Deliverable D2.1

Report on the implementation of the architecture

Project Acronym	OASIS
Grant Agreement number	297210
Project Title	Towards a cloud of public services

Project co-funded by the European Commission within the ICT Policy Support Programme

Deliverable reference number and title	OASIS_D2.1
Status	Final

Dissemination level[1]	PU	**Due delivery date (project month)**	M7
Nature[2]	R	**Actual delivery date**	16/02/2014

Lead beneficiary	ATOL CONSEILS ET DEVELOPPEMENTS SAS
Contributing beneficiaries	Atol, Open Wide, Pôle Numérique
Author(s)	Christophe Blanchot, Yannick Louvet, Jérôme Poittevin, Marc Dutoo, Sylvain Chambon

Revision History

Revision	Date	Author and Organisation	Description[3]
0.1		Christophe Blanchot (PN)	Creation
vf	14/02/2014	Bruno Thuillier (Pôle Numérique)	Final corrections and Approval

[1] Dissemination level: **PU** = Public, **CO** = Confidential, only for members of the consortium and Commission services

[2] Nature of the deliverable: **R** = Report, **P** = Prototype, **D** = Demonstrator, **O** = Other

[3] Creation, modification, revision, final version for evaluation, revised version following evaluation

Statement of originality:

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Deliverable abstract

The European CIP (Competitiveness and Innovation framework Programme) OASIS project aims at a shared data and services Cloud platform for e-government that eases up creation, publishing, communication and reuse tasks for public administrations, citizens, entreprises and associations alike, by breaking up traditional information silos. OASIS architecture is implemented in order to meet requirements defined in D1.2 "Architecture design", as well functional ones such as data management and collaboration, catalogs, notifications ; as non-functional ones such as security and scalability.

Prior to actual development, several candidate technologies have been assessed on key points for OASIS so that the best one can be choosed for a given job.

Actual architecture implementation entails OASIS Datacore, its cross-business collaborative Linked Data platform ; OASIS Kernel, its security, directory and integration platform ; OASIS Portal, its user frontend web application ; and all federated Services provided to users through OASIS Portal, sharing business data in Datacore, integrated thanks to Kernel services. All are integrated in a decoupled, scalable manner using REST APIs in a Web-like overall architecture. Datacore and Kernel implementations are detailed in terms of technologies they build on and technical components they are made of (both being written in Java and using MongoDB storage), as well as models and algorithms used to achieve detailed business features and non-functional requirements, while their use is let to D2.2 User Guidelines to describe. Portal and federated Services architectures are mainly detailed at their infrastructure level, Portal because users still have to be consulted on its features in the upcoming pilot phase, and federated Services because implementation of their various business domains are out of scope of OASIS platform core architecture.

Implementation of OASIS architecture has produced two loosely-coupled though well-integrated, secure, scalable technical platforms which are available to Service Providers to consume in their collaborating applications, deployed on their own infrastructure and offered for installation through user Portal. In addition, it has led to an updated outlook on how to achieve OASIS goals, be it how to best implement a feature or innovating tools to provide to best help federated Service developers.

Project Management Review

	Reviewer 1: WP leader			Reviewer 2: B. Thuillier		
	Answer	**Comments**	**Type***	**Answer**	**Comments**	**Type***
1. Is the deliverable in accordance with						
(i) the Description of Work and the objectives of the project?	☒ Yes ☐ No		☐ M ☐ m ☐ a	☒ Yes ☐ No		☐ M ☐ m ☐ a
(ii) the international State of the Art?	☒ Yes ☐ No		☐ M ☐ m ☐ a	☒ Yes ☐ No		☐ M ☐ m ☐ a
2. Is the quality of the deliverable in a status						
(i) that allows to send it to European Commission?	☒ Yes ☐ No		☐ M ☐ m ☐ a	☒ Yes ☐ No		☐ M ☐ m ☐ a
(ii) that needs improvement of the writing by the originator of the deliverable?	☐ Yes ☒ No		☐ M ☐ m ☐ a	☐ Yes ☒ No		☐ M ☐ m ☐ a
(iii) that needs further work by the partners responsible for the deliverable?	☐ Yes ☒ No		☐ M ☐ m ☐ a	☐ Yes ☒ No		☐ M ☐ m ☐ a

* Type of comments: M = Major comment; m = minor comment; a = advice

Table of Contents

1 Introduction

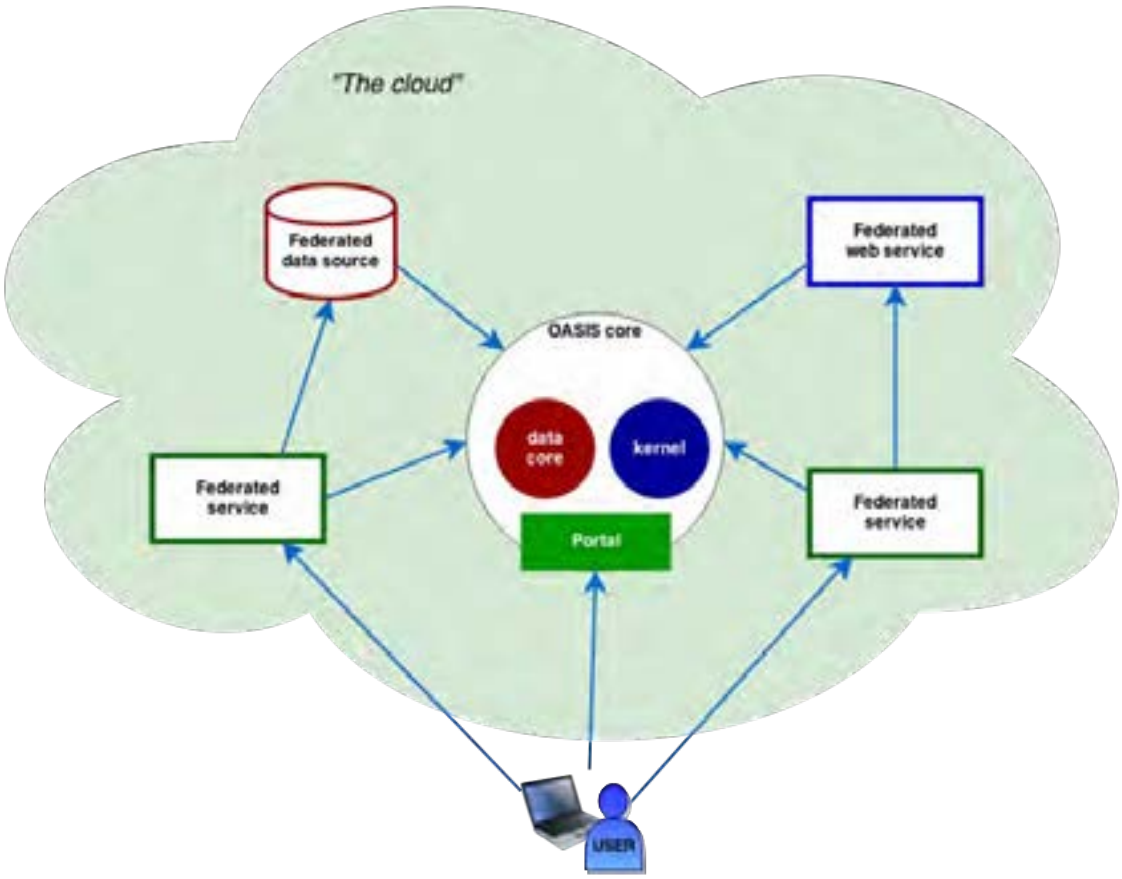
1.1 Context

OASIS is:

- a federation of services
- an interoperable database
- an advanced social network
- a smart and user friendly graphical user interface

And OASIS work on the Internet, with Internet and cloud computing technologies.

To achieve these objectives, OASIS is made up of several subsystems :



The federated services ensure the « business » functionalities required by the users.

The **portal** is the user interface to access these services, set OASIS and have a unified user experience.
The **kernel** provides the required functionalities to federate the services : authentication, management of users and organisations, directory, management of persoanl data, communication and notifications, catalogue of available resources.
The **datacore** can store shared data and guarantee the respect of the governance rules of these data.

OASIS relies on a REST architecture : the API are then resources, the communication protocol between the various components is http (more exactly https to guarantee the confidentiality), and the components are weakly coupled.

1.2 Document objective and content

This document relies on:

- D1.1 : requirements
- D1.2 : architecture
- D2.3 : study of the security and confidentiality

It presents the choice of the technologies used to make OASIS and describes the implementation of the various modules and services for the implementation of the pilot.

This document is divided into several parts :

- First part : evaluation of the technologies (chapter 2).
  For each technology we used, we evaluated the different available solutions so as to make the most adapted choice to the OASIS context (context integrating the available skills in theconsortium).

- Second part : General architecture and implementation of sub-systems (chapters 3 to 6)

  We present a report of the implementation of each sub-system (datacore, kernel and portal), explaining how we implemented the selected technologies to make the functional architecture described in deliverable D1.2.

- Third part : implementation of the pilot (chapters 7 to 10)
  We present in the third part the implementation of the pilot including :
  The hosting of the datacore, the kernel and the portal
  - A focus on the elasticity and the security of the pilot
  - The adaptation and the implementation of the federated services within the pilot phase

2 Technology assessment

2.1 Methodology

Technical choices are often driven by personal convictions. These beliefs are often based on a first negative experience or otherwise on a positive experience (and based on an earlier version of the product), or on feelings or reputations weakly substantiated.

To avoid this pitfall, we defined the main important criteria and assessed the available options for each criteria and then averaged the results out.
We commissioned the evaluation independently from different partners, outside experts, and then we made the synthesis. The synthesis is not the average score but a consensus resulting from reasoned discussions around the assessments.
Of course, it is weighted by the control and skills on the solutions by the partners (for a good reliability of the final developments, it is important that the technical team has a great experience in the chosen solution).
This method enabled us to establish the technical team on the basis of a technology selection, and not to base the technical choices on a pre-defined team.

The main selected criteria are:

- Elasticity that is the ability of technology to manage the use of a potentially unlimited number of servers in several data center.
- Functional wealth.
- Adoption by a large community of developers and maturity of the technology (it is important to ensure the OASIS sustainability).
- License type (open source and free solutions are preferred).
- Compliance with standards.
- Reliability.

The specific criteria depend on the type of technology and appear in the comparative tables in this chapter.

We assess each criteria with a note from 1 to 5, 5 is the best.
For licence, 1 means proprietary, and 5 full open source and public licence (GPL, BSD, etc…).

2.2 Languages

2.2.1 Presentation

Languages have been grouped into several categories to suit different use cases. Most treatments that do not require critical performance are developed in a common language that enables high-quality software through the integration of software factories, such as Java.
If some treatments should be very efficient in a highly competing environment, the use of Go language development can meet the performance requirements, even if tooling is less well-stocked.
Finally, web scripting languages have been studied since the ecosystems they provide are in line with some needs (portal, store ...).

2.2.2 Java

Introduction
Java today stands as a prime candidate for the development of Web Oriented Architecture applications.
This is a mature and widely used language, enjoying a rich ecosystem. It is therefore easy to find people mastering this language or technologies based on it, ensuring ease to enhance or fix software.
Due to its maturity (almost 20 years of existence), a very rich ecosystem exists around this language and many open source projects, framework, libraries are available to treat a wide variety of issues.
Software factory tools to industrialize processes of development are numerous, allowing to test and deliver software efficiently.
Over the years, the JVM has become more and more intelligent, allowing optimization of the execution of programs beyond what a simple compiler can do: a virtual machine does not just gather static information about the instructions to execute but also metrics collected following the observation of the program itself, allowing optimizations more effective and targeted.

Use case for OASIS
Java will is perfect for developping :

- REST API development
- Services used by interfaces
- Server side apps
- Identity Management
- Broker

Comparison with competitors

Criterion	Java	Groovy	Scala	node.js	C#
Sustainability and membership of community involvement (can we easily find developers?)	5	4	3	3	5

Efficiency	5	4	4	2	3
Reliability and structuring (typed, untyped)	4	3	5	3	4
Workshop tool of software engineering (IDE, build tools, testing)	5	5	4	3	4
Standardization of language	3	4	4	5	5
VM License	4	4	4	5	2
Average	**4.3**	**4**	**4**	**3.5**	**3.8**

Scala: uncommon language, poorer tooling
Groovy: good alternative to Java, although less common. Choice of Java has been guided by the choice frameworks and libraries mentioned below
Node.js:.performance lagging behind compared to its competitors
C#: language well designed and well equipped but in a mainly proprietary ecosystem

2.2.3 Go

Presentation
Go is a language which grew in many huge projects in recent times. A large number of major web actors are turning to the language, among which may be mentioned Amazon, Google, Heroku, Canonical, 10gen, Cloudfoundry, bit.ly.
Go is a language compiled to native code, thus having very good performance execution, while higher-level languages like C or C++ and providing a better security code and a better memory management (presence of a garbage collector).
Competing handling primitives (goroutines and communication channels) are one of the key points of this language, allowing the development of parallel treatments effectively without deadlock. The language also includes at its core an efficient http server, making it very appropriate for the development of web APIs.
The purpose of this language is to keep the best parts of C and Python, while avoiding the bad C points and correcting some problems of Python.

