WeKnowIt

Emerging, Collective Intelligence for Personal, Organisational and Social Use

FP7-215453

D6.1.2

Identification of architecture elements and relations, version 2

Dissemination level	Confidential
Contractual date of delivery	Month 12, 30.03.2009
Actual date of delivery	
Workpackage	WP6, Architecture and Integration
Task	T6.1 Identification of architecture elements and relationships
Type	Report
Approval Status	
Version	0.5
Number of pages	102
Filename	

Abstract

This document describes the final architecture of the WeKnowIt system. It discusses requirements for the system and technical solutions (technologies selected and their place in the architecture of the system) that meet them.

In the second part integration plan for the work of all WPs is presented, along with description of tools and measures that will be used in this process.

co-funded by the European Union

History

Version	Date	Reason	Revised by
0.1	05.03.2009	First version, plan of document.	T.Kaczanowski
0.2	09.03.2009	Technologies descriptions. Development environments, version control system, bug tracker.	R.Janik, T.Kaczanowski
0.3	12.03.2009	Technologies updates & fixes. Introduction & Executive Summary enhanced. Architecture – SOA & ESB – written.	T.Kaczanowski
0.4	02.04.2009	Final version.	A.Boruch, R.Janik, T.Kaczanowski, G.Tabor
0.5	07.04.2009	Final version. Language fix.	J.Kwiecińska

Author list

Organization	Name	Contact Information
Software Mind	A. Boruch	a.boruch@softwaremind.pl
Software Mind	R. Janik	r.janik@softwaremind.pl
Software Mind	T. Kaczanowski	t.kaczanowski@softwaremind.pl
Software Mind	J. Kwiecińska	j.kwiecinska@softwaremind.pl
Software Mind	G. Tabor	g.tabor@softwaremind.pl

Executive Summary

Second version of "Identification of architecture elements and relations" deliverable provides detailed information about technologies used in the WeKnowIt System, and reflects changes that were introduced to the architecture since D6.1.1.

The selection of technologies for the WKI System has been completed. As it was described in D6.1.1 (M6) the WeKnowIt architecture is based on service-oriented architecture (SOA) and uses enterprise service bus (ESB) as backbone for connecting the services. It uses OSGi component framework to manage the loosely coupled services.

Description of the architecture starts with requirements, which were enhanced by analyzing the real services declared by WPs. Next, architecture solution is discussed, showing how the technologies that were selected for the WeKnowIt system, and the whole architecture concept meet the requirements.

Thanks to the fact that technical architecture is only a part of the integration task of the WP6, this document also describes the integration plan, which contains:

- tools that will be used by all partners during development,
- environments for developing and testing of the WKI services and the WKI System,
- quality assurance process which ensure high quality of the final WKI System.

Abbreviations and Acronyms

API	Application Programming Interface
BPEL	Business Process Execution Language (synonym to WS-BPEL)
CI	Continuous Integration
CSG	Consumers Social Group
CXF	Apache project for service creation and execution
DAO	Data Access Object
DI	Dependency Injection
DS	WeKnowIt Data Store
DSL	Domain Specific Language
EIP	Enterprise Integration Patterns
ER	Emergency Response
ESB	Enterprise Service Bus
FAQ	Frequently Asked Questions
GUI	Graphical User Interface
IEP	Intelligent Event Processor
HTML	HyperText Markup Language
HTTP(S)	HyperText Transfer Protocol (Secure)
IDE	Integrated Development Environment
J2EE	Java Platform, Enterprise Edition
J2ME	Java Platform, Micro Edition
J2SE	Java Platform, Standard Edition
JAR	Java Archive
JAX-WS	Java API for XML Web Services
JBI	Java Business Integration
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JSR	Java Specification Requests
JVM	Java Virtual Machine
KB	Knowledge Base
NMR	Normalised Message Router
OO	Object Oriented
ORM	Object-Relational Mapping
OS	Operating System
OSGi	OSGi Alliance (formerly Open Services Gateway Initiative)
POJO	Plain Old Java Object
POM	Project Object Model
QA	Quality Assurance
RAID	Redundant Array of Independent Disks
RDBMS	Relational Database Management System
RDF	Resource Definition Framework
RSS	Really Simple Syndication

SCA	Service Component Architecture
SOA	Service-Oriented Architecture
SOAP	(formerly) Simple Object Access Protocol
SPRINGDM	Spring Dynamic Modules for OSGi(tm) Service Platforms
SVN	Subversion
TIFF	Tagged Image File Format
UDDI	Universal Description, Discovery and Integration
UI	User Interface
WP	Work package
WS	Web Service
WSDL	Web Service Description Language
XML	eXtensible Markup Language

Table of Contents

List of Pictures

List of tables

List of Listings

1. Introduction

Deliverable D6.1.2 constitutes continuation and enhancement of description of the WKI System architecture that was presented in deliverable D6.1.1 "Identification of architecture elements and relations, version 1".

This document begins with the presentation of requirements gathered from: "Definition of Work", directly from WPs, and derived from services declared by WPs.

Then, the WKI System is described from the technical point of view. This section starts with overview of: layers, modules and overall picture of technologies. Next all the technologies are presented, for each of them reasons of use and benefits they provide for the WKI System are enumerated.

After the technologies are described, it is presented how they fit into the WKI System architecture. The description of architecture solution starts with the smallest parts of the WKI System (single service), than it is shown how to combine many services, and finally how to make them accessible by third party applications.

The second part of this deliverable deals with the process of integration, that will result in creation of the fully-functioning WKI System. It starts with description of two sub elements of this process: "common infrastructure", which describes technical aids for the development process, and "quality assurance", where tools and processes that will be used to achieve high quality of the created software are presented.

Integration plan binds these two areas together showing how they will contribute to the creation of the final WKI System.

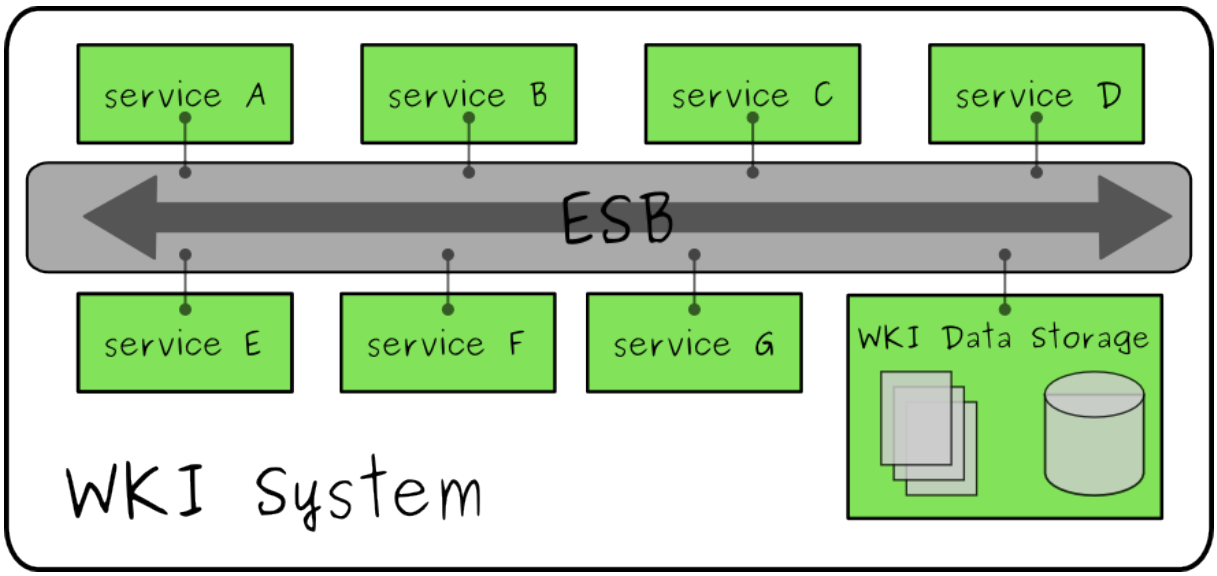
2. The WKI System

According to the Definition of Work, the WKI project shall provide *"system platform and the necessary user interfaces for creation, storage, manipulation and consumption of Collective Intelligence"*. The WKI System groups services created by all WPs, and integrates them in order to provide functionality, that allows to create various applications with different user interfaces.

The goal of the WKI project is to create the WKI System and two client sets of applications - one for each case study: Emergency Use Case and Social Group Use Case.

Picture 1 presents an overview of the WKI System. All the WKI services are connected to the Enterprise Service Bus. The WKI Data Storage (presented in D6.3) is also treated as service of the WKI System.

The applications mentioned in the previous paragraphs are not visible on this picture, as they do not belong to the WKI System, and should be regarded as clients which use functionality, provided by the WKI System.



Picture 1 Overview of the WKI System

Detailed description of the WKI System architecture and its connection with external applications is provided in section 5.

3. WKI System requirements

To develop the most suitable architecture for the WKI System, requirements analysis was prepared. The requirements on several different levels have been gathered and analyzed:

- General requirements – Mostly non-functional, general requirements related to the general project ideas, research and development organization.
- Specific WPs requirements – Detailed requirements for each Work Package (functional, software and hardware) as each WP is responsible for development of particular software which should be integrated within the system.
- Use-cases (WP7) requirements – Requirements which result from the use case WP7 responsible for end-user applications creation. These requirements strongly influence overall system architecture.

3.1. General requirements

This section lists the general WKI System requirements. The first version of those requirements was presented in the deliverable D6.1.1. Since then they have been intensively discussed and reviewed, some of them have been removed as no longer valid.

#	Requirement	Details/Motivation
General requirements		
R1	Flexibility	• For defining interfaces and dependencies, no fixed service schemes, but rather pattern usage. • Separate logical layer encapsulating the dependencies between the work packages. • System should be ready for changes at each stage of its development lifecycle as the research in WPs goes.
R2	Loosely coupled components	• Research teams in WPs use different and separate technologies suitable for specific research area. Nevertheless, the system as a whole should be integrated. • Loosely coupled components are usually easier to deploy and test (encapsulation).

#	Requirement	Details/Motivation
R3	In general, system should be efficient	In the context of the WKI System and the end-user applications built on the top of it: • Large volume and computational intensive processes (like image indexes generation) should be separated as backend/batch jobs not to influence the online part of the system. • Online-purpose services should respond to all requests in the limited time (event if it is possible to return only partial or uncertain results). This is absolute must in case of the Emergency Response scenario.
Processes		
R4	Support of asynchronous processes	Asynchronous processes support is usually a good pattern for mixing conflicting requirements: a) on-line access (required for both scenarios) when some results should be returned in a matter of seconds; b) some processes require more time to give complete results. In such case user gets some results quickly but they can be incomplete, more complete results are provided later and they are available asynchronously.
R5	Parallel execution of workflows or processes	The system should support many users and processes working in parallel. This requirement is absolute must in the context of both ER and CG use cases.
R6	Workflow definition by developers	• System logic should be defined by means of workflows. In such case the application logic can be relatively easily understood by the non-researchers (e.g. ER personnel) and modified according to the evolving requirements. • Developers should be able to modify workflow definitions in order to change the system logic.
R7	Support of on-line jobs as well as batch jobs	• The WKI System should definitely support on-line access for both ER and CG use cases. In parallel, massive data processing (images, video, voice) is required for fulfilling project objectives (knowledge extraction). • Batch jobs can be optimized using dedicated hardware and software resources. Additionally, they can be executed when the system resources are not fully utilized (e.g. during night).

#	Requirement	Details/Motivation
R8	Event-driven architecture features	The possibility of defining internal and external triggers that cause starting specific workflows/processes when certain conditions are met. For example, when a file of the specific type (e.g. image, video, speech recording) enters the system, some particular actions are undertaken (e.g. voice processing for speech recording).
Access and interaction with end users and external applications		
R9	Communication prioritisation	Communication prioritisation based on social, environmental and other factors. For example, specific types of end users can be cut off from communication temporarily while, at this same time, others are preferred (e.g. in emergency situations).
Technical pre-requisites and requirements		
R10	Integration of C/C++ and Windows software (DLLs)	Some WP2 offline processes may need C++/Windows-DLL environment (and direct, local knowledge base access); WP2 online processes are possible in Java or as (Web) Services, e.g. search in images, analyse picture. Some C++ pre-processing tools might be necessary for WP1.
R11	Interfaces for data exchange and system coupling	It should be possible to exchange data with other systems, (accessing and retrieving) as well as to couple other software systems to the WeKnowIt system. In general, this requirement also means that the system should be open and standard-based.
R12	Support for transactions (process and database)	To keep the system state and its data consistent, especially when the system is built of the loosely-coupled components, it is important to guarantee transactions support.
R13	Platform independency	It should be possible to run the WeKnowIt system under various operating systems (Windows, Unix, Linux). This requirement is especially important during the development phase – it gives the research and development teams flexibility to use that OS environment which is the most suitable for the team.
R14	Ontology support	Data exchanged internally or with external systems should conform with the common ontology(-ies). The main reason for that is to build a "common language" understood by all system modules (specifically services prepared by the WPs).

#	Requirement	Details/Motivation
R15	Handling of multimedia content	This is rather obvious requirement as one of the main project objectives is multimedia processing.

Table 1 General requirements

3.2. Specific WPs requirements

Besides general requirements listed in the previous section, the specific Work Package requirements have been collected. As the Work Packages research teams explore different research areas, use (or plan to use) different software tools and methodologies, the differences in their specific needs are quite significant. The WKI System architecture and the set of selected technologies should address these requirements.

For each Work Package the requirements are divided into two categories:

- functional requirements
- hardware and system requirements

In the context of the WKI System, the requirements mentioned above refer to those internal WKI System features or resources which should be implemented or guaranteed to let each WP services expose specific functionalities. For the detailed service definition please refer to "The WKI System" section or to D6.2.1 document.

3.2.1. WP1

The research and development performed by WP1 concerns the design, development and testing of the methodologies and technologies for Personal Intelligence management where citizens or users are enabled to interact with the WeKnowIt system, in terms of both: uploading new information and accessing available information.

WP1 Functional requirements

Service name	Requirements
WP1_AccountManager	• The system should be able to store information about users' accounts. • The system should be able to store information about users' profiles. • Entities stored by the system can be modified and deleted. • It's possible to create links (references) between entities stored in the system.
WP1_LogIn	• The system allows to store permissions. • Permissions stored by the system can be modified and deleted.

Service name	Requirements
WP1_ManageItem	• The system should be able to store multimedia content (photo, video, audio, document). Each stored file is identified with an unique identifier (in form of URI). • Stored content can be deleted. • The system is able to put some restriction on the size of uploaded files. • Access modifiers (e.g. "public", "private", "friends only") can be assigned to multimedia content stored in the system. Each access modifier can be used by application to decide if a particular user is entitled to access this resource. The list of access modifiers is application-specific.
WP1_Tag	• To each uploaded item additional information (of textual kind) can be assigned. These information can be modified and removed (certain permissions required). • To each uploaded item a list of tags can be assigned. Tags can be modified and removed. Operations on tags require certain permission.
WP1_Comment	• To each uploaded item additional information (of textual kind) can be assigned. These information can be modified and removed (certain permissions required). • The system allows to add comments to uploaded items. Comments can be modified and removed. Operations on comments require certain permission.
WP1_Rate	• To each uploaded item additional information (of textual type) can be assigned. These information can be modified and removed (certain permissions required). • The system allows to add a rating value to uploaded item. Ratings can be modified and removed. Operations on ratings require certain permissions.
WP1_SearchKB	Effective search of data stored in the Knowledge Base part of the WKI DS.
WP1_UsersMessaging	No system requirements.
WP1_RSSManager	No system requirements.
WP1_UpdateUProfile	Users' profiles can be updated.
WP1_PushInfo	No system requirements.

Table 2 WP1 Functional requirements

WP1 Hardware and Software requirements

Service name	CPU	RAM	Storage	Database	Additional software
WP1_AccountManager	0.5-2 sec depending on the operation.	512Mb+	50Mb should be more than enough for storing user information.	*SQL/TripleStore (the first prototype does not use a triple store but the future version will require one).	OpenID Provider Server (given by UShef)
WP1_LogIn	0.5-1 sec / Connection. The process is mostly relying on two servers (RP/OP). Each server should not require a lot of resources for authenticating a user.	128Mb			OpenID Provider Server (given by UShef)
WP1_ManageItem	1-5 sec depending on the operation.	128Mb+	1mb/Item metadata.	Triple Store	
WP1_Tag	<1 sec depending on number of tags.	512Mb+	<1mb/Tag. (Tags are cheap to store but the storage size depends o the number of tagged items).	Triple Store	
WP1_Comment	<1 sec depending on the comment size.	128Mb	<1mb/Comment. (Tags are cheap to store but the storage size depends on the number of commented items).	Triple Store	
WP1_Rate	<1 sec.	128Mb+	<<1mb/Rate.	Triple Store	
WP1_Search KB					
WP1_Users Messaging				*SQL (MySQL-PostgreSQL)	XMPP Server? (OpenFire)
WP1_RSS-Manager					
WP1_UpdateUProfile	0.5-2 sec depending on the operation.	512Mb+	50Mb should be more than enough for storing user information.	*SQL/TripleStore (the first prototype does not use a triple store, but the future version will require one).	OpenID Provider Server (given by UShef)
WP1_PushInfo					

Table 3 WP1 Hardware and software requirements

3.2.2. WP2

While WeKnowIt deals with modelling and management of Personal, Mass, Social and Organisational Intelligence, most of related knowledge and information originates from raw content, in the form of e.g. text, images, video, or speech. Human annotation or tagging used in social networks is a way of representing or handling the underlying knowledge, yet despite the human intervention, content remains highly unstructured, and it is quite difficult to extract semantics and correlate with other sources of information. WP2 aims at the development of intelligent, automated content analysis techniques for different media to extract knowledge from the content itself.

WP2 Functional requirements

Service name	Requirements
WP2_Text_Classification	Metadata about uploaded items (text files) is stored in the WKI DS.
WP2_Text_Clustering	No system requirements.
WP2_Text_Annotation	No system requirements.
WP2_VisualAnalysis	The system is able to store image files. Uploaded images can have tags and geo-location information assigned.
WP2_IndexSpeech	The system is able to store audio files.
WP2_SearchInSpeech	No system requirements.

Table 4 WP2 Functional requirements

WP2 Hardware and Software requirements

Service name	CPU	RAM	Storage	Database
WP2_Text_Classification				Triple Store
WP2_Text_Clustering				Triple Store
WP2_Text_Annotation				Triple Store
WP2_VisualAnalysis	Depends on database size, for a database of 2000 images, an online query can be processed in about 15 seconds by an Intel 3.2GHz Pentium 4.	2+GB	25Mb/1000 images for indexing + images files + around 500Mb vocabulary size	MySql database, the size of the database tables is identical to the required disk storage size for indexing mentioned before.

Service name	CPU	RAM	Storage	Database
WP2_Index-Speech	Depends on settings, can take 0.1-1x real time (a 1min call will be indexed in 6-60secs)	2+GB	Proportional to the time of recordings, depends on pruning parameters, 10-200GB	RDBMS/tripple store for metadata
WP2_SearchInSpeech	Depends on the complexity of the query and the type of indices, can take several seconds for complex queries.	2+GB	The internal memory is usually sufficient for the result sets, so no additional disk space (in addition to the search indices) is necessary	RDBMS/tripple store for metadata

Table 5 WP2 Hardware and Software requirements

3.2.3. WP3

Mass Intelligence is recognition and understanding of facts and trends by exploitation of massive user contributions. Mass Intelligence is always useful when the aggregation of data, metadata and behaviour from and of a large mass of users gives new insight that would not be possible by investigating the contributions at the individual level only.

WP3 Functional requirements

Service name	Requirements
WP3_AnswerSpamDetector	
WP3_AnswerQualityEvaluator	
WP3_LocalTagCommunityDetector	
WP3_QuestionExpertFinder	
WP3_DetectLatentTopics	

Table 6 WP3 Functional requirements

WP3 Hardware and Software requirements

Service name	CPU	RAM	Storage	Database
WP3_Answer-SpamDetector	16 msec / answer @ Intel Core 2 Quad 2.4GHz	64MB	~120MB (56.3MB Lucene index + 62MB Wordnet)	Lucene index

Service name	CPU	RAM	Storage	Database
WP3_AnswerQualityEvaluator	online: 10 sec for analysis offline: creating RDF snapshot - few hrs (mostly: uploading RDF from Wikipedia and creating indexes)	online: 2+Gb, Unix system as memory mapped file is used offiline: creating RDF snapshot from Wikipedia can be up to 8Gb; reimplementing for using DB and much less memory	~6-8Gb with current in-memory implementation of RDF (Wikipedia snapshot); reimplementing to use DB with Sesame	expected: 8-10 Gb (depends on Sesame indexes; can be Oracle or MySQL) current: all stored on disk
WP3_LocalTagCommunityDetector	0.5-1 sec / community of 100-200 nodes in a network of 20K nodes and 200K edges	1024MB (could be reduced if we use neo4j for disk-based graph persistence, however, this would negatively affect speed).	Depending on the size of the graph (e.g. a graph of 20K nodes, 200K edges takes up ~2MB in custom text format).	Text files (could also be in database).
WP3_QuestionExpertFinder	1-5 sec - depends on size of input and size of used model. Most time taken by DB communication. offline - can take up to few days to generate LDA model (whole English LycosIQ dataset), in-memory implementation, CPU intensive.	online - minimum RAM requirements offline model calculations - 2-4GB preferable (the more, the better)		SQL DB (for keeping LDA model and Lycos dataset)
WP3_DetectLatentTopics	1-5 sec - depends on size of input and size of used model. Most time taken by DB communication. offline - can take up to few days to generate LDA model (whole English LycosIQ dataset), in-memory implementation, CPU intensive.	online - minimum RAM requirements offline model calculations - 2-4GB preferable (the more, the better)		SQL DB (for keeping LDA model and Lycos dataset)

Table 7 WP3 Hardware and software requirements

3.2.4. WP4

WP4 targets the analysis, recognition and understanding of needs and capabilities of communities and communication interaction patterns. It will exploit existing work in Social Network Analysis and the dynamics of social systems improving upon state-of-the-art in terms of visualisation, navigation, as well as community analysis and management techniques.

