for SQL in 11g R2 release based on the popular SQLite API by including a version of SQLite in Berkeley DB. Consequently, as is common with SQLite, there is now 3rd Party support for PL/SQL in Berkeley DB through Metatranz StepSqlite.

A program accessing the database is free to decide how the data is to be stored in a record. Berkeley DB puts no constraints on the record's data. The record and its key can both be up to four gigabytes long.

Despite having a simple architecture, Berkeley DB supports many advanced database features such as ACID transactions, fine-grained locking, hot backups and replication.

5 COMPONENT VIEW

This section provides a detailed view on how the components identified in the logical view will be implemented considering the chosen infrastructural components.

Two different perspectives are provided:

- The Portlet Component View, which shows the GUI components, the related core components and their relationship;
- The Service Component View, which focuses on the core components, the exposed services, the external services used and how they are related.

5.1 PORTLET COMPONENT VIEW

This view shows the components which will implement the User Interface of the system. Since a portal based architecture has been chosen, each GUI component is identified by a portlet, a specific GUI component having a well defined role and interacting with the underlying services to support the necessary interactions with the users.

The diagram of Figure 3 shows the main portlets and how they are related with the core components. New portlets are shown in red, while existing portlet are identified in yellow. Core components' descriptions can be found in the logical view diagram of paragraph 2.1. Only the components which need a GUI are included in the following figure.

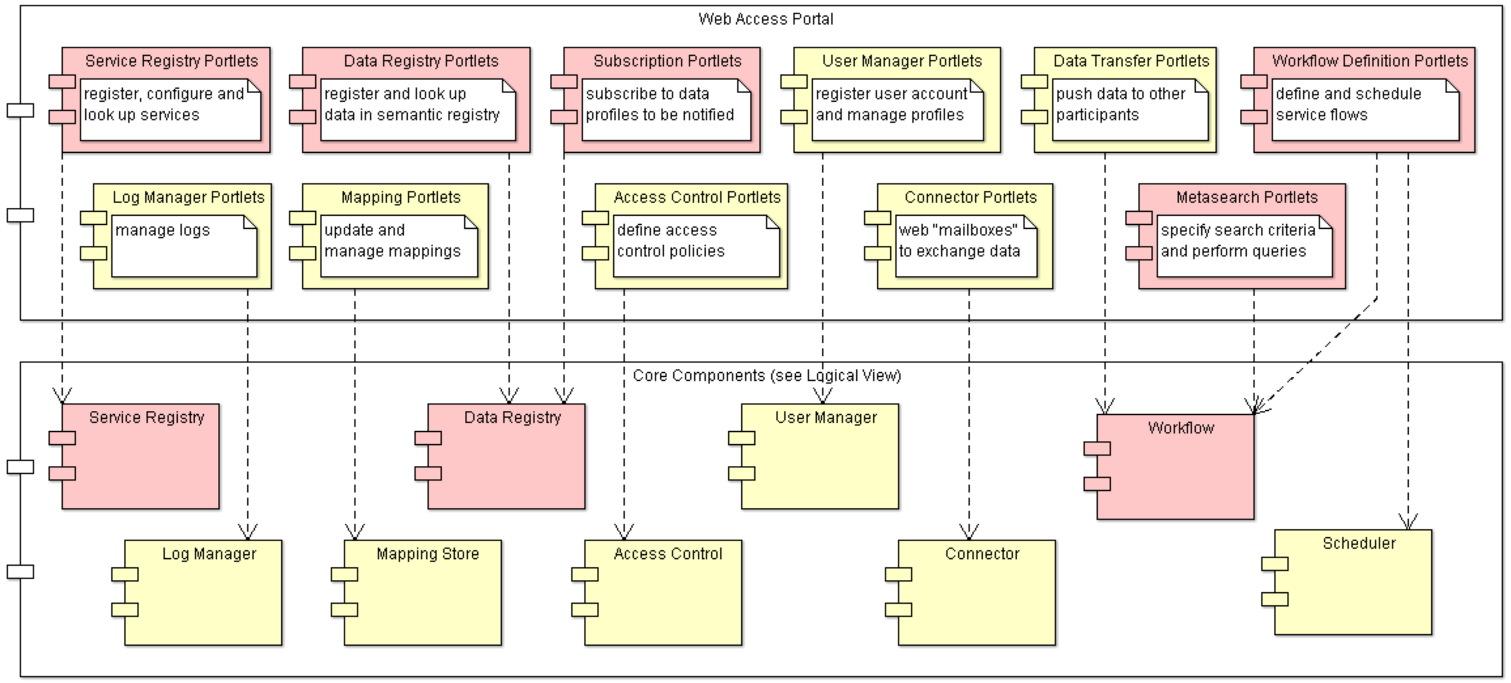

Web Access Portal
Service Registry Portlets
register, configure and look up services
Data Registry Portlets
register and look up data in semantic registry
Subscription Portlets
subscribe to data profiles to be notified
User Manager Portlets
register user account and manage profiles
Data Transfer Portlets
push data to other participants
Workflow Definition Portlets
define and schedule service flows
Log Manager Portlets
manage logs
Mapping Portlets
update and manage mappings
Access Control Portlets
define access control policies
Connector Portlets
web "mailboxes" to exchange data
Metasearch Portlets
specify search criteria and perform queries
Core Components (see Logical View)
Service Registry
Data Registry
User Manager
Workflow
Log Manager
Mapping Store
Access Control
Connector
Scheduler

Figure 3 Portlet Component View

5.1.1 User Manager Portlets

The *User Manager Portlets* allow managing organisations and users, thus providing the user interface to:

- **Register a new organisation** or edit its settings.
- **Register new users which are administrators of the organisation**, thus in charge of configuring the services offered or consumed by that organisation.
- **Register new users belonging to the organisation**, who are authorized to act on behalf of the organisation.

It will exploit the user management Liferay features, which allow defining and assigning roles to users and managing permission in a hierarchical way.

5.1.2 Access Control Portlets

The *Access Control Portlets* provide the following user interfaces:

- **Access control policies configuration**. It offers the possibility to add, edit, delete and configure access control policies to restrict access to data and services, by specifying the data or service to be protected, the participants which are allowed to access the data or to use the service and the constraints under which access is allowed (e.g. time period or number of times). In particular it allows to:
    - Configure the access rights for each data provider in different granularities (e.g., for specific Harmonise participants or a specific group of participants). See 5.1.7, "**Fehler! Verweisquelle konnte nicht gefunden werden.**";
    - Configure the access rights for each service in different granularities (e.g., for specific Harmonise participants or a specific group of participants). See 5.1.8, "Service Registry Portlets".
- **Access control policies visualisation**. It provides capacity to view all the access control policies currently active.

5.1.3 Log Manager Portlets

The *Log Manager Portlets* provide the following user interfaces:

- **System event monitoring configuration**. Allows the user to edit the current monitoring configuration. It offers the possibility to add or delete registered system activities.
- **Visualization of current monitoring status**. Describes the possibility to list actively monitored system activities ordered by type or name. The activities are listed in tabular form.
- **Registration of new system activities**. Provides the possibility to define and register new system activity types. A new system activity may be described by a specific workflow including necessary parameters and services.

5.1.4 Workflow Definition Portlets

The *Workflow Definition Portlets* provide the following user interfaces:

- **Workflow editing**. If the current user has the appropriate user rights, it offers the possibility to add, edit, delete and configure workflow processes. A workflow may include the set of services to be executed, a recipient (e.g. whom to send data to or whom to query) and other parameters (e.g. type of data to be pushed or to be queried).
- **Workflow monitoring**. To obtain information on the workflows currently in execution and on their status.
- **Workflow instantiation and execution**. To select a specific workflow and schedule its execution. Each participant can schedule a job specifying the time when the workflow has to be triggered, e.g. every day, every week, every month, etc.

5.1.5 Data Transfer Portlets

The *Data Transfer Portlets* provide the following user interfaces:

- **Possibility to push data** to other participants in the Harmonise network. The interface supports the data transfer definition by means of a form which allows to select the data to be pushed and the recipient, e.g. whom to send data to.