Use cases for OASIS
Servers treatments requiring high performance (frequent requests to be served quickly as tickets checks and other authentication tokens).

Comparison with competitors

Criterion	**Go**	**C**	**C++**
Sustainability and membership of community involvement (can Easily we find devoppers)	2	4	4

Performance	5	5	5
Reliability and structuring (typed, untyped)	5	2	3
Workshop tool of software engineering (IDE build tools, testing)	2	4	4
Development competitor	5	2	2
Average	**3.8**	**3.4**	**3.6**

2.2.4 PHP

Presentation
Php is a widely used scripting language in the web, enjoying a rich ecosystem. Despite all the additions and improvements of the latest versions, the language is still suffering from its origins and its history regarding its robustness. But the large number of tools and libraries available make it a competitor of choice for some types of development (Drupal).

Use case for OASIS
Web Application Development relying on existing php bricks (Drupal): portal, store.

Comparison with competitors

Criterion	**PHP**	**Python**	**Ruby**
Sustainability and membership of community involvement (can we Easily find devoppers)	5	4	4
Performance	4	5	2
Reliability and structuring (typed, untyped)	1	3	2
Workshop tool of software engineering (IDE build tools, testing)	3	3	3
Average	**3.25**	**3.75**	**2.75**

The choice of language is guided by the choice of framework.

2.3 Web framework

The developed REST API are accessible via HTTP. This choice allows a higher level of interoperability and better performance than the SOAP standard. Although this way to provide API is not a standard, strictly speaking, it is the state of the art in 2013 and can be considered a *de*

facto standard. The HTTP REST approach has the advantage of being based on all the mechanisms induced by the HTTP protocol such as the cache.

2.3.1 JAX-RS

Presentation
JAX-RS is a specification for developing REST services which is part of Java EE. The second version of the spec has just been published. Several implementations are available that can be deployed in a servlet engine or used directly as a standalone application with an embedded HTTP server such as grizzly netty.

Use cases for OASIS
REST API development
Services and server side web application processing

Comparison with competitors

Criterion	JAX-RS	Play	Play2 (Java)	JSF	Spring MVC
Sustainability and membership of community involvement (can we Easily find developers)	5	3	3	4	5
Respect of web oriented architecture	5	5	5	1	5
Compliance with standards (REST, MVP ...)	5	5	5	2	5
Functional Richness (notification, thread ...)	4	4	4	3	4
Performance	5*	4	4	2	4**
Scalability	5*	4	5	1	4**
Average	**4.8**	**4.1**	**4.3**	**2.1**	**4.5**

* varies depending on the implementation. Grizzly + Jersey or Netty + RESTEasy go well
** varies depending on the servlet containter

JSF: Stateful model, inefficient, not scalable
Play1: uncertain framework sustainability
Play2: scala core = expensive adoption
Spring MVC: spec techno prefered (possible implementation change)

2.4 Storage

2.4.1 Why NoSQL

The world of NoSQL databases is about non relationnal database management systems, ie which do not rely on tables as the unit of information storage. Different types of NoSQL databases exists, that all have common goals of simplicity and performance, allowing elasticity deployment through horizontal scaling, ie by adding nodes to a group of machines, rather than increasing the capacity of a single node.

2.4.2 MongoDB

Presentation

At the time of its conception, the creators of MongoDB have sought to meet the following objective: keep the maximum capabilities of a relational database, while achieving a good level of storage performance, whether in access speed or handled data size in a cluster configuration.

Data are organized in collections of documents, which are stored and transferred in a BSON format (a binary variant of the JSON format).

Queries can be expressed in JSON for querying and updating data. Map/Reduce scripts, running in the database engine, can be written in javascript. MongoDB also includes an aggregation framework for expressing treatments applied to the data on the server. A mechanism known as GridFS is also proposed to store binary stream large (> 16MB) in the same database enabling the same sharding and replication options.

The concept of transaction is absent but many atomic operators allow the manipulation of document as a whole to achieve, in many cases, the same objectives as the transactions transactional storage engine.

Referential integrity is also an absent concept of this type of storage, but the data are grouped as documents which are aggregated data of potentially consistent information. Data modeling has indeed to be adapted to the manipulation possibilities offered by the storage engine.

To achieve its performance goal, MongoDB stores documents in pre-allocated disk files, accessed via Memory Mapped Files. The indexes are stored as B-tree, and used properly, allow optimal search through collections, potentially accessing only data which are already in memory, thereby ensuring a high level of performance.

MongoDB storage engine uses a journaling mechanism, to ensure consistency of data even in case of sudden interruption of service, or to consistently backup a snapshot of the storage medium.

Another journal is also dedicated to the replication between nodes in a replica set.

MongoDB provides two mechanisms for the distribution of data on different nodes: the replica sets and shards.

A replica set is a set of nodes containing the same data. One of these nodes is elected replica set master, the others being the secondary nodes and replicated from the master. Only the master node is updated. You can choose the level of consistency of writing on different nodes from the client (writing can only be considered valid only after most or all nodes have received the information, the price of an obvious additional latency). It is also possible to choose whether the information in the secondary nodes can be used for reading or not. According to the need for

consistency manipulated information, we choose a setting or another.
It is also possible to share data between different nodes or different replica sets, thanks to the concept of shards: data is distributed among the nodes using a discriminating key.

Use case for OASIS
Large data storage with scalability and elasticity needs.

Comparison with competitors

Criterion	MongoDB	Cassandra	CouchDB	PostgreSQL
Elasticity	4	5	5	1
Efficacy read	4	3	5	3
Efficiency Writing	4	5	2	3
Wealth storedstructures	5	4	5	2
Data consistency control	5*	5*	3	3
Stability, reliability	4	5	5	5
Crash recovery	5	5	5	4
Perenity	4	5	4	5
License	5	5	5	5
Average	**4.4**	**4.6**	**4.3**	**3.4**

* control per request

2.4.3 Redis

Presentation
Redis is a first key-value memory database. However, it is possible to persist data to disk.
Redis can handle values with five native data structures. Rather than having a generic system in which everything can fit (as proposed by the relational systems), Redis offers special low-level structures for optimized data access. Several requests may then be necessary to access the desired information but the overall access time remains low. Redis provides a transaction mechanism, allowing to link several queries atomicly.
Redis main use is handling data with very fast access constraints, controlled volume and whose loss does not affect the consistency of the system
It perfectly fulfills the function of data cache (data expiration function is natively present), but is also completely relevant to store information whose structure is simple and are therefore suitable for the tool operation mode.

Use cases for OASIS
Storage of data to be accessed quickly, and where writings are rare or temporary (it's a memory database)

Comparison with competitors

Criterion	Redis	Memcached	Ehcache	Infinispan
Performances	5	4	3	4
Robustesse	4	5	3	4
Elasticity	2	3	4	4
Feature set	5	2	2	2
Average	**4**	**3.5**	**3**	**3.5**

2.4.4 Neo4j

Presentation
Neo4j is a graph oriented database. It is based on node concept, relationship and properties belong to it. This model helps to represent the real world, it gives it a real advantage to treat data from social network. Neo4j is the most popular solution on the market (of database oriented graph).

Use case for OASIS
Neo4j is well adapted to the social graph, wich represents the relations between persons and entities.

Implementation in a relational database is not direct. There exist several ways to answer this problem, pattern query is one of them but it's fastidious. Database such Neo4j, HyperGraphDB and InfoGrid seems to be the best answer. Indeed a graph stores 2 kinds of information : nodes and edges.
Each node could own several links which point to other nodes. Thanks to that a relationship could be establish between nodes. Indeed it allows us to organise them. Each node could have several properties or attributes to store data under form key / value.

A graph stores data in nodes which have properties. Relations organise data in nodes between them. We could sort data in list, tree or in a more free way depending on our needs. They could have several directions input or output or both of them. And nodes could have properties to help us with organisation.
Nodes are organised by relations which have properties too. It exists two ways to recover data in a graph. The first one with crossing, the other one with index. For the first one we have to "across"

graph nodes after nodes, with an algorithm. We could define several options for the crossing : retrieve node with a value of his properties, if node owns at least one output relation.
A crossing navigates in the graph with a node and identifies ways or under ways with ordered nodes according to the options.
A second way to retrieve a node or a relation with a more specific method is to use index. Thanks to that, we could recover a node directly according to the value of its properties. If we take the same example again, database could give us a according to its attribute “name”.
An index is mapped by properties of nodes and relations.
And Neo4j ?
Database allows to manage all these objects, nodes, relations and index. Thanks to algorithms, internal tools and external modules such as Apache Lucene, Cypher or Gremlin, recovering data is easier.

Comparison with competitors

Criterion	**Neo4J**	**OrientDB**	**InfoGrid**	**HyperGraphDB**
Elasticity	3*	3	2	2
Performance	3	2	3	2
Managed structures complexity	4	3	4	5
License	5	5	5	5
Average	**3.75**	**3.25**	**3.5**	**3.5**

* CA, AP on the roadmap

2.5 Security protocols

2.5.1 OAuth2

Presentation
We propose to use the standard of authentication OAuth2 as a technical specification to implement server authentication and authorisation. We choose this standard for the following reasons :
● Single management of authentication
● mastering propagation of data access rights for users.
● security : this protocol is open and specified and used by leaders in web architecture (Google, Facebook, Twitter, 37signals, ...)

● standard and extensible : works performed for this job could be used as a base for future evolutions of USN of OASIS.

Use Cases for OASIS
Authorization management.

Comparison with competitors
No competitor

2.5.2 OpenID Connect

Presentation
Simple IdentityLayer on top of OAuth 2

Use Case for OASIS
Identity Management.

Comparison with competitors

Criterion	**OpenID Connect**	**CAS**	**SAML**
Integration with OAuth 2	5	3	2
Complexity	4	5	2
Efficiency	5	4	3
Security	5	3	5
Web compliant	5	5	2
Existing Open Implementations	2	4	4
Average	**4.3**	**4**	**3**

CAS: non standard
SAML: not well suited to the web

2.6 Client Framework

2.6.1 Angular.js

Presentation
JavaScript MVC Framework for HTML5/Ajax application development.

Use Case for OASIS
Administration User Interfaces
Client Application

Comparison with competitors

Criterion	Angular.js	jquery	GWT	Extjs	YUI
Community	4	5	4	4	3
Functional Coverage	4	3	4	3	4
perenity	5	5	5	4	2
licence	5	5	5	4	5
Average	**4.5**	**4.5**	**4.5**	**3.75**	**3.5**

2.6.2 HTML5 + CSS3

Presentation
The use of HTML5 and CSS3 allows to capitalize on quick evolution from renderers browsers.Quantity of code to write on a customer side is light. An important part of RIA (Rich Internet Application) which was to implement before is now available with HTML5.

Another important point concerns new possibilities offered by HTML5 and CSS3. These technologies are made to build modern extensible and semantic web applications.

Use cases for OASIS
All the Man Machine Interface of OASIS has to be based on HTML5 standard.

Comparison with competitors
OASIS target is the web, its standard is HTML, and the future is its fifth version.

2.7 Middleware components

2.7.1 HornetQ

Presentation
Java event bus embeded or standalone, clusterable providing a publish/subscribe mode.

Use Case for OASIS
Event bus

Comparison with competitors

Criterion	HornetQ	RabbitMQ	ZeroMQ
Complexity (simple is best)	5	3	4
Performances	4	4	5
Integration	5	4	4
Average	**4.66**	**3.66**	**4.33**

RabbitMQ : non intégrable, complexité de mise en oeuvre
ZeroMQ : non intégrable, disparité technologique

2.7.2 Logstash

Presentation

Tool aggregation of events extracted from system logs. Data compatible with Kibana which allows to benefit from dashboards on the events to be monitored. Ability to add one's own events, to centralize all the metrics (businesses and techniques) and to retrieve KPIs.