WP4 Functional requirements

Service name	Requirements
WP4_Community_Design_Language	The system must be able to store CDL entities.
WP4_ClosenessCentrality	The system must be able to store graph structures.
WP4_InverseDistanceClosenessCentrality	
WP4_BetweennessCentrality	
WP4_NewmanClustering	
WP4_SNAGraphStatistics	
WP4_AffiliationNetworkMeasures	

Table 8 WP4 Functional requirements

WP4 Hardware and Software requirements

Service name	CPU	RAM	Storage	Database
WP4_Community_Design_Language	~2s/Access Request (very rough estimation)	~1024 MB (very rough estimation)	~100MB (very rough estimation)	~4GB (very rough estimation)
for all graph related services	loading graph: 10min for ~200.000 nodes	~2-4GB	~1GB	PostgreSQL, ~10GB (depending on number of concurrently stored graphs)
WP4_Closeness-Centrality	CPU and RAM requirements depend on graph size, algorithms do not scale linearly, no general statement possible	>2GB RAM might be required for large graphs, not including RAM required for storing graphs		
WP4_InverseDistanceCloseness-Centrality				
WP4_BetweennessCentrality				
WP4_Newman-Clustering	<1sec for 100 vertices (service intended for smaller graphs)	<64MB for 100 vertices		
WP4_SNAGraphStatistics				
WP4_Affiliation-NetworkMeasures				

Table 9 WP4 Hardware and software requirements

3.2.5. WP5

WP5 brings the innovation of Web 2.0 technologies to the field of Organisational Intelligence where processes and workflows are set up in order to bring the right piece of knowledge at the right time to the right person in the organisation in order to support decision making. In WeKnowIt, however, this knowledge is not necessarily produced by the individual knowledge worker, but rather by the interaction with the layers of Personal, Media, Mass and Social Intelligence.

WP5 Functional requirements

Service name	Requirements
WP5_GroupManagement	The system should be able to store information about: • groups, communities and their profiles. • user roles, • user organizational profile.
WP5_TaskManagement	The system should be able to store information about tasks. Tasks can be assigned to users or groups. Resources (uploaded items) can be assigned to tasks. Relations between tasks are possible ("subtasks").
WP5_IncidentLog	
WP5_LogMerger	

Table 10 WP5 Functional Requirements

WP5 Hardware and Software requirements

Service name	CPU	RAM	Storage	Database
WP5_Group-Management	10+ seconds for importing group structures (includes online access to other URLs)	128 MB	100 MB	Triple Store
WP5_TaskManagement	3 seconds for preserving integrity of KB	128 MB	100 MB	Triple Store
WP5_Incident-Log	3 seconds for preserving integrity of KB	128 MB	100 MB	Triple Store
WP5_LogMerger	0.5-1 sec for merging and indexing 3-5 moderate size log files	128MB	Depends on whether WP2 will provide NE and verb recognition. If not, then thesauri and lexical resources are estimated to ~200MB.	No.

Table 11 WP5 Hardware and software requirements

3.2.6. Use-cases (WP7) requirements

According to the definition of the WKI System, the WP7 applications are not a part of the WKI System. However, use-cases Work Package (WP7) plays a special role among others WPs. End-user applications developed by the WP7 decide about the general usability of the whole system and its capabilities.

This section lists the requirements (mainly related to efficiency and scaling) which should be fulfilled by the WKI System in order to let the WP7 applications provide their functionalities and do it in an efficient way.

Use-case - Emergency Response

Parameter	Value
Total number of users	500,000
Number of registered users	5,000
Number of concurrently working users	200
Average number of files per user	2
Average size of file	500 kB
Upload of file [number of requests per minute]	40
Search [number of requests per minute]	100
Advanced search [number of requests per minute]	5
Retrieval of file [number of requests per minute]	500

Table 12 WP7 ER requirements (sizing)

Use-case - Consumer Social Group

Parameter	Value
Total number of users	500,000
Number of registered users	50,000
Number of concurrently working users	200
Average number of files per user	2
Average size of file	1 mb

Upload of a file [number of requests per minute]	5
Search [number of requests per minute]	10
Advanced search [number of requests per minute]	5
Retrieval of a file [number of requests per minute]	50

Table 13 WP7 CSG requirements (sizing)

4. The WKI System – technologies

Before discussing the architecture, technologies used in the WKI System are presented. Main features of each technology are described. Moreover, information about reasons for using it in the WKI project is included, as well as remarks regarding other competing technologies that were taken into consideration during technology research and work on the WKI System architecture design.

Selected aspects of some technologies – e.g. dependency management provided by Maven – are described in details, because their features have significant influence on other topics – i.e. Quality Assurance (section 7) and Integration Plan (section 8).

4.1. Selection of technologies

Today almost for every technology there exist a wide range of substitutes. Choosing the suitable technology, framework or library from the plethora of solutions is not an easy task. Competition on technology market is fierce and every project claims its superiority. One can try to evaluate a technology by reading its documentation and executing examples presented in tutorials. Many aspects of the technology can be examined in this way, but it is very time consuming[1]. Besides, as many projects have adopted the "release early, release often" mantra[2], there is a possibility that new version of particular used technology will be released during the development phase so that all tests will have to be regressively applied once again.

On the other hand, even if time was not limited and new versions were not released so frequently it still would be questionable if such a "right" choice could be made at all. It is only possible to choose the most suitable technology on the basis of system characteristics known at a given moment in time. Nothing certain can be claimed about its future development – will the commonly known errors be fixed? Will the project be abandoned? Will the community gathered around the project still be active or will it shrink to a couple of people? Will the other projects be developed better and faster? Moreover, there may exist much better solution not known to the decision maker. Thus, appropriately designed and applied tests are insufficient to prove that a particular technology is a perfect choice. This implies that choice cannot be based solely on the technical aspects.

The decision should be also partially based on individual experience and intuition. Other reasonable factors are technical skills of the development

[1] The amount of time needed to evaluate a technology is especially high, if one goes beyond simple examples from tutorials and creates a test projects which resemble a real enterprise situation.

[2] This idea comes from the agile software development.

team and time reserved for the decision process that can narrow the possible choices.

4.1.1. Technologies for the WKI System

During the process of choosing the most appropriate technologies for the WKI System, WP6 team has stumbled across all of the problems described in previous paragraphs.

Requirements

In general, each technology has been reviewed against the requirements of the WKI System. The general requirements for the technologies, along with common factors like: robustness, stability, good community support, etc., included:

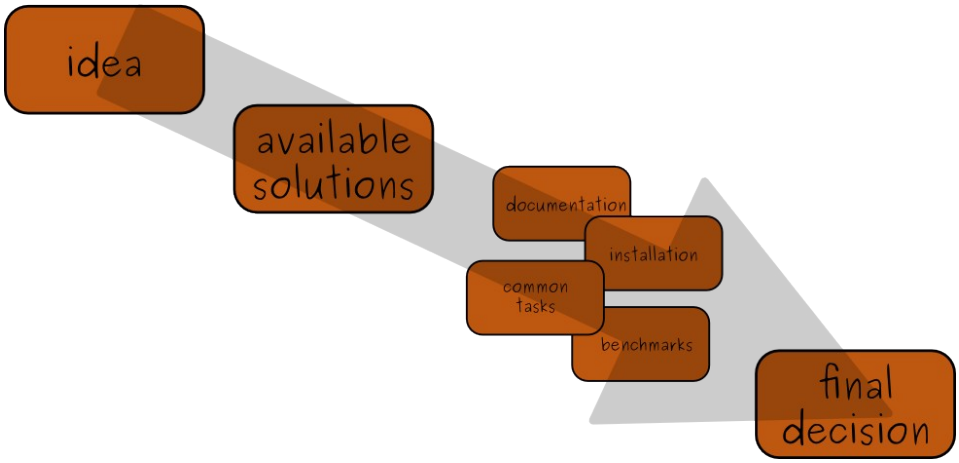
- perspective of development,
- ease of use,
- availability of IDE/tools support,
- open-source,
- compatibility with standards (e.g. with JSR[3]),
- ease of integration with other leading frameworks.

From the very beginning the technology choice was narrowed to the Java-compatible technologies – this decision was made by the whole project consortium. Two companies – LYC and SMIND – were involved in technology research.

Process of technology selection

Picture 2 presents the process of technology choice realised in the WKI project. At the beginning, a general conceptual approach must be chosen, what refers to decisions like "SOA or not SOA". Next, some existing solutions were identified (e.g. Apache ServiceMix, OpenESB and Mule) and carefully examined in terms of documentation, benchmarks and another significant aspect. Some technologies were rejected at very early stage (e.g. because of lack of documentation or no community activity) but some of them required more detailed investigation before being recognized as unsuitable by the WP6 research team.

[3] JSR, Java Specification Request is the actual description of proposed and final specifications for the Java platform. JSRs are reviewed by the Java Community Process and the public before a final release of a specification is made.



Picture 2 Process of technology assessment

Obviously, the process was neither linear as presented on Picture 2, nor a "1 or 0" choice. For example, during the analysis of content repository solutions it occurred that Alfresco SDK installation is very troublesome,[4] while, on the other hand, Apache JackRabbit is very easy to set up. Unfortunatelly, this was not enough to make a decision. Because of importance of content repository to the whole WKI System Alfresco has been examined thoroughly and it was rejected much later due to benchmark results.

It has to be emphasized that choice of a one concrete solutions influences other decisions. Namely, it narrows further choices to some subset of technologies possible to integrate. In case of the WKI project, such a situation took place with Apache ServiceMix selected as SOA solution. Apache ServiceMix integrates out-of-the-box with many other libraries, including the ones developed by Apache community (e.g. Apache Camel, ActiveMQ, etc.) and it is justified to use these already existing frameworks.

Dead-ends

Presented process supports the choice of technologies but actually it is not adaptable to all possible circumstances. To exemplify this issue we discuss problem with Apache ServiceMix. The latest available version of the technology is based on JBI concept. Process of examining and testing of JBI was very complex and time consuming. The framework seems to be quite promising so that some of the guidelines prepared by WP6 team refers to this solution. However, despite the effort put into this task, such technology has never emerged – OSGi[5].

Such "dead-ends" should never be considered as a waste of time, but rather as a necessary step towards choosing the most suitable, mature and stable solutions. Experience gathered during such research helped to make better future decisions.

[4] To be more precise the real problem was to build it from the sources with Maven, because Alfresco uses many libraries that are not available in any popular repo.

[5] Apache ServiceMix version 4 supports OSGi and JBI at once.

4.1.2. Results of the technology selection process

The process of technology selection has resulted in a set of technologies, that will be used in the WKI project. The chosen solutions are claimed to be:

- robust – proved to work in enterprise architectures,
- modern – build on the experiences gathered by previous projects,
- compatible with standards and possible to integrate with other frameworks, e.g.:
  - Apache JackRabbit – JCR compatible,
  - Apache Camel – ease of integration with Spring,
  - Maven – able to use Ant tasks,
  - MySQL – JDBC compliant.
- prospective – currently developed and supported by serious companies or communities, e.g.:
  - Apache Camel is close to release version 2.0 – first 2.0 milestone has been released 17.03.2009,
  - Apache JackRabbit – getting close to release 1.6.0 version,
  - OSGi – promising efforts towards distributed OSGi (R-OSGi project),
  - Apache ServiceMix – moving towards version 4.0 release,
  - Maven – version 2.1.0 has just been released (23.03.2009).
- well documented.

Not all of the selected technologies are easy to learn but developers are provided with a set of guidelines[6] which are expected to help them with potential problems.

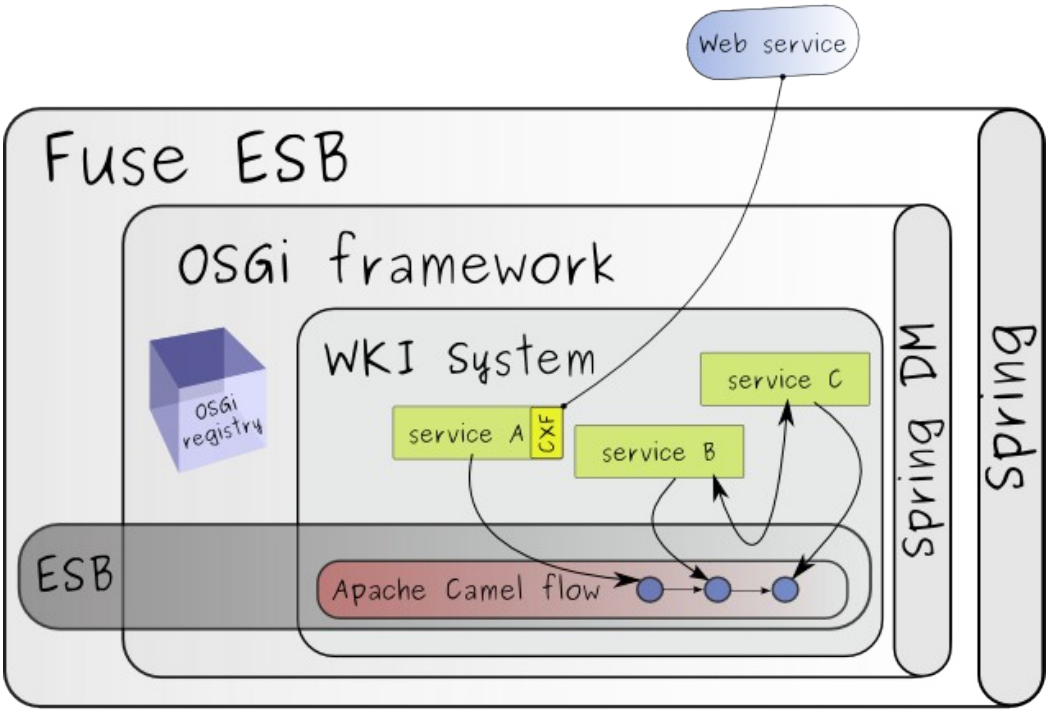
4.1.3. Versions of technologies used in the WKI System

Next sections contain description of technologies including information about concrete version used in the WKI System. Due to the long development phase it is planned to use newer versions of some libraries which provide necessary additional features, improve robustness or are free of potential newly found bugs.

4.2. WKI technologies stack

This section provides short explanation of place and role of all technologies used in the WKI System. This is illustrated in the Picture 3. Following sections provide detailed descriptions of every framework mentioned here.

[6] For more information on guidelines, please refer to D6.2.1.



Picture 3 WKI technologies stack

WKI service (section 5.3.1) is an OSGi bundle (section 4.6) registered in the OSGi registry. It can be exposed as Web service (section 4.12) by Apache CXF (section 4.13). WKI services can communicate with each other directly via ESB (section 4.4) or can be bound by Apache Camel (section 4.11) which uses ESB for messaging underneath.

All WKI services live in OSGi framework and are configurable by Spring DM (section 4.10) using Spring (section 4.9) configuration files. OSGi framework is a part of Fuse ESB (section 4.8).

4.3. Service Oriented Architecture (SOA)

In D6.1.1 detailed information about the reasons for choosing SOA and ESB as general architecture approach can be found.

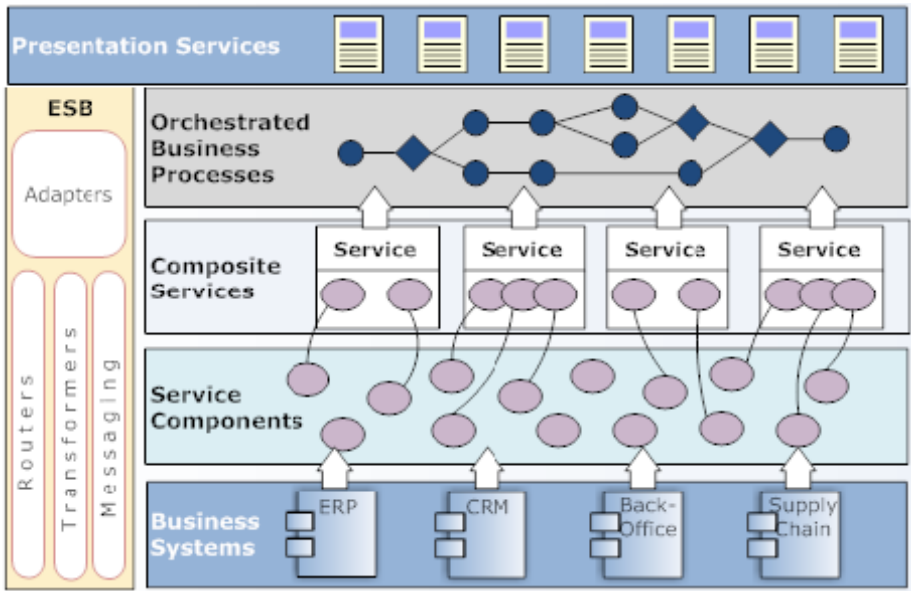
Service Oriented Architecture (SOA) promotes the usage of reusable services. Ideally, these services are well-defined and stateless, and can be combined and reused in order to create business applications.

SOA does not pose any restrictions on the implementation of the services, as long as they can cooperate with one another. On the contrary to the popular beliefs, SOA does not demand any particular communication protocol neither – even though, the communication via web services (HTTP[7] + XML[8]) is the most popular choice.

[7] Hypertext Transfer Protocol, http://www.w3.org/Protocols/

[8] Extensible Markup Language, http://www.w3.org/XML/

The idea of SOA architecture has gained a lot of attention because of its applicability to integration tasks. SOA based solutions are flexible, which addresses the needs of many business applications.



Picture 4 SOA environment (from 12)

Picture 4 presents typical layers of SOA application. Each of the elements inside Service Components tier can communicate with one of Business System and allows the layers above to interact with these systems. In Composite Services layer functionalities of service components are grouped into more course-grained services. Not all components functionalities should be accessible by external components. Composite services can be used to create flows that realizes some more complex functionality – this is represented by Orchestrated Business Processes. All of this layers (except presentation layer) use Enterprise Service Bus (ESB) to communicate with one another.

4.3.1. SOA alternatives

Alternatives to SOA approach are discussed in D6.1.1. Three of them were described:

- data level integration (applications communicate through reading/writing operation to shared data sources (e.g., files or databases) with agreed formats),
- classical application programming interfaces (APIs which need for the exchange of "header files", that approach results in strongly coupled services),
- client-server architecture (centralisation of some parts of the application and system management).

All of the above-mentioned alternatives do not provide flexibility comparable to the one offered by SOA solution. Adding new services or changes in the services cooperation manner might enforce substantial changes in the system.

4.3.2. Reasons for using SOA in the WKI System

To explain why SOA approach fits so well as a main architecture solution for the WKI System, it is necessary to realize how development tasks are shared between WPs. The WKI services (which constitute building blocks of the WKI System) are prepared, in most cases, by independent teams of researchers. Each service is produced and based on the specification provided by a particular WP. It can use any 3rd party libraries that development team finds suitable and it can offer API that is suitable from the point of a particular WP[9].

The final WKI System needs to bind these small components together. In such environment the main challenge is to integrate these parts and create a cohesive application. Integration based on SOA works even for such heterogeneous elements.

The WeKnowIt system is meant to serve as a platform which provide a collection of tools and methods that can be recombined into several different scenarios or user interfaces. It means that the system should not impose any strong coupling between services. Flexibility and future modifications are the top priority for the WKI System architecture. SOA, which promotes use of loosely coupled services, is able to fulfil these requirements.

4.4. Enterprise Service Bus (ESB)

Enterprise Service Bus can be considered as an infrastructure (a "backbone") for SOA – it is responsible for protocol conversion, message format transformation, routing, accepting and delivering messages from various linked services and applications. To cut the story short, ESB is responsible for communication between services integrated in SOA.

In case of WKI System ESB allows to achieve flexibility. By using ESB solution two goals are achieved:

- WKI services are not required to manage the communication – it is handled by ESB,
- use of ESB promotes loosely coupling of services, which may be unaware of specification of other services (e.g. service A does not need to know input structures of service B, because ESB takes care of necessary message transformations).

[9] In D6.2.1 efforts towards creation of coherent APIs are described.

4.4.1. ESB alternatives

SOA can be realized also with other tools than Enterprise Service Bus. They were however rejected because they do not guarantee flexibility and scalability required by the WKI System.

Point to point

In Point to Point approach communication for each pair of application is defined separately. This solution is well suited for systems, where number of integrated services is relatively small and communication paths are well defined right from the beginning. If this is not the case, then Point to Point approach is not recommended, as it does not offer flexibility required for present applications.

Hub & Spoke (also known as Message Broker)

In Hub & Spoke approach all applications are connected to one central Hub. Message transformation and routing takes place within the Hub.

Hub & Spoke architecture has serious advantage over Point to Point approach:

- in terms of required number of connections which is O(n) instead of $O(n^2)$,
- spokes can be replaced much easier.

The major drawback of Hub & Spoke architecture is the single point of failure (Hub).

Compared to ESB, Hub & Spoke is much more centralized solution. On the other hand, ESB supports distributed operations and management, which results in better scalability.

4.5. Java

Java is an object oriented, OS-independent language of general use developed by Sun Microsystems. Source code written in Java is transformed by Java compiler to bytecode, which is executed by Java Virtual Machine (JVM). This approach allows to achieve the "write once, run anywhere" feature and makes the execution of Java applications on any platform for which JVM exists possible.

There are many reasons why Java has been chosen as a primary language for the WKI System development. Java is very popular and all WKI partners have some experience with it. There exist many flourishing Java communities and infinite amount of information, documentation, user guides, tutorials, FAQ on every Java-related topic. Plenty of great tools of any kind are available, many of them are open-source and free to use, including IDEs (e.g. Eclipse IDE[10], NetBeans IDE[11]). There are also many lib-

[10] http://eclipse.org

[11] http://netbeans.org

raries and frameworks ready to use, well-documented and exercised by thousands of projects and developers.

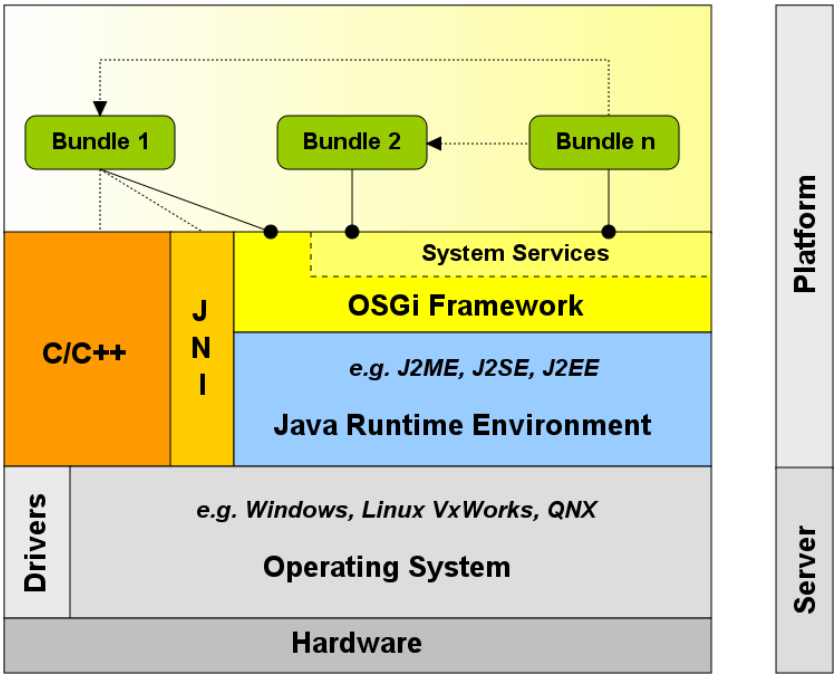
Java has been proven as a robust and enterprise-ready platform - nowadays it is an approved industrial standard. Java is still being developed and new libraries and frameworks, which facilitate creation of applications, emerge every day.