5.1.6 Metasearch Portlets

The *Metasearch Portlets* provide the following user interfaces:

- **Possibility to specify search criteria** to perform queries on data instances available in the Harmonise network. The interface supports the query definition by means of a dynamic form which allows selecting the data type to be queried and the definition of the search criteria by combining attributes to be queried, operators and values to be matched. According to the attribute type, the interface supports the user to enter the correct values (e.g. for date fields, it allows to specify only dates). For enumerated values, the interface exploits the available metadata to support the selection of the values among the ones supported for that specific field.
- **Display of the result set**. The items found are shown as a list, in a paginated way. Items can be then sorted according to specific sort criteria. Slow queries could show the results asynchronously as they become available.

5.1.7 Data Registry Portlets

The *Data Registry Portlets* provide user interfaces to:

- **Register as a data provider**. This means that after the normal registration for the Harmonise participant and the upload of appropriate mappings, the content of the data provider is described in a semantic way. The description process is aided by the analysis of the uploaded mapping, which provides a first impression of the available data. Further explication of what kind of content the data provider can supply is done through dynamic forms similar to the composition of a search query in the metasearch portlets (see 5.1.6, "Metasearch Portlets"). As with a search query, the participant can provide restrictions to describe what kind of data is available.

- **Configure the connection** for providing data access, normally by implementing a Harmonise data connector providing web-service access to the data.
- **Discover data providers** in order to configure access control or to select specific data providers when configuring a workflow (see 5.1.2, "Access Control Portlets" and 5.1.4, "Workflow Definition Portlets").

5.1.8 Service Registry Portlets

The *Service Registry Portlets* provide user interfaces to:

- **Register a service** in the Harmonise network, thus creating a new service, describing its effect textually and by means of a description of the required input and the returned output data.
- **Configure the service** connection, i.e., in which form (normally as web-service) the service is provided and what the parameters (e.g., web-service URL) are.
- **Discover services**, especially by description and name. This allows Harmonise participants to discover appropriate services in order to use them in specific workflows (see 5.1.4, "Workflow Definition Portlets").

5.1.9 Subscription Portlets

The *Subscription Portlets* provide the following user interfaces:

- **Data subscription**. It offers the possibility to subscribe to data profiles in order to be notified if new data are available which may be of interest for the user. Participants can add, edit, delete subscriptions and specify the actions to be triggered if new data or updates to data corresponding to the data profile they have subscribed to are published in the Data Registry. Actions may include e.g. sending a notification; execute the process which allows getting data from the service provider and sending them to the subscriber, etc.
- **Visualisation of the subscribed data**. Possibility to view all the data subscriptions currently active.

5.1.10 Mapping Portlets

The primarily task of the *Mapping Portlets* is to provide a view on the registered mappings (registration actually takes place using the Mapping Tool). In addition, it may also provide editing capabilities in order to configure and manage mappings. The main editing capabilities are:

- **Functions to manage the lifecycle of uploaded mappings**, such as activation and de-activation of mapping versions as well as deprecating and removing of mapping versions.

- **Assignment of SW-components** to mappings that perform the actual transformation such as a particular XSLT Processor.

5.1.11 Connector Portlets

Connector Portlets allow each participant to interact with the system. They implement mailboxes that allow participants to send data to other participants, receive data from other participants or receive data by querying the harmonise network. In particular they offer:

- **Possibility to store uploaded data** and to download data received by other participants. These GUIs follow an inbox/outbox/sent-box scheme: the outbox stores information on data which are in the process of being uploaded, the sent-box contains information on data which has already been uploaded and the inbox allows downloading content received from other participants.
- **Possibility to receive the results** of a distributed query as a file. This GUI allows downloading the results of the queries performed by the participant.

5.2 SERVICES COMPONENT VIEW

This section focuses on the core components of the system, thus the main building blocks implementing the business logic.

Their interactions, i.e. how they interact with external services and the interfaces they expose to the outside world, are highlighted in the diagram of Figure 4. New services are shown in red, while existing services are identified in yellow. Core components' descriptions can be found in the logical view diagram of paragraph 2.1. In the following figure the Service Connector logical component is split in the specific connectors to the most important external services that could be possibly plugged in.

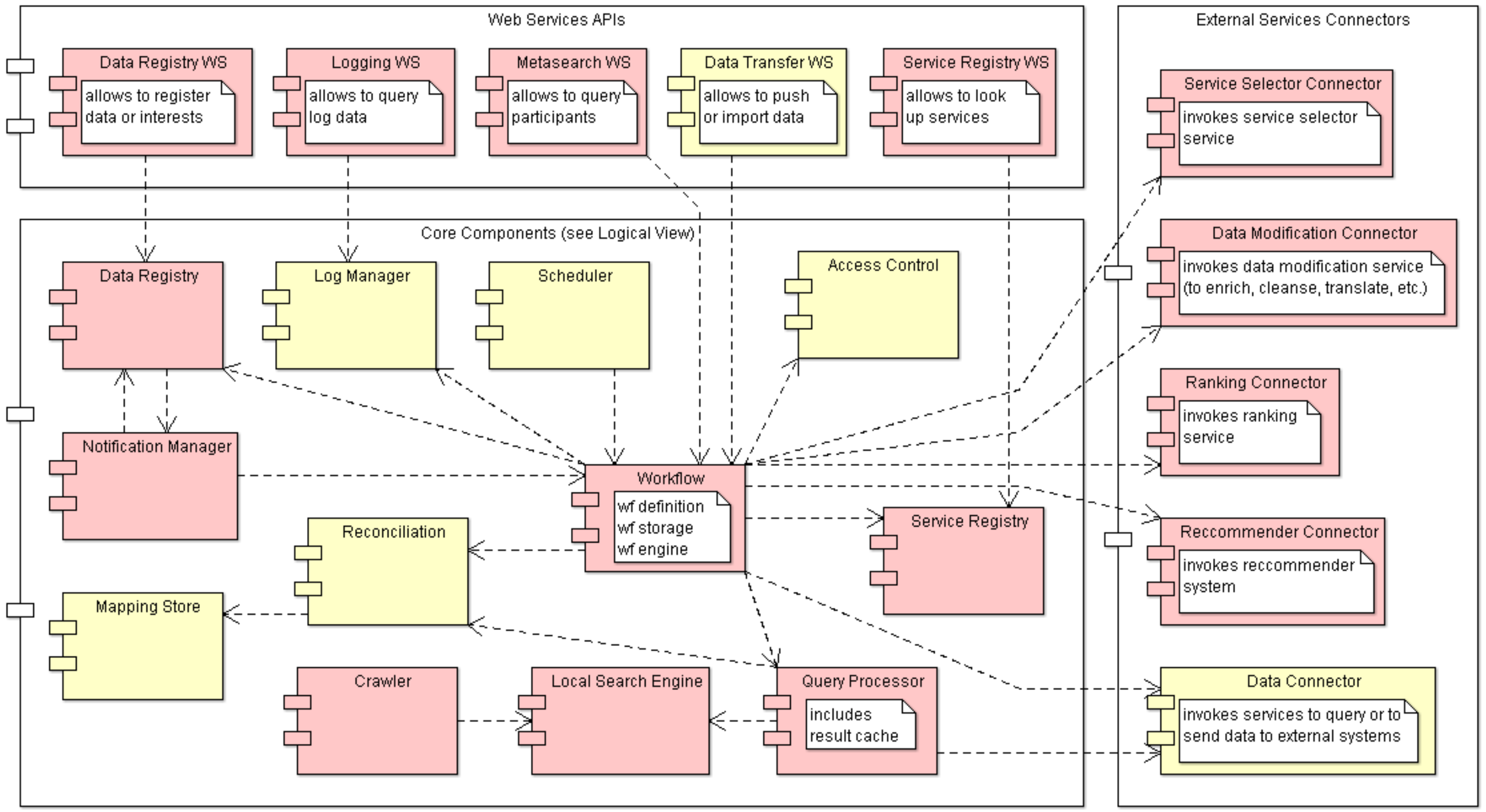

Web Services APIs
Data Registry WS
allows to register data or interests
Logging WS
allows to query log data
Metasearch WS
allows to query participants
Data Transfer WS
allows to push or import data
Service Registry WS
allows to look up services
Core Components (see Logical View)
Data Registry
Log Manager
Scheduler
Access Control
Notification Manager
Workflow
wf definition
wf storage
wf engine
Service Registry
Reconciliation
Mapping Store
Crawler
Local Search Engine
Query Processor
includes result cache
External Services Connectors
Service Selector Connector
invokes service selector service
Data Modification Connector
invokes data modification service (to enrich, cleanse, translate, etc.)
Ranking Connector
invokes ranking service
Reccommender Connector
invokes reccommender system
Data Connector
invokes services to query or to send data to external systems

Figure 4 Services Component View

Here below is presented a more detailed specification of each core component, including implementation approach, interfaces exposed and interfaces used of other components. It is not the aim of this section to give a full specification of each interface, so only some examples of possible methods are outlined. A full specification will be provided in the following project deliverables (D3.1 "Ontology for the query model", D3.2 "Ontology for the registry model", D4.1 "Semantic query – Query language specification", D5.1 "Registry Requirements analysis").