Use Case for OASIS
Log event aggregator and indexer

Comparison with competitors

Criterion	Logstash	FluentD	Flume	Scribe
Tools integration	5	4	3	3
Performances	4	4	3	5

perenity	5	4	5	3
Average	**4,75**	**4**	**3,6**	**3,6**

3 Architecture and deployment

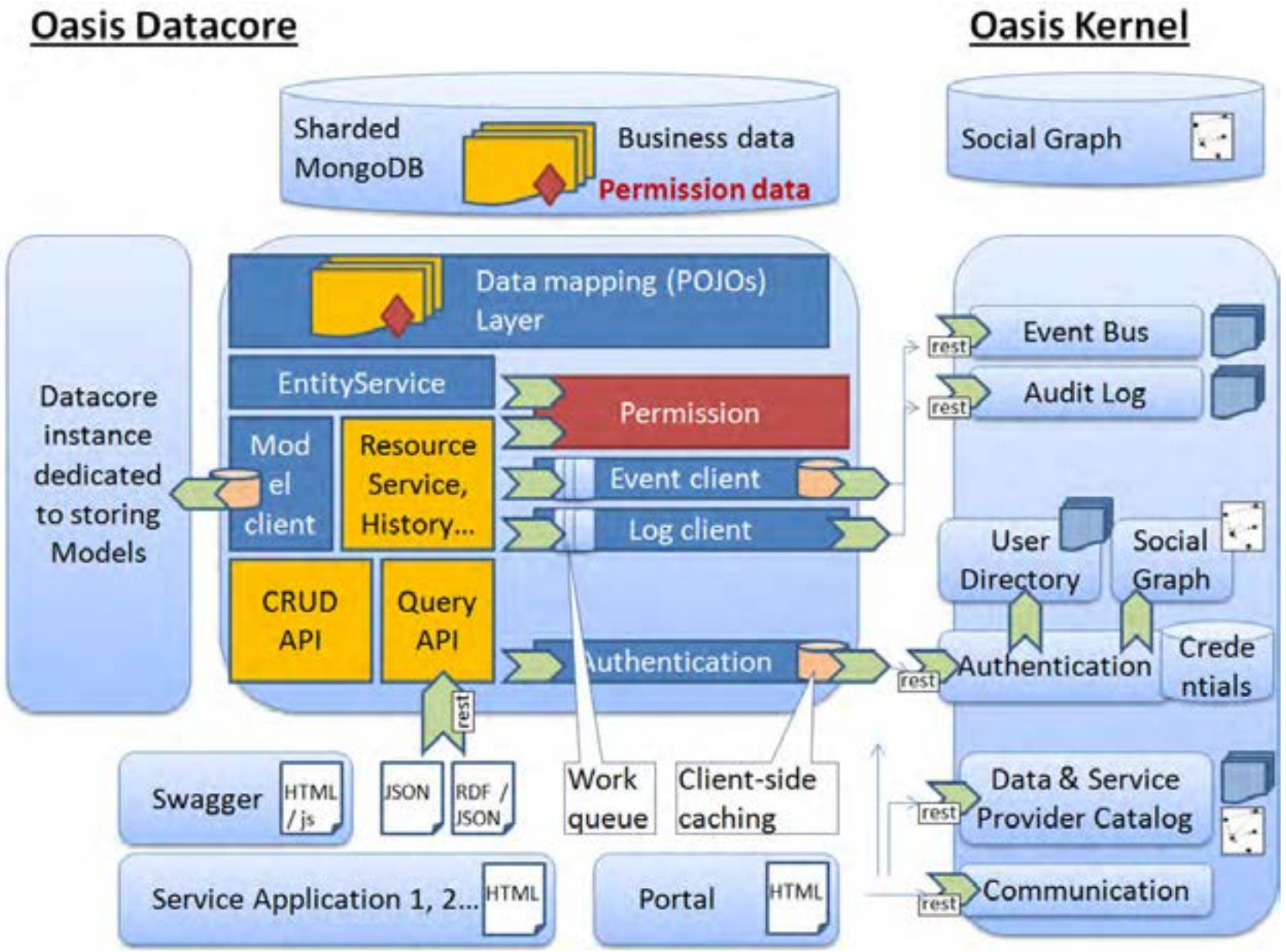
3.1 Architecture overview

The global technical architecture relies on the functional architecture described in deliverable D1.2 (the sub-systems ands the functional modules are sub-systems and technical modules)

Several criteria guide the definition of this architecture :

- We manage a federation of services, some having their own architecture constraints with different availability, security levels and response times.
- We must ensure a high availability and a good elasticity of the datacores, the kernel and the portal.

We then retained a REST architecture which allows a weak coupling between the sub-systems and which offers a high independency between the sub-systems.



Global architecture of the platform

3.2 Architecture deployment

3.2.1 Deployment architecture components

The OASIS platform's own deployment architecture is a typical Java over MongoDB one. Its atoms are Virtual Machine (VM) nodes of two different types : Java nodes, and MongoDB nodes. This is because that though software is computing-bound while database is IO-bound, both do consume RAM, especially the latter, therefore software has to be on different VMs than data. This way, within the platform, MongoDB and Java nodes will be able to scale up separately (see below).

The overall OASIS architecture, as already said, is Web-oriented, meaning decoupled through REST APIs. Having an overall Web-oriented architecture means that actual individual OASIS endpoints (Kernel and Datacore) will have to be used by clients (service providers but also OASIS Portal) as much as possible instead of using a single generic one, in order to foster client-side caching. Besides that, DNS routing may be used to provide a single entry point, typically to support Datacore URIs. Note that if it is not enough to provide a single URL for all Java nodes serving the same MongoDB shard with High Availability, we will use HA Proxy in front.

3.2.2 MongoDB deployment architecture

At the bottom layer, OASIS MongoDB storage is replicated for resilience and high availability, and sharded to allow to scale writes. More precisely, overall MongoDB architecture is made of several MongoDB shards, each serving different data, and each being a replicated MongoDB cluster, that is a set of MongoDB VM nodes serving the same data with one of them being the primary one. Application servers are stateless (or as if, by using web caches), and they must be clustered to provide high availability. Using the Java MongoDB driver (or its Spring wrapping layer) and mongos unix services make the link between both transparent. Indeed, MongoDB provides optimal and transparent routing from its client to where it actually stores data, through its mongos routing process, meaning that each Datacore application server should have a mongos process. Finally, where data is actually stored, i.e. in which shard, depends on the sharding strategy (i.e. the shard key) ; it is dealt later in its own chapter, but for now suffice it to say that data should be classically stored near code nodes that uses it, that is near code nodes used by close users to read said data, assuming OASIS use is read-heavy.

3.2.3 high availability and backup

As hinted at in the global architecture, high availability is achieved :

- at data level, by live replication and sharding of data.
- at web service level, by putting in front of each shard its own load balanced cluster of stateless application server nodes.

Backup is achieved by live replication of data. Therefore being able to replace faulty data VMs in a

timely manner is paramount. IT Governance will design an appropriate answer, which includes SLA and KPIs such as Time to repair, but possibly also requiring at least 2 additional replica data VM rather than a single one in addition to the master one.

However, some backup is required to get a copy of all or some data to feed other environments with meaningful business data. Indeed, especially with a generic business (meta)model such as Datacore's, the only meaningful enough data is the production data (or at least the end user staging environment data, but this one should also be fed from production). Whether it is done by snapshotting at VM level (which MongoDB allows) and copying all production VMs to a new environment, or by database commands (ex. MongoDB slave mechanism, rather than mongodump), or application utilities that ease export (because those procedures require additional operations, such as temporarily disabling MongoDB's own load balancer ; useful ones are Open Source, ex. Wordnik's), or several of those methods at once, will be studied by infrastucture and application providers together. The chosen design will then be put in place, documented and up to some point automated as well.

Other maintenance processes (MongoDB compaction...) will be documented and, up to some point to be defined in pilot phase, automated as well. Pieces of advices and operational best practices that will have to be followed have already been gathered.

3.2.4 Monitoring-driven scalability

Scaling up can't be automated out of the box, because it is not known ex. which type will encounter the most load, or even which technical or business KPI(s) will best tell when which kind of scaling up is required. This will be learned step by step along the project's deployment and increasing use, thanks to knowledge of infrastructure and application behaviour by operators and resp. developers. But the best information must be easily and timely available to infrastructure operators, to help them make the right decision about when and how to scale up.

This is why monitoring is paramount to achieve elasticity : because it crucially allows to know when OASIS has to be scaled up. Indeed, the key to OASIS scalability is to have all information allowing to decide when (anticipate) and how (ex. add a Java node rather than a Mongo node) to scale up.

As a side note, let's say that automated scalability (such a script reacting on such alert and automatically adding a VM) would be possible, but won't work all the time. Indeed, it can't handle all scalability problems, at least not those who are rooted in business-level causes, such as one application having a runaway success.

3.2.5 Monitored Key Performance Indicators (KPIs)

Monitored Key Performance Indicators (KPIs) break up in two categories :

- IT Governance (D3.4 deliverable) is expected to design the monitoring architecture to gather the technical infrastructure-related KPIs among those listed in D5.1b.
- Application-level KPIs (such as number of users, number of piece of Data of each type and overall average, biggest piece of Data & type & user...) must be provided by application providers, that is Open Wide (Datacore) and Atol (Kernel Service, Social Graph).

All KPIs must be gathered in a single monitoring UI, whose (infrastructure and application) architecture is to be designed together by all KPI providers (that is IT Governance, pilot sites and service providers), so that it allows easily (but not necessarily graphically, ex. scripting) to set up

alerts, to highlight some KPIs rather than others and to add new ones (assuming they are already provided by infrastructure or application).

3.2.6 Gathering application-level Key Performance Indicators (KPIs)

To gather application-level Key Performance Indicators about its operations, all OASIS comoponents use the Kernel Audit log architecture. In practical terms, all Datacore components emit to the Kernel Audit Log endpoint logging information about their operation, comprising operation details and especially tags that tell how it relates to business concepts (to a given Resource, Model type, business domain, application, user account...). Such tags allow, using the Kernel Audit log architecture (Logstash etc.), to filter, assemble and analyze operation logs in indicators that are meaningful to scalability, for instance that show that a given business domain is enjoying dangerously increasing success and that actions should be taken to make its resources scale up (such as sharding its Model's MongoDB collections). Those indicators are finally monitored through graph or dashboard visualization and alerts, to which operators should react accordingly.

4 Datacore implementation

4.1 Reminder - Business of Datacore

OASIS Datacore aims at providing a Big Data-level amount of shared data to access and query for any possible use. But for this to happen, this data must have been contributed first, meaning that **contributing data** must also scale to Big Data level, and that contributing data must be made as easy, efficient and painless as possible. Far from being a mere Open Data repository, OASIS Datacore provides collaboration across business and services on data, and that's the critical added value (besides confidentiality and other additional feature) that drives Data providers to share their data. So **cross-business collaborative linked data** is the first (not the most important, but the most urgent) business of OASIS Datacore.
Therefore, the default OASIS Datacore implementation should be **efficient at collaboration on linked data (as much as "native" business applications), while being efficient on common (according to expected use), simple queries**. All other queries (up to RDF-like free form queries) should still be allowed, at worse by doing several successive such simple queries. Note that other data consumption needs (from graph-like traversals rather than query on criteria, to Map/Reduce and other data aggregation and processing frameworks) are out of scope here and have to be optimized by deploying dedicated solutions on top.

4.2 Design of REST Data Model for Cross-business collaborative Linked Data

On one side of the spectrum, semantic modeling such as using **RDF**, where each attribute or relation is a line in the same database table for all data**,** is optimally flexible and allows to model any kind of data, support any kind of model evolution, but at heavy costs in consistency management and performance.
On the other side of the spectrum, traditional, dedicated business applications, such as those of OASIS service providers, have models that follow closely their business, where each business concept is modelled by a separate class or object type and stored in a separate database table. It makes them very efficient at reading and writing, and while making it evolve is costlier, at least it can be done in a consistent and maintainable way.
The collaborative linked data business of OASIS sits **somewhere in the middle** : not as free-form, all-data-is-equal as RDF, but still more flexible than dedicated business application models, in order to be able to serve several business applications that are different but happen to talk about a few same things.

In order to achieve this, and based on the previous technology analysis, the Datacore REST API model has to be:

* **RDF compatible (Optimal flexibility of data modeling and cross-business linking)**
to allow optimal data modeling flexibility (i.e. be able to model any business and support any

change), data compatibility (i.e. data homogenization)

- thanks to an "almost" **JSON-LD** representation which is the most JSON-friendly alternative among natively RDF-compatible representations (and increasingly becoming the norm, as shows it having become a W3C Recommendation on January the 16th, 2014[4]),

- **W3C LDP** (Linked Data Platform)-like REST query filters, which is the most REST-friendly alternative among RDF-compatible query languages. For instance, to get all cities of France sorted by name with a population of more than 500000 (with a default maximum of 50 results) :

http://data.oasis-eu.org/city?inCountry=http://data.oasis-eu.org/country/France&name=+&population=>500000

- but also a **"mini-SPARQL"** built on these last ones.