The cross-platform feature of Java allows to produce applications not only for desktop computer but also for various mobile devices – which was considered very important for the WKI use-cases.

The use of Java as primary language for the project was accepted by the WKI consortium at the very beginning of the project.

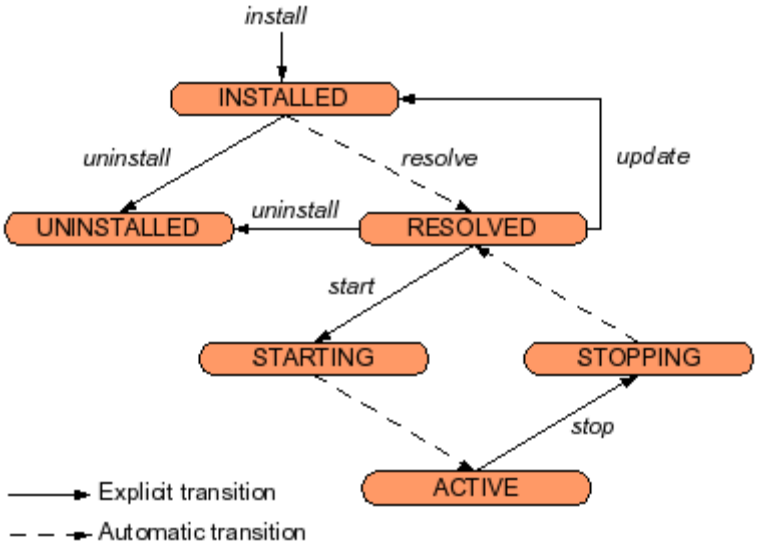
4.6. OSGi

OSGi specifies the "dynamic module system for Java" [18]. The idea of OSGi comes from networked and mobile devices. If OSGi Service Platform is running on such a device it is possible to manage the lifecycle of software components deployed that device remotely from anywhere in the network.



Picture 5 OSGI Layering (from 2)

OSGi component is called a "bundle". OSGi specification defines bundle lifecycle. Picture 6 presents available states of bundle and possible transitions between them. Some transitions can be performed on demand while others are triggered automatically by OSGi framework.



Picture 6 OSGi bundle lifecycle (from 1)

OSGi technology facilitates building of flexible and dynamic solutions. OSGi bundles can be installed, updated or removed on the fly without having to disturb the operation on the device (for example it is possible to deploy a new component, reset it or uninstall it without restarting the whole container). OSGi frameworks comes with set of services that allows to monitor and change status of deployed modules.

Some of the features offered by OSGi were regarded as very valuable for the WKI project, for example:

- hot-swap is required in "mission critical" applications (e.g. ER use case),
- new functionalities may be added in run-time, which is important for globally available services (e.g. CSG use case),
- OSGi components can be managed (monitored, activated and deactivated) which helps to manage running production systems,
- OSGi technology encourages developers to create small components that all together constitute an application. These components can later on be composed into the system and co-operate. Reusability is provided as well. Concepts of modularity and component-based software facilitate development of applications by dispersed and independent teams like the WKI team.

4.6.1. OSGi service model

OSGi service model allows JAR[12] files to import and export services made up of classes located in this concrete JAR, or classes available in other JARs[13]. One can think about this in analogy to concepts of access modifiers

[12] JAR, Java Archive, file format used to distribute java classes, based on popular ZIP format.

[13] A JAR file with OSGi information included In MANIFEST.MF file is called a „bundle".

for Java classes and Java packages. These two allow to expose from the package to the "outside world" only selected methods (by using public access modifier) while the rest of classes remain hidden inside a package and is available only for internal use. OSGi service model takes this concept one level of abstraction higher, and provides similar functionality to libraries (JAR files). This enables better control of the usage of each service.

OSGi offers very strict visibility control – only services exposed by bundle creator are visible by other services. It is not possible to make use of unexposed methods even with reflection or any other classloading trickery.

The WKI project can benefit from this strict visibility control in a way ensure that only certain parts of the code developed by WPs will be visible to others. This allows WPs to utterly control the way their code is accessed (it is not possible to bypass the exposed API and call the service internal classes) what increases manageability and simplifies changes to existing configuration.

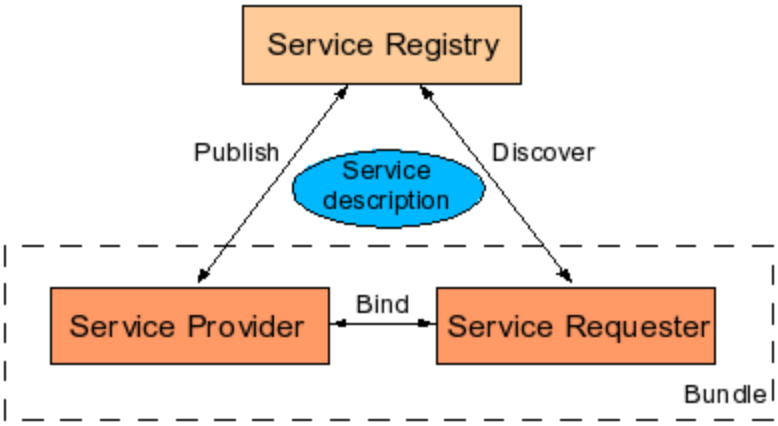
4.6.2. OSGi registry

When JAR file is deployed to OSGi framework it is inspected by OSGi in order to discover services. OSGi framework keeps track of all services exposed by JARs installed on JVM. No other service can look up this OSGi registry. This model resembles the well-known WSDL/SOAP/UDDI[14] and JNDI[15] approach, however, there is one important difference: OSGi operates on a single JVM level and has a minimal overhead comparing to UDDI or JNDI lookup.

Services platform is a software platform that supports so called service orientation interaction. This interaction involves three main factors - service providers, service requesters and service registry, however, only service registry belongs to the services platform. Service providers publish service descriptions while service requesters discover services and bind them to service providers. Publication and discovery are based on service description.

[14] WSDL – Web Services Description Language, SOAP – Simple Object Access Protocol, UDDI - Universal Description Discovery and Integration are all parts of the web services model, see http://www.w3schools.com/WSDL/default.asp.

[15] JNDI, Java Naming and Directory Interface, part of the Java platform providing applications based on Java technology with a unified interface to multiple naming and directory services, see http://java.sun.com/products/jndi/.



Picture 7 Service orientation interaction (from 1)

The process of registering and looking up OSGi services can be further simplified by using dependency injection mechanism provided for OSGi by project Spring Dynamic Modules for OSGi™ Service Platform.

4.6.3. Open-source and enterprise usage

OSGi is getting a lot of popularity nowadays and few robust implementations already exists – i.e. Apache Felix[16], Equinox[17], Knopflerfish[18].

OSGi is used in big and well-recognized projects, for example:

- Eclipse[19] Runtime and JOnAS[20] server are based on the OSGi notion of bundle,
- popular DI Spring Framework[21] offers integration with OSGi bundles via Spring Dynamic Modules subproject,
- many enterprise products from IBM (e.g. WebSphere[22] 6.1), Bea and Oracle are making heavy use of OSGi modularization concept.

4.6.4. OSGi vs. JBI

There are many other specifications aiming at the creation of modular Java components environment. One of the alternatives to OSGi is Java Business Integration (JBI). Both OSGi and JBI offer a platform for services componentization but OSGi has serious advantages over JBI:

1. OSGi offers hot-deployment mechanism,
2. OSGi services configuration is easier than JBI
    a. OSGi: MANIFEST.MF file and typical JAR file structure

[16] http://felix.apache.org/

[17] http://www.eclipse.org/equinox/
[18] http://www.knopflerfish.org/
[19] http://eclipse.org
[20] Java Open Application Server, http://jonas.objectweb.org
[21] http://springframework.org
[22] http://www.ibm.com/websphere/

   b. JBI: many XML configuration files and JBI-specific file structure

3. OSGi services creation is better supported by popular tools (i.e. Maven[23]),
4. R-OSGi[24] is a promising distributed version of OSGi, which allows to access remote services (across JVM) in an entirely transparent way.

The above-mentioned reasons made Apache ServiceMix switched (between versions 3 and 4) from JBI to OSGi as main component technology and because of this the WKI System will also be based on OSGi technology. Previously, as stated in D6.1.1, the WKI System was intended to use JBI.

4.6.5. OSGi and the WKI System

During the work on the WKI System architecture various aspects of OSGi technology were examined. This task was especially important because all the WKI services, developed by WPs, will take form of OSGi bundles. This made WP6 look for tools and techniques of creation of OSGi bundles, that are not only robust and reliable, but also easy to use. In particular, creation of bundles using BND tool[25] and Maven Bundle Plugin[26] was examined.

Creation of OSGi bundles is only a beginning, other aspects of OSGi technology, and cooperation of OSGi bundles with other frameworks of the WKI system, were also examined:

- bundles activation by extending org.osgi.framework.BundleActivator class (which allows bundles to execute some initialization code during deployment),
- registration of services in the OSGi registry,
  - programmatically (by using org.osgi.framework.BundleContext class),
  - declaratively (with Spring-DM),
- listeners (allowing services to react on various system events),
- hot deployment of bundles.

The next three options that were examined are very important from the integration point of view. They allow to configure services deployed in form of OSGi bundles and bind them without introducing any changes to the original bundles:

- configuration of services deployed separately (in different bundle) than the service itself,

[23] http://maven.apache.org

[24] http://r-osgi.sourceforge.net/
[25] http://www.aqute.biz/Code/Bnd
[26] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

- deployment of Camel flows packed as OSGi bundles,
- use of AspectJ[27] for bundles enhancement.

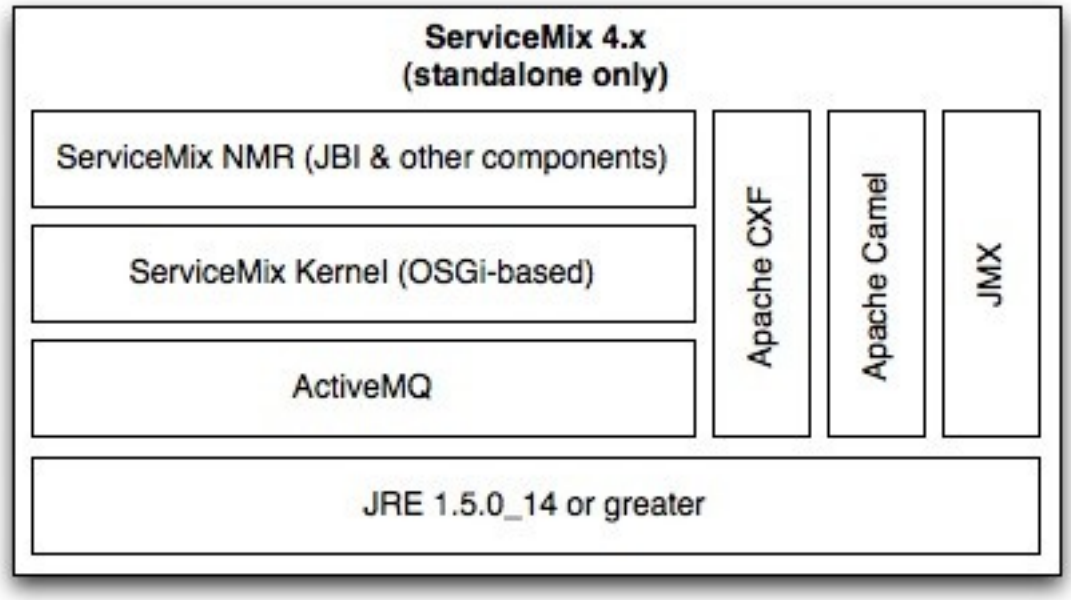
The guidelines presented in D6.2.1 describe process of writing of the WKI services in form of OSGi bundles. Maven POMs (see sections 4.15 and 7.3) prepared by WP6 facilitate this process.

The WKI System uses version 1.2 of Apache Felix - open source implementation of the OSGi Release 4 core framework specification[28].

4.7. Apache ServiceMix

Apache ServiceMix[29] is lightweight and easily embeddable open source distributed ESB and SOA toolkit with integrated Spring support released under the Apache License. It is a popular framework with good community support.

Previous versions of Apache ServiceMix used JBI to integrate components and services ,but since version 4.0, Apache ServiceMix is also an enterprise OSGi container offering full OSGi support (via Apache Felix OSGi framework). Apache ServiceMix provides plug and play mechanism to install and run components.



Picture 8 Apache ServiceMix 4 layers (from 3)

Apache ServiceMix makes use of various open-source projects. Apache ActiveMQ[30] - the message broker which is a complete implementation of the Java Message Service[31] 1.1 (JMS) – is used for messaging. The SOAP

[27] AspectJ is an aspect-oriented extension of the Java language; http://www.eclipse.org/aspectj/

[28] http://www.osgi.org/Specifications/HomePage

[29] http://servicemix.apache.org

[30] http://activemq.apache.org/

[31] http://java.sun.com/products/jms/

support is provided by fully featured Service Framework Apache CXF[32]. Routing is achieved with the use of Apache Camel[33].

The advantages of ServiceMix are as follows:

1. it is a lightweight solution,
2. it supports POJO making it easy to unit test the components,
3. it is easy to debug components and applications,
4. it can be clustered (allowing high availability),
5. it makes use of other open-source projects, not trying to reinvent the wheel but focusing on integrating them in order to provide an enterprise-ready ESB solution.

Use of POJO means that no special steps must be taken by the WPs during the creation of services. The classes provided by WPs does not need to implement any interface, inherit from any base class or even contain annotations. This facilitates unit testing (see section 7.1.1) which allows to achieve better code quality.

The WKI System uses version 4.0 of Apache ServiceMix (which is a foundation of Fuse ESB).

4.8. Fuse ESB

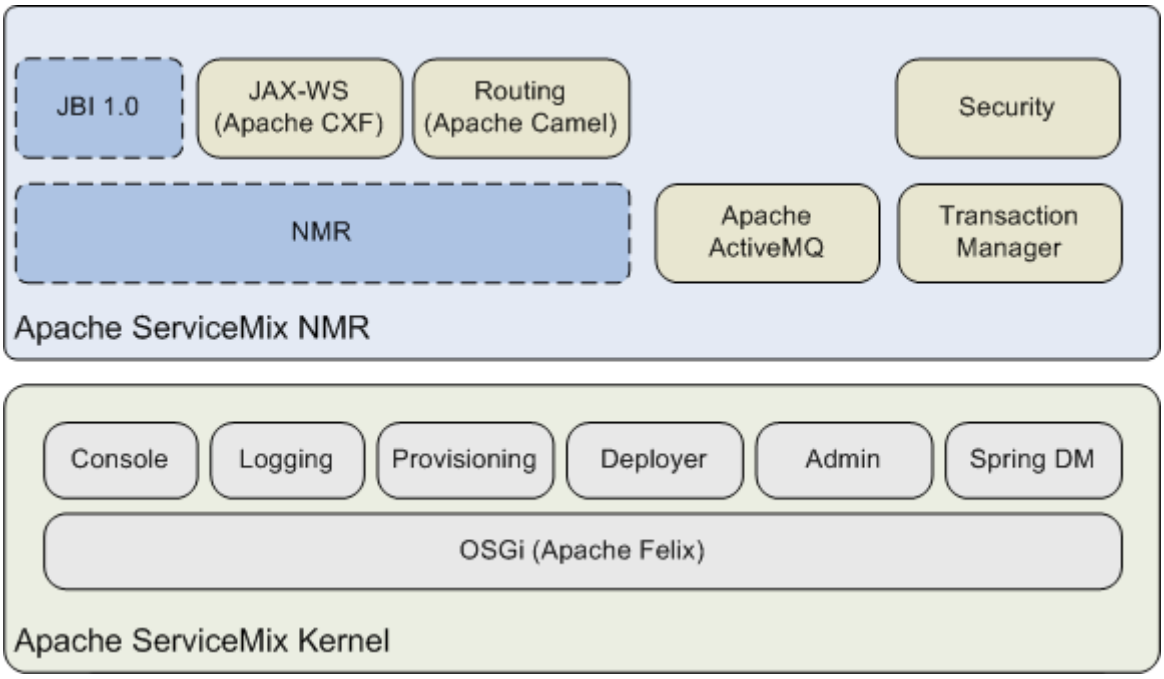
Fuse ESB[34] is an open source Apache ServiceMix based on integration platform with JBI 2.0 and OSGi support. Fuse ESB still implements JSR-208[35] so it supports JBI 1.0 too (till version 3.4 only JBI 1.0 was supported). Fuse ESB is ServiceMix enterprise version with SOA infrastructure that provides a server, standardized methodology and tools to integrate components in applications.

[32] http://cxf.apache.org/

[33] http://camel.apache.org

[34] http://www.fusesource.com

[35] Specification of JBI 1.0 made by JSR community: http://jcp.org/en/jsr/detail?id=208



Picture 9 Fuse ESB 4 architecture (from 5)

Fuse ESB was chosen instead of "bare" Apache ServiceMix because it offers some additional features. Apart from enhanced documentation and community support, Fuse ESB can be regarded as additionally tested version of Apache ServiceMix. Released versions of Fuse ESB often include features that are not yet available in Apache ServiceMix. This is the case of OSGi support, which, at the time of writing this document, is available only in snapshot versions of Apache ServiceMix. But stable release of Fuse ESB already provides OSGi support, thanks to additional work made by Fuse ESB team.

The WKI System uses version 4.0.0.4 of Fuse ESB.

Enterprise usage

Fuse ESB has been proved to be enterprise-ready. Lately (March 2009) Sabre Holdings[36] has announced that in order to "ensure continuous uptime and customer ease of use within its integration platform, known as the Sabre Supplier-Side Gateway"[28] Fuse ESB has been chosen as a part of underlying technology. The requirements of Sabre open systems and legacy hosts reach a peak of 32000 transactions per second. Some advantages of Fuse products were stressed by Sabre:

- its open-source character,
- strict adherence to industry standards,
- flexibility allowing for easy adjustment to rapidly changing industry.

Also other spectacular uses of Fuse ESB are known. Belgium Ministry for Education uses Fuse ESB "to integrate the multiple applications, servers

[36] http://www.sabre.com/

and databases in order to facilitate real-time registration and validation of over one million student records." [29] This task involves integration of over 8,000 various client applications, which uses different formats for storing ~1.5 million of school students records.

4.9. Spring framework

Spring framework[37] is based on DI[38] mechanism. It simplifies the creation of J2EE applications and promotes good coding practices (use of interfaces, easy testing by using POJOs, flexible architecture etc.). Spring framework[39] is de-facto standard in Java world. The success of Spring has resulted in emerging of many other frameworks and libraries that benefit and enhance Spring capabilities.

Many technologies used by the WKI project use Spring internally – e.g. Apache Camel, Spring Dynamic Modules, Apache JackRabbit – and many of them are configured with configuration files using Spring namespaces. The configuration files of the WKI services also use these namespaces.

4.10. Spring Dynamic Modules

The Spring Dynamic Modules for OSGi Service Platform[40] (Spring-DM) is a subproject of Spring. This project integrates two popular technologies – Spring framework and OSGi. It gives Spring applications access to all features of OSGi technology (e.g. hot-deploy of services). At the same time it gives OSGi services access to the world of Spring-compatible technologies.

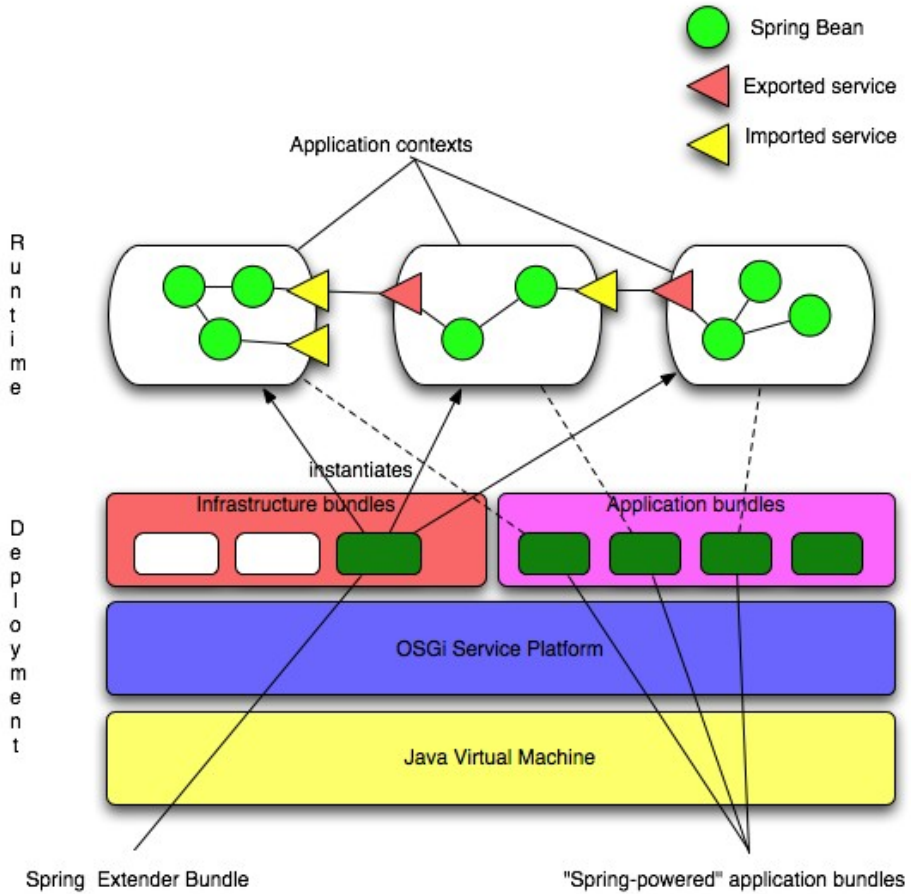
Spring-DM is easy to use and for most OSGi applications provides the low-level coding and could be responsible for creating, running and life-cycle managing. Spring-DM also supports unit and integration tests of OSGi components.

[37] http://www.springsource.org/

[38] DI, Dependency Injection – the concept of managing dependencies of services, where service is not responsible for acquiring required dependencies (via for example service registry) but the dependencies are provided to it by framework. This concept has been proved to simplify many problems with traditional way of managing services dependencies.

[39] http://www.springsource.org/

[40] http://www.springsource.org/osgi



Picture 10 Spring-OSGI integration (from 6)

Spring-DM is tested for compatibility with three OSGi frameworks: Eclipse Equinox 3.2.x, Knopflerfish 2.2.x and Apache Felix 1.x.