5.2.1 User Manager

The *User Manager Component* is to manage users and organizations. It is implemented by exploiting the Liferay user management component.

Interfaces exposed:

- *UserManager*
  - ***login***. Allows to log in a given user (by username and password)
  - ***getCurrentUser***. Returns a data structure representing current user properties
  - ***updateUser***. Allows updating properties about the current user.
  - ***getRoles***. Returns the roles of the current user. Roles are used by the services to understand if a given user can access or not certain functionality.
  - ***getCurrentUserOrganization***. Returns the organization(s) to which the current user is associated.
  - ***getOrganizations***. Returns the list of organizations registered in the system.

The component does not depend on other components.

5.2.2 Access Control

The *Access Control component* is used to allow the configuration of Access Control policies (ACP). An ACP restricts or accepts access to specific data or services. It will be implemented as a Java module.

Interfaces exposed:

- *AccessControlManager.* Is used to manage access control policies. Therefore, the main functionality allows access to certain policies, update policies, and to create and assign new policies to a given service.
  - ***getAccessControlPoliciesForService***. Returns all access control policies that have been committed to a given service.
  - ***getAccessControlPoliciesForUser***. To receive all access control policies which have been specified by the user.
  - ***updateAccessControlPolicy***. To update an existing access control policy.
  - ***createAccessControlPolicy***. To create a new access control policy.

- **_isAccessableByServiceConsumer_**. Determines if a given service may be accessed by a specific user. The user is identified by username and password.

5.2.3 Log Manager

The *Log Manager component* is in charge of monitoring certain system activities that reflect the use of the Harmonise system.

The *Log Manager component* will be implemented using the popular open-source logging framework log4j.

Interfaces exposed:

The component exposes the following primary interfaces:

- *LogManager*. The main functionality of the LogManager interface is to allow access to certain log data, save this data and configure the type of system event which should be monitored.
    - **_extractLogData_**. Extracts relevant meta-data out of a given system activity.
    - **_storeLogData_**. Writes the gathered log information to the target log-destination.
    - **_addSystemActivity_**. Adds a new observable system activity to the monitoring pool.
    - **_removeSystemActivity_**. Un-registers a given system activity.
    - **_prepareLogData_**. Prepares the meta-data extracted from an observed system activity for logging. The format depends on the target log-destination.
    - **_getLogDestination_**. Returns the path to the target log-destination.
    - **_setLogDestination_**. Sets the target log-destination.
    - **_getLogData_**. Returns a result set including log data that corresponds to given search parameters.
- *LogManagerService [WebService]*. Web Service API which allows looking up log data.

5.2.4 Workflow

The role of the *Workflow Component* is to allow the definition of specific processes and to control their execution.

It is implemented by exploiting the JBPM/Activiti Workflow engine technology, described in the infrastructural view.

Interfaces exposed:

- *WorkflowStorage*. It allows loading new workflows, editing their definition and configuration, deleting them and obtaining a list of defined workflows (considering the current user capabilities).

- *WorkflowEngine*. It allows starting a new process, checking its status, and killing one process.
- *WorkflowMetasearchService [WebService]*. Web Service API which allows running distributed queries among the participant in the Harmonise network.
- *WorkflowDataTransferService [WebService]*. Web Service API which allows pushing content for a well defined recipient or importing content from selected data sources.

Interfaces used:

- *ServiceRegistry*. To obtain the specifications on how to invoke an external service.
- *AccessControlManager*. To checks whether a participant has the rights to execute a service.
- *DataRegistry*. To find appropriate data providers, to look up the provider's configuration and to add additional information to the data registered.
- *MetasearchQueryProcessor*. To run distributed queries.
- *ReconciliationEngine*. To transform data among different data formats.
- *DataConnector*. To push data from one participant to another one.
- *ServiceConnector*. To perform external services to be executed by the components participating in the workflow.
- *LogManager*. To log relevant information.
- *UserManager*. To obtain user profile information.

5.2.5 Scheduler

The role of the *Scheduler Component* is to schedule specific workflow processes to be automatically executed at defined interval times.

It is implemented by exploiting the Quartz job scheduling service, described in the infrastructural view.

Interfaces exposed:

- *SchedulerManager*. It allows scheduling new jobs, editing their definition and configuration, deleting them and obtaining a list of scheduled jobs.

Interfaces used:

- *WorkflowEngine*. To run the scheduled process.

5.2.6 Reconciliation

The *Reconciliation Engine Component* is responsible for the harmonisation of queries and query results.

Interfaces exposed:

- *ReconciliationEngine*. Service which offers translation of queries and results from source format to target format.

  - ***requestToProvider***: transforms a particular request from a source format to a target format.
  - ***resultToProvider***: transforms a particular result from a source format to a target format.

Interfaces used:

- *MappingStore*. In order to get a hold of a suitable mapping for a particular request or result format.

5.2.7 Notification Manager

The role of the *Notification Manager Component* is to monitor the system to discover when a specific process should be started.

Interfaces exposed:

- *NotificationManager*. It allows triggering the execution of a process when a given event occurs, e.g. when new or updated data are published to the Harmonise network.

Interfaces used:

- *DataRegistry*. To look up the data registry to find which are the new or updated data and which are the users who subscribed to those data and need to be notified.
- *WorkflowEngine*. To trigger the execution of the requested process.

5.2.8 Metasearch

The role of the *Metasearch Component* is to run distributed queries among the participants in the Harmonise network and to aggregate the results.

Content available to be queried can reside in external repositories as well as on a local repository which indexes external content. The local search engine is implemented by exploiting the Solr search platform and the Lucene Java search library, described in the infrastructural view.

Interfaces exposed:

- *MetasearchQueryProcessor*. It allows performing distributed queries aggregating results coming both from the local repository and from external data sources.
- *MetasearchResultCache*. It allows storing temporarily search results in order to be further elaborated.
- *MetasearchSearchEngine*. It allows indexing web pages, storing locally external content, and running queries on this local repository.

Interfaces used:

- *ReconciliationEngine*. To transform queries and search results among different query languages and data formats.
- *DataConnector*. To query external data providers and get the results.

5.2.9 Crawler

The role of the *Crawler Component* is to browse external web pages and to create a copy of all the visited pages for later processing by the local search engine.

It is implemented by exploiting the Nutch web-search software, described in the infrastructural view.

Interfaces exposed:

- *CrawlerManager*. It allows configuring the crawler.

Interfaces used:

- *MetasearchSearchEngine*. To index locally the web pages downloaded by the crawler.

5.2.10 Data Registry

The *Data Registry Component* stores and manages metadata describing also the interests of Harmonise participants in specific data but mainly describes what kind of content a data provider has available.

The data registry builds upon open source base technology (see section 4.6,"Semantic Registry"). The metadata schema of the data registry will be described but not necessarily implemented as RDF(S) ontology.

Interfaces exposed:

It provides the following main interfaces:

- *DataRegistry.* To set up and use the metadata about what kind of information a Harmonise user is interested in or provides. This also includes configuration information dealing with how to access that content.
    - ***setDataDescription****.* To set the metadata describing either the content a data provider can deliver or the content a data consumer is interested in (e.g., for alerts). Furthermore, the possibility to describe the provided data in a textual way is also supported.
    - ***annotateDataDescription***. Adds annotations to the data description.
    - ***setProviderConfiguration****,* ***getProviderConfiguration***. To set up and look up the provider's configuration (e.g., the web service for calling the provider's search routines).
    - ***findDataProviders***. To list and look up data providers according to specific search criteria like name and (free text) description.
    - ***findDataProvidersForQuery****,* ***findParticipantsForData****.* These interfaces provide methods for finding data providers which can possibly answer a given search query. Also, a very similar functionality is provided to look up users interested in specific data objects.
- *DataRegistryService [WebService]*. Web Service API which allows from one side data providers to update the meta-information associated to their data, on the other side data consumers to specify their interest in certain data items.