* But at the same time, to optimally support the collaborative linked data business, it has to handle the following **differences from a native RDF** implementation :

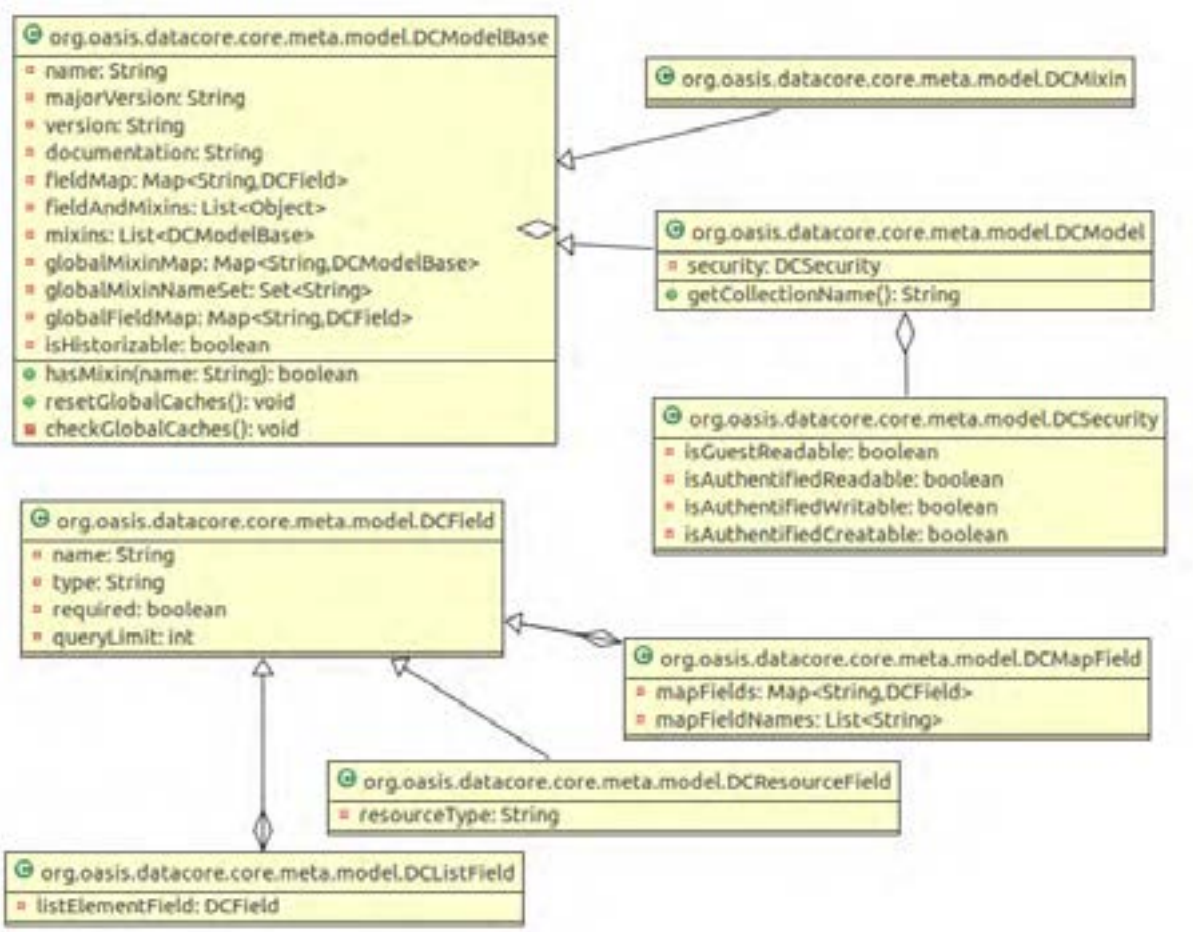
- all data are not equal : each data Resource has **a primary collaboration "Model" type** (the first "rdf:type" of JSON-LD resources) which corresponds to its collaborative data business use. Such a "Model" type is born each time a community of service providers agrees on collaborating together on a given kind of data, and will follow the specific governance rules decided by this community. If a new service provider wants to contribute to the same kind of data, either it subscribes to the existing community and rules, possibly even only partly (such as its contributions having to undergo approval before being accepted), or manages to get the community to change them, or it has to do it all separately, which OASIS discourages though.

- granularity of the RDF triple is too fine to be efficient, easily handled, modeled or understood by service providers. Therefore **an intermediate lightweight "Mixin" type granularity** is promoted, similar to the notions of Aspect, Trait and Facet. A Mixin type aggregates triples that talk about the same business domain and have the same lifecycle (authors and owners, associated business behaviour, overall governance). It roughly corresponds to a an RDF predicate's namespace. Mixin types allow **multiple typing, shared types (such as Address), data layers** (allowing different source to manage different fields).

- allow complex property types : lists and maps (allowing all kind of complex data, including internationalized text and XML ; which RDF actually allows through XML properties). This comes in addition to regular primitive types boolean, int, float, long, double, date and string / text (including to store geolocalization information using textual format such as Well-Known Text (WKT)), and to resource (embedded or reference, including cross container ; allowing classification in categories...) which is inspired by RDF.

This begets
- the **Datacore Resource REST API**, its **"DCResource"** Java model (see next chapter)
- its **ResourceService** CRUD management service and **LdpEntityQueryService** native query service (see chapters)
- and the **"DCModel"** Java description of available properties etc. on a Resource according to its type :

[4] http://www.w3.org/TR/2014/REC-json-ld-20140116/



4.3 Datacore REST API Java client

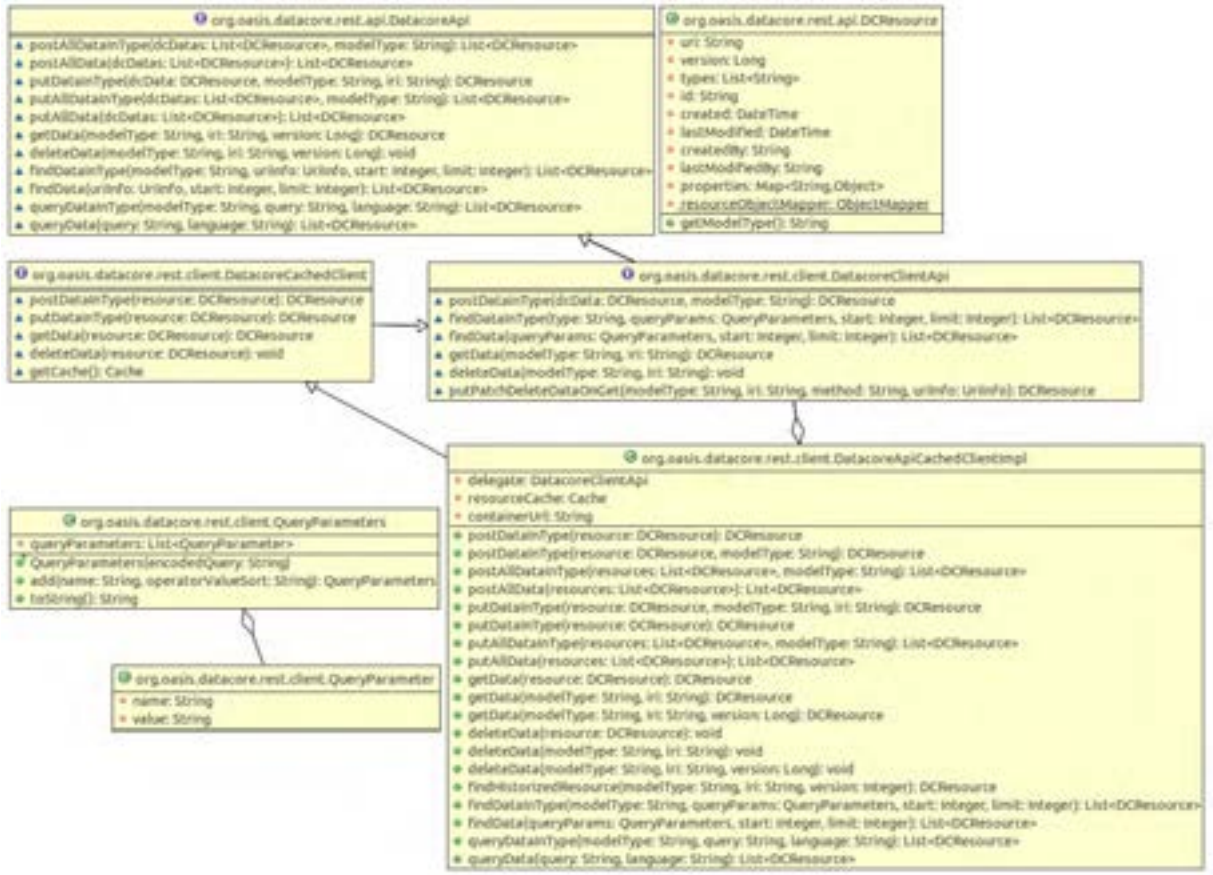
To show client developers how Datacore REST API clients should be best developed to take advantage of all its features, but also to help testing those features and other ones, an exemplary Java client has been developed for the Datacore REST API.

It offers various helpers methods, and allows to take advantage of all REST API features, even those that are not available directly in a JAXRS proxy client (such as POSTing a single Resource directly instead as the single item of a JSON Resource list / array).

It offers **client-side caching** by storing retrieved resources in a Spring Cache-powered cache, which then allows it to transparently handle HTTP 304 (Not Modified) responses from the server by returning looking up resources in said cache instead.
It also offers **Spring Security-powered security** : the appropriate Bearer (OAuth authentication, production & integration) or Basic (dev mode only) HTTP authentication headers are sent, according to the current thread's logged in Spring Security user token.

This Java client uses a JAXRS-defined CXF Proxy to talk to the server using the same business Java interface. In fact, advanced JAXRS features, client-side caching and Spring Security-powered security also use CXF Interceptors to achieve their aims, for instance to add HTTP headers.

Here is the client architecture and its point of view on the Datacore API.



4.4 Datacore server application architecture

To address efficiently the collaborative linked data business, based on the previous analysis of requirements and technologies, we make the following architecture and technology choices for the Big-Data-level scalable default Datacore implementation (merely named “Datacore” from now on) :
- stored in **MongoDB** (see motivation in following chapter)
- implemented in **Java** for its robustness, scalability and maintainability, using the **Spring** Framework to speed as much as possible developments, for its wide coverage and high flexibility (meaning that better solutions can be chosen on a case-by-case basis and coexist later), and for Open Wide’s advanced expertise on it
- using its **JAXRS** standard to model the REST API, helped by **Jackson** annotations (which are the obvious choice to to control its JSON formatting), and the **Swagger** library to document it (the best of the market, with online testing playground and client library generation)

- and the **Apache CXF** service engine to expose it, for its integration with Spring, performance and flexibility.

This is the OASIS Datacore stack.

4.5 Datacore Resource persistence

4.5.1 Datacore storage technology choice

Neo4j (and graph databases) is not suited to Datacore needs because :

Overall, OASIS needs are rather queries than graph traversal
OASIS graph traversals are done by hopping from one business type to another through URI references and are therefore very well directed, oriented and guided. OASIS business types are at the opposite of generic nodes linked by generic links. Let's give the usual simple example : OASIS is far less interested in finding any connection between Obama (who's born in Honolulu) and Hawaii (where is Honolulu), than from Obama looking up information about where he's born (Honolulu) and from there information about where it is (Hawaii).
Main use cases have been tested in a prototype (*).
- Neo4j has transactions (will be mandatory for read in 2.0 !!), but they are logically mutually exclusive with sharding and write scaling
, which are required for OASIS to go Big Data-level on data contributions.
- there are indexes, but no compound ones (on group of fields). Worse, they are not automated (have to be filled explicitly) before 2.0 beta (**). Both make it bad at queries.
- property types : no map & list (less business domain modeling flexibility, ex. Internationalized text or XML). As an aside date must be indexed as a long
.
- there are no way to define node types, before 2.0 beta (**) which allows free-form only “labels”. Sure RDF is just as free-form, but from the point of view of service providers there will have to be very efficient ways of separating their data from others' and it would have been better to have a builtin solution.
- bad at analytics
(which could have been another way beyond queries to get knowledge from data)
- ... even if there are weighted relationships, (XA) transactions, GIS
(*)
(**) 2.0 still beta at the time of this writing and has a regularly breaking API

Relational Database Management Systems a.k.a. SQL databases (including PostgreSQL) on the whole are not suited to Datacore needs because :
- can't scale to terabyte without sharding (which is a complex, heavy and error-prone operation unless automated) and forbidding transaction locks. Wikipedia does both using MySQL, but it doesn't need any data consistency, given its very simple business domain model (pages).
- no complex / content-oriented properties (less business domain modeling flexibility, ex. Internationalized text or XML)
- setting up mongodb tables and indexes on demand is far easier and less costly than in RDBMS.