Spring-DM shields developers from the internal workings of the OSGi and allows to express dependencies between modules (OSGi or simple Spring beans[41]) in an uniform way via XML configuration, which is well-known to every Spring developer. It allows to omit programmatic initialization of OSGi bundles and replace it with much easier XML configuration.

Spring-DM is used in the WKI System because it provides features valuable not only for integration layer, but also for every WP that develops services:

- configuration of OSGi bundles is much easier with Spring DM than using standard, programmatic technique – all the WKI services (which technically are OSGi bundles) are instantiated and configured with use of Spring DM,
- Spring DM allows OSGi bundles to be treated like "first-class" Spring citizens, which makes them easy to use by all Spring-compatible technologies (e.g. Apache Camel).

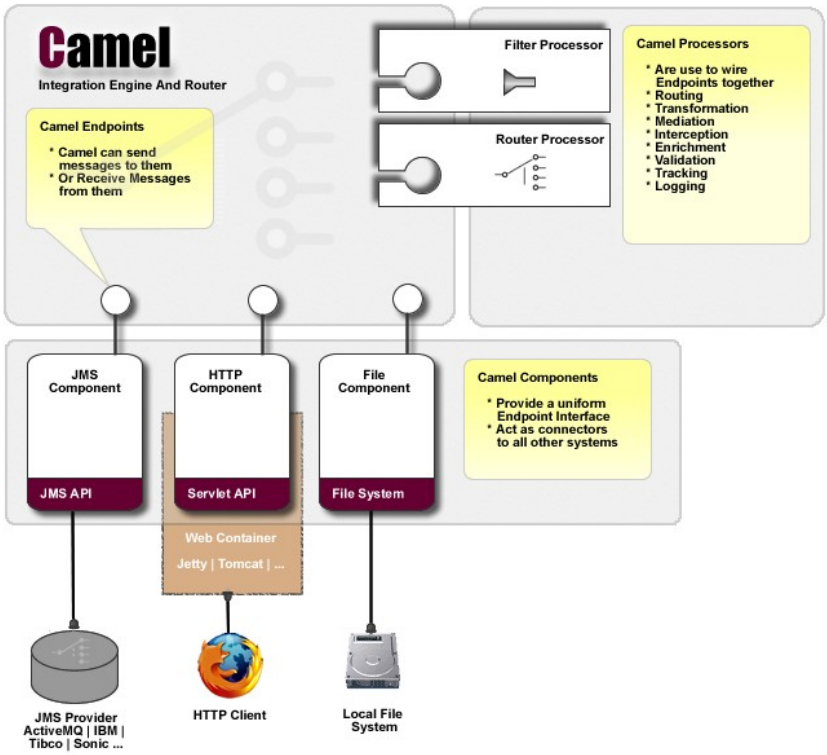
[41] Spring bean is a class managed by Spring framework, see http://static.springframework.org/spring/docs/2.5.x/reference/index.html

The WKI System uses version 1.1.0 of Spring Dynamic Modules.

4.11. Apache Camel

Apache Camel is an open-source, rule-based mediation and routing engine. It combines the power of the Enterprise Integration Patterns (EIP) (see [1]) with the ease of POJOs and beans usage. The EIP patterns facilitates integration tasks while use of simple objects offers much easier testing, thus helping to achieve overall higher quality.

Apache Camel is able to work with any kind of messaging or transport model such as HTTP, ActiveMQ, JMS, JBI, SCA[42], MINA[43] or CXF Bus API and provides a common API regardless of the kind of transport solution used. It also integrates with popular DI frameworks like Spring or Guice[44].



Picture 11 Apache Camel components (from 7)

4.11.1. Apache Camel and the WKI System

Reasons for choosing Apache Camel for the WKI System architecture are as follows:

1. it allows to build very complex flows and supports many EIP[45],

[42] SCA, Service Component Architecture – a specification of technology-agnostic SOA components.

[43] Apache MINA, http://mina.apache.org/
[44] http://code.google.com/p/google-guice/
[45] http://camel.apache.org/eip.html

2. it is flexible and extensible,
3. it can use Spring beans directly as message receivers (in case of BPEL POJOs must be first wrapped as Web services),
4. it offers a wide variety of endpoint components[46] which can be addressed via URIs in the flows configuration,
5. it can be configured via Spring-based XML configuration files or with DSLs (written in Java, Scala[47], ...),
6. Apache Camel integrates "out-of-the-box" with Fuse ESB.

Apache Camel plays a prominent role in the WKI System integration layer. It is used to bind services together. Examples of usage of Apache Camel are presented in section 5.

During the work on the WKI System architecture, various features of Apache Camel were tested, especially:

1. ability to create and deploy camel-based flows included in OSGi bundles,
2. use of various EIP,
3. declarative (XML-based) and programmatic (Java DSL) configuration of flows,
4. use of Spring beans and OSGi bundles as members of a flow.

The WKI System uses version 1.5.2 of Apache Camel.

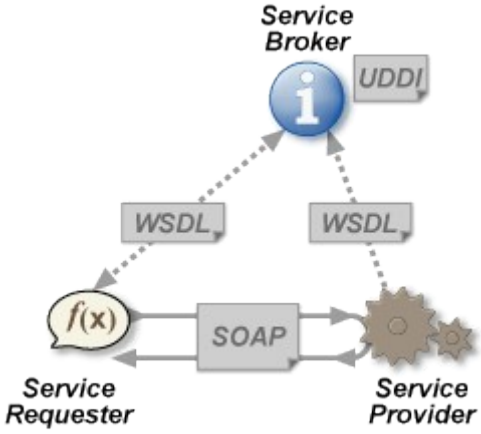
4.12. Web services

According to W3C[48] definition Web service is "a software system designed to support interoperable machine-to-machine interaction over a network" [26]. In other words, it is an application programming interface (API) which can be accessed via network and used by other applications.

[46] http://camel.apache.org/components.html

[47] Java based object oriented language running on the JVM, see http://www.scala-lang.org

[48] W3C, The World Wide Web Consortium, http://www.w3.org/



Picture 12 Interaction via Web services (from 8)

The API of the WKI System is based on Web services. External applications, running on remote machines, can access functionalities provided by the WKI System via Web services. Section 5.4.1 describes this topic.

4.13. Apache CXF

Apache CXF is an open source services framework. It allows to "build and develop services using frontend programming APIs, like JAX-WS[49]. These services can speak a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and work over a variety of transports such as HTTP, JMS or JBI." [23]

The most important areas covered by CXF are:

- support for Web service standards – SOAP, WSDL, WS-*,
- tools that facilitate building Web services – e.g. Java2WS or WSDL2-Java which helps developing Web services starting from WSDL or code development,
- full support for JAX-WS 2.0 client/server, MTOM[50] and others,
- integration with Spring for easy and intuitive creation of endpoint and client configurations.

In the WKI System, Apache CXF is used to generate WSDL files from Java classes with Maven cxf-java2ws-plugin[51]. It is also used internally by Apache Camel.

The WKI System uses version 2.1 of Apache CXF.

[49] https://jax-ws.dev.java.net/

[50] MTOM, SOAP Message Transmission Optimization Mechanism, http://www.w3.org/TR/soap12-mtom/

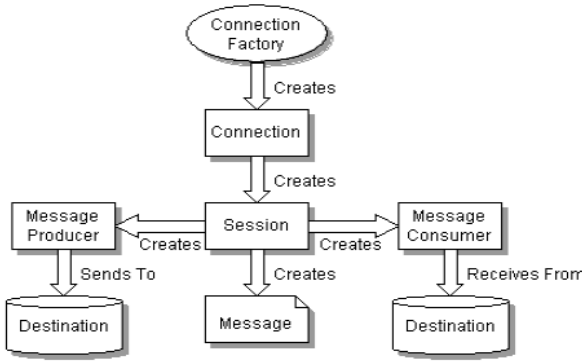
[51] http://cwiki.apache.org/CXF20DOC/maven-integration-and-plugin.html

4.14. JMS & ActiveMQ

"The Java Message Service (JMS) API is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition (J2EE) to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous." [31]

Java Message Service (JMS) provides API for sending and receiving messages in vendor-neutral manner. JMS API plays similar role in messaging as JDBC API plays in relational database – it creates an industry standard thus helping to avoid vendor lock-in.

JMS API abstracts interaction between messaging clients and message-oriented middleware.



Picture 13 JMS object relationship (from 9)

Message producer creates a message and passes it to JMS provider for delivery to destination. Message consumer receives message using JMS client.

ActiveMQ is an open source message broker and provider of Enterprise Integration Patters. It implements JMS ver. 1.1 and integrates out of the box with Apache Camel.

4.14.1. JMS in the WKI System

Integration layer of the WKI System benefits greatly from usage of JMS. JMS realizes the publisher-subscriber pattern which helps to reduce coupling between services. Publisher access the message queue while consumer uses callback for receiving messages. Data exchange is provided by JMS and both parties are completely unaware of the existence of the other side. This allows to combine services whose cooperation is completely transparent. Of course such integration might require various transformations of messages (because there is no guarantee that provider and consumer use the same format of messages) – this is handled through the use of EIP via Apache Camel.

An example of usage of JMS queue in the WKI System can be found in section 5.3.2.

4.15. Maven

Maven[52] is a project management tool based on the concept of project object model (POM). Maven provides support covering many aspects of project development such as building, reporting and documenting.

4.15.1. Use of Maven in the WKI development

In the WKI project Maven is utilized both by individual developers and by CI server (section 6.5). Integration plan for the WKI System presented in section 8 makes heavy use of Maven. Hence it is justified to clarify some of its main features, as it is done in following sections.

4.15.2. Maven and Ant comparison

In the world of Java development there exist many tools supporting building process. Among them Ant[53] and Maven are the most popular. Table 14 contains comparison of features of Ant (ver. 1.7.0) and Maven (ver. 2.0.9). Aspects that were considered the most important for WP6 are printed in **bold**. Each feature is labeled with weight from 1 (weak) to 5 (great).

	Ant	Maven
CI Server integration	**5**	**5**
Dependency management[54]	**2**	**4**
Documentation	5	5
Ease of use (first project)	4	2
Ease of use (common tasks)	3	5
Ease of use (not-so-common tasks)	4	2
Flexibility	5	4
IDE support	5	5
OS independent	**5**	**5**
OSGi integration	**2**	**5**
Robustness	**5**	**4**
Standard project layout	2	4

Table 14 Comparison of Ant and Maven

As Table 14 shows, both tools present some advantages and disadvantages. The chosen framework is Maven, because of better fulfilment of requirements specific for the WKI System. Some shortcomings of Maven (steep learning curve, troubles with specific tasks, less flexibility and robustness) may be easily addressed by WP6. In fact, these issues have already been solved by WP6:

[52] http://maven.apache.org

[53] http://ant.apache.org

[54] Dependencies management for Ant possible with Ivy (http://ant.apache.org/ivy/).

- guideline for developing services with Maven has been prepared and additional support for WPs developers will be provided if such needs arise[55],
- default POM files have been created by WPs, which rezlizes WPs from need of learning and understanding all complicated concepts of Maven.

In case of Ant, some of it's shortcomings (dependency management, lack of standard project layout, weak OSGi integration) would affect the entire project and would be significantly harder to overcome.

4.15.3. Maven features

The following sections explain some of the most important Maven features:

- Maven automatically manages dependencies,
    - very important for distributed teams,
    - significantly reduces size of created artifacts[56],
- common tasks are very easy to accomplish with Maven, and they require almost no coding at all,
- Maven offers consistent, standardized naming and project layout,
- artifacts created with Maven have unique versions,
- Maven can be extended with various plugins (installed on demand) which allow some extended features including:
    - execution of unit tests (created with various frameworks – TestNG, jUnit),
    - generation of reports (code coverage, OO metrics, dependencies),
    - execution of Ant tasks (important for non-common tasks).
- Maven supports creation of OSGi bundles,
- all CI solutions offer integration with Maven,
- structure of POM artifacts support inheritance so that one can define default building properties by the usage of "parent POM's". This feature is used in WeKnowIt project to configure some plugins which standardize the process of building services by WPs.

The WKI System uses version 2.0.9 of Maven.

[55] For information about guidelines please refer to D6.2.1.

[56] According to Maven an "artifact" is something created during the building of a project from the source files (e.g. jar file, a html documentation).

4.15.4. Dependencies management

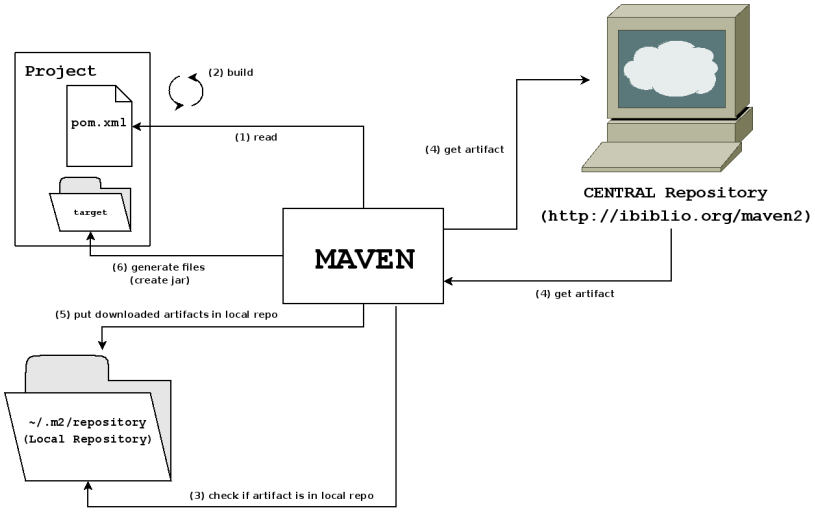
Main reason for the popularity of Maven is its ability to take care of dependencies. Every project uses some 3rd party libraries and it is crucial to ensure that all developers use exactly the same versions of libraries. In a dispersed team this might be a serious issue resulting in some errors significantly hard to find. This is the purpose of using Maven in projects such as the WKI System.

Every library used in the project is virtualized as dependency identified by three attributes: groupId, artifactId and version. For example, if project requires the latest release of Hibernate[57] library, this dependency needs to be specified in POM file:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate</artifactId>
  <version>3.3.1.GA</version>
</dependency>
```

Listing 1 Dependency in POM file

Above declaration guarantees that every developer of this project will use exactly the same release of Hibernate library. Maven automatically downloads appropriate files and additional dependencies from one of the repositories accessible in the web.



Picture 14 Maven first searches local repository for required artifact (from 11)

Picture 14 presents Maven behaviours/actions during retrieval of dependent artifacts. Each library is downloaded once and stored in local repository, which was previously created in the file system.

[57] Hibernate, a popular Object-Relational-Mapping library, http://hibernate.org

4.15.5. Maven build process

Build process is performed on the basis of configuration provided in POM file. Typical build involves following phases:

- cleaning – local environment is prepared what includes removal of all files left from previous builds,
- compilation – source code is compiled by appropriate compiler (e.g. javac, ajc),
- testing – unit tests are applied,
- packaging –appropriate artefacts are created, for example JAR archives
- reporting – many kind of meaningful reports are prepared, including:
    - general project information (e.g. description, list of dependencies, SCM information, team),
    - status of tests and code coverage,
    - documentation (javadocs, source code included in html files),
    - code analysis (static checkers).

Created artifacts might be deployed into the Maven repository. If such activity refers to new release of a library it also should involve :

- update of version in POM file,
- tagging of SVN trunk or particular branch.

In fact, the flexibility offered by Maven and CI servers ensures that all requirements addressing build process are met.

4.16. The WKI Data Storage technologies

Apart from the technologies related to overall architecture and integration layer presented in previous sections, the WKI System also uses few additional technologies dedicated for the WKI Data Storage component. Detailed description of these libraries is included in D6.3 "Design, architecture and implementation of the knowledge base". Table 15 presents basic information about each technology.

Name	Description	Version
MySQL	Popular open-source relational database management system (RDBMS). http://mysql.org	5.1
Jena	Semantic database providing persistent storage of RDF (Resource Description Framework) triples.	2.5.7

	http://jena.sourceforge.net	
Apache JackRabbit	Open-source hierarchical content storage supporting both structured and unstructured content, full text search, versioning, transactions, observation and more. Implements standard Content Repository for Java Technology API (JCR). http://jackrabbit.apache.org/	1.5.0

Table 15 Technologies used in the WKI Data Storage

4.17. Summary of selected technologies

Previous sections describe technologies used in the WKI System, where solutions chosen for all components of the WKI architecture, from data storage and integration layer to API exposure were discussed.

It is difficult to identify common features of such a heterogeneous environment, however, some conclusions can be made. Most of the presented technologies are open-source and gather wide range of user and developer communities. All of them are modern and follow the actual trends of the IT world. Moreover, all of them evolve and are under continuous development. No "legacy technologies" (that means old, unmaintained, vendor-specific or lacking community support) are going to be used. Wherever possible, approaches based on some well defined standards were chosen which makes the architectural component interchangeable. All of the selected technologies smartly integrate with one another using some extended features of Spring framework.

5. Architecture

In this section the final version of the WeKnowIt system architecture is described. First, a broader picture of layers and modules is presented. Then, the actual architecture description follows.

5.1. Layers

"In object-oriented design, a layer is a group of classes that have the same set of link-time module dependencies to other modules. In other words, a layer is a group of reusable components that are reusable in similar circumstances." [from 21]

The concept of layers is commonly adopted in software architecture. It simplifies the design by strictly limiting the possible communication in the system. Elements of every layer are allowed to exchange messages only with the nearest layers – any other communication is prohibited.

Layers also help to identify the potential problems that may occur if some changes are introduced to the API of a particular layer. Proper layering of the system is also very important for testing. It helps to examine the integration aspects of the system (see 7.1.2).

For a web application, the typical layering scheme includes:

- presentation layer,
- web controllers layer,
- services layer,
- DAO[1] layer.

5.1.1. Layers of the WKI System

Picture 15 presents not only layers of the WKI System itself, but also layers of the web applications build on top of it[2].

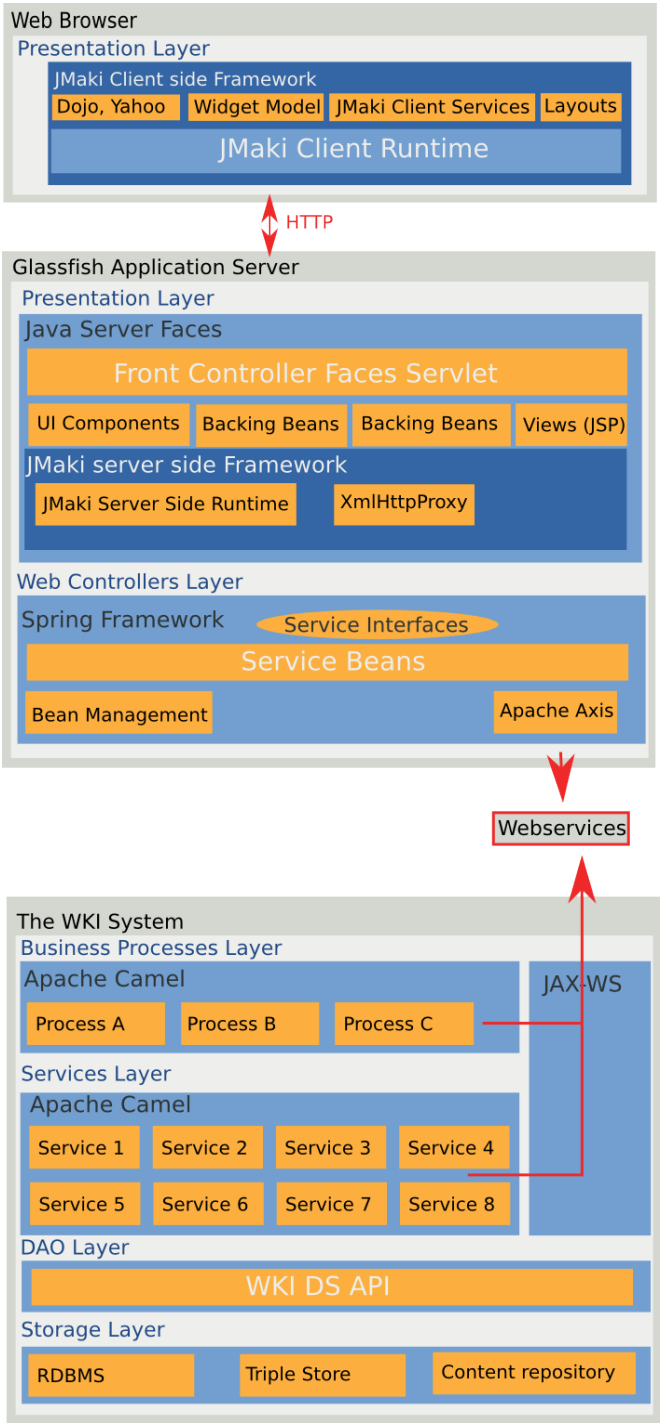
Starting from the top of picture:

- Presentation Layer (client-side) is responsible for the actual rendering of the content provided by server-side Presentation Layer.
- Presentation Layer (server-side) is responsible for preparing views for client-side Presentation Layer, based on information delivered by Web Controllers Layer.

[1] DAO, Data Access Object, an object that provides access to some persistent storage (e.g. database, content repository) hiding storage internals behind easy to use API.

[2] Names of the technologies that appear in server and browser should be considered as proposals. They were discussed in the consortium, and temporarily accepted, but no final decision regarding the stack of GUI technologies was made.

- Web Controllers Layer contains thin, application specific business logic related mainly to the presentation purposes. It uses functionality exposed as Web services by lower layers.
- Business Process Layer, groups services from Services Layer into processes and exposes some of them via Web service endpoints.
- Services Layer offers functionalities used by higher layer. It uses DAO layer to access WKI Data Storage. Some services are exposed via Web service endpoints.
- DAO Layer provides access to the storages hiding the complexity and multitude of used storages behind an API.
- Storage Layer is responsible for storing resources (files, semantic and relational data).



Picture 15 Layers of the WKI System (with applications)

5.2. Modules of the WKI System

The WKI System will be comprised of many services providing very different functionalities. To make the whole system easier to understand, it is convenient to group these services into larger structures based on their functionalities.

In case of the WKI System, modules are only a conceptual aid for developers. They are not meant to provide API - this is done on services level, as described in section 5.3.1.

5.2.1. Identified modules

Based on the Definition of Work document, and on the list of services declared by WPs[3] few main modules of the WKI System can be identified. This decomposition might change during the further development of the WKI project as new services will emerge.

User and group management module

This module constitutes of modules created by WP1, WP4 and WP5. Their task is to provide a comprehensive set of functionalities that allow to manage user accounts and communities.

Data storage module

This component is provided by WP6. The WeKnowIt Data Storage[4] (WKI DS) is responsible for storing all of the data in the WeKnowIt System: both files (e.g. multimedia files) and semantic data.

More detailed information about the WKI DS is presented in D6.3 deliverable ("Design, architecture and implementation of the knowledge base").

Search module

Services from this module provide search functionality. They cooperate very closely with services grouped in "data storage" module (section 5.2).

Data processing module

This module includes work of WP2. It contains services dedicated to e.g.: speech, visual and text analysis. Depending on the application needs, services included in this module will extract various metadata from the resources stored in the data storage.

Common services module

"Common services" module includes utility services, that will be used frequently by other services, or by external applications. This includes e.g.:

- WP6 services that provide common API for file upload, logging and performance analysis
- WP5 "task management" service
- WP1 add tags service

[3] For the full list of WKI services please refer to D6.2.1 – in this section only selected services are mentioned.

[4] In the Definition of Work document this component is named "Knowledge Base" but after some discussion in the WeKnowIt consortium it was renamed to the "WeKnowIt Data Storage'. It was argued that the former name does not reflect the true purpose of this component, which stores both files and semantic data.

WKI Web service API module

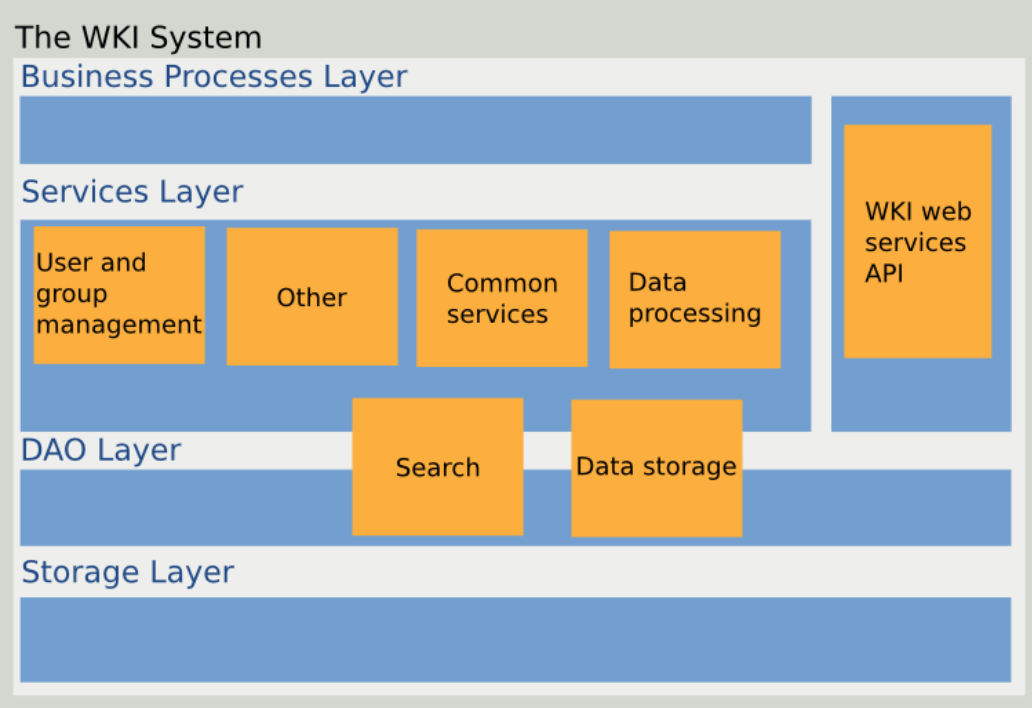
This module provides Web services API that is used by external applications (see section 5.4.1) which are not deployed in the same OSGI environment.

Other module

That module contains all services which provide "low level" functionalities used by other modules, or provide very specific functionalities which makes them hard to include into any other previously described module.

5.2.2. Modules and layers of the WKI System

Picture 16 presents how modules and layers fit together. Due to the fact that modules group services with common functionalities, they mostly fit in one layer each.



Picture 16 Layers and modules of the WKI System

5.3. The WeKnowIt system architecture

This section presents the actual architecture of the WeKnowIt system.

As far as the WeKnowIt system is concerned, SOA and ESB approach is used. Fuse ESB is used as an ESB implementation. It is worth noticing that SOA in WeKnowIt is implemented with two protocols. One is a standard Web services approach. The other one is based on OSGi bundles.

The WKI System consists of integrated WKI services. Single WKI service is the smallest element of the WKI System architecture.

5.3.1. WKI service

Conceptually, every WKI service is a building block that provides well-defined functionality and can be used as element of larger structures. Each WKI service is accessible via its API, which specifies available methods and defines input and output structures. API provided by each service is defined by service developer. The WKI System strictly follows SOA approach, which is based on the notion of well-defined, loosely-coupled services.

Technically, a WKI service is a OSGi bundle. This implies, that every WKI service is manageable via OSGi framework and can benefit from features described in section 4.6. OSGi framework guarantees that only methods from certain classes (enumerated in descriptor file) are accessible.

Example of the WKI service: WP1_Tag

An example of WKI service is a WP1_Tag service. This service has well-defined functionality - its purpose is to add metadata (simple text tags) to resources. It can be used by other services – e.g. one of the beneficiaries of WP1_Tag's functionality is WP2_Text_Annotation service. WP1_Tag service will be distributed in form of OSGi bundle, which makes it compatible with the WKI System architecture.

D6.2.1 contains a list of services declared by WPs.

Registration of service in OSGi registry

All the services of the WKI System are registered in OSGi registry via Spring DM. This approach does not involve any programmatic configuration – all information required by OSGi framework is stored in MANIFEST.MF[5] file and Spring DM XML configuration files[6].