Interfaces used:

It uses the following interfaces:

- *UserManager.* To look up the corresponding Harmonise user in order to associate the user with the data description or configuration.

5.2.11 Service Registry

The *Service Registry Component* stores the description of available services to be used in workflows. It is based on the same technology as the data registry and is possibly implemented in the same module.

Interfaces exposed:

It provides the following main interfaces:

- *ServiceRegistry.* To set up and look up what external services are provided by Harmonise participants. This also includes configuration information dealing with how to access these services.
    - ***registerService, setServiceDescription***. To make a new service known to the Harmonise network. The service is associated with a specific Harmonise participant as service provider and described in both a textual and a formal way (including the expected input and output of the service).
    - ***setServiceConfiguration, getServiceConfiguration***. To set up and look up the service configuration (e.g., the web-service interface for calling the service).
    - ***findService***. To list and find available services according to specific search criteria like name, textual description and processed data items.
- *ServiceRegistryService [WebService]*. Web Service API which allows looking up services stored in the registry.

Interfaces used:

It uses the following interfaces:

- *UserManager.* To look up the corresponding Harmonise user in order to associate the user with the service description or configuration.

5.2.12 Mapping Store

The *Mapping Store Component* holds the mappings and provides management functionality.

Interfaces exposed:

- *MappingStore*. Gives access to the database that holds mappings.
    - ***getMapping***: Takes as input a mapping ID and returns the mapping with that particular ID. Has as restriction that user is allowed to get the artefact.

- ***listMappings***: Gives a list of mappings from the database that the user is permitted to see, i.e., it returns not the mappings but only mapping IDs.
- ***setState***: Sets the current state of a particular mapping identified by a mapping ID. States may be either activated, de-activated, deprecated.
- ***getState***: gets the state of a particular mapping identified by a mapping ID.
- ***setVersion***: Sets the version field of a particular mapping identified by a mapping ID.
- ***getVersion***: Gets the version field of a particular mapping identified by a mapping ID.
- ***setPolicy***: Sets the policy to a given mapping, i.e., defines who is permitted to access the mapping with the given mapping ID.
- ***getPolicy***: gets the policy to a given mapping, i.e., defines who is permitted to access the mapping with the given mapping ID.
- ***uploadMapping***: Loads a new mapping into the mapping store, returns a newly generated unique mapping ID.

5.2.13 Connector

A connector represents a proxy for external services to be invoked, which can provide access to data integrated in the network or other kind of services (e.g. data modification, ranking, recommendation, etc.). Thus it provides a common interface to the other components of the system.

Interfaces exposed:

- *DataConnector*
  - ***query***. Allows to send a query to an integrated system
  - ***push***. Allows to push data to the integrated system
- *ServiceConnector*
  - ***execute***. Executes an external service. It receives an XML file and returns another XML file as result.
    - *select.* The external service either selects the service to be executed among a set of possible services passed as parameters or selects the service on its own by invoking the ServiceRegistry web services.
    - *modify.* Modifies data passed as parameter as an XML file.
    - *rank.* Ranks and sorts a given result set passed as parameter as an XML file. Within the input file, profile information of the current user is provided.

- *getRecommendations.* Selects items to be recommended among the ones passed as parameter as an XML file. Within the input file, profile information of the current user is provided.

5.3 WORKFLOW ENGINE

5.3.1 General Terms

A workflow consists of a sequence of connected steps and it may be seen as a virtual representation of actual work. Each step implements an activity. A process is a more specific notion than a workflow, and may be distinguished from the former by the fact that it has well-defined inputs, outputs and purposes. It contains a start, an end and at least one activity between them. There are two kinds of activities: task and sub-process.

- A task is an atomic activity which represents work that cannot be broken down.
- On the contrary, a sub-process represents work that can be broken down to a finer level of detail.

When designing a workflow it is important to determine the right granularity of the single activities to properly define the boundaries of each step. In order to build a significant brick of the workflow frame, a step should not be too big or too small and, if necessary, it should be possible to reuse it in other similar contexts.

In the scope of the HarmoSearch project, a task is an activity that indeed can be composed by several minor operations, but that can be still considered as a single operation (e.g. check permissions or reconciliate data); a sub-process groups together a series of tasks needed to fulfil a macro-operation (e.g. query a data provider).

Each user may create his own workflows, selecting the service to use and configuring it. He may also chain different services together to create a service flow to be executed by the workflow engine (e.g. run a query and get the results sorted, filtered and paginated).

Services can be selected among core services, i.e. built-in functions of the Harmonise platform, like pushing data to a selected recipient or run a distributed query, and external services, i.e. functionalities offered by an external component which can be plugged in the Harmonise platform, like recommending, ranking or data modification.

A HarmoSearch workflow or process can therefore be defined as the set of activities that are needed to fulfill one or more services requested by the participants.

5.3.2 HarmoSearch Workflows

HarmoSearch Activities

Here are the main activities that compose the processes needed to fulfill the services offered by the Harmonise platform.

- *Metadata Upload*: data provider uploads to the Harmonise portal some information on his data in form of metadata

- *Access Control List (ACL)*: determines if the user has the proper right on the requested operation
- *Data Transfer*: writes data/metadata to or reads data/metadata from a determined user by using the file system connector provided by Harmonise or by invoking an appropriate service provided by the user
- *Metadata Registering*: stores metadata in the semantic registry
- *Notification*: notifies the system that an event occurred that may need to start a new process
- *Data Upload*: data provider uploads some data to the Harmonise portal
- *Data Enrichment*: add additional information to the data published in the semantic registry
- *Data Reconciliation*: translates data from one format to another one based on a mapping rule
- *Mapping Rules*: retrieves the mapping rules for the data reconciliation
- *Data Subscription*: data consumer specifies the data profiles he is interested in to regularly import data into his system
- *Query Submission*: sends the query to the data provider (by invoking an appropriate search routine provided by the user) and gets the results
- *Query Reconciliation*: the translation of queries from the Harmonise query language to the data provider's one
- *Query Definition*: data consumer specify some search criteria and sends the query to Harmonise
- *Local Repository Lookup*: queries the local repository
- *Data Providers Selection*: find appropriate data providers, i.e. providers which own data a user may be interested in
- *Results Visualisation*: displays to the users the results of a query (possibly asynchronously)
- *Data Storing (external task)*: stores data in a CMS
- *Service References*: retrieves the references to the external service to be invoked
- *Data Selection*: data consumer selects some data he wants to download
- *Data Retrieval (external task)*: retrieves data from the CMS
- *Recommendation Request*: data consumer asks for recommendations about items related to a specific topic of interest
- *User Profile Retrieval*: retrieves user profile information
- *Recommended Data Providers Selection (external task)*: find recommended data providers, i.e. providers which own data matching a recommendation request

- *Recommended Results Selection (external task)*: identifies the results which match a recommendation request
- *Data Accumulation and Caching*: accumulation and caching of the results of a query
- *Data Sorting / Filtering / Paginating (external task)*: sorts /filters / paginates incoming results from queried data providers
- *Data Modification (external task)*: transforms data received from data providers (e.g. controlling, correcting, completing, transforming or translating content according to the user's needs)

HarmoSearch Processes

Here are the main processes implemented and managed by HarmoSearch Workflow Engine to fulfil the services offered to the participants. Each service or service chain will be implemented as a process and the list of tasks needed to compose each process is specified below.

Let's start with the main processes that will be used to implement some of the core services, i.e. the ones involving data exchange.