So **MongoDB is chosen** as Datacore storage because :
- can scale to Big Data level, in a heavily automated manner (using sharding), including on writes (using optimistic locking rather than server-side locks)
- efficient query & index with the most featureful API

- has the most flexible storage schema : tree-like JSON document, with Javascript types : Map (Object), List (Array), date (ISODate)
- has even built-in computed analytics (Map/Reduce etc.)

4.5.2 Storing Datacore API model

In order to choose the best way to store it, here's how the Datacore API Model **differs** from the traditional RDF approach : this model would be similar to native RDF storages (i.e. store triples) if
- native RDF storages (i.e. triplestores) store all Resource properties in the same collection / table / indexed storage. But here we have a clear business-level partitioning, which is along business collaborations, i.e. Model types. So the right way is to store Resources of each Model type in separate collections, each dedicated to a given "collaborative data" business in order to allow per-business optimization, such as queries using business-specific indexes.
- native RDF storages (i.e. triplestores) store all Resource properties in a "flat" manner. This is actually the case most of the time. However the model offers more freedom on this side by also allowing trees, which is useful to store any complex data values, such as first and foremost internationalized text and XML. For instance, an internationalized text value is a "map" of language to translations.

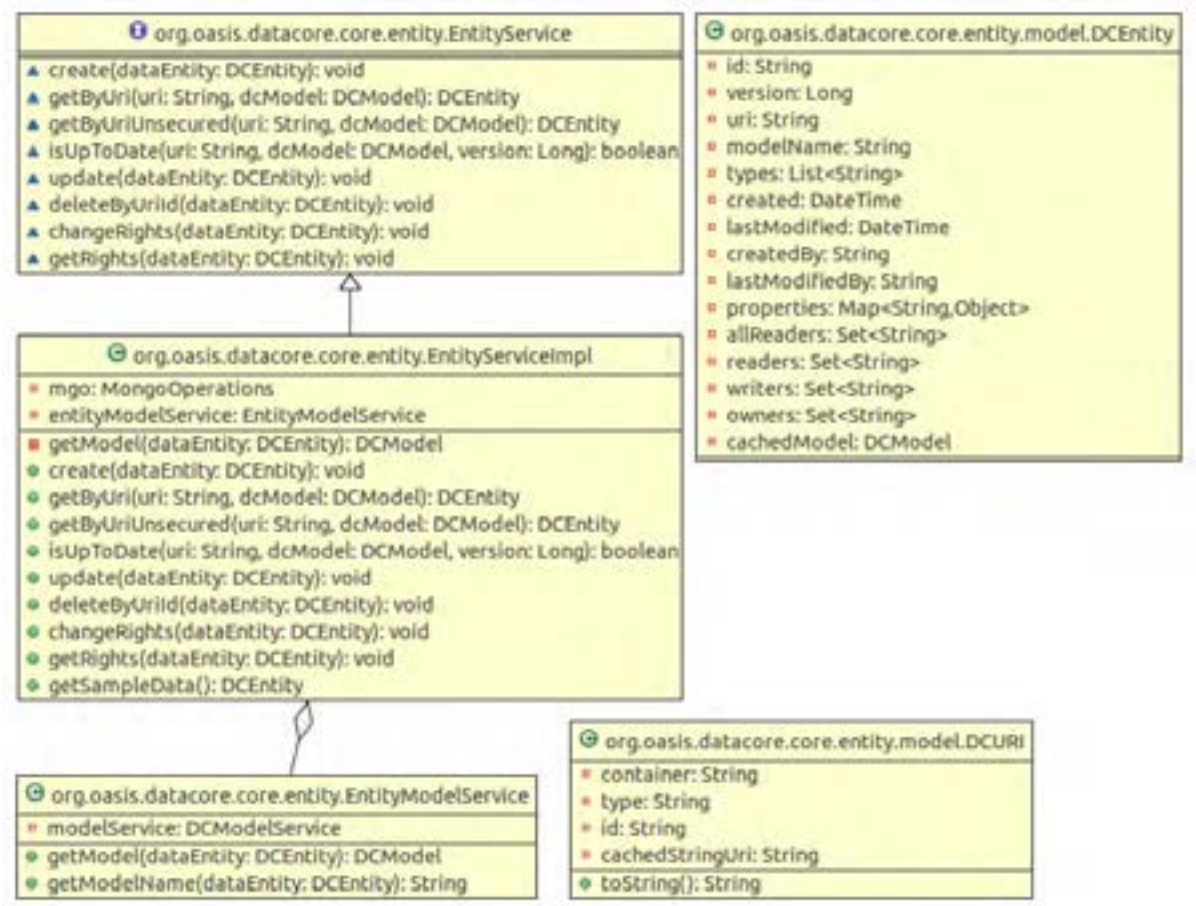
This allows the model to fullfill its requirements of not being worse performance-wise on free form queries on the long term, but to have performance similar to non-OASIS business applications that collaborate together on the same "common good" data.

This begets the **"DCEntity"** persistence model in MongoDB of Datacore Resources, and its **EntityService** management service.

In concrete terms, a DCEntity contains all of its Model-typed Resource's properties (business fields, but also technical ones such as version), but goes beyond those to store also :
- properties of Mixin-typed Resources, with Mixin type acting as another business view on the same Entity
- additional information required by other features, such as Resource-level permissions for authorization (see chapter)
- in addition, MongoDB indexes are created and managed on queriable technical properties (uri, version, allReaders) and business properties (those featuring a queryLimit > 0).

Spring Data for MongoDB is used to persist DCEntity, as well as to version (ensuring optimistic locking) and audit (retrieve and store user and date of creation and last modification) it.

Here is the architecture of DCEntity management.



4.6 Resource CRUD Service

The ResourceService Java component implements Datacore Resources creation, individual read, update and deletion (CRUD). It provides those features to the Datacore Resource REST API, which exposes them to remote REST clients. It also provides them internally to other Datacore components (if only for unit testing), along with helper methods to easily build URIs and Resources programmatically.

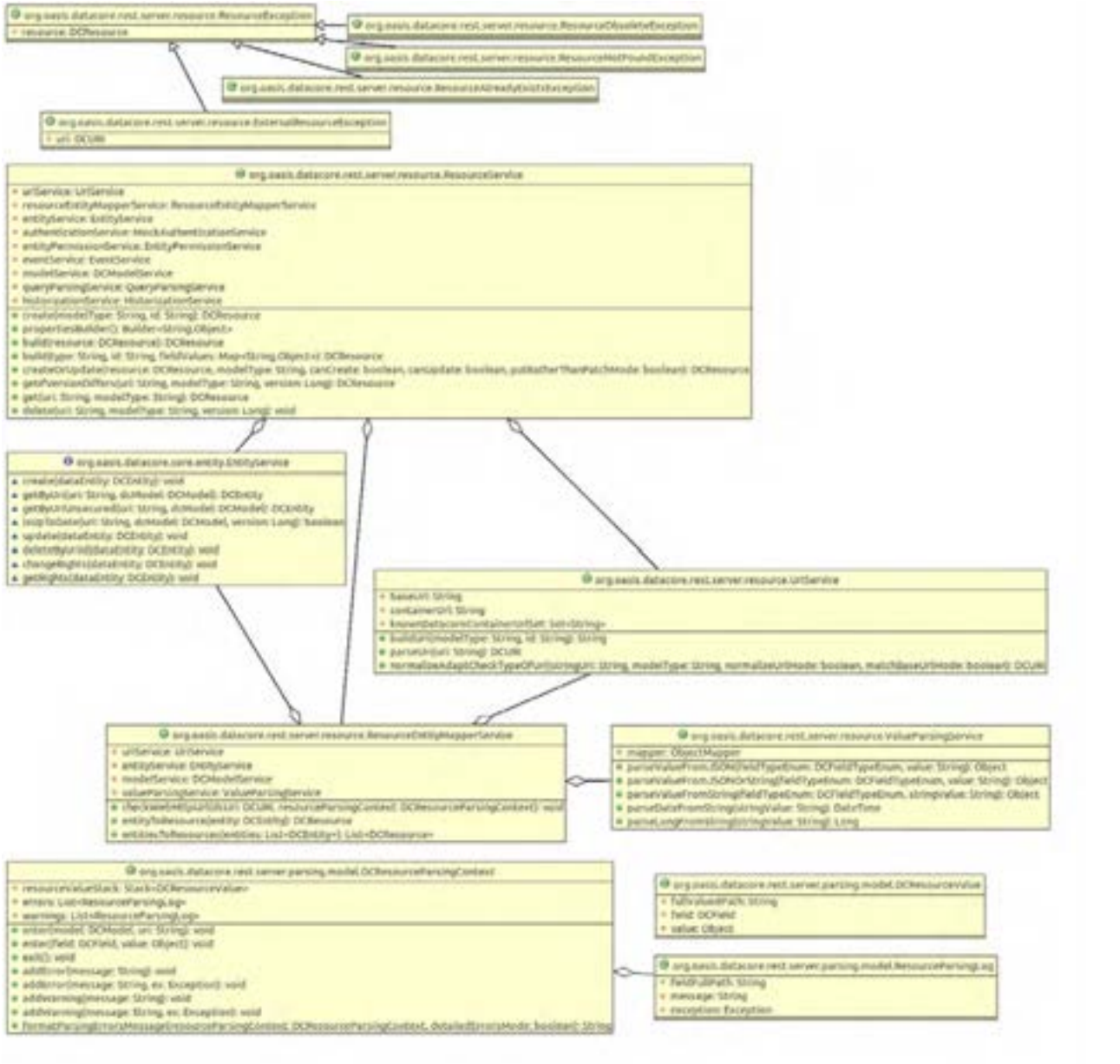
ResourceService uses MongoDB-backed EntityService to implement **individual read and deletion** of Resources, both using Resource version. Read uses version to implement client-side caching : providing an up-to-date version doesn't return the version but only that it's still up-to-date. Deletion uses version to implement optimistic locking : deletion is successful only if the up-to-date version is provided, and otherwise it doesn't do anything (which is easier and proper per REST principles).

To **create or update Resources**, the Resource Service :
- uses EntityService to retrieve corresponding existing data or update it, with version-based optimistic locking. Note that (most) EntityService methods are protected against access by a user without sufficient rights, which is how authorization-level security is applied (see chapter).
- uses ResourceEntityMapperService to check that uploaded new or changed Resources are compliant with their Model type (DCModel provided by ModelService). This is done in Visitor

pattern-style, with state of checking visit being stored in a dedicated ResourceParsingContext, allowing to help client developers by returning all errors and warnings at once if check fails.
- uses HistoryService to save Resource to History (see chapter)
- uses EventService to emit a local Datacore event, that may be listened to and reacted on by any plugged EventListener-implementing interface. This allows behaviour such as computing some fields out of others, consistency checks, sending distributed events to the Kernel EventBus API...

Resource consistency
- is implemented for a single Resource by having a “unique” index on the Resource URI (which is its identifier) and by using version-based optimistic locking.
- beyond, must be implemented by client applications. Usually, using the workflow / rights token collaboration model (see Mediation chapter) is enough. The generic advanced solution is for them to manage their own transaction locks (see perspectives), but it is harder and must be used parcimoniously (REST principles such as optimistic locking should be preferred) to avoid breaking up scalability of collaboration.

Here’s the architecture of the ResourceService component and its dependencies.



4.7 Resource Query

Resource query is available either through the native LDP (Linked Data Platform) criteria-like API and service, or through query languages supported by plugged in Query Engines.

The LdpEntityQueryService Java component implements Datacore query on top of MongoDB using LDP (Linked Data Platform)-style criteria on Model fields. It provides those features to the Datacore REST API, which exposes them to remote REST clients. It also provides them internally to other Datacore components, and especially Query Engines built on top of it.