```
Import-Package: pl.swmind.koda.*, org.apache.servicemix.common,
[...], !*

Export-Package: eu.weknowit.uploader,
eu.weknowit.uploader.wrappers,
eu.weknowit.uploader.wrappers.types, eu.weknowit.uploader.jaxws

Private-Package: eu.weknowit.uploader.*

Bundle-Description: WKI Uploader Service
```

Listing 2 Fragment of MANIFEST.MF configuration file

Listing 2 presents a fragment of OSGi bundle configuration. Only classes from packages named after "Export-package" will constitute API of this bundle. The rest of classes will not be accessible.

[5] MANIFEST.MF file is used in JAR files to provide metadata about the content of the file.

[6] The process of preparation of a OSGI bundle that is compatible with the WKI architecture is described in "How to create OSGi service with Spring Dynamic Modules" guideline. Please refer to D6.2.1 for more information.

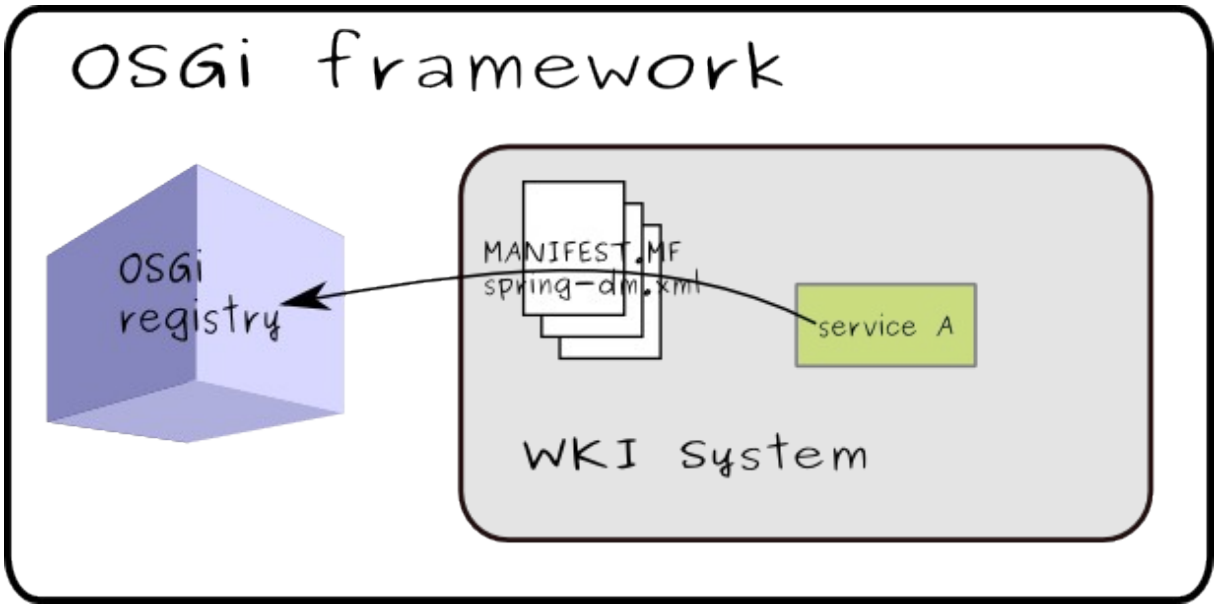
```
<bean id="wkiDataStorageService"
    class="eu.weknowit.datastorage.impl.WkiDataStorage">
    ....
</bean>

<osgi:service id="dataStorageOsgiService"
    interface="eu.weknowit.datastorage.IWkiDataStorage"
    ref="wkiDataStorageService" ranking="0">
</osgi:service>
```

Listing 3 Fragment of Spring DM configuration file

Listing 3 presents a fragment of Spring DM configuration file. First, it defines Spring bean with ID "wkiDataStorageService", then it registers this bean in OSGi registry with ID "dataStorageOsgiService".

Picture 17 shows that information included in both configuration files is used to register a WKI service into OSGi registry.



Picture 17 MANIFEST.MF and Spring DM configuration files used to register service in OSGi registry

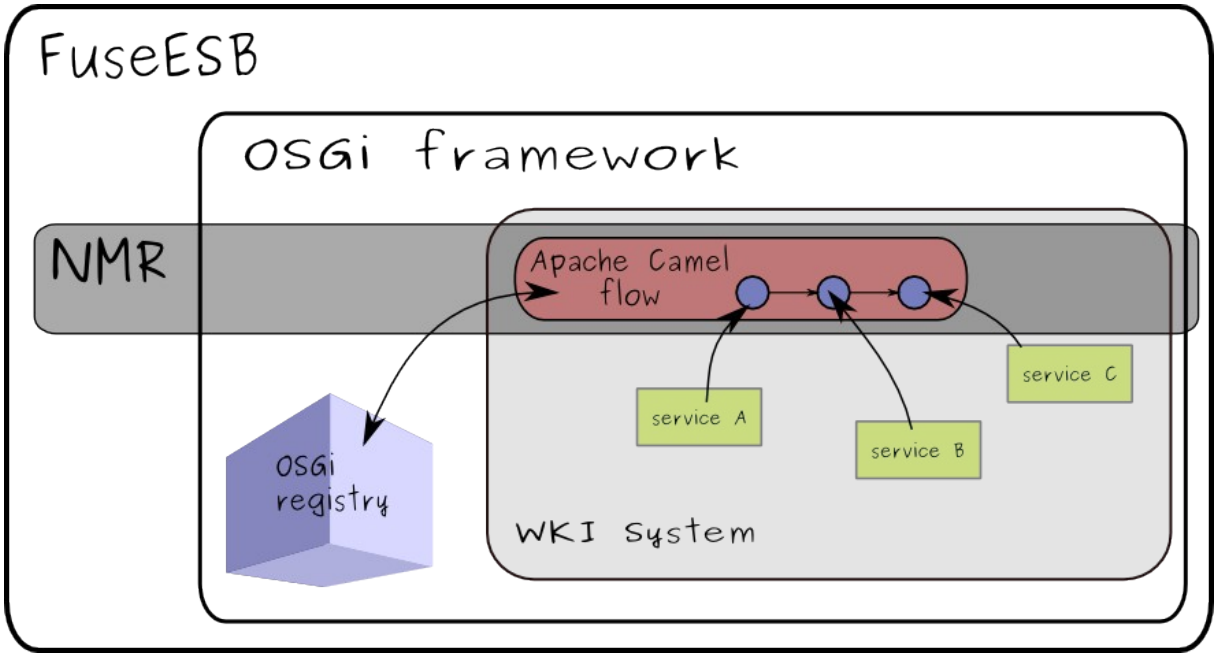
Every WKI service, apart from being accessible via OSGi registry, can also be exposed via Web services. This option is described in section 5.3.4.

5.3.2. Combination of services

The WKI services in order to provide real "business" value have to cooperate. This cooperation is possible via OSGi registry where all the services are registered and can be looked up.

Every service can search and acquire references to needed services by looking up the OSGi registry by itself. Unfortunately, such approach would endanger the flexibility of application by introducing strong bonds between services. Because of this, an additional layer is used for connecting services which helps to achieve loose- coupling.

In order to create a message flow between services additional frameworks – Apache Camel – is used. Apache Camel allows to avoid programmatic configuration[7] thus providing solution that is more flexible and easier to change.



Picture 18 Apache Camel allows to express exchange of messages between services, and takes care of communication with OSGi registry

The existence of ESB remains hidden from the point of view of service developers, but in fact all the messages between services are exchanged via Normalized Message Router[8] (NMR).

On the Picture 4 some orchestrated business processes are shown. They are constructed from services connected and cooperating? with one another. In SOA approach this is exactly the point in which loosely coupled ser-

[7] It is still possible to declare flows programmatically. This option is used in case of complicated flows - xml configuration might not be sufficient to express them.

[8] "The Normalized Message Router is the component which routes normalized messages from a source component to its eventual destination using some kind of routing policy to decide which endpoint to use" (from 22).

vices start to provide some business values. In case of the WKI System the situation is similar. Independent services created by WPs in order to provide a business value (to provide a functionality that is really valuable to external clients) have to be connected and organised in some way.

In the WKI System this effect is achieved by combining services into a Camel "flow". For example, if user of ER application uploads a file, this file, before being stored in the WKI Data Storage, will be examined by many services. Which services will receive this file[9], in what order, and under what conditions – this is specified by a particular flow. For example, if a picture is uploaded, it undergoes some visual analysis, and then, in case it is of TIFF[10] mime-type, it will be converted to JPEG[11] format, before being written to the WKI Data Storage. These four steps (visual analysis, routing depending on mime-type, conversion to JPEG, saving to the WKI Data Storage) are all elements of one flow.

Example of Camel flow