1. **Publishing of new data**
    a. Metadata Upload
    b. ACL
    c. Data Transfer
    d. Metadata Registering
        i. Notification
2. **Pushing of data to a selected recipient**
    a. Data Upload
    b. ACL
    c. Data Transfer
    d. Data Enrichment (optional)
    e. Data Reconciliation
        i. Mapping Rules
    f. Data Transfer
3. **Importing data from selected data sources**
    a. Data Subscription
    b. ACL
    c. Query Submission
        i. Query Reconciliation
    d. Data Enrichment (optional)
    e. Data Reconciliation

            i. Mapping Rules
    f. Data Transfer
4. **Metasearch**
    a. Query Definition
    b. ACL
    c. Data Providers Selection
    d. Query Submission
        i. Local Repository Lookup
        ii. Query Reconciliation
    e. Data Enrichment (optional)
    f. Data Reconciliation
        i. Mapping Rules
    g. Results Visualisation

Besides the processes described above, the workflow engine allows managing also processes which include the execution of external services.

1. **Data hosting**
    a. Data Upload
    b. ACL
    c. Data Reconciliation
        i. Mapping Rules
    d. Data Storing
        i. Service References
2. **Data download**
    a. Data Selection
    b. ACL
    c. Data Retrieval
        i. Service References
    d. Data Reconciliation
        i. Mapping Rules
    e. Data Transfer
3. **Request recommendations**
    a. Recommendation Request
    b. ACL
    c. User Profile Retrieval
    d. Recommended Data Providers Selection

        i. Service References
    e. Query Submission
        i. Query Reconciliation
    f. Data Enrichment (optional)
    g. Recommended Results Selection
        i. Service References
    h. Data Reconciliation
        i. Mapping Rules
    i. Results Visualisation

Finally services (core and external ones) can be chained together to set up a service flow. Here are a couple of examples of processes that implement such a service flow.

4. **Ranking, filtering and pagination of search results**
    a. Query Definition
    b. ACL
    c. Data Providers Selection
    d. Query Submission
        i. Local Repository Lookup
        ii. Query Reconciliation
    e. Data Enrichment (optional)
    f. Data Reconciliation
        i. Mapping Rules
    g. Data Accumulation and Caching
    h. Data Sorting / Filtering / Paginating
        i. Service References
    i. Results Visualisation
5. **Modification of imported data (e.g. cleansing, translation, etc.)**
    a. Data Subscription
    b. ACL
    c. Query Submission
        i. Query Reconciliation
    d. Data Enrichment (optional)
    e. Data Reconciliation
        i. Mapping Rules
    f. Data Modification (e.g. cleansing, translation, etc.)
        i. Service References

g. Data Transfer

5.3.3 Workflow Definition

Many different languages have been defined to describe tasks and the flow of activities that constitute a process. From a first analysis it came up that the best candidate for describing HarmoSearch workflows is BPMN 2.0 (see 4.3.1).

As an example, the following figure shows a BPMN graphical notation of a simple business process with a start event, a sub-process, a gateway, two user tasks and an end event. The sub-process internal structure is depicted in the figure below.

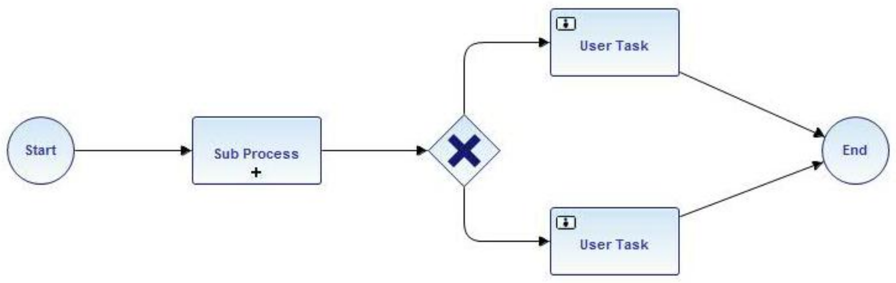


Figure 5 BPMN Main process

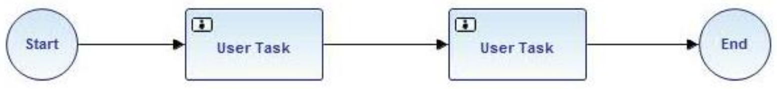


Figure 6 BPMN Sub-process

The related BPMN 2.0 XML file is the following:

```
  <process id="mainprocess" name="mainprocess">
    <startEvent id="startevent3" name="Start"></startEvent>
    <exclusiveGateway id="exclusivegateway3" name="Exclusive
Gateway"></exclusiveGateway>
    <userTask id="usertask2" name="User Task"></userTask>
    <sequenceFlow id="flow24" name="" sourceRef="exclusivegateway3"
targetRef="usertask2"></sequenceFlow>
    <userTask id="usertask3" name="User Task" ></userTask>
    <sequenceFlow id="flow25" name="" sourceRef="exclusivegateway3"
targetRef="usertask3"></sequenceFlow>
    <endEvent id="endevent4" name="End"></endEvent>
    <sequenceFlow id="flow26" name="" sourceRef="usertask3"
targetRef="endevent4"></sequenceFlow>
    <sequenceFlow id="flow27" name="" sourceRef="usertask2"
targetRef="endevent4"></sequenceFlow>
    <subProcess id="subprocess1" name="Sub Process">
      <startEvent id="startevent4" name="Start"></startEvent>
      <userTask id="subprocess1_usertask4" name="User Task"></userTask>
      <sequenceFlow id="subprocess1_flow30" name=""
sourceRef="subprocess1_startevent4"
targetRef="subprocess1_usertask4"></sequenceFlow>
      <userTask id="subprocess1_usertask5" name="User Task"></userTask>
      <sequenceFlow id="subprocess1_flow31" name=""
sourceRef="subprocess1_usertask4"
targetRef="subprocess1_usertask5"></sequenceFlow>
      <endEvent id="endevent5" name="End"></endEvent>
      <sequenceFlow id="subprocess1_flow32" name=""
sourceRef="subprocess1_usertask5"
targetRef="subprocess1_endevent5"></sequenceFlow>
    </subProcess>
    <sequenceFlow id="flow28" name="" sourceRef="startevent3"
targetRef="subprocess1"></sequenceFlow>
    <sequenceFlow id="flow29" name="" sourceRef="subprocess1"
targetRef="exclusivegateway3"></sequenceFlow>
  </process>
```

HarmoSearch processes can easily be described using BPMN graphical notation, as shown in the following figures that represent how the metasearch process may be modelled.

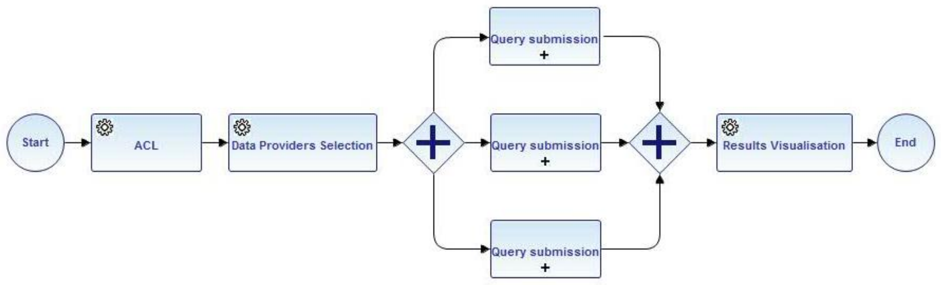


Figure 7 Metasearch process

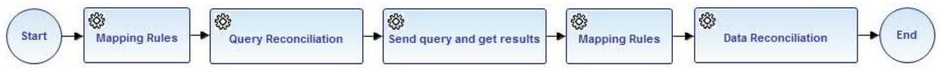


Figure 8 Query submission sub-process

5.3.4 Workflow Execution

Many different workflow engines and Business Process Management (BPM) platforms have been developed to manage and execute modelled computer processes. From a first analysis it came up that the best candidate for managing and executing HarmoSearch workflows is Activiti (see 4.3.2).

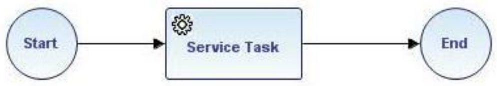
A workflow consists in a sequence of tasks. Tasks may differ according to the type of operation that they describe. Examples of tasks are:

- *User task*. It is used to model work that needs to be done by a human actor. When the process execution arrives at such a user task, a new task is created in the task list of the user(s) or group(s) assigned to that task.

- *Script task*. It is an automatic activity. When a process execution arrives at the script task, the corresponding script is executed.
- *Service task*. A Java service task is used to invoke an external Java class.