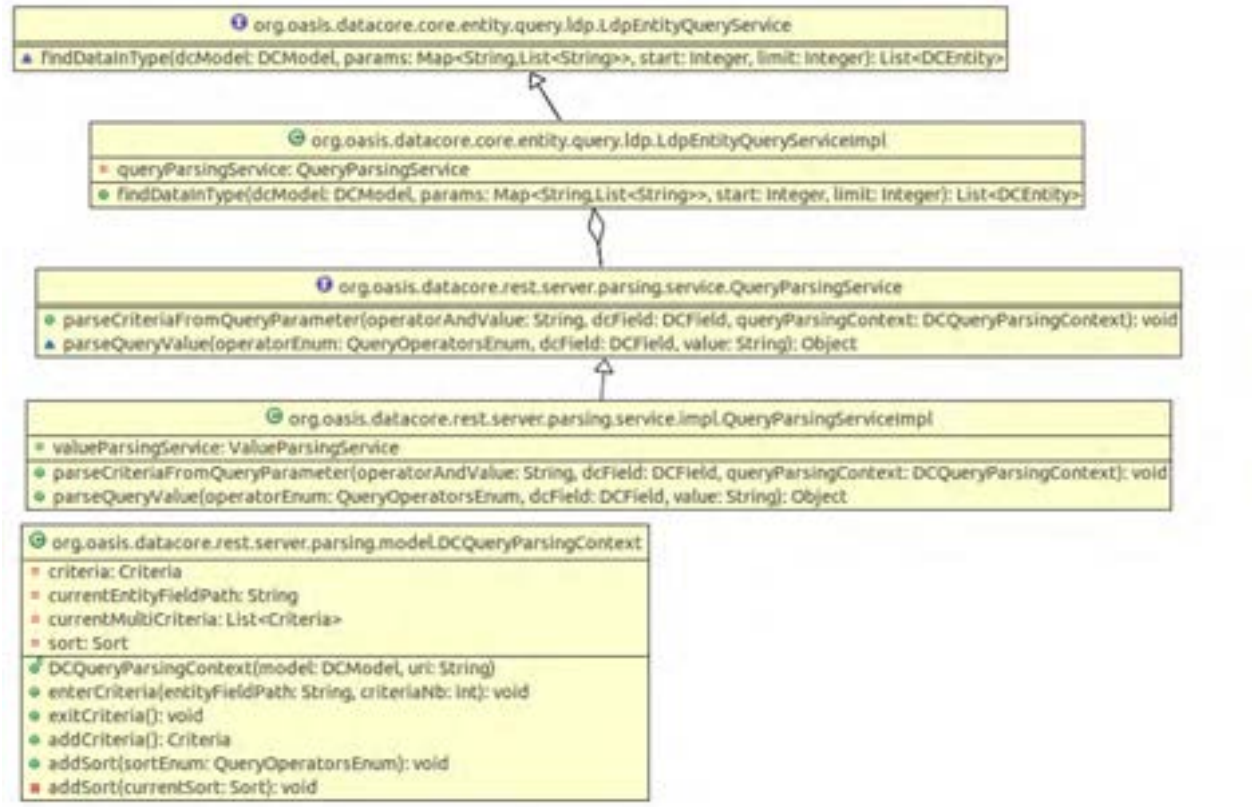
The LdpEntityQueryService implementation is built on top of MongoDB using the Spring Data MongoDB Criteria query API, and provides the following features :
- multi-valued criteria on fields at any place in Resource tree and of any type (including dates, lists, Resource references),
- criteria parsing (several representations allowed: Java-like, SQL-like...),
- can be restricted to fields that are indexed (to avoid performance drop), which are those whose DCField definition has a queryLimit > 0 (which could also be used as pagination maximum).
- pagination. If there is none, or if the client asks for too many results, it reverts to a default value.
- ordering

Here are all accepted operators, in their logical/XML/MongoDB/Java forms if any (else -) :

=/-/-/==
>/>/$gt/- (ex. 3)
=/>=/$gte/-
<=/<=/$lte/-
<>/<>/$ne/!=
-/-/$in/- (in JSON list, ex. [0,1])
-/-/$nin/- (not in JSON list)
-/-/$regex/- (Perl's ex. /Lond.*n/i or only Lond.*n)
-/-/$exists/- (field exists, ex. name)

list operators :
-/-/$all/-
-/-/$elemMatch/-
-/-/$size/-

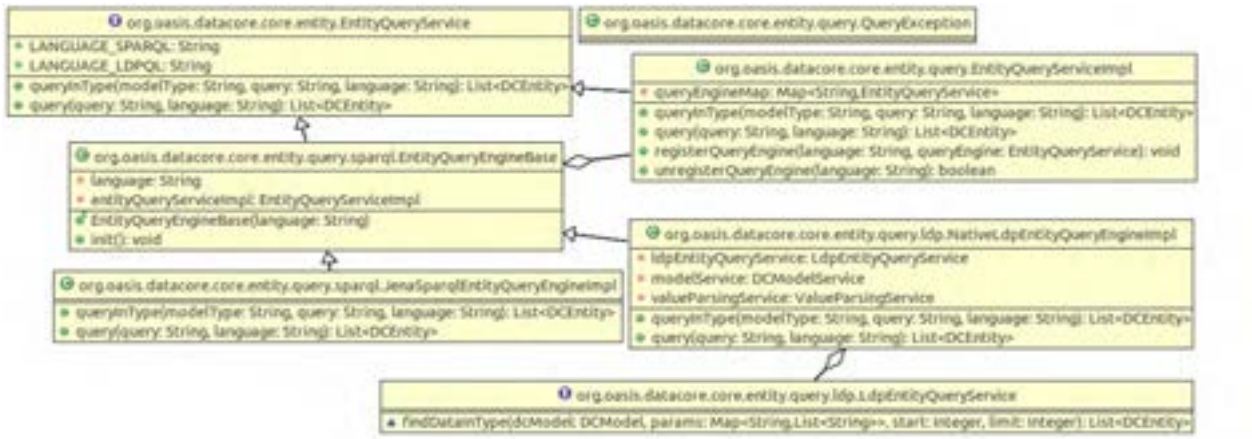
Here is its whole architecture.



In addition to the Mongo-based LDP-style criteria API and service, the Datacore provides a query language API whose Java service is EntityQueryService. It can be plugged by QueryEngine implementations in order to support various other query languages. The ones currently available are :
- NativeLdpEntityQueryEngineImpl
- JenaSparqlEntityQueryEngineImpl (see SPARQL Facade chapter)

Both are built on top of the Mongo-based LdpEntityQueryService.

Here is this additional (non-native) Query component architecture :



4.8 Security

4.8.1 Authentication

In OASIS authentication is made possible with OAuth2.
As said in Wikipedia : "OAuth2 provides a method for clients to access server resources on behalf of a resource owner".
In OAuth2 there are differents entities that are working together :

- Authorization Server
- Resource Server
- Client

To sum up the workflow between these three entities, let's say that the Client asks the AS (Authorization Server) to get access to a data stored in the RS (Resource Server). If the access is granted by the AS, the client receives a code that he next redeems against an access token, which is like a temporary access ticket, and he can then access the data on the RS by putting the token in the request against the RS.
In this workflow, the Datacore is the Resouce Server. It's where data are stored and protected.

We use **Spring Security to implement OAuth2** authorization in the Datacore.
Spring Security provides library that implements the mecanism to authenticate (OAuth, LDAP, Basic HTTP) the user and get informations about him (authorities, roles...).
We can handle guest mode and also Basic HTTP Authentication for development purposes only (so-called "dev mode" and can be triggered manually).

In the Datacore case, we are not in the standard implementation of OAuth2 using Spring Security, because the Resource Server and the Authorization Server do not share the same Web Application.
Indeed, in a standard implementation, the AS and RS share a TokenStore where are stored all the tokens. In our implementation, AS and RS are separated and are located on 2 differents webserver (not even on the same physical machine).
Now, the only way we can grant access to the data is by validating the token against an introspection endpoint. An introspection endpoint is a webservice which gives information about a token, such as its user identifier ("sub"). This introspection endpoint is hosted on the same web application as the AS, which means it can access tokens locally.

So in Datacore the basic OAuth2 flow is :

- Receive request
- Get the access token in the header
- Validate the access token against the introspection endpoint
- Grant or revoke access to the data

The access token is in the header part of the request, as following :

```
Authorization: Bearer <access_token>
```

In this part I'll explain each part of the spring configuration, which is done in an XML file.

```
<oauth:resource-server
        id="oauth2ProviderFilter"
        resource-id="oasis-datacore"
```

```
        token-services-ref="tokenServices"
/>
```

This is the definition of the resource server by using the OAuth2 Spring Security library.
The "token-services" parameter is where are defined which services are used to validate our token.

```
<bean id="tokenServices"
        class="org.oasis.datacore.oauth.RemoteTokenServices">
        <property
                name="checkTokenEndpointUrl"
                value="<introspection_endpoint_URL>"/>
    <property name="clientId" value="<datacore_provider_ID>" />
    <property name="clientSecret"
                value="<datacore_provider_password>" />
</bean>
```

This is the definition of the TokenServices.
This class intends to verify the validity of a token, get all scopes and groups associated to it.
At the end, if the token is checked as valid, the user (Spring Security User) is filled with groups associated to the token, as well as the access / Bearer token string itself.

```
<security:http pattern="/dc/**" entry-point-ref="oauthAuthenticationEntryPoint" access-decision-manager-
ref="accessDecisionManager" authentication-manager-ref="authenticationManager" create-session="never">
        <security:intercept-url pattern="/**" access="SCOPE_DATACORE" />
        <security:custom-filter ref="oauth2ProviderFilter" before="PRE_AUTH_FILTER" />
        <security:access-denied-handler ref="oauthAccessDeniedHandler" />
</security:http>
```

This is the Spring Security HTTP filter deployment configuration.
We ask Spring Security to intercept all traffic on /dc/* and make it go trought the entire security chain before allowing access to the data.

In our case, we define the following components :

- An authentication entry point, wich is used when an un-authenticated user wants to access a protected resource. Spring will inform him that it must authenticate and will either return an HTTP Status code or redirect him to a login form.
- An access decision manager, wich is the entity at the end of the chain that will decide if the user can access the data or not (we will see the definition later in this text).
- An authentication manager which validates the authentication and fills the Spring Security Context with an authentication object. This context **materializes the fact that the user has been authentified** and is attached to the current thread. It can be accessed at any further time during handling of the current request to get information about the current authentified user, such as **its scopes, groups and overall authorities** (property “sub_groups”, see next chapter).
- create-session=never says that we want the user to send the token in each request against the Datacore, and go trough the whole authentication process. The alternative is to ask him to only provide a session cookie if the authorization has already been granted, which may be used to improve scalability, by skipping the whole authentication chain. A better intermediate alternative is to not create sessions, but cache recently validated token and user information, and reuse it when an HTTP request bears a cached access / Bearer token rather than call the Kernel Service Introspection endpoint to validate it.
- intercept-url pattern="/**" access="SCOPE_DATACORE, IS_AUTHENTICATED_FULLY"gives information to Spring Security about the different authorities that are granted access. In the Datacore case, usersmust logically be a scope named "datacore" linked to ytheirour token to get access.

- custom-filter ref="oauth2ProviderFilter" before="PRE_AUTH_FILTER" is where Spring Security is told that it will have to use the OAuth2 filter at the "PRE_AUTH_FILTER" position in the chain (it's at the start).
- access-denied-handler ref="oauthAccessDeniedHandler" is used by Spring Security to inform a user that his authentication information isn't right (wrong token, expired token...)

```
<security:authentication-manager id="authenticationManager">
        <security:authentication-provider user-service-ref="clientDetailsUserService" />
</security:authentication-manager>
```

This is the authentication manager definition. It tells to use a custom user service (used to load all of a user's information)

```
<bean id="clientDetailsUserService"
        class="org.springframework.security.oauth2.provider.client.ClientDetailsUserDetailsService
">
        <constructor-arg ref="clientDetailsService" />
</bean>
```

This custom UserService is the OAuth2 implementation of a UserService.

```
<bean id="accessDecisionManager"
class="org.springframework.security.access.vote.UnanimousBased">
        <constructor-arg>
                <list>
                        <bean
class="org.springframework.security.oauth2.provider.vote.ScopeVoter" />
                </list>
        </constructor-arg>
</bean>
```

This is the definition of the accessDecisionManager. This brick is essential to the good behavior of Spring Security. It will look at the intercept-url access and check if requirements are met. Here it will check that the user has the scope "datacore" on the authenticated user, and if not will deny access.

Finally, let's say that access / Bearer token expiration and refreshing them is not the job of the Datacore but of client applications, and that anonymous access is forbidden but dedicated system accounts can be provided per service provider / application to handle its role as "guest" accounts.

4.8.2 Authorization

4.8.2.1 Authorization model

As we've seen in the last part, we authenticate users using the Spring Security framework. This framework can also be used to manage authorization around the application. The main difference between authentication and authorization is that a user can be logged in the datacore (authentication) but still not be able to access a given piece of data (authorization).

In OASIS, there are many business level authorizations (organizations, Social Graph groups, scopes, user itself), so we had to translate them into a simple but powerful way to manage **permissions** within the Datacore.