```
<bean name="httpProc" class="eu.weknowit.HttpProcess" />
<route>
     <from uri="jetty:http://0.0.0.0:8080/myapp/myservice "/>
     <to uri="bean:httpProc?methodName=processHttpReq"/>
</route>
```

Listing 4 Example of Camel flow

Listing 4 presents a minimal flow, which starts with a HttpRequest sent to given URL, and this message is passed by jetty component to the "processHttpReq" method of bean "httpProc".

Each flow has input (in Listing 4 represented by "from" tag) and could have many outputs or middle states (Listing 4: "to" tag). Input flow component could be of various type[12], for example it could be:

- a directory (e.g. any change inside directory starts this flow),
- ActiveMQ (flow is run when new message is received/flow is triggered when a new message is received.),
- jetty (when new request is received then flow will be run/flow is triggered when a new request is received).

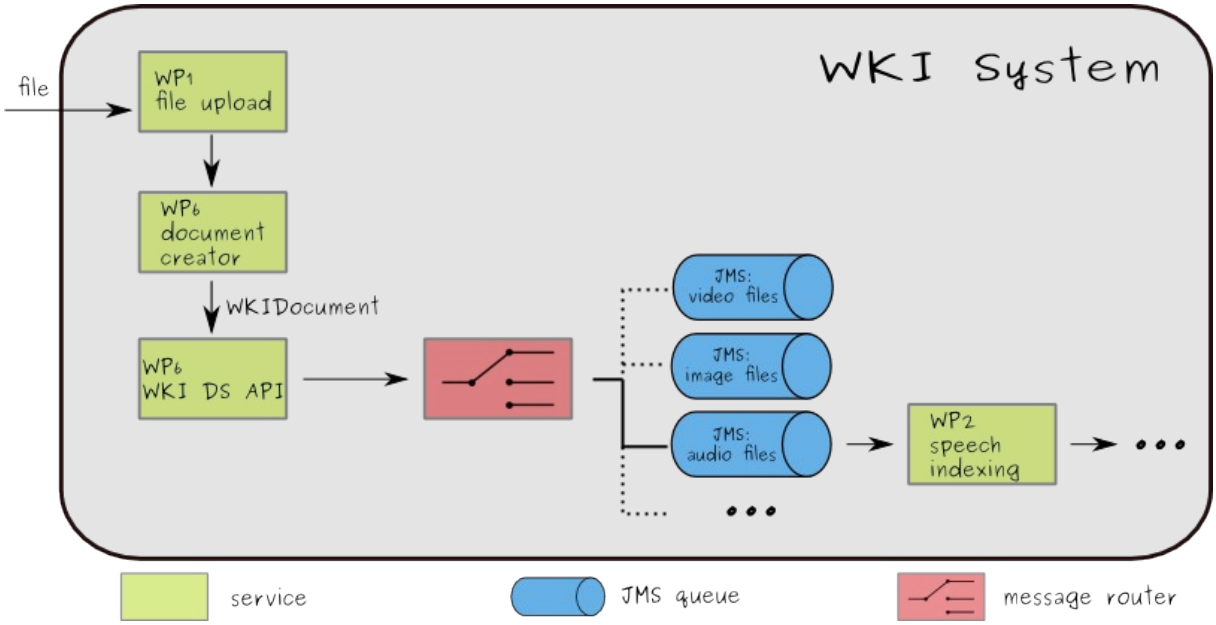
The transitions between components are realized with EIPs, as described in section 4.11.

[9] In fact not a bare file will be transferred, but rather it will be wrapped in some message.
[10] TIFF, Tagged Image File Format, a file format for storing images.
[11] JPEG, compression format, used in various file formats for storing images.
[12] Full list of components offered by Apache Camel is available at http://camel.apache.org/component.html

Usage of JMS queues

To avoid direct connections between services and achieve loose-coupling, JMS queues are used (see section 4.14 for more information). Picture 19 presents a simplified version of real life scenario of file being uploaded to the WKI System

First a file is uploaded. It is converted to a common WKI data format (of type WKIDocument) by "WP6 document creator service", and saved to the WKI DS by "WP6 WKI DS API" service. Then, depending on the mime type of the file, message is routed to one of many JMS queues. The routing is performed by one of the EIPs[13] provided by Apache Camel. Assuming that this file is an audio file, it'll be passed into "JMS audio files" queue. At this point, "WP2 speech indexing" service, which is registered as a listener to the "JMS audio files" queue is notified about the new file that appeared, it receives this file, and performs the indexing. Further processing of the file is not shown on the picture.



Picture 19 JMS queue example

This scenario shows how use of JMS queue increases flexibility of the system. Picture 19 shows that "WP2 speech indexing" service is not aware of any of the services from the left side of the picture. There is no coupling between them, which means that these parts of the system can be modified independently. The only common element chaining both sides together is a JMS queue.

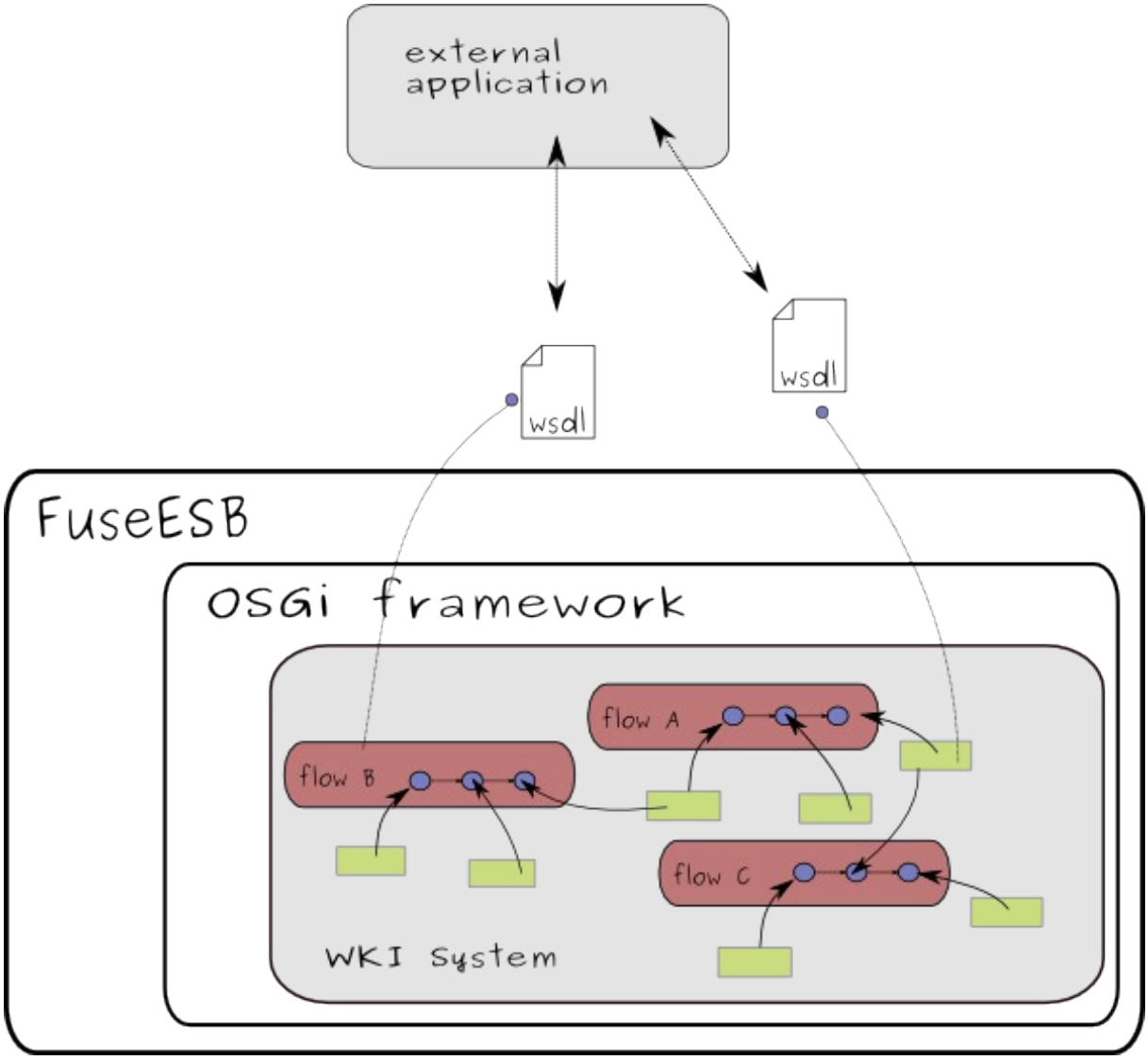
Such flows as presented on the Picture 19 can be configured declaratively (in XMl files) which also results in better flexibility of the system.

[13] Message router - http://camel.apache.org/message-router.html

5.3.3. Exposing functionalities as Web services

A single service can be exposed as Web service. The same is possible for any composition of services (grouped in flow by Apache Camel) that can be configured with use of Spring DM configuration files as being accessible via Web services by external applications.

This way, an external API of the WKI System (API, as perceived by external applications) is not limited to methods provided by single services, but also to more coarse-grained functionalities expressed by Camel flows in the WKI System. This can make such API easier to use, because the exposed functionalities of the WKI System can shield end users from the internal complexity of the system.



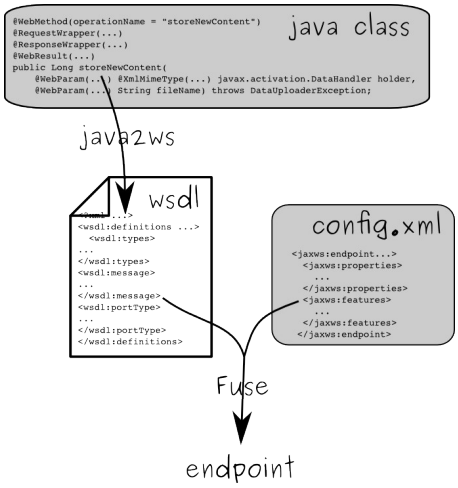
Picture 20 Both services and service composites can be exposed via Web services

5.3.4. Exposing single service as Web service

Apache CXF is used internally by Fuse ESB to expose single service as Web service.

Picture 21 presents a process of exposing a service as a Web service. The following steps are required[14]:

1. service source code must be annotated with JAX-WS annotations[15],
2. WSDL file is generated with "Java to WSDL" Maven tool,
3. service endpoint/client is configured using Spring configuration files,
4. Fuse ESB takes care of the actual exposition of configured endpoint (using other technologies underneath).



Picture 21 Endpoint creation

5.3.5. Exposing Camel flow as Web service

Section 5.3.2 described how services are integrated into processes. It is a common thing, that an external client is more interested in such processes (which reflects business actions) than in separate services (which provide some atomic behaviour). Because of this, an entry point of process should be accessible to external clients.

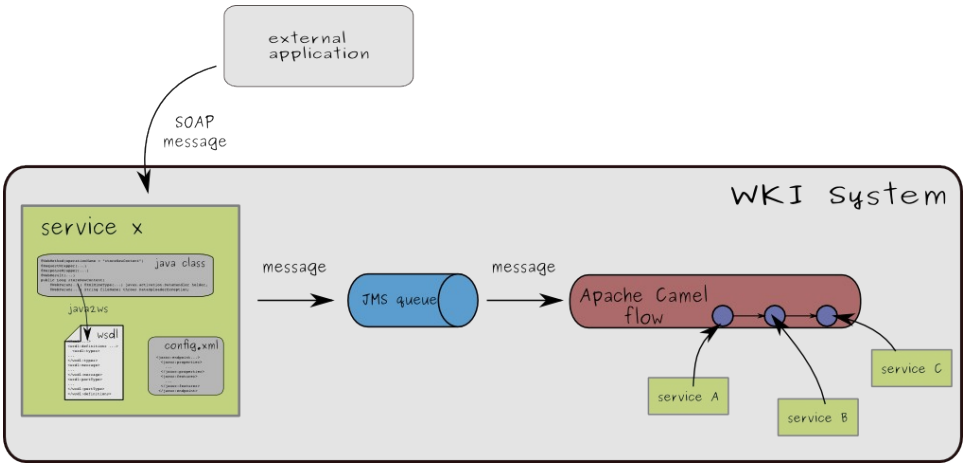
For example, in case of the WKI System ER use case application is very interested into printing to users list of 10 latest information on some topic. Let us assume that to get this information three different services must be used. It would be cumbersome for client (ER application) to talk with each of these services separately – that would require client to know and understand API of all three services. Because of this, the WKI System API provides a Web service, which allows to access a process that provides a

[14] This process will be described with details in a separate guideline (please refer to D6.2.1).

[15] Is possible to create web service without adding any annotations by use CXF "simple frontend" which builds services using Java reflection mechanism.

functionality required by client in one step. This makes the WKI System much more convenient to use.

Not every component type can be used as input. ActiveMQ is used to expose input of flow as Web service. In this way, external application can send SOAP messages to invoke a particular functionality of the WKI System.



Picture 22 Camel flow as Web service

On the left side of Picture 22 Web service component (Service X) puts message to JMS queue in order to execute one of Web service methods. Flow has ActiveMQ component as input and when new message is received then flow is run. At the end of the flow, http Camel component can be used to send response message to client web application.

5.4. Applications of the WeKnowIt System

The WKI system offers various functionalities in form of single services or composites of services that can be accessed via the system API. An application build on top of the WeKnowIt system can use these services in order to provide valuable services to the end users.

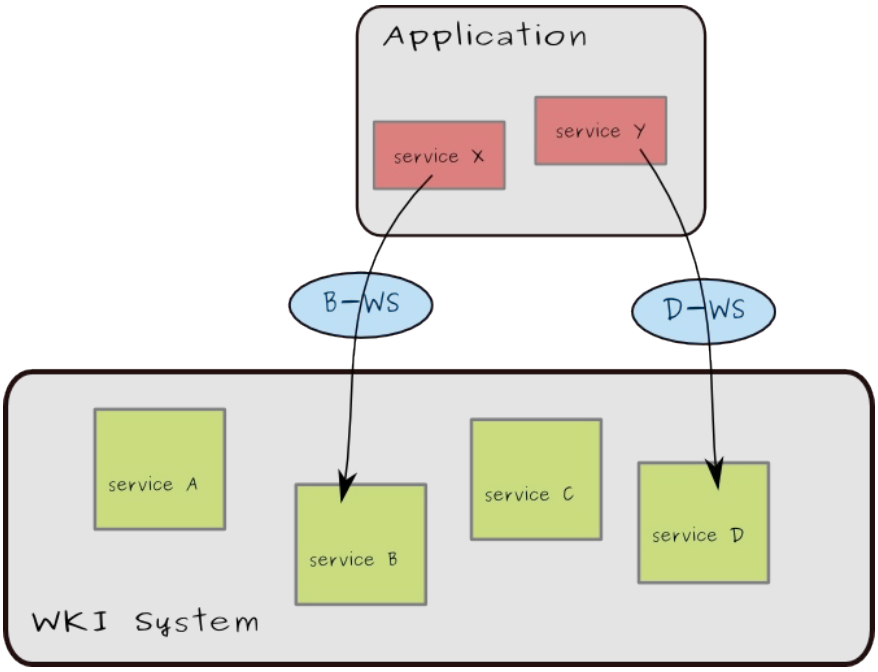
To make the picture of the WeKnowIt architecture complete, it has to be mentioned, that the communication between an external application and the WeKnowIt system might be realized using two different approaches:

- access via Web services,
- access via OSGi registry.

This section describes technical aspects of using both solutions.

5.4.1. External - communicate via web-services

An application can communicate with WeKnowIt services via Web services layer. This is the most common, and recommended solution.



Picture 23 An application accessing WKI services via Web services

Client application in order to use WKI System functionality exposed via Web service must do the following:

- acquire WSDL file which describes exposed methods and input/output types,
- create Java classes corresponding to the information in the WSDL file,
- call methods of the generated classes to invoke WKI functionality (transfer of data from the WKI System to the client application will be handled internally by generated classes)

Advantages of web-service solution

- The application and the WeKnowIt system can run on different machines (different JVMs).

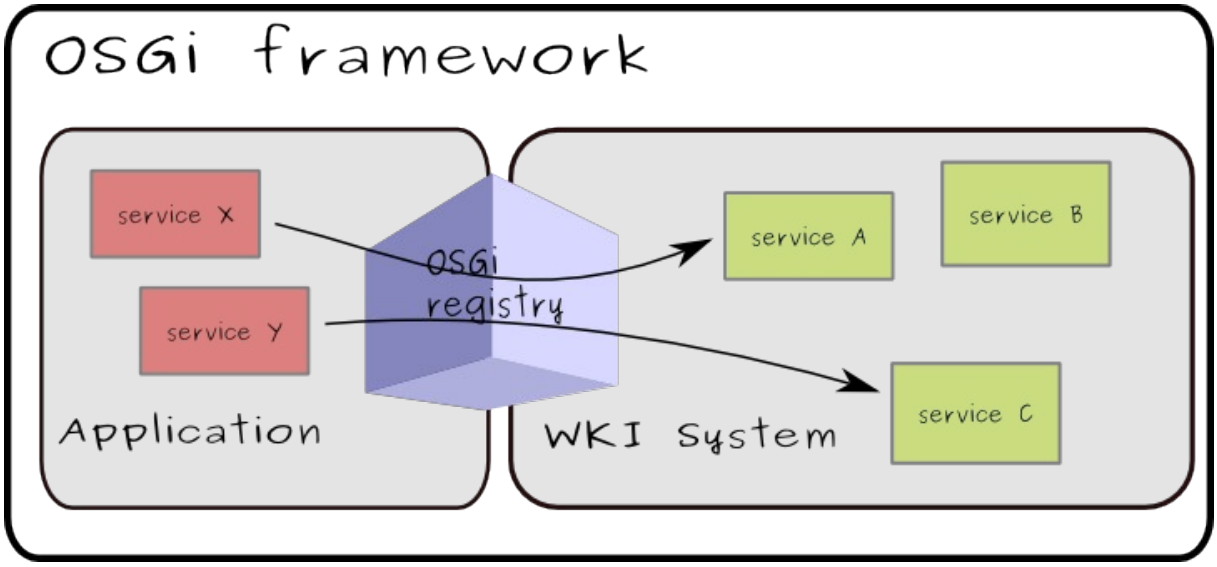
Disadvantages of web-service solution

- It requires particular service first to expose its functionality (all of it or just a subset) as a web-service.
- Communication via web-services involves use of HTTP protocol, which adds some overhead (depending on net connection quality) to every method call. Also passing/receiving large amount of data is strictly related to the bandwidth available.

5.4.2. Internal – communicate via OSGi registry

The second option is to deploy an application in the same runtime environment (i.e. Fuse ESB), and access WeKnowIt services via OSGi registry.

This option might be interesting for applications with very high performance requirements. Communication via web-services (with use of HTTP & XML) adds significant overhead which is avoided in this solution.



Picture 24 An application deployed to the same JVM accessing WKI services via OSGi registry

Such "internal" application can communicate with WKI services via OGSi registry. This process is described in two guidelines: "How to create OSGI services using Spring Dynamic Modules", and "How to deploy OSGi bundles to the WKI System".

Advantages of OSGi solution

- Services discovery, lookup and communication via OSGi registry, is much faster than in case of Web services.

Disadvantages of OSGi solution

- Both the WKI System and the application must reside on the same machine (in same JVM).[16]

This option is only possible if full access to the runtime environment is available.

5.5. Hardware architecture

Preparing hardware configuration is usually difficult task. In all cases prediction of number of users and operations which will be performed by them in the future is a challenge. In this case, we have additional difficulties in predicting how much time some operation will take because some components of the system are in early stage of development. In

[16] This shortage might be overcome in the near future, if distributed OSGi project (R-OSGi) will fulfill its promises.

most cases WPs deliver only very rough estimation (see hardware requirements in section 3).

Hardware configuration is required at this stage, so some assumptions have to be made. This section describe proposed hardware architecture of the system with description on how figures were calculated. Each WPs team should read this section, and try to figure out if estimated hardware is enough for their purpose and if proposed hardware is enough to process assumed in this section operations/number of users.

This section describes the recommended architecture, that results from estimations of requirements of the WKI System.

5.5.1. Number of events (uploaded files) and users

Emergency Response (ER)

In case of Emergency Response scenario, the average number of events is expected to be very low, but it is during emergency when the activity of users might be extremely high. System should take into account worst case scenario, and should be ready to handle all requests during peak caused by some emergency situations.

It is assumed that number of events during peak could be about 2000 events per hour. System load could stay high for several hours, probably less than a day. Most of registered events will be phone calls, let's say 80-90%. In our opinion number of pictures (photos) delivered to system could be about 10%. Assumed number of video streams is 1%. It means that it will be only a few video streams in the same time. Number of text messages (SMS, emails) will be probably relatively low, besides processing of text documents is relatively simple in comparison to images, video and audio streams. For handling ER users will feed data anonymously, there will be no time to register users. Users registered in the system will be staff like FD, Police and other similar organisations, number of concurrently working users could be about 200. Number of serious emergency event per year is relatively low. The assumption is that it will be about 10 days per year. Those assumptions lead to annual resource requirements:

- about 5000 uploaded video streams
- about 430000 uploaded audio stream (phone calls)
- about 50000 pictures (photos)
- total 485000 uploads

Consumer Group Study (CGS)

In this use case number of read operation performed every day is much higher than in case of ER. Write (upload) operation will be similar to peek load generated by ER application. In this case peaks are not expected. Of course, some fluctuation of usage depending on time of the day, day of the week or part of a year are possible. Number of users registered in the system will be, for sure, much higher if system will be open for public.

Number of users working in the same time will be only a fraction of registered users current assumption is about 100 in the same time.

- about 40000 uploaded video streams
- about 20000 uploaded audio stream
- about 300000 pictures (photos)
- total 360000 upload

5.5.2. Calculation of Required Resources

Knowledge Base

For each document stored in the system additional data related to this document will be added. It could be some keyword, selected by user or automatically by the system. In some cases of adding some keyword it will be required to add more than one triple to select proper ontology for entry. Current assumption is that average number of triples for each document will be about 1000 triples.

Total number of files stored per year is about 1 000 000 (for both ER and community portal). It means that total number of triples stored every year is about 1 000 000 000 ($1*10^9$). Based on our previous experience and other publications [30] 350 bytes of disk space for each stored triple is required. It means that, 1.6 TB of disk space per year will be required. This is only storage space for knowledge base, not for files body. To keep database efficiency some of data have to be loaded into memory as indexes for faster searching. Our assumption is that it will be required to store about 1% of data in RAM. It means that for each year of system work, 16GB of memory will be required. For fist years of system usage vertical scalability of system will be enough, later some cluster will be required.

Proposed server for database server is:

- CPU: 64bit Intel Xeon architecture, 8-Cores (two Quad-Core processors)
- RAM: 48 GB of memory
- Storage: RAID Disk Array, 5 disks 750GB each

Content repository

Storing multimedia content is a challenge for many years and it is still difficult due to increasing size and quality of stored media. According to our consideration, the above mentioned system should be able to (to do what?) annually of: ?????

- 45000 video streams 200MB each
- 450000 audio streams 5MB each
- 350000 pictures 3MB each

It means that there are 12TB of disk space required every year.

Suggested server for content repository:

- CPU: 64bit Intel Xeon architecture, 8-Cores (two Quad-Core processors)

- RAM: 32 GB of memory
- Storage: external NAS 200TB

It is possible to replace expensive NAS server to cluster or PCs. In case of this amount of data and required replication level costs of such cluster will be also high. Additionally, effort required to configure and maintain is high.

5.5.3. Processing resources

Previous subsections specify only storage components for content and knowledge. WKI System requires processing power to handle user request and sustain communication between components. For components integration ESB will be used. The easiest way of adding components is keeping all of them as threads in the same Java Virtual Machine as long as it is possible. Limitation to single JVM is too strict for this project, it is planed that in case of some components which require heavy processing, two stage integration will be required. The first part of component will be connected directly to the integration bus (ESB), while the second one executed in separate JVM and on separate server will be connected to the first part of component. Splitting component into parts will be required only for CPU and/or memory intensive components. Currently only WP2 visual and audio (speech) processing have to be executed on separate machine(s)

Resource requirement based on data delivered by WPs are:

- WP1
  - RAM – 2GB
  - CPU – 1 core
- WP2
  - RAM – 2 GB
  - CPU – 6 cores
- WP3
  - RAM – 7.5 GB
  - CPU – 3 cores
- WP4
  - RAM – 7.5 GB
  - CPU – 2 cores
- WP5
  - RAM – 0.5 GB
  - CPU – 1 core
- Integration
  - RAM – 10 GB
  - CPU – 1 core

Main processing and integration server:

- CPU: 64bit Intel Xeon architecture, 8-Cores (two Quad-Core processors)
- RAM: 64 GB of memory
- Storage: internal disks for system and temporary storage 2*0.5TB

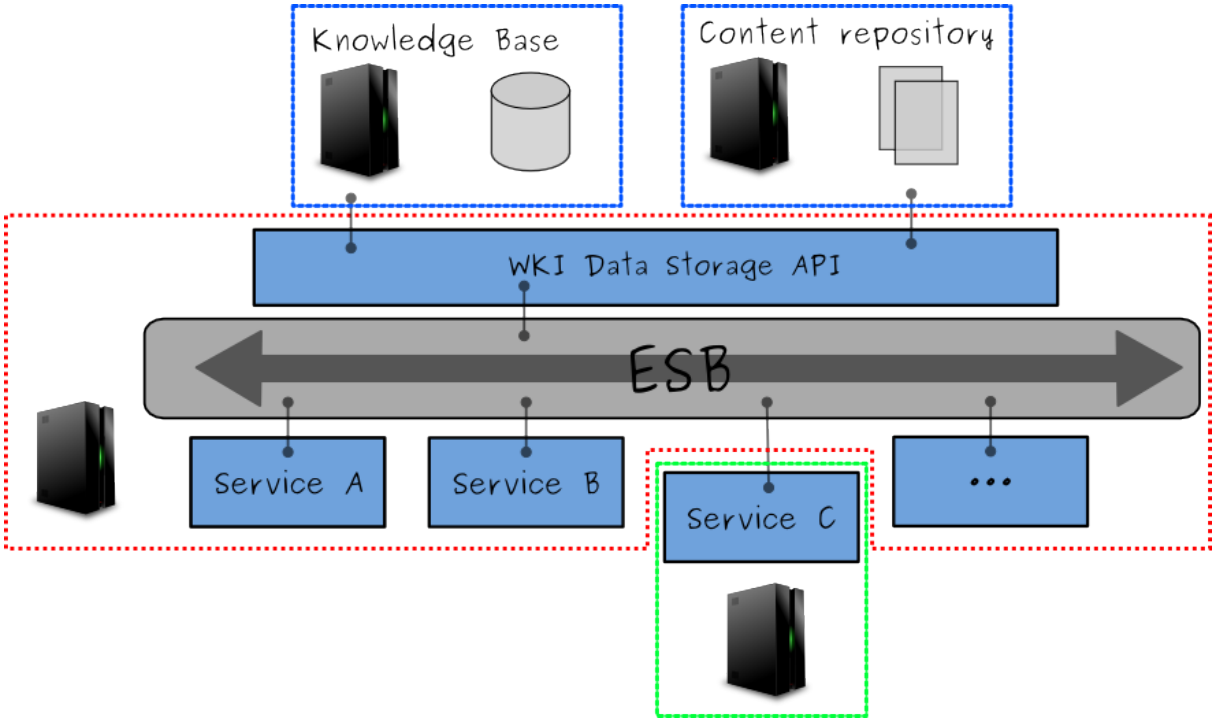
Auxiliary processing server:

- CPU: 64bit Intel Xeon architecture, 8-Cores (two Quad-Core processors)
- RAM: 64 GB of memory
- Storage: internal disks for system and temporary storage 2*0.5TB

5.5.4. Hardware configuration

Based on the system requirements which are described in previous sections, system will be de composited minimum to four separate machines as presented on Picture 25. Two machines will be dedicated to the WKI Data Storage - first to the Knowledge Base, second to the Object Storage (content repository). On the third machine integration platform (Fuse ESB) will run. Fourth server is dedicated to hosting of services that are expected to make a heavy use of the CPU. It is expected that such situation might take place with visual analysis services.

Such configuration should ensure that no single activity performed by one component will block the whole system. This architecture is still open for further enhancements – it is possible to move more services to separate machines, if such need arises.



Picture 25 Hardware architecture

5.5.5. Development hardware architecture

Architecture used during test and integration activities should be similar to the one described in the previous sections, but it does not need to be identical. For example the performance characteristics of machines might be different (i.e. less powerful).

Staging environment, described in section 6.3.2, will be run on a single machine, which is enough for the purposes of this environment. The production environment (section 6.3.3), which is also used during development, should resemble more closely the recommended hardware architecture – i.e. it should be set on more than one machine in order to test how components deployed on separate machines communicate with each other.

6. Common development infrastructure

This sections describes infrastructure used during WKI System development. This infrastructure provides different services to the WKI developers and integrators. The overall goal of the development infrastructure is to standardize and unify the development process.

6.1. Source code management

The code of the WKI System and services developed by WPs will be stored in a Subversion[1] code repository (SVN). This repository will be maintained by SMIND and protected against unauthorized access. Every WKI services developer will get a login/password that will allow to commit code to repository.

Some purposes of having a common code repository are as follows:

- it is industry standard – code repository is recognized as one substantial tools for development of every project,
- code in the repository is protected against accidental loss (backups)
- it is possible to go back to the previous versions of file (and compare versions with diff)
- it is possible to maintain various versions of file (via branching and tags mechanisms)
- code is easily accessible by the whole development team (which is very important for teams which are so geographically dispersed as WeKnowIt)
- it fits well into general software development process (good integration with CI servers),
- having all code in one place helps to control quality.

Among various possible solutions, Subversion was chosen because:

- it is well-known, robust and popular - good client tools exist for various operating systems and for all popular IDEs (e.g. Subclipse[2] plugin for Eclipse IDE)
- it has some advantages over popular CVS[3] (e.g. "all or nothing" operations, cheap branching and tagging, easy file moving/renaming which retains file history, etc.) [20] [21]

[1] http://subversion.tigris.org/

[2] http://subclipse.tigris.org/

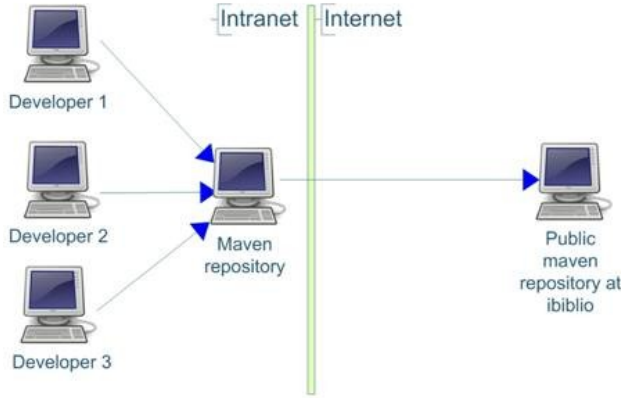
[3] http://cvs.nongnu.org/

6.2. WKI Maven repository

Maven build tool uses repositories (see section 4.15.4) from which it downloads artifacts required to build a particular project (i.e. JAR files). There are a lot of public repositories existing, where publicly available artifacts can be found, but for a WKI project, there is a need of own repository, where WKI-specific artifacts can be stored.

SMIND hosts a Maven repository for the WKI project. In this repository all the WKI artifacts are stored. It is logically divided into two parts:

- WKI artifacts (built by CI server from source files provided by WPs and stored in SVN code repository – see sections 6.1 and 6.5)
- 3rd party libraries necessary to build WKI artifacts



Picture 26 Local and public Maven repository (from 14)

Picture 26 present the idea of having own repository. The WKI Maven repository will serve following purposes:

- it allows to share the artifacts created by WPs,
- it helps to assure that all the WPs use same versions of 3rd libraries,
- it acts as a cache of public repository which can significantly shorten the download time of public artifacts.

As far as the WKI System is concerned, the cache functionality will be used only by SMIND developers, because the WKI Maven repository is hosted on SMIND infrastructure.

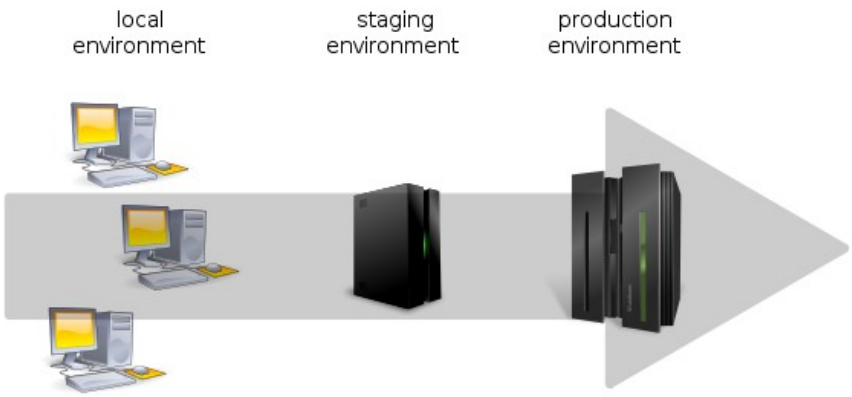
WKI Maven repository uses Nexus[4] Maven Repository Manager which provides a GUI (in form of a web application) for managing the repository and its content.