HarmoSearch workflows will mainly make use of system tasks to fulfil the services described in 5.3.2.

Using Activiti, to implement a class that can be called when the process execution arrives at a service task, this class needs to implement the `org.activiti.engine.delegate.JavaDelegate` interface and provide the required logic in the `execute` method. When process execution arrives at this particular step, it will execute this logic defined in that method and leave the activity in the default BPMN 2.0 way. Here is an example of a simple helloWorld workflow composed by start event, end event and one system task that prints "Hello World".



The related BPMN file is the following:

```
  <process id="helloworld" name="helloworld">
    <startEvent id="startevent1" name="Start"></startEvent>
    <serviceTask id="servicetask1" name="Service Task"
activiti:class="it.cpr.JavaService"></serviceTask>
    <sequenceFlow id="flow1" name="" sourceRef="startevent1"
targetRef="servicetask1"></sequenceFlow>
    <endEvent id="endevent1" name="End"></endEvent>
    <sequenceFlow id="flow3" name="" sourceRef="servicetask2"
targetRef="endevent1"></sequenceFlow>
  </process>
```

The `activity:class` attribute of the `serviceTask` tag there contains the connection between the workflow and the Java code invoked when the task is executed. Here is the implementation of the `JavaService` delegate class:

```
package it.cpr;
import org.activiti.engine.delegate.DelegateExecution;
public class JavaService implements org.activiti.engine.delegate.JavaDelegate {
        @Override
        public void execute(DelegateExecution arg0) throws Exception {
                String ss = (String)arg0.getVariable("name");
                System.out.println(ss + " Hello World!");
        }
}
```

The `execute` method contains the reference to a shared environment where the tasks in the workflow can read and write different variables. In the example above the string variable `name` is read in the `JavaService.execute` method.

The objects shared by process tasks and Java delegate classes cannot only be primitive types but also user defined classes. In this case they have to implement the `java.io.Serializable` interface.

As an alternative, it is also possible to inject values into the fields of the delegated classes using BPMN 2.0 `activity:field` element. If available, the value is injected through a public setter method on the delegated class, following the Java Bean naming conventions. If no setter is available for that field, the value of private member will be set on the delegate.

In order to execute the simple process described above, here is a junit test class:

```
public class ProcessTestHelloworld {
@Rule
public ActivitiRule activitiRule = new ActivitiRule();
       @Test
       @Deployment(resources="diagrams/helloWorld.activiti.bpmn20.xml")
       public void startProcess() {
              RuntimeService runtimeService = activitiRule.getRuntimeService();
              Map<String, Object> variableMap = new HashMap<String, Object>();
              variableMap.put("name", "Activiti");
              ProcessInstance processInstance =
runtimeService.startProcessInstanceByKey("helloworld", variableMap);
              assertNotNull(processInstance.getId());
       }
}
```

This test class contains:

- the reference to the BPMN file containing the process description and the connection with the Java delegate class;
- the instantiation of the HashMap for sharing data between the workflow task and the Java delegate class;
- the method for starting the process with id "helloworld".

The output of this process is, as expected:

```
Activiti Hello World!
```

Using Activiti with Spring

A different and more efficient way to connect workflow service tasks with Java code is to use the integration between Spring and Activiti. In this way there is no need for delegate classes neither to extend the org.activiti.engine.delegate.JavaDelegate interface nor to implement the `execute` method. Spring objects with their methods can be directly referenced in the BPMN files using the `activity:expression` element. Let's have a look at how the helloWorld process described above will look like using Spring. Here is the Java class that implements a `helloworld` method:

```
package it.cpr;
public class PojoClass {
       public void helloworld(String s){
              System.out.println(s + " Hello World!");
       }
}
```

And this is the spring definition of a bean for that class:

```
<bean id="springRef" class="it.cpr.PojoClass" />
```

The BPMN description of the helloWorld process will look as follow:

```
  <process id="callSpringProcess" name="Call Spring example">
    <startEvent id="theStart" />
    <sequenceFlow id="flow1" sourceRef="theStart" targetRef="callSpring" />
    <serviceTask id="callSpring"
activiti:expression="${springRef.helloworld(name)}" />
    <sequenceFlow id="flow3" sourceRef="callSpring" targetRef="theEnd" />
    <endEvent id="theEnd" />
  </process>
```

The `activity:expression` property in the `serviceTask` tag allows to use an expression that resolves to a delegation object; in this case it allows to invoke the `helloworld` method defined in the `springRef` bean.

Here is the code for the class that starts the process:

```
public class CallSpringExample {
       public static void main(String[] args) {
              // setup
              ApplicationContext context = new
ClassPathXmlApplicationContext("spring-context.xml");
              ProcessEngine processEngine =
context.getBean(ProcessEngine.class);
              RepositoryService repositoryService =
processEngine.getRepositoryService();
              RuntimeService runtimeService = processEngine.getRuntimeService();
              // Deploy process
              repositoryService.createDeployment()
                     .addClasspathResource("callSpringProcess.bpmn20.xml")
                     .deploy();
              // Run process
              Map<String, Object> variables = new HashMap<String, Object>();
              variables.put("name", "Activiti");
              runtimeService.startProcessInstanceByKey("callSpringProcess",
variables);
       }
}
```

This class configures the environment in the same way as in the previous example, but in the spring class there is no reference to that environment, but just input and output variables.

As expected, the result of the execution of this process is again:

```
Activiti Hello World!
```

The return value of a service execution (for service task using expression only) can be assigned to an already existing or to a new process variable by specifying the process variable name as a literal value for the `activiti:resultVariable` attribute of a service task definition. Any existing value for a specific process variable will be overwritten by the result value of the service execution. When not specifying a result variable name, the service execution result value gets ignored.

In general the output of a method can be used also by conditional elements to choose the flow to execute.

Exception handling

When custom logic is executed, it is often required to catch certain exceptions. One common use case is to route process execution through another path in case some exception occurs. The following example shows how this is done.

```
  <serviceTask id="javaService" name="Java service invocation"
activiti:class="org.activiti.ThrowsExceptionBehavior">
  </serviceTask>
  <sequenceFlow id="no-exception" sourceRef="javaService" targetRef="theEnd" />
  <sequenceFlow id="exception" sourceRef="javaService" targetRef="fixException"
/>
```

Here, the service task has two outgoing sequence flow, called `exception` and `no-exception`. This sequence flow id will be used to direct process flow in case of an exception:

```
public class ThrowsExceptionBehavior implements ActivityBehavior {
      public void execute(ActivityExecution execution) throws Exception {
            String var = (String) execution.getVariable("var");
            PvmTransition transition = null;
            try {
                  executeLogic(var);
                  transition =
            execution.getActivity().findOutgoingTransition("no-exception");
            } catch (Exception e) {
                  transition =
            execution.getActivity().findOutgoingTransition("exception");
            }
            execution.take(transition);
      }
}
```

5.4 MAPPING TOOL

The goal of work package 6 of the HarmoSearch project is to develop a tool that visually supports the user in the task of creating the necessary mapping artefacts with little technical knowledge. The artefacts are then used to perform a translation from the data model of one organization to the data model of Harmonise and vice-verse.

The mapping tool is considered to be a standalone application and should not be dependent on any particular domain. Basically, it consists of a graphical User Interface to show and manipulate mappings, a mechanism to import and display schemata, a pluggable set of algorithms to support automatic mappings, a generator to create mapping artefacts and to export these artefacts, and an infrastructure in order to manage a mapping project.

In a first step we have evaluated a number of existing solutions that claim to support creation and maintenance of mapping projects (see WP6, task 6.1 Scanning of existing open source projects). As evaluation criteria we used functional and non-functional features of existing projects. The functional features refer to the basic components of the envisioned mapping tool, which are a Graphical User Interface, an import and export component to read a variety of schema formats and export transformation rules to the Harmonise platform, a matching component that gives access to a variety of matching algorithms, and a transformation creation component. The non-functional features comprise the terms and conditions of usage and the support to users by community and documentation.