As said earlier, Spring Security can help us to grant authorization to users on data.
We chose the **ACL (Access Control List)** way to do permissions.
The basic flow of a request against the datacore is :

- Receive request
- Authentication check (access granted or not)
- Authorization check (access granted or not)
- do operation on Data

In the datacore we implements 2 level of permissions :

- Model (business, high level)
- Entity (technical, low level)

These permissions are stored in the same object : the User in the Spring Security Context (into 3 differents list for the business level permissions, and into a unique one for the technical permissions).

To help understanding Datacore principes, let's say that a DCModel is an equivalent to a table into a database (model), while a DCEntity is an equivalent to a row into a table (data).

4.8.2.2 Business-level permissions

Here we introduce the permissions on a business level, on the DCModel.
In the previous, authentication-themed chapter, we've been talking about filling the user informations in the Spring Security Context with groups/authorities/token.
In order to fill this Security Context with rights information, we had to introduce a formalism in the group names (these are the groups in the sub_groups array that we get when we ask informations about a token against the introspection point) :

- Model type Resouce Admin : rom_<model_type_name>
- Model type Admin : mo_<model_type_name>
- Model type Resource Creator : rcm_<model_type_name>

For instance, let's say there is a user named Bob who is requesting the datacore with a token ABCD.
When the Datacore asks information about the token ABCD, the Kernel Introspection endpoint returns an array called "sub_groups". This array contains all the groups of the user Bob, which can now be parsed them.
A user has 3 lists defined on him, one for each Model-level permission (Resource Admin, Admin, Resource Creator). In these lists, we store name of the Model type it is related to, for example :

```
sub_groups : [ "rom_sample.marka.city", "mo_sample.brand.car", "rcm_sample.marka.company" ]
user.resourceAdminForModelTypes : [ "sample.marka.city" ]
user.adminOfModelTypes : [ "sample.brand.car" ]
user.resourceCreatorForModelTypes : [ "sample.marka.company" ]
```

We now know that the user Bob is Resource Admin for the model type "sample.marka.city", Admin for the model type "sample.brand.car" and Resource Creator for "sample.marka.company".

4.8.2.3 Entity-level permissions

Let's now talk about the permissions on a technical level.
First of all, a DCEntity is the object stored in the mongo database (see chapter). This is the lowest level with the finest granularity (a DCEntity is where pieces of data are).
Storage of permissions differs for business versus technical level :

- DCModel : stored on User in security context (and not on the DCModel)
- DCEntity : stored on Entity

Technical permissions are fetched the same way as we do for the DCModel, the only thing that differs is that we don't parse them, we just store them in a different list of the User.

Our user is now having permissions, but against what do we verify them ?
Against permissions stored on the DCEntity object stored in the database.

But before checking each permissions of the user against the one on the entity we need to add permissions on the entity.

On the DCEntity the permissions are stored by using 3 list :

- readers : can read the entity
- writers : can write on the entity
- owners : can read/write/delete entity

4.8.2.4 Rights Management API

To let users manage permissions on a DCEntity, a Rights REST API has been developed that can :

- add rights on a DCEntity (union)
- remove rights on a DCEntity (difference)
- flush rights on a DCEntity (erase readers, writers, put owner = current user)
- replace rights on a DCEntity with the DCRights in payload

This API requires information modeled by a DCRights object that can be defined in JSON as :

```
{
  "readers": [ "string" ],
  "writers": [ "string" ],
  "owners": [ "string" ]
}
```

Only owners of a DCEntity can modify its permissions.

For instance, user Bob is admin of the model type “sample.marka.city” (bob.adminOfModelTypes : ["sample.brand.city"]).
He wants to add a writer permission called “sample.marka.city.writers” to a DCEntity he owns (type = sample.marka.city, iri = 1, version = 0).
This technicallymeans need adding the group "sample.marka.city.writers" to the list of writers on the DCEntity. To do that, he sends the following DCRights to the Rights API (add rights is a POST method) with the corrects parameters :

params :

- type : sample.marka.city
- iri : 1
- version : 0

payload :

```
{
    "writers": [ "sample.marka.city.writers" ],
}
```

Permissions are now stored on the entity, and every user that is a member of “sample.marka.city.writers” will be allowed to write on this DCEntity.

4.8.2.5 Enforcing rights on CRUD operations

But before letting them write the DCEntity, the Datacore must first check that their permissions match the ones on the DCEntity. Here is how the decision is taken for all CRUD operations.

We have an EntityService that is providing CRUD operations, including write.
On each method, there is an annotation telling Spring Security that before calling the method, it has to check something by calling a PermissionEvaluator, which plays the role of **point of decision** of authorization :

```
@PreAuthorize("hasPermission(#dataEntity, 'create')")
void create(DCEntity dataEntity);
```

In Spring Security, you can define a custom PermissionEvaluator that will be used to verify the permission. Datacore’s such permission evaluator is called **EntityPermissionEvaluator** and has a "hasPermission" method with this signature :

```
public boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission)
```

The first argument is the Authentication of the Spring Security context of the currently executed authentified thread. It contains the authenticated user, including its scopes, groups & overall authorities.
The second argument is the "#dataEntity" in the annotation, it corresponds to the DCEntity object passed to the create method of the EntityService.
The third argument is the permission type : create/read/write.

In the hasPermission method, permissions can be checked easily because it has at hand :

- User in the Authentication object who contains
  - 3 lists for the business level permissions (Resource Admin, Admin, Resource Creator)
  - 1 list for the technical level permissions
- The DCEntity which contains
  - a list of authorized readers
  - a list of authorized writers
  - a list of owners
- Action we want to do (create/read/write)

The hasPermission() method will at first check whether the user is an administrator by calling the isAdmin() method on the user.

Then for each action (create/read/write) it will check the business level permissions (e.g. create / user.isModelTypeResourceCreator()), and then, if not yet granted, the technical level permissions (e.g. read / match user.technicalPermissions[] with entity.readers[])

At the end the method returns a boolean : if true then the EntityService method is called, if false an error message is returned back to the user saying that this action wasn't authorized by the Datacore.

4.8.2.6 Enforcing rights on query operations

The operations that query data in the datacore (e.g. city=Paris), be it using LDP criteria (LdpEntityQueryService) or various query languages (such as SPARQL) supported by query engines are also protected according to rights. This is achieved by enforcing rights directly in LdpEntityQueryService, which is actually the only kind of query available natively in the Datacore, all others (ex. SPARQL) being built on top.

Here is how it is done. LdpEntityQueryService is not protected with annotations as EntityService because the idea is not to check rights for each Resource returned by an executed MongoDB query, which would be highly inefficient in the case of sparse rights among found Resources, such as a user wanting all documents that he can read (10 documents maybe) among a collection of billions of documents across all document management application users. So the idea is rather to inject rights checking within the query itself, as an additional query criteria : “and the allReaders field should contain one of the groups that the user bears”. This shows that the same rights configuration is used to secure queries as to secure CRUD operations, that is using the same business-level and entity-level permissions.

4.8.2.7 Conclusion

To sum up permissions :
- Business level permissions are used when a user want to do CRUD operations to all the data stored in a DCModel (e.g. resource creator on a DCModel, you can create all the data you want in that model)
- Technical level permissions are used when a user want to do CRUD operations on some data of a DCModel (e.g. writer on the companies (DCModel) with iri 1 (DCEntity) and 2 (DCEntity))

4.9 Model Service and API

The Datacore’s main component ResourceService uses Model types through their DCModel modelization, which is directly accessed in memory since it does not take much place and allows the optimal efficiency required to achieve its goals (i.e. checking that all submitted Resources are compliant with their Model definition). However, besides being used by ResourceService, crucial though it is, Model types have to be able to be **created and updated by type administrator** users (be it through the OASIS Portal’s type administration User Interface, or through other dedicated User Interfaces provided by applications). This also implies that those created and updated versions have to be stored in persistent manner, and made available from there to said type

administrators, and obviously (as first mentioned) to the Datacore's ResourceService. This is done by the Datacore's ModelService and its Datacore-powered API.

Model persistence and REST exposure actually reuse a lot of what has been done for storing regular Resources in Datacore, and in some way is like **storing Model type definitions as Datacore Resources**, with their own "root" Model type "oasis.datacore.model". This is because it offers all features that are required :
- **REST** exposure, with version-based optimistic locking and client-side caching, which is especially important to tak advantage of the fact that Model types seldom change
- efficient, indexed **queries**, in order to know "which Model types bear a given Mixin types", allowing for instance to look up for all kind of Resources that have the "oasis.address" Mixin whose city field is Lyon.
- collaborative management of Models, where security and global business-level as well as instance-level **permissions** are crucial, and History and Contribution features come as very useful bonus ones.

The business of Model management has however some important additional rules compared to regular Resources management :
- besides the classical "version" technical field allowing for optimistic locking, there is a business "**modelVersion**" field, which is used to build a Model type's uri, for instance "http://data.oasis-eu.org/dc/type/oasis.model/altTourism.place/3" . It allows a given Model's older versions to exist at the same time than its current one, so that Resources being compliant to an old version can choose their own time to migrate to the newest Model version, making their **migration** more scalable and putting its responsibility in the hand of type administrators rather than OASIS technical teams.
- in the same spirit, there are fields about the previous or planned Model **lifecycle** : whether a Model version has already been published and can and should be used to create new Resources, or is still only a draft one being collaboratively evolved (though test Resources can be built for it, and possibly up to period of validity, of required migration, accepted migration target.
- for **RDF compatibility**, there are fields about how each Model and Field maps to RDF, which are accessible in the DCModel and used by RDF compatibility components such as JenaSparqlEntityQueryEngineImpl to convert Resources to their RDF representation.
- there are fields to help developers code applications using Models, such as providing translations of name of Model and Mixin types and of fields, that applications can display in forms for instance. Those fields are not required to be in the DCModel business model of Model types because they are not used in Resource parsing and checking, but only in the REST-exposed Model storage. Those fields being stored in Datacore Resources as well, such translations can therefore also be contributed and edited in a collaborative, crowdsourced manner. However, the most common case if for applications to hardcode field names in their User Interface and not show fields that it does not know, so this is developed step by step as such requirements appear among use cases.

4.10 History Service and API

Historization is really important when we talk about business data that we must keep for a certain time, this is why there is way to historize your data in the datacore.

In the datacore we care about versionning, all the datas comes with version.
Historization is about storing the old version of your data so you never lose it.

In technical term, it's a copy of the data at the time you insert/modify it.
All those copies are stored in a different collection (mongo) than the one where you store your data.

Historization is made by putting a boolean on the DCModel to true (isHistorizable). Therefore Historization can be enabled by using the Model API.
Then every data you insert/modify (with the Datacore API) will be copied to the historized collection.

We also have an API that permits you to grab an old version of your data which is stored in the historized model.
This API only require a modelType, an IRI and a version.

Both insertion and retrieval are implemented in the HistorizationService. This service's use is secured not by itself, but by calling secured EntityService operations, either within its own operations or at the place where they are used.

4.11 Contribution Service and API

The main goal is to be able to contribute changes to data that already exist and that the user does not own.
The principle is for a user using an application to get an existing data Resource, modify it locally, then submit the changed Resource data to the owner so he can choose to merge it or not (i.e. play the role of data Mediator).

In technical terms, here is how it works.
The **DCContribution** object has been designed to store (this is not the entire object but only important data) :

- Title
- Comment
- Validation Comment
- Destination Model
- User (that initiated the contribution)

- Status
- A list of DCResources attached to the contribution

Under the hood, this contribution object is actually stored as a DCEntity in a dedicated collection (whose name derives from the original model's).
The resources linked to the contribution are stored into a dedicated collection too.
In order to remenber where does the data come from (and when) we added two fields on the DCResource object :

- originVersion
- originURI

It's important to remember that a contribution can only modify data of a resource and not technical fields (such as URI, Model name, author, permissions...).
When posting such a contribution, the Datacore will handle each of its attached Resources, check that they all have the same Model type, and check that they are compliant with it.
Note that you obviously can't add or delete a field of a resource when contributing (the Datacore will reject the contribution), but you can add or modify an existing resource.

A contribution has different statuses :

- DC_CHECK_FAIL (checked by Datacore, problem detected, contribution is refused)
- VALIDATED_NEED_REVIEW (checked by Datacore, Model are the same among contributed resources)
- REFUSED_BY_DATAOWNER (the data owner has refused the contribution and explains why)
- APPROVED_BY_DATAOWNER (the contribution has been approved and the owner is now merging the data manually, which consists in getting the target Resource, applying contributed changes, and finally saving (PUT) it in the Datacore)
- MERGED

In order to make all of that possible for users through applications, a new API called ContributionAPI has been called.
It allows to :

- add a contribution (with DCContribution + List<DCResources>)
- modify a contribution status
- get a contribution and associated resources
- copy an entire model + resources to another model

Here is an example.
The city of Lyon is putting some data about the different places of the city in a Model named "city.france.lyon.places".
A citizen from Lyon copies the entire Model and data to a Model that he owns, or in its own local database.
He adds some places in his own copy and then decides that he wants to share his contribution with the city of Lyon (who owns the city.france.lyon.places Model).