6.3. Development environments

In order to help with development and assure the high quality of the final results, three different environments for deploying and testing of the WKI

[4] http://nexus.sonatype.org/

System and services are planned: local, staging and production. The actual role of these environments are described in this section.



Picture 27 Development environments

6.3.1. Local environment

The role of local environment is to let developers quickly test developed services. Every developer is able to run the WKI System on his/her local machine and deploy created services to the local WKI System.In this way one is able to check if the developed service integrates with the architecture of the system and if it interacts correctly with other services.

Due to the fact that the hardware and software environment on local machine is not similar to the production one (possible different operating system, processor, RAM available, etc.), there is a need for other environment that mirrors hardware and software used in production environment.

Local environment is already available since January 2009. The WKI System can be downloaded from the WeKnowIt wiki. A comprehensive set of guidelines (describing installation of the WKI System and development of services) is available since February 2009[5].

6.3.2. Staging environment

The staging environment (also known as pre-production environment) is used to assemble, test and review new versions of the WKI System and services developed by WPs before it goes into production. The hardware and software of staging environment mirrors the one used in the production environment.

This environment will be managed by SMIND. It will be provided soon after first services are developed by WPs.

The staging environment will be accessible by SMIND and selected partners if it is necessary for their work.

[5] For information about guidelines please refer to D6.2.1.

6.3.3. Production environment

The production environment is a final place for deploying of the WKI System and its services. It plays two roles. In the first phase of the WKI System development production environment will serve as an ultimate testing environment. When the project is mature enough to be evaluated this environment will host running WKI System.

In both cases, before deploying an updated version of the WKI System or new versions of services to the production environment, the updates must pass all tests on the staging environment.

This environment will be managed by SMIND. It will be provided soon after the staging environment is set up.

6.4. Bug tracking system

A bug tracking system (or "bug tracker") is: "a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work. It may be regarded as a sort of issue tracking system." [27]

Usually, such a system supports the concept of the life cycle for a bug, which means, that a status can be assigned to every bug, describing its phase of life. Strict rules (which can be configured) govern the possible transitions between statuses, and system administrators can configure permissions based on bug status.

Bug tracker interface allows client and administrators to browse, search and manage (comment, change status, assign to developer) bugs. Bug trackers also provide reporting and can be integrated with various other development tools (e.g. wikis, source code repositories).

SMIND provides a Mantis Bug Tracker[6] (MantisBT) for the WKI project. It is a popular web-based bugtracking system written in the PHP scripting language. ManitsBT is platform independent, works with a webserver and can use MySQL, PostgreSQL or MS SQL databases.

Bugs can be reported to MantisBT via web interface or via email. Information about bug processing can be pushed to clients with use of mail notifications or RSS feeds.

MantisTB is released under the terms of the GNU General Public License (GPL)[7].

6.5. Continuous Integration server

In order to avoid problems with different versions of libraries created on developers machines, SMIND will provide Continuous Integration server.

[6] http://www.mantisbt.org/

[7] http://www.gnu.org/licenses/

The goal of CI server is to provide common environment for artifacts creation. It is important, that all builds(?) of in project are performed with the same environment setting (e.g. Java version) and with same libraries (retrieved from the same Maven repository). This is crucial for projects like WeKnowIt, where teams are geographically dispersed, and local development environments differ substantially.

In fact, CI server offers much more. It's secondary goal is to provide information about the projects in form of documentation (project information, build logs, various reports). Generated reports are of great importance – they show actual and potential problems with tests or bad/wrong? code.

All documents generated during builds are accessible via web interface by all developers. In this way all partners are well informed about the progress of work on every service.

Only artifacts created on this server will be used on staging and production environments.

CI server is also a part of the WKI Quality Assurance plan (section 7).

Team City 4.0[8] implementation of CI server has been chosen for the WKI project.

6.5.1. Builds

Builds are performed on the source code retrieved from the SVN repository (section 6.1) and with use of 3rd party libraries retrieved from the Maven repository (section 6.2). Created artifacts (JAR files) are deployed to the Maven repository.

The builds might be characterized by their type and frequency. In the following sections popular types of builds are presented. Due to configuration possibilities offered by Maven and CI server, types of build may be tailored to users needs.

Build types

Some build phases (see section 4.15.5) require a lot of CPU/RAM (e.g. compilation, creation of reports for static code checkers) which means, they should not be executed too often. Moreover, it is not required that a full build is performed every time. Thanks to this, many types of builds are available. Below, three widely used types are described.

- Quick check – The simplest build. It's role is to make sure that the whole project compiles and unit tests pass not only on developer's machine but also on CI server.
- Snapshot – Allows to build artifacts (in SNAPSHOT version) on the CI server. Also generates all reports. This build is used before releasing the new version of artifact.

[8] http://www.jetbrains.com/teamcity/

- Release – Full build. Executed rarely, only when new version of software should be released. It is usually preceded by snapshot build. During release build new version of artifact is installed in Maven repository, SVN is tagged, and project POM files are updated with next version number.

Table 16 presents comparison of various build types.

build phase	quick check	snapshot	release
clean	+	+	+
compile	+	+	+
test	+	+	+
package		+	+
report[9]	±	+	+
Release			+

Table 16 Comparison of build types

Build frequencies

Because of shortcomings previously described (amount of CPU power and/ or RAM needed by some builds) it is very important to execute builds at such time, that they do not disturb each other. This can be achieved by using different build frequency:

- on every commit – CI starts build after each commit. This type of build might be very resource-consuming if commits are frequent.
- on change – CI server frequently (e.g. every 30 minutes) checks code repository for changes. If any change is detected, than the build is started.
- on demand – Build is started by user.
- nightly – Once a day (usually at late night hours) build is performed. The aim is to deliver building reports (including bug reports) to the developers on the start of their work day.

Table 17 presents typical combinations of build types and build frequencies. A simple pattern of builds usage can be discovered:

- "heavy" builds should be executed rarely (on demand or in time of lesser activity),
- "light" builds should be executed as often as possible.

	quick check	snapshot	release

[9] In case of „quick check" build, only the most important reports (i.e. unit test report) is available.

on every commit	+		
on change	+		
on demand	+	+	+
nightly		+	

Table 17 Combinations of build types and build frequencies

7. Quality assurance

Quality assurance (QA) can be defined as "systematic process of checking to see whether a product or service being developed is meeting specified requirements" [25] The main goals of QA are to create a product which:

- is suitable for intended purpose,
- is free of mistakes.

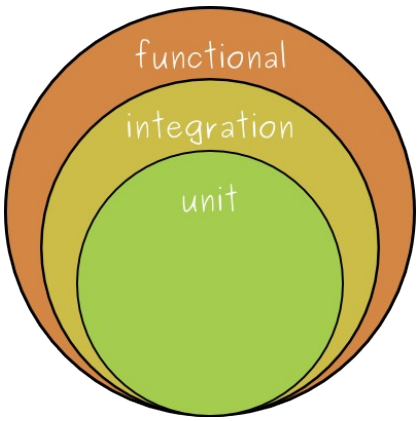
The WKI System, in order to serve as a platform for various applications, must be reliable, error-free, predictable, fast and easy to use. This sections describe tools used in the WKI project to ensure that the final result will be of highest quality.

Thanks to the fact that the quality can be regarded as a technical task, WP6 will lead and monitor activities of the quality assurance. But it is obvious, that overall high quality of the project can be achieved only if every element of the project is of high quality. For the WKI project, it means that every WPs should treat the quality of its work as top priority.

7.1. Tests plan

In order to insure that the WKI System works properly, various tests are planned. First group of tests checks if the system works as planned in terms of the provided functionality. These test must be performed on different levels:

- examining single classes,
- examining cooperation between layers,
- checking the system from the end-user perspective.



Picture 28 Functional tests

The second group of tests addresses non-functional requirements and covers wide range of aspects from system scalability to the end-user experience (usability aspects).

7.1.1. Unit tests

Unit tests allow to examine if a single, small part of a project (i.e. a single class) works fine. The idea is to isolate the SUT[1] and use test doubles[2] (mocks, stubs, test-spies, fakes, etc.) for every collaborator of the SUT. This way few goals are achieved:

- Thanks to the fact that all the collaborators are under control, it is clear that likely bug must be in the SUT itself. Such a bug can be pointed out with great precision (i.e. line number can be traced).
- Unit tests are quick to execute because no heavy dependencies are used (all external entities like application context or data base connection are stubbed). This allows developers to get rapid feedback and makes regression testing cheap.
- Adoption of Test Driven Development[3] (TDD) technique or even its part called test-first[4] results in a cleaner, testable code with well-defined class responsibilities.

Unit testing frameworks and tools

Among the Java developers two testing frameworks are especially popular:

- jUnit[5] (the ancestor of all unit testing frameworks),
- TestNG[6] (the runner-up, with scope broader than unit tests only).

Both frameworks are supported by popular IDEs, both can be integrated easily with the build process (via Ant or Maven) and thus be a part of continuous integration.

Code coverage tools, which shows the percentage of code examined by tests, are also available– with Cobertura[7] being the most popular (and free) solution. These tools can also be easily integrated into the building process.

[1] SUT – System Under Test, [24].

[2] For the full unit-testing vocabulary explanation please refer to [24]

[3] Test Driven Development, a software design methods, that promotes short development iterations based on pre-written test cases.

[4] "Test first" is a coding technique which promotes writing tests before the actual code. This way the minimal amount of code is written, because no code is created if there is no failing test. Also high (even 100%) code coverage is achieved.

[5] http://www.junit.org

[6] http://testng.org

[7] http://cobertura.sourceforge.net/

Package	# Classes	Line Coverage	Branch Coverage
jtge.util	2	95%	80%
jtge.util.grid	7	72%	100%
jtge.util.grid.direction	4	92%	100%
jtge.util.grid.hex	1	67%	N/A
jtge.util.grid.hex.builder	4	52%	60%
jtge.util.grid.hex.directiongroup	6	92%	100%
jtge.util.grid.iterator	2	89%	100%
jtge.util.grid.square	1	67%	N/A
jtge.util.grid.square.directiongroup	2	91%	100%
jtge.util.grid.torus	7	73%	N/A
jtge.util.random	4	38%	30%

Picture 29 Code coverage report generated with Cobertura (from 16)

Unit tests in the WKI project

Service developers are responsible for writing and running unit tests. These tests are by default executed during each building process (performed with Maven Surefire Plugin[8]). The CI server will also run unit tests and create code coverage report during every build. No fixed threshold of code coverage is set[9]. Coverage reports will be examined by code review group (see section 7.4). The absence of unit tests will be treated as a serious threat to the code quality.

7.1.2. Integration tests

While unit tests exercise single classes, integration tests check if "our code" works well with the code provided by 3rd parties. Such tests are usually used to see if, for example, the classes from DAO[10] layer works well with the ORM[11] mapping tool (which API is outside the control of the developers).

Setting integration tests usually requires much more work than setting unit tests (for example, the application context and database connections must be created). To overcome this problem popular frameworks and libraries provide utility classes aimed at this tasks (e.g. classes from org.springframework.test package provided with Spring framework), or facilitate the whole integration testing process from the very beginning (e.g. Grails[12]).

[8] http://maven.apache.org/plugins/maven-surefire-plugin/ - a Maven plugin which executes tests and generates test reports

[9] Code coverage tools allows to set a minimal expected percent of code coverage and report errors if the project does not fulfil it.

[10] DAO, Data Access Object layer is responsible for communication with various data sources, i.e. databases.

[11] ORMs, Object Relational Mapping framework facilitate the process of conversion of objects (e.g. Java objects) into relational databases (and vice versa). Hibernate (http://hibernate.org) is the most popular ORM for Java.

[12] http://grails.org

Integration tests by nature execute much slower than unit tests[13]. Also, if an integration test fails, the exact reason for the error might be hard to find.

Integration tests in the WKI project

Every service must be compatible (must integrate well) with the WKI System architecture. This includes proper OSGi entries in MANIFEST.MF and Spring-DM configuration files. The role of the integration tests is to check if every service can be successfully deployed to the WKI System.

Integration tests can be run (and should be run by service developers) on local environments, but the value of such tests is limited - the local environment configuration which might be very different from the production one. To gain a better confidence in integration tests, and to control the whole testing process WP6 will perform such tests on staging and production environments. Reports of functional tests will be available for insight.

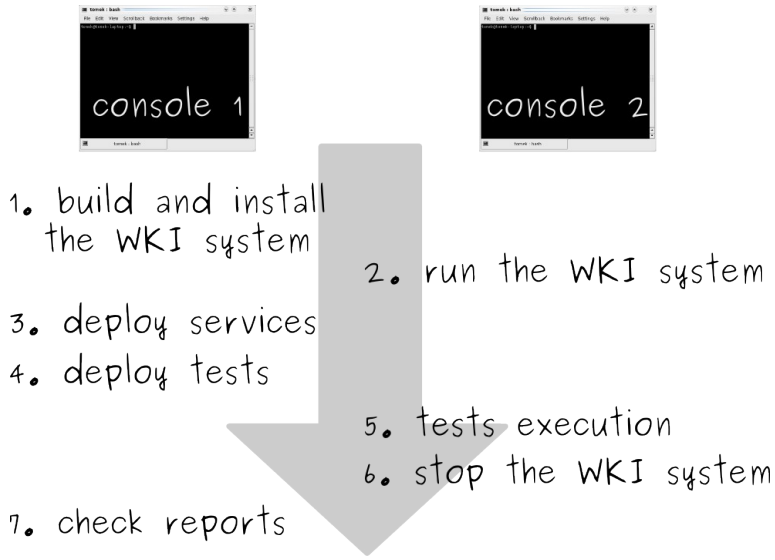
Automation of integration tests

Due to the fact that integration tests involve deployment of artifacts to the WKI System, they might be hard to integrate into the CI process. Some preliminary solutions has been already created by WP6, yet no full automation of the process has been achieved.

Picture 30 presents the current version of this process, as performed by WP6. It uses the in-container-testing technique which in this case means that tests are also services (OSGi bundles) deployed to the WKI System. This allows them to interact with other services via OSGi registry.

All the steps presented on Picture 30 are automated with Ant or Maven scripts. For example step "3. deploy services" is performed by a Maven script. This script retrieves services with given versions from local Maven repository and copies them to the "deploy" folder of the WKI System.

[13] The us of in-memory databases has significantly shortened the execution time of integration tests. The downside of this approach is that such tests use different database than the one used on production environment.



Picture 30 Semi-automatic process of integration testing

This process will be further developed in order to achieve full automation. In addition to that, it can be used as a base for creation of similar processes for functional tests.

WP6 will share the process of automation of integration tests to facilitate running of such tests by service developers.

7.1.3. Functional[14] tests

Previously described unit and integration tests examine various parts of the system. Both unit and integration tests are "developers tests", that is, they are useful for developers to assure them that the classes and components works fine. To gain the confidence about the system as a whole, different kind of tests is needed. Functional tests are performed "on customer side", which means they answer the question that is vital for client of every system: "Does it work as expected?".

The goal of functional tests is to examine the system in the same way in which it will by used by client. This implicates the following:

1. system must run on the environment which closely resembles the production environment,
2. system must be exercised via interface (usually GUI).

Production-like environment

In case of the WKI System, the first point will be assured by existence of the production environment (see section 6.3.3). Of course, the sole existence of such environment dedicated to end-to-end tests does not solve all the problems related to functional tests. Issues typical for functional tests - like setting up of resources (i.e. database), dependencies among tests,

[14] These kind of tests are also known as „acceptance", „system", "end-to-end" etc. – no coherent naming has been accepted by software community to describe such tests.

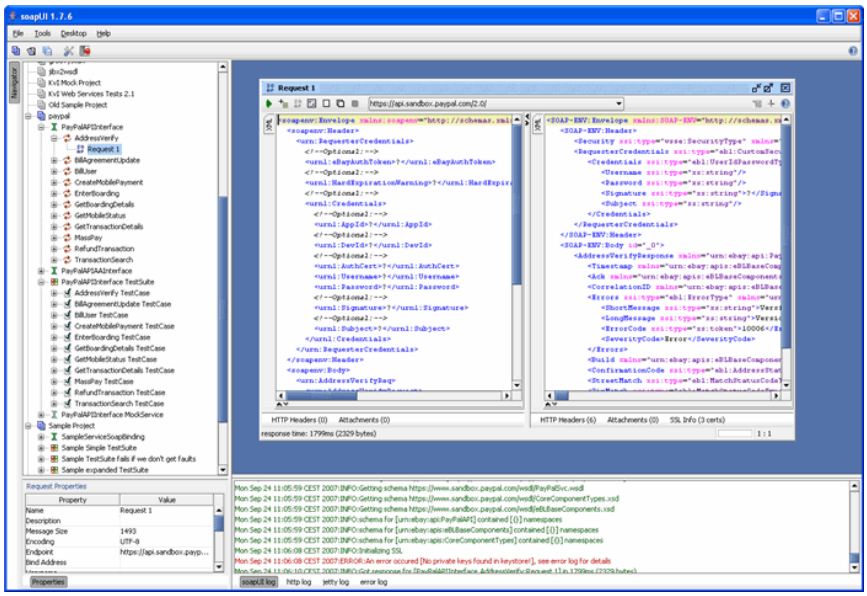
slow execution time, etc. - will have to be resolved during the development.

Testing via interface

As for the second point – exercising the system via interface - in case of the WKI System, different interfaces need to be tested: Web service interface and GUI interface (see section 5.4).

Webservice interface

In case of services and Camel flows defined inside the WKI System and exposed via Web services the process of testing requires use of tools such as SoapUI[15], that are able to send a SOAP[16] message, and analyze the received answer. SoapUI greatly reduces the amount of work needed to create tests that use SOAP messages.



Picture 31 SoapUI testing tool (from 18)

SoapUI integrates with jUnit with SoapUITestCaseRunner class. SoapUI provides a Maven SoapUI Plugin[17] which makes it easy to run tests via Maven and integrate them into CI process.

GUI interface

Functionality of WKI applications (for both use-cases) can be tested via GUIs. The GUI for these applications can take different forms, the most important being web interface created as HTML pages.

There are many good functional testing tools that are capable of simulating the browser in order to mimic the behaviour of user and then to exam-

[15] http://www.soapui.org

[16] http://www.w3.org/TR/soap/

[17] http://www.soapui.org/plugin/maven2/index.html

ine the resulting html pages. Available tools differ in many aspects, for example:

- they simulate the browser, or take over the real one,
- they allow to use various languages for tests writing (Java, scripting languages, XML, html, proprietary text-base formats, ...),
- they promote the idea of recording tests, or they deter user from doing this,

The time of tests execution and reports provided are also very different. It is planned for the WKI applications, that they will have rich interfaces created with use of AJAX[18]. This excludes some of the testing tools that are not capable of testing such rich web pages.

Picture 32 presents a typical functional testing via GUI scenario. First, test cases are created (e.g. recorder with Selenium IDE).Then, a test runner executes these tests, which means it issues commands for browser (e.g. "click link titled 'Home'", "type 'John' into form field with ID='name'", etc.). After this, it examines the resulting pages according to assertions included in tests (e.g. "check if the title of the page is 'Home page'", "check if error message 'No permission to see this page' was printed", etc.). The browser responses to the requests issued by the test runner by communicating and requesting data from the tested system.

Picture 32 Functional testing via web GUI with Selenium IDE

Because SMIND has experiences with Selenium[19] and WebTest Canoo[20], these two tools will be promoted to be used for functional testing of the WKI System. These two tools have very different characteristics, and it is also possible that both of them will be used to examine different parts of the WKI System and it's applications.

Tests scenarios and tests execution

Scenarios for functional tests will be derived from the use-cases of both WKI applications. This will be done by WP7 in cooperation with WP6.

[18] AJAX, Asynchronous JavaScript and XML, a group of techniques used to create rich internet applications.
[19] http://seleniumhq.org/
[20] http://webtest.canoo.com/

A process similar to the one described in 7.1.2 is being developed by WP6 for functional tests. Once achieved, it will allow to run functional tests often, which helps to avoid regression bugs.