- Open source: Is the source available? This is in effect a knock-out criterion since we consider only Open Source projects for HarmoSearch.
- Graphical Interface: Does the tool provide a GUI to show and manipulate mappings?
- Programming Language: which programming language is employed?
- Transformation Technologies: which transformation technology is employed to transform data from one format to the other?
- Matching Algorithms: Are there any algorithms to propose matches between data formats?
- Input-Formats: which input formats are supported?
- Output-Formats: which output formats are supported?
- Extensibility: which techniques exist to provide extensions to the tool?
- Technological Requirements: what are the dependencies to particular technologies and platforms (Win/MAC/Unix)?
- Community: Is there some kind of community that discusses the tool?
- Documentation: scope, actuality and extensiveness of documentation
- License: what type of license model is used? Is it suitable for the HarmoSearch project?

The evaluation showed that none of the Open source tools is actually mature or comprehensive enough to fulfil all of our requirements. Nevertheless, some candidates have parts that can provide valuable input to the HarmoSearch mapping tool. For instance, the Alignment API (http://alignapi.gforge.inria.fr/align.html) provides an open environment for schema matching with easy means to integrate and test new matching algorithms. The Smooks project (http://www.smooks.org/), on the other hand, has no support for automatic matching but provides extensive means to generate and execute transformations from source to target data. None of the evaluated tools have a really suitable GUI.

As a result from this investigation, we see as the most promising approach a mix of self-developed SW components tied together with (parts) of existing software components. As a glue to combine these components we use Eclipse, a well established environment for SW development in various areas.

The architecture of the Mapping tool is depicted in Figure 9 and comprises the following components:

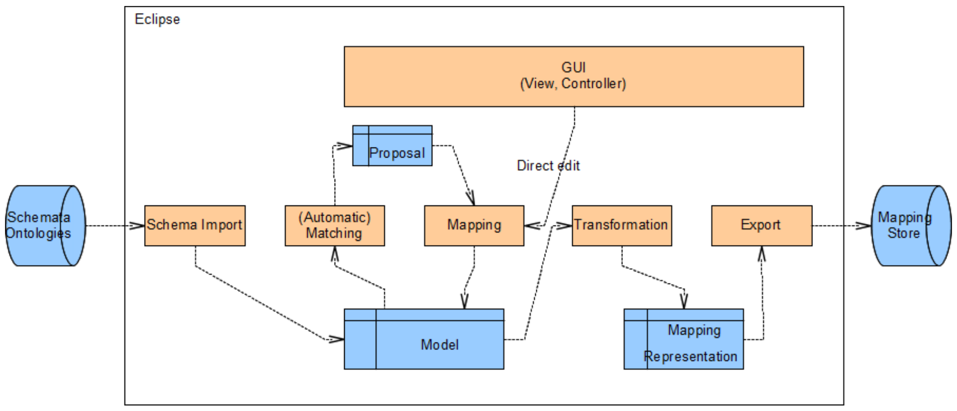


Figure 9: Basic architecture of Mapping Tool

5.4.1 Eclipse

This is actually not a component but a whole environment with a set of built-in core components and a set of optional components. Using Eclipse has a number of advantages: First of all, it is well-known and established as a software development tool and is used in many projects worldwide. Then, Eclipse is highly configurable using a plug-in mechanism; additional components can be integrated easily, and many components are available from a huge community. Finally, it is available for all major platforms and has only the single dependency that Java must have been installed on the local machine.

5.4.2 GUI

The GUI consists of editors and views to navigate and manipulate artefacts and additional elements such as tool bars or menus. It serves to visualise the source and target schemata, the proposed matches between source and target data elements, and the created mappings between source and target. Furthermore, it provides functionality to select matching algorithms and to manipulate and edit mappings. In addition, The GUI provides means to manage mapping projects. These include the creation of new projects, creation of new mappings within a project, and registration of data schemata from various locations.

5.4.3 Schema Import Component

Imports a schema file from local storage device or from an URL over the Web and transforms the schema into a suitable internal representation. While the Harmonise schema is available in XSD and RDF format, we do not want to restrict our approach to just these two kinds of schemata, but also take into account additional formats such as data descriptions of relational databases. Basically, a schema importer is a component that takes as input the location of a schema and returns a suitable representation of the schema. Currently, we use an XML Document format as representation means, but other representation formats may evolve during the development of the mapping tool.

5.4.4 Matching Component

This component provides means to automatically derive equivalences between source and target schema elements. Basically, the component provides an Interface called Matching Service that has a method getMatching (source schema, target schema) which returns a distance matrix for the source and target elements.

In addition, a configuration method addContext (Context context) allows setting a single matching algorithm or a sequence of consecutive algorithms. Algorithms are added using the Strategy pattern, that is, algorithms are not pre-configured but can be added and removed at runtime.

5.4.5 Mapping Component

Holds an actual mapping from a source to a target schema, that is, some representation that expresses equivalences of source schema elements to target schema elements and is the result of a mapping task. It takes as input matching proposals from the matching component as well as user generated input from the GUI.

5.4.6 Transformation Component

It is responsible for the creation of the actual transformation from source to target. In the Harmonise project XSLT (Extensible Stylesheet Language Transformations) is used as a transformation means, but other transformation mechanisms should not be excluded. The component offers an interface that provides the main functionality of transformation. It takes as input the source and target schema and the actual mapping from the mapping component and has as output a representation of the transformation according to the actual configuration such as an XSLT file.

5.4.7 Export Component

The result from the Transformation process is uploaded to and registered at the Harmonise platform using this component.

All components are being realized as Eclipse plug-ins. In a first step, simple mock-ups are being developed that allow studying the overall behaviour of the system and to detect any shortcomings and design flaws of our approach. Additionally, optional components are scanned that may be of interest for the mapping tool. In particular, Eclipse has a number of plug-ins that can give aid in the management of mapping projects, such as collaboration tools, task management or bug tracking.

6 DEVELOPMENT VIEW

This chapter defines the development environment, how the software components should be structured and some common issues related to error and log management and testing.

6.1 SVN

Apache Subversion (often abbreviated SVN) is a software versioning and a revision control system founded and sponsored in 2000 by CollabNet Inc. Subversion uses the Apache License, making it free software and open source.

HarmoSearch developers will use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation.

A necessary step before starting development is to check out the project source tree from the SVN repository: https://62.149.192.167/repos/harmosearch. Access to the repository requires username/password authentication.

The project and the component directory hierarchy are still to be agreed. Here is a possible package structure.

- *codes/java/*
    - *framework* (Spring)
    - *component1* (component1 subtree)
    - *component2* (component2 subtree)
    - *component3* (component3 subtree)
    - …
    - *componentN* (componentN subtree)
    - *application* (Liferay web application)
    - *lib* (components generated jar files)
    - *extlib* (needed external libs)
    - *instLiferay* (installation and configuration instructions)
    - *build.xml* (main build script)

The idea is that each component has its own separate build which produces a jar file. The application build is in charge of integrating all the components' jars and creating the war package to be deployed on the Tomcat application server.

Here is a possible directory structure for a component or application subtree.

- *componentN*
    - *main*
        - *src* (source code)
        - *resources* (needed resources)
    - *build* (place for building the component)

        - *classes* (generated classes)
    - *web* (web interfaces)
    - *test* (test cases)
        - *src* (source code)
        - *resources* (needed resources)
    - *build.xml* (component build script)

6.2 BUILD

Apache Ant is a software tool for automating software build processes. It is implemented using the Java language, requires the Java platform, and it uses XML to describe the build process and its dependencies. By default the XML file is named build.xml. Ant is an Apache project. It is open source software, and is released under the Apache Software License.

The following Ant targets may be used by HarmoSearch developers from the application or component build file during development process:

- *clean* the component build results
- *compile* the component
- *build_jar*, generates the component .jar file
- *test*, runs the component test cases
- *doc*, generates javadoc
- *build_war*, generates and deploys the application .war file

The following Ant targets may be used from the main build file:

- *all*, builds all components, generates and deploys the .war file
- *clean*, cleans all temporally build files

6.3 TESTING

Software testing can be costly and burdensome, but not testing software is even more expensive and often causes frustration in a later stage of software development. For that reason testing is part of every software engineering process.

Since mistakes are inevitable and most of them are made in early stages of development, testing the software must begin right from the beginning. Therefore, HarmoSearch approach will be based on JUnit [http://www.junit.org] in order to test each component.