Functional tests will be executed on the staging environment. Reports of functional tests will be available for insight.

It is possible that a subset of functional tests ("smoke-tests"[21]) will also be executed more frequently on local machines in order to get rapid feedback during development.

7.1.4. Non-functional tests

Apart from tests that check if the WKI System works properly in terms of its functionality, other tests are also planned. These tests examine different, non-functional requirements that the WKI System should fulfil.

All reports of the non-functional tests executed on stating or production environment will be available for insight. Found errors will be analyzed by WP6 and the request to fix them will be send to responsible developers.

Usability testing

The applications of the WKI System build for each use-case should be examined towards usability. This involves checking of time and number of steps required to perform basic tasks and number of mistakes that users make during this process. Moreover, user's feelings about the system should be taken into account.

This aspect is critical/crucial for the ER use case. User interface of this application must be especially user-friendly and shall allow to perform certain actions (like reporting a dangerous situation) intuitively, using minimal number of steps.

It has not yet been decided, how exactly usability tests will be performed. It is expected that WP7 will provide test scenarios, that will allow to examine typical actions performed by users.

Load tests

Load tests are used to examine the behavior of the system in statistically normal load. In general, application shouldn't have any problems with serve any request from simulated client (or not greater then allowed number of errors) and responses times should be less then established.

This kind of tests will be performed on the staging environment by WP6 in cooperation with WP7. Tools are yet to be chosen – JMeter[22], SoapUI and Grinder[23] are taken into consideration.

[21] A subset of tests used to make sure (quickly) that a system under test does not fail catastrophically.
[22] http://jakarta.apache.org/jmeter/

[23] http://grinder.sourceforge.net/

Performance tests

Every application has limited scalability and is not able to serve requests above some threshold value effectively. Performance testing is used to examine some system properties like scalability, reliability and system usage.

In performance testing, load is changed from a minimum to the maximum level that the system can sustain without running out of resources, or having transactions suffer excessive delay. It allows to check some system properties like scalability, reliability and system usage.

This kind of tests will be performed on the staging environment by WP6 in cooperation with WP7. Same tools as for load tests will be used.

7.2. Development Conventions

Development conventions are very important especially in geographically dispersed teams, where many developers create code separately. Without such conventions code will be hard to understand by any other team member than the author himself. Code conventions improve the readability of the software, allowing developers to understand new code more quickly and thoroughly.

Such development conventions has been set for the WKI System. They are included into "Development Conventions" guideline.

Some of the conventions used in the WKI System are based on two Sun's documents:

- Code Conventions for the Java Programming Language[24],
- How to Write Doc Comments for the Javadoc Tool[25].

7.2.1. Naming conventions

Naming conventions proposed for the WKI introduce rules about how specific elements should be named or should adhere to some pattern. For example, Java class names should start with upper case, variables should have meaningful names which should start with lowercase, all packages in WKI should be started from *eu.weknowit.{Workpackagename}.*

7.2.2. Code & documentation conventions

Development conventions are not only about naming, but other rules are also important:

- formatting – how lines should be wrapped, maximum line length, etc
- files organization in project – each project should be created using specified maven archetype

[24] http://java.sun.com/docs/codeconv/

[25] http://java.sun.com/j2se/javadoc/writingdoccomments/

- documentation conventions – for example, descriptions (javadoc) are needed in all interfaces and classes, and additionally every external interface must be described on project site with respect to the proposed service declaration template.

7.3. Common WeKnowIt POM's

As previously stated, all WeKnowIt services will be build with Maven. Two POM's where created in order to facilitate and standardize this process[26].

7.3.1. Weknowit-bundle-pom

Weknowit-bundle-pom is an archetype[27] of WeKnowIt projects. The project generated from(albo from this archetype albo with use of this archetype) with use of this archetype has a special folders and files structure for OSGi and Spring configuration files, necessary to build OSGi bundles (WKI services). It also creates a POM which inherits weknowit-parent-pom.

7.3.2. Weknowit-parent-pom

Weknowit-parent-pom contains common configuration for build process, reports etc. It contains preconfigured plugins of static code checkers (PMD[28] & Findbugs[29]) and code coverage tool (Cobertura). Developers of services are encouraged to examine reports generated by these tools regularly, in order to improve the quality of code.

WKI parent POM also enforces use of certain versions of software (via Maven Enforcer Plugin[30]) which helps to avoid bugs related to e.g. usage of different Maven versions.

All Maven project in the WKI project, should inherit from this POM file. It is not intended to be used standalone, but rather to be referenced by other POMs.

Weknowit-parent-pom configures a set of plugins for WeKnowIt projects[31]:

Plugin name	Task
maven-enforcer-plugin	Checks environment preconditions (e.g. Java version).
maven-compiler-plugin	Compiles source code.

[26] For more information regarding Maven POMs please refer to "How to create a Maven project" guideline.

[27] Maven archetype can be understood as a template for creating new projects.
[28] http://pmd.sourceforge.net
[29] http://findbugs.sourceforge.net/
[30] http://maven.apache.org/plugins/maven-enforcer-plugin/
[31] Not all plugins listed here as some of them provide low-level functionalities used by other plugins and not directly visible by end-user.

maven-source-plugin	Generates artifacts with source code (e.g. for use in IDEs).
maven-javadoc-plugin	Generates javadocs.
maven-jxr-plugin	Generates html, browsable version of source code.
taglist-maven-plugin	Generates report about tags encountered in source code (e.g. TODO, FIXME, etc.)
findbugs-maven-plugin	Generates report of Findbugs tool (static code checker).
maven-pmd-plugin	Generates report of PMD tool (static code checker).
cobertura-maven-plugin	Generates code coverage report.

Table 18 WKI parent pom - list of configured plugins

7.4. Code reviews

As previously stated, many tools will be used to analyze the code in order to find problems. These automatic tools are useful, but not perfect - they can't spot some problems that a human will find.

Code review is a very simple, and yet very efficient technique. The main idea is that another developer (not the author of the code) examines the source code searching for suspiciously looking statements, weird design decisions or evident bugs. This technique has two goals:

- improve the overall quality of software (including not only bugs removal but also standardization of names and patterns used),
- improve developer's skill (by discussing found problems with him).

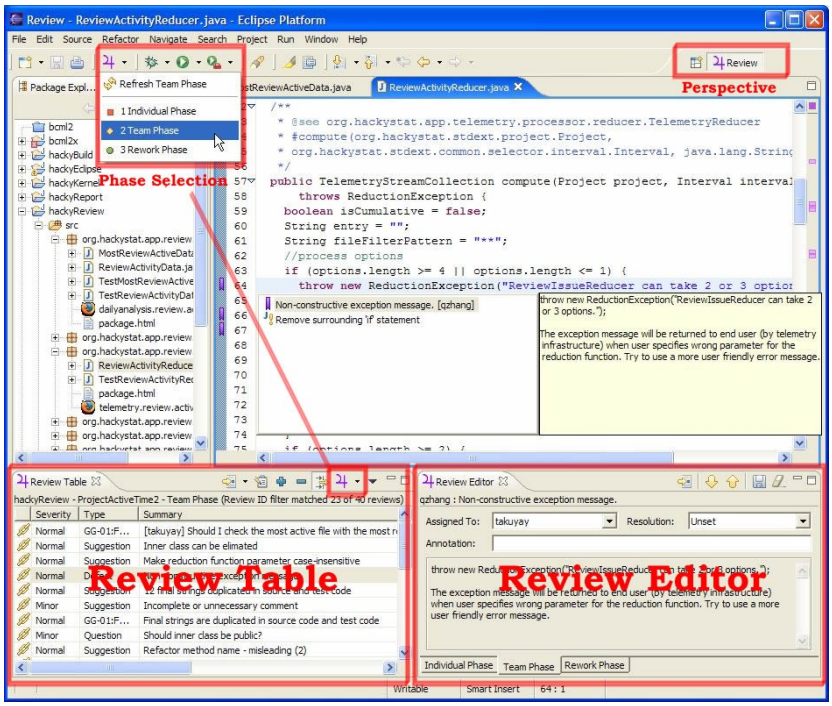
Code reviews will be performed regularly by WP6 in a group consisted of various consortium partners. The members of this group and the frequency of code reviews has not yet been chosen. It is expected that presence of developers from different WP will improve the understanding of the WKI services by all partners and will lead to overall quality improvement.

Code reviews will be run in order to find:

- evident bugs and potential problems (e.g. weak design decisions, abnormal values of code complexity etc.),
- untested code (via examination of code coverage reports - section 7.1.1),
- deviations from development conventions (described in section 7.2).

All spotted problems will be reported to the responsible developers with a request to fix them.

Use of Eclipse Jupiter plugin[32] is considered to facilitate the communication with WPs developers. This plugin allows to attach comments directly to source code under review.



Picture 33 Code review with Eclipse Jupiter plugin (from 19)

[32] http://code.google.com/p/jupiter-eclipse-plugin/

8. Integration plan

Components of the WKI System are developed by all the participants of the consortium. This makes the development process a real challenge and makes integration a vital part of the project. This section of the document describes the integration plan for the WKI System.

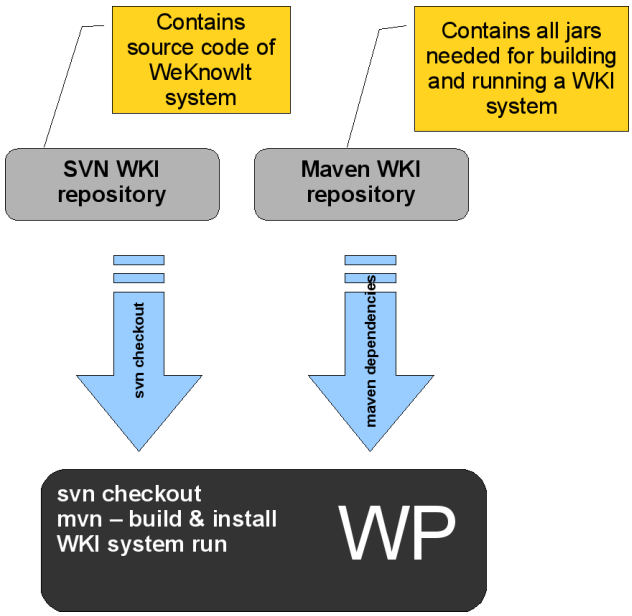
The ultimate goal of the integration is a fully functional WKI System working on the production environment.

8.1. Services development

Services of the WKI System are produced by all WPs. They have to be developed according to the guidelines provided by WP6. This assures, that the created services will be compatible with the WKI System architecture and that integration will be possible.

8.1.1. Local development

The development process starts, when developers install the WKI System on their local machines. The process of environment setting is described in "How to install Java & Maven" guideline.



Picture 34 Local environment - installation of the WKI System

During the development phase, the recommendations presented in section 7.2 should be followed, in order to produce high quality code. This includes writing of unit tests (described in 7.1.1), following the development conventions (section 7.2), using common POM files (section 7.3) and

checking for potentially dangerous code fragments with static code checkers (section 7.3).

The source code of services should be stored in common code repository management (as described in section 6.1). This makes it possible to use common CI server (section 6.5) to build services, unit-test them, generate reports and documentation.

Services developers should consult following guidelines to create services compatible with the WKI System architecture:

- "How to create Maven project"
- "How to create OSGi service using Spring Dynamic Modules
- "How to deploy OSGi bundles to the WKI System"

Local integration tests

After a service is created it should be tested if it is compatible with the WKI architecture. Preliminary integration tests (section 7.1.2) can be performed on the local environment (section 6.3.1) by service developer. This can be done by deploying the service to the WKI System installed on the local machine (as described in "How to install the WKI System"). Only after the service has been confirmed to work properly on the local environment, the development can move to the further steps.

8.1.2. Artifacts creation

After the service is checked on local environment, an artifact can be build by the CI server (section 6.5) using "release build" (section 6.5.1). The build depends on configuration of the CI server and on information included in service POM file.

8.1.3. Successful build

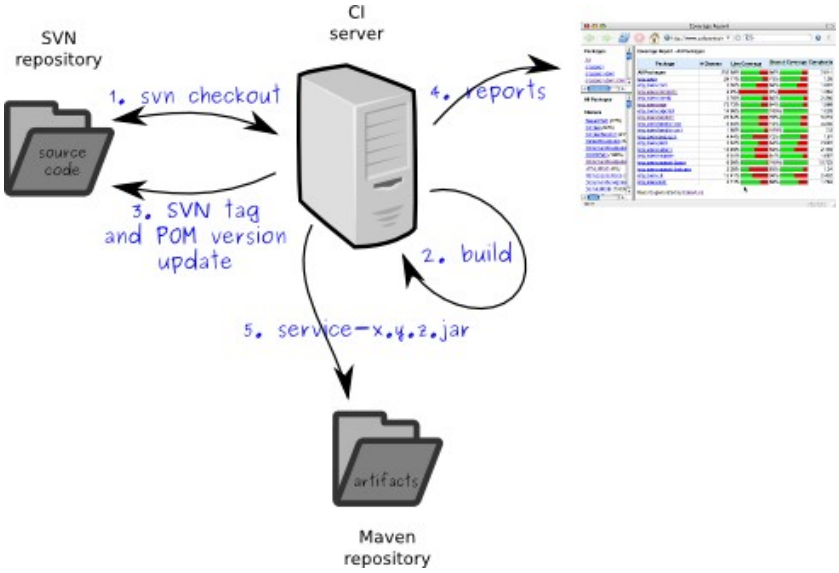
Picture 35 presents actions taken by CI server in case of successful release build.

In case of successful build CI server tags the SVN trunk (or branch depending on SVN repository structure). This step is very important, because it allows to exactly identify what files (what exact versions of classes) were used to create this particular version of service, which is very valuable in case of errors that need to be fixed.

Each artifact build by CI server is given a unique version number (this task is handled by Maven Release Plugin). The uniqueness of versions is guaranteed by the Maven Release Plugin which is used to perform such builds[1]. After tagging, CI sever also updates artifact version in POM file which guarantees that the next version will have different number.

[1] It is still possible, but rather harmful, to overcome this restrain.

Successfully created artifacts are deployed to the WKI Maven repository and reports generated during build are put on the web server.



Picture 35 CI server full release build

8.1.4. Build failure

In case of errors during build process, the SVN trunk is not tagged, no artifact is created and deployed to the Maven repository, and the service is returned to the service developer. Information about the failure (build log which contains detailed information about the build) is stored and is available for inspection.

8.1.5. Staging

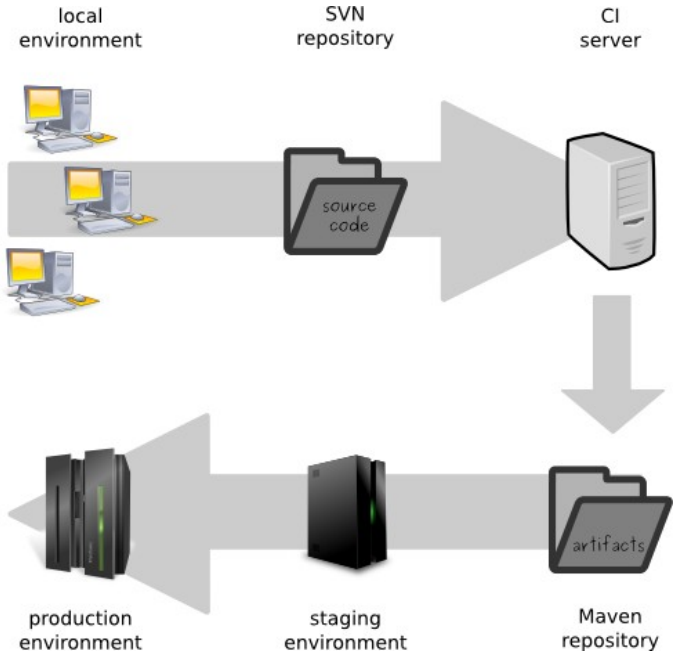
After the development on the local environment is finished and the artifacts built by CI server are available in the WKI Maven repository, the development moves to the staging environment (section 6.3.2). Services, retrieved from the WKI Maven repository, are deployed to this environment in order to check their integration with the WKI System in environment that closely resembles the production environment. If there were no errors, functional tests can be performed.

In case of errors, the service is returned to the service developer, and the cycle of development is repeated starting with the local development environment.

8.1.6. Production

After being successfully deployed and tested on staging environment, service can be moved to production environment (section 6.3.3). This is the final step of the development process.

8.2. Development cycle



Picture 36 Development cycle

Picture 36 presents the order that various environments and infrastructure elements takes in development. It also shows important role of CI server and Maven repository. Only artifacts created by builds performed on CI server and stored in the Maven repository are allowed to be deployed on staging and production environments. As previously stated, such artifacts have unique IDs which allow to control the versions used.

Table 19 presents environments and people involved during different phases of the service development.

Development phase	Environment	Participants
Coding	Local	Service developer
Unit testing	Local	Service developer
Integration testing	Local	Service developer
Integration testing	Staging	WP6 or Service developer
Functional testing	Staging	WP6 or Service developer
Integration testing	Production	WP6
Functional testing	Production	WP6

Table 19 Development cycle – environments and participants

8.2.1. Usage of CI server during development

The role of CI server has already been mentioned regarding the release builds which result in artifact creation. This section presents other uses of CI server during development of the WKI System.

Build types

Following build types are planned for each service of the WKI System. Column "type" refers to build types described in section 6.5.1.

Name	Execution	Type	Reports (available via web interface)	Artifact deployed to the Maven repository
Developer	on demand	quick check	subset	no
On change	every 30 minutes[2]	quick check	subset	no
Release	on demand	release	all	yes
Nightly	every night	snapshot	all	yes

Table 20 CI sever builds for the WKI System

In case of failure of any of automatically executed build ("On change" and "Nightly"), the responsible developers (developers who have committed code since the last successful build) are informed via emails.

Local development

Picture 36 does not show the fact, that the CI server can be used by service developers during their work on the local environment. As long as all the source code is stored in the SVN repository CI server can perform snapshot (section 6.5.1).

Such snapshot builds might be very valuable for two reasons. First, reports and documentation created during such build might be reviewed by any member or the WKI project. Second, using CI server for build from the very beginning also allows service developers to avoid the "it builds on my machine" error. This error can be caused, for example, by existence of some proprietary artifacts in the local repository of developer which are not available on the WKI Maven repository, and thus can not be retrieved by CI server during build process. In fact, developers are encouraged to perform builds on CI server at least once a day.

8.3. Integration schedule

This section presents integration schedule. To every activity a date is assigned.

[2] The build is fired only if any changes in source code were noted after 30 minutes period.

8.3.1. Development infrastructure

	Responsible WP	Date and description
Maven repository	WP6	M12, Available
SVN	WP6	M12, Available
Common Maven POM's	WP6	M10, Available
Local environment	WP6	M10, Guidelines available
Staging environment	WP6	M15
Production environment	WP6	M18
WKI System installation version	WP6	M10, Available
Bug Tracking System	WP6	M12, Available
CI Server	WP6	M15

Table 21 Integration plan schedule - development infrastructure

8.3.2. Specification of system behaviour

	Responsible WP	Date and description
Workflows specification	WP7, WP6 & all WPs	M14 (initial version) M18 (final version)
Test-cases for functional tests	WP7 & WP6	M19

Table 22 Integration plan schedule - specification of system behaviour

8.3.3. Quality control

	Responsible WP	Date and description
Code reviews	All WPs, lead by WP6	Regular reviews of lately committed code – every 2 weeks. Starting M15.
Integration tests	Local – all, Staging – WP6, Production – WP6	Performed regularly. Frequency depends on automation level of tests achieved. All tests executed on staging environment after any new service or new version of the WKI System is created.

		Tests on production environment only after all tests passed on staging. Starting M16.
Functional tests	Local – all, Staging – WP6, Production – WP6	Performed regularly. Frequency depends on the automation level of tests achieved. All tests executed on staging environment after any new service or new version of the WKI System is created. Tests on production environment only after all tests passed on staging. Starting M20.

Table 23 Integration plan schedule - quality control

9. Conclusions

Deliverable D6.1.2 presents the architecture of the WKI System. It begins with requirements. Then, it explains the technical solutions used, and describes how the selected technologies work together in the WKI System. But this deliverable goes beyond narrow view of the architecture as a "technology thing". It shows the integration plan prepared by WP6, that is intended to deliver the development of services and the evolution of the whole system to success.

Two other WP6 deliverables of M12 (D6.2.1 and D6.3) provide additional information that make the picture of the WKI System complete. D6.2.1 presents prototype of the WKI System, while D6.3 is a prototype of the vital part of the WKI System - the WKI Data Storage. Both deliverables prove, that the prepared system architecture described in D6.1.2 actually works.

10. References

[1] http://gravity.sourceforge.net/servicebinder/osginutshell.html

[2] http://en.wikipedia.org/wiki/OSGi

[3] http://soatechlab.blogspot.com/2008/09/whats-new-in-servicemix-4x.html

[4] http://servicemix.apache.org/5-jbi.html

[5] A Look Inside FUSE ESB 4: An OSGi-Based Integration Platform, Tijs Rademakers, http://java.dzone.com/articles/fuse-esb-4-osgi-based

[6] http://static.springframework.org/osgi/docs/1.1.3/reference/html/app-deploy.html

[7] http://camel.apache.org/architecture.html

[8] http://www.w3.org/TR/ws-gloss

[9] http://java.sun.com/products/jms/docs.html

[10] G. Hohpe, B. Woolf, "*Enterprise integration patterns: Designing, building and deploying messaging solutions*", Addison-Wesley Longman, Amsterdam 2003.

[11] http://blogs.exist.com/oching

[12] Jeff Davis , "Open Source SOA", Manning Publications, 2008.

[13] T. Rademakers, J. Dirksen, "*OpenSource ESBs in action*", Manning Publications, 2008.

[14] http://www.theserverside.com/tt/articles/content/SettingUpMaven-Repository/article.html

[15] T. Erl, "*Service-Oriented Architecture: Concepts, Technology, and Design*", Prentice Hall International, 2005.

[16] Jean-Laurent de Morlhon weblog, http://morlhon.net/blog/2005/08/

[17] D. A. Chappell, "*Enterprise Service Bus: Theory in Practice*", O'Reilly Media, 2004.

[18] http://www.soapui.org/userguide/overview.html

[19] http://purnank.in/2008/07/jupiter-plugin-for-code-review

[20] https://open-esb.dev.java.net/public/pdf/JBI-Components-Theory.pdf

[21] http://en.wikipedia.org/wiki/Layer_(object-oriented_design)

[22] http://servicemix.apache.org/nmr.html

[23] http://cxf.apache.org/

[24] George Meszaros, "*xUnit Test Patterns*", Addison-Wesley, 2007

[25] http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci816126,00.html

[26] http://www.w3.org/TR/ws-gloss/

[27] http://en.wikipedia.org/wiki/Bugtracker

[28] "Progress FUSE Selected by Sabre Holdings as Their Enterprise Standard ESB", retrieved from http://finance.yahoo.com/news/Progress-FUSE-Selected-by-bw-14523272.html, on 30th March 2009.

[29] http://www.iona.com/pressroom/2007/20071204.htm

[30] http://labs.ingenta.com/2006/05/juc/paper.pdf

[31] http://java.sun.com/products/jms/

[32] Leok Bakker, "Goodbye Hub-and-Spoke, Hello ESB? Integration Architecture With BizTalk 2004", retrieved from http://dotnet.sys-con.com/node/121831, on 30th March 2009.