JUnit is an easy-to-use Java testing framework that serves two main purposes:

- Avoid errors in a component's application logic.
- Avoid regression errors - while fixing an error you could introduce new error into the system.

The JUnit philosophy is quite a simple one: "code a little, test a little" - which means that test cases should be written at the same time of the main code. With JUnit checks are encapsulated in a TestCase and assertions should be made to let JUnit check if the expected results correspond to the result a certain piece of code (e.g. a method, class) delivers. The advantages of the JUnit approach is that checks don't have to be placed into the program code and all of them are reproducible.

JUnit itself is just a jar file (junit.jar) that provides the framework to write test cases and to group them into (nested) test suits. Furthermore, it contains a graphical as well as a textual environment to run the test cases.

For HarmoSearch, the JUnit rule of thumbs are:

- Write one test case for each java class of a component by creating a new test class that extends the JUnit TestCase class.
- Test all methods that perform a given functionality by writing one or more corresponding test methods. A test method contains several assertions that reflect what it is expected to be the result of a certain test scenario.
- Group test classes by defining classes that extend JUnit's TestSuite class.

Integration tests can be written using the same JUnit approach by writing test cases that reflect the functionality of the use cases defined for each component or set of components.

Generally spoken, test cases should be written in a way so that all tests for one component can run independent from other components within the system architecture. However, due to the distributed architecture of Harmonise, it might happen that one component relies on objects and service interfaces defined in other components. Since in an early stage of development only the predefined interfaces are available, but not the according implementations, it is possible to write Mock Objects that simulate the expected behaviour of other components, providing some dummy values that represent the values expected to get from those interfaces. This is the case not only for component interfaces but also for database connectivity.

6.4 LOGGING

Logging information during run-time is very important for tracking down what is going on. Proper logging helps a lot in finding and discovering bugs and potential weird behaviours.

HarmoSearch project will use Apache Log4J [http://jakarta.apache.org/log4j/] and Apache Common Logging [http://jakarta.apache.org/commons/logging/] for getting *Logger* instances.

Logger supports logging of 5 different types of messages (debug levels):

- *DEBUG*, used for showing detailed information about what is going on in the system
- *INFO*, just some common info messages
- *WARN*, warnings that the software can handle properly
- *ERROR*, errors that the software can handle properly
- *FATAL*, fatal errors after which the software cannot continue

6.5 EXCEPTION HANDLING

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution.

A piece of code is said to be exception-safe, if run-time failures within the code will not produce ill effects, such as memory leaks, garbled stored data, or invalid output.

Checked exceptions in Java are a special set of exceptions. They represent invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages and absent files). The checked exceptions that a method may raise are part of the method's signature. On the other hand, unchecked exceptions (Runtime Exceptions and Errors) represent defects in the program (bugs) - often invalid arguments passed to a non-private method – and they remain unhandled.

Checked exceptions can, at compile time, reduce the incidence of unhandled exceptions surfacing at runtime in a given application. However, they can either require extensive throws declarations, revealing implementation details and reducing encapsulation, or encourage coding poorly-considered try/catch blocks that can hide legitimate exceptions from their appropriate handlers. It is possible to reduce the number of declared exceptions by either declaring a superclass of all potentially thrown exceptions or by defining and declaring exception types that are suitable for the level of abstraction of the called method, and mapping lower level exceptions to these types, preferably wrapped using the exception chaining in order to preserve the root cause.

In HarmoSearch the super class of all potentially thrown exceptions is called *HarmoniseException*, while the super class of all unchecked exceptions is called *HarmoniseUncheckedException*.

7 PHYSICAL VIEW

This chapter discusses possible approaches to deploy on the network the HarmoSearch solution.

The figures below show how the developed components can be deployed on the HarmoSearch servers and the relationships with external systems.

The first option is the simplest one. All the Harmonise components are deployed in one central server, which acts as web server, application server and database server.

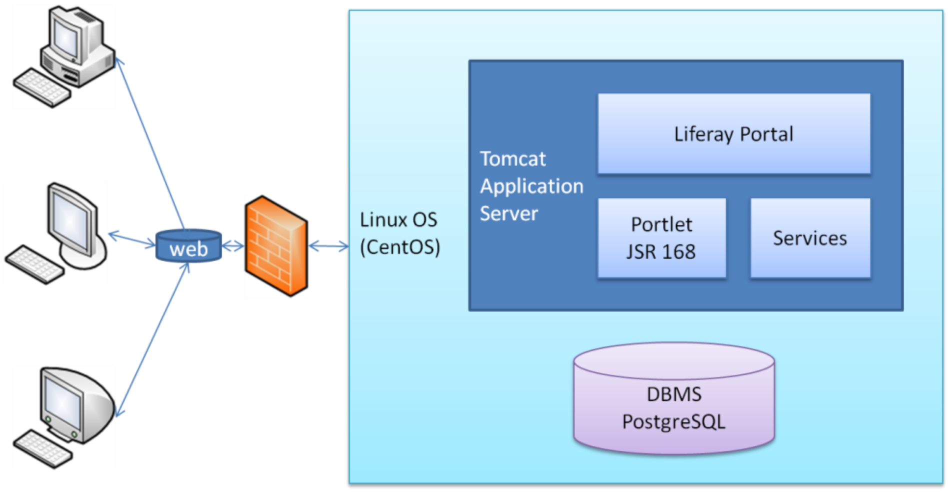


Figure 10 Physical View: simple approach

This server may connect to other external servers (mainly through web services) when there is the need to push data to / query data from a Harmonise participant or when the execution of an external service has to be triggered (e.g. recommendation or data modification services).

The second option is based on a load balancing cluster approach. In networking, load balancing is a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load balancing, instead of a single component, may increase the reliability and the scalability of the system, i.e. its ability to handle growing amounts of work in a graceful manner or to be enlarged to accommodate that growth. To be able to optimize access considering geographical distribution of the clients it will be possible to adopt the new cloud computing (for example Amazon S3) approach, where specific services are made available to the client from a network server optimized considering the location of the client.

The adopted infrastructural technology, in particular the Liferay Portal Server, is well suited to scale to complex high availability and fault tolerant scenario as such needs arise. The actual architecture will be chosen according to the business needs of the solution.

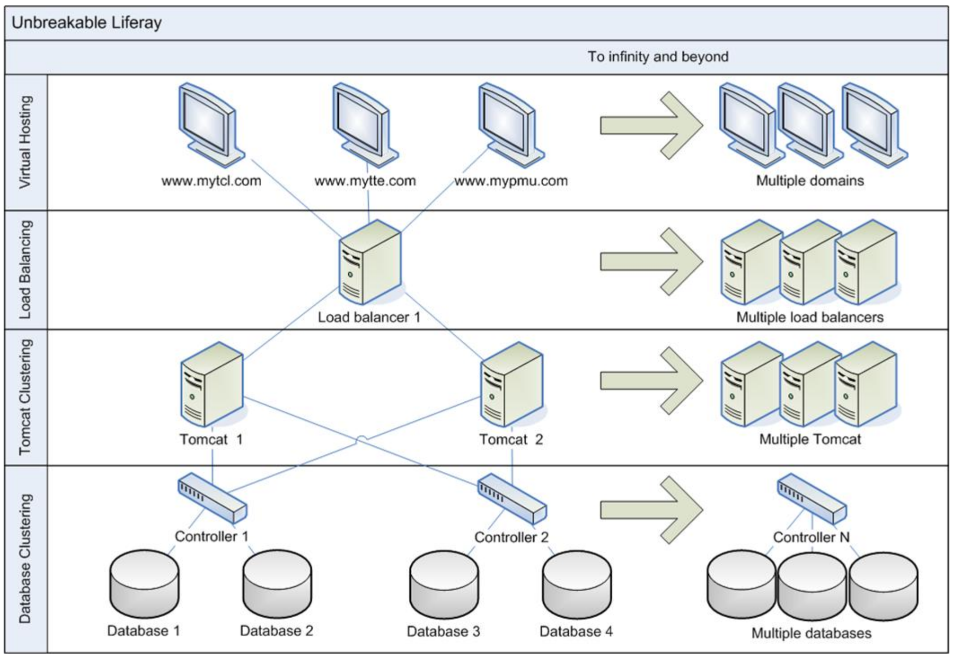


Figure 11 Cluster

8 LIST OF FIGURES