For this purpose, he submits a contribution with :

- 3 new places : 3 new resources
- 1 modified place : 1 existing resource
- The contribution itself

The contribution is successfully checked by the Datacore, which changes the status to : VALIDATED_NEED_REVIEW
The Lyon Cityhall employee in charge of the places in the City of Lyon sees the contribution and the associated resources.
He decides to merge the contribution with the actual model (status : APPROVED_BY_DATAOWNER)
To do that, he retrieves the contribution and associated resources, modifies URIs on the resources with the ones from his Model (so that they will target his own Model's Resources), modifies the originURI to point to the citizen model (if the contributed changes originated in the Datacore, otherwise if they originated in a local database it is empty), then use the regular Datacore API to PUT/POST the data to his Model.
At the end, he changes the status to MERGED so that the contributing citizen can see that his changes are now in the original Model.

The point of this example is to show that we offer simple APIs to make a contribution and the merge part is made by the owner, and not the Datacore himself. Merging is manual because it offers maximal flexibility, and because we don't want to put business code in the Datacore, since it has to stay as generic and also as RESTful as possible.

4.12 Generic properties (data quality, duplicate, request priority, business context, billing transaction id)

("transversal features")

4.12.1 Generic Resource properties

Generic Resource properties are properties of each resource and don't have any business logic / custom behaviour (at least in Datacore), therefore modeled as *Mixin(s)* (rather than dedicated DCResource/Entity fields), and it opens up their definition & is cleaner :
- **data quality** : int 0-10 ; API right : writer (so RDF-like API) ; semantic is relative to collaboration business ; there could be additional quality-themed criteria on ex. contribution origin
- **duplicate** : boolean ; means "at some point, this Resource was the only existing version of the data it represents"
- author : string (i.e. original author of the changes, not necessarily an OASIS user) (will se later if it might be dc:author), and same for **comment** / motivation (might be dc:description)

Such generic Resource properties are therefore

- defined in OASIS-wide Mixin(s) that store per-resource properties,
- accessible in Datacore Event system, which can be plugged by any code using them.

4.12.2 Generic Request properties

Generic request properties are (at least used) elsewhere (log, notifications, request handling) and have business logic, at least for how to get them (HTTP header accessed through (business interface built on top of) (Cxf)JaxrsApiProvider injected by JaxrsServerBase ; chosen because simpler than alternative local service arguments or gathered in a Context argument or in threaded transversal call stack Context, THOUGH it may not be enough for async & batch tasks) :
- request **priority** : int 0-10 ; applications must provide a lower one when ex. batch / background / mass operation ; Datacore could use it later, for instace for query routing / messaging. Request priority is lower if : no model type, hits default containerUrl (and not a shard endpoint's baseUrl), query criteria or sort on non-indexed (Queriable) fields, auto join query
- business **context** ("functional chain") : string ; for now saved in log AND Resource (because it is audit information like owner and modifiedBy) ; it could be used later to trigger notifications i.e. call Kernel API EventBus
- **billing transaction id** ("free code" / "transaction" / "champ libre") : string ; provided as request parameter, saved in log (and not in Resource, which would be possible only on writes and not on reads), where it might later be aggregated and used to power a business model

Such generic request properties are therefore
- stored in OASIS-wide generic Mixins, reusable by all Models
- passed to API calls in HTTP headers
- retrieved on server-side through a dedicated business interface GenericRequestProperties implemented using injected JaxrsServerBase, itself built on top of the CXF MessageContext, available all along the handling of the request in the same thread (rather than adding Java interface method parameters that would make them more complex, less business and less RESTful)
- a pluggable Java interface allows to handle it on server side (see test)
- provided on Java client side by CXF interceptors, fed by client-wide configuration of properties : set/getProperty on Datacore client Java interface
- available in Swagger UI and client libraries by being defined as implicit parameters on API

4.13 RDF/SPARQL Facade

The SPARQL Facade is implemented as one of the available Entity Query Engines (see Resource Query chapter), which are exposed to client applications through the REST Datacore Query API. It provides Datacore native query capabilities, but in the SPARQL language.
Its JenaSparqlEntityQueryEngineImpl implementation
- uses **Apache Jena**'s SPARQL parser and Java model to parse incoming SPARQL queries
- visits this model to translate it to LDP-like Datacore finder criteria queries

- which are then executed using Datacore's LdpEntityQueryServiceImpl. Note that this means that it has no features beyond LdpEntityQueryServiceImpl's own.

See architecture schema in Query chapter.

4.14 Sharding data close to users

For OASIS Datacore, the goal of sharding is to optimize queries by having data near its consumers, in a way that can work for all data. Here we expose how it is done.

Note that specific business (such as log monitoring) might however prefer optimizing writes, in which case their own Model types' collections might be sharded using a shard key that is randomly distributed (across all shards) such as a hashed one.

4.14.1 MongoDB sharding principles

MongoDB (tag-aware) sharding is partioning data per shard key field value ranges. Shard key values must therefore be geolocalized,
- (by default) either through location of use (used endpoint / application / current user etc.),
- or through **geolocalized** perimeter of business Resources i.e. scopes (requires all business Datacore scopes to be defined and never change).
- it might also be sharded according to **Resource fields** as said in Model or scope, ex. address of Resource or of its owner / creator / modifier / "mediator" or of other business field-related Resource (site...).

4.14.2 Strategy 1 - location of use

i.e. Geolocalized sharding key according to context
- a "location" request context property is configured in each Datacore endpoint, or application frontend or even browser (telling their backend to use said endpoint), or on each end user (better than application frontend) or their organization (which is more scalable) or group (and further, on business Datacore scopes)...
- this "location" property values are defined so that distance between values is comparable to geographical distance between endpoints it is configured on, either tree ex. continent/country/region/city/district.
- the default "location" property is based on the user's configured address, his business (**organization**) address if it's a company account, his home address otherwise. This is because using an default, contextual, technical location (endpoint, application, browser...) has problems : if the location of use is "bad" the first time (ex. Resource created by employee working from remote home), it will then always have (relatively) bad performance
- it has an additional hash-like complement field. This is because low cardinality location (endpoint, application, zipcode, organization without office room number...) is bad because Resources will

almost never change shards on finer resharding (ex. if using endpoints, resources created when there was only one endpoint will always be in its single corresponding shard).

4.14.3 Strategy 2 - business Datacore scopes

i.e. Business-level (model / scope) sharding key
- scope requirements : a Resource must have a single business Datacore scope (or at least a primary one), it must NEVER change (indeed, scopes are given to users but don't change on Resources, ex. Torino's plot of lands will remain Torino's, or at the very worst very close to it)
- it must be provided in queries (not providing it is allowed, but then the query is sent to all shards, negating the benefits of sharding), it must be stored in a field that is the prefix of the (compound) sharding key. Since a business Datacore scope is checked this way for rights, these constraints are a given.
- shard key follow up may then be : (hierarchical) sharding-only "sub-scope", Model type then Resource id i.e. URI (allows further sharding according to Model type then Resource id which can be hierarchical ex. for city "France/Lyon"), or another sharding (location of use, or Resource fields)

4.14.4 Such shard key is then :

- On REST operation on DCResource(s) whose Model(scope).sharded is enabled, the shard key value is extracted out of the use context (i.e. endpoint, scope...), and out of resource, according to Model(scope).shardKeyStrategy
- On creation, it is stored as a "technical property" defined in DCEntity (rather than Mixin).
- On other CRUD operations, it is ADDED TO URI (if URI does not contain scope, which is better ex. http://.../dc/type/city/France/Lyon) in MongoDB interactions. This allows routing to the right shard.
- on queries, it (full context-based key or partial Resource query-based key) is added to the query.

4.14.5 MongoDB sharding configuration

For this to work, the following Mongo configuration is required :
- tell Mongo to shard said Collection using said shard key
- define shard tags (global, or if use cases require it per Model type if they want a specific sharding) by telling Mongo to which range (of said shard key) they apply.
- tell Mongo (only once if global shard tags) to which shard each shard tag directs.

4.15 Mediation

Collaboration of several actors (users, organizations) on the same Resource(s) is possible by :

- the **workflow** collaboration model. In this model, all actors take turns to work on a Resource and agree on whose turn it is. In concrete terms, each actor checks whether it's his turn (by reading a "currentEditingActor" field or a Resource Access Control List (ACL) such as “owners”), contributes its own changes, then gives the "token" to the next actor (sets "currentEditingActor" field or if has “owner” rights gives them).
- setting up the proper **rights** on the Resource (see Security / Authorization chapter). This allows to prevent any unduly write and authorize welcomed ones. This may be combined with the above workflow collaboration model to prevent any untimely write.
- **a posteriori** moderation, i.e. reviewing changes after they’ve been done. Client applications must use History API (which must be enabled for this Resource’s Model) to retrieve the previous version in order to compare them and review changes (see History Service chapter). Client applications can define a “reviewed” property can be defined (a Mixin is the proper place) to say whether such review has been done or not, or even provide additional information (though original author and creation date are already in available in audit properties).
- submission of **changes to be approved** by a mediator, usually the Resource owner(s). Such a priori moderation is the simplest way of using the Contribution API (see Contribution Service chapter).
- further **contributions** from one Model to another from which it derives (that it extend and whose Resources it copied at some point) can be managed by combining all the above : Contribution API to submit changes, History API to retrieve any version and review changes, and client applications can define merge information properties (at least source Model type and Resource version).

4.16 Specific Datacore deployment architecture

4.16.1 Datacore scalability options

Scaling up is achieved first vertically, by adding RAM (ex. from 8 to 36go or even 128go...) so that more data can be mapped in memory, or putting MongoDB Journal on its own physical volume (if possible SSD ex. 2x64go RAID-1 which are very efficient but costly), or CPU if monitoring shows that it is the limiting factor, but not beyond what infrastructure and cost efficiency allow. Vertical scalability is easy and inevitable, but limited and obviously not enough for OASIS goals.

Scaling up is achieved secondly horizontally, by adding new VMs. Java VM nodes can be added transparently to handle more web HTTP requests. MongoDB nodes can be added to either :

- scale up reads is achieved by adding a new read-only data VM to the MongoDB cluster
- scale up writes is achieved by adding a full shard, which comprises at least 2 VMs to the MongoDB cluster. Here MongoDB's full power becomes apparent, by making this option quite easy thanks to it automating most of the tasks.

The said 3 processes of adding VMs (Java VM, read-only MongoDB VM, MongoDB shard VMs) must be documented and industrialized up to a good enough level of automation by their application providers to be efficient and not error prone. Ultimately further industrialization at the infrastructure level (up to being able to add such VMs by merely clicking on a button, by ex. using Puppet and / or dedicated APIs...) will be studied based on project results.

4.16.2 Datacore VM requirements

In a first phase, only on-demand VMs will be required.
- Datacore : one VM (for the Java REST API and mongodb) with 1 CPU, 4go RAM, 20go HD ext4, network, running Debian stable (wheezy) 64 bits, to be available on Friday 11th

In a second phase, the full production environment of a full OASIS standard Container will be required, to be available at the start of February (so that the end March pilot deployment)

Datacore :
- three MongoDB Config servers (each residing on a different discrete system)
- at least 2 shards, each being replicated on at least 1 additional VM ; so at least 4 VMs. Each VM is as initially, save for : 4CPU, 8go RAM, 2x500go RAID-1, 1gb network (TODO to be discussed and updated). For reliability, in each shard, there should be at least one replica (if possible all) on a different discrete system. Note that if a shard has an even member count, an arbiter must be started on a (typically related, see below) application server.
- Application servers : in front of each shard (to let it serve its data according to geographical repartition), a cluster of at least 2 VMs behind (DNS) load balancing (each also running its own mongos client process), at least one (if possible all) being on a different discrete system. Each VM is as initially, save for : 4CPU, 4go RAM, 1gb network (TODO to be discussed and updated). And in front of all clusters, routing (possibly itself clustered and fronted by DNS routing) that respects geographical distribution.

Those requirements will be completed step by step as the project goes forward, especially on the side of additional deployment environments required beyond first deployment, and of further steps of scaling it up obviously.

4.17 Technical perspectives

4.17.1 Multiple datacores

Several Datacores can coexist and collaborate, their data having links to some of each other's data, one Datacore being a single sharded MongoDB database fronted by Datacore Java engines. There is only one requirement : they must all have the same data definitions i.e. the same Models, otherwise cross-Datacore data will be inconsistent. The solution is for them to share the same client-side cached Model repository, as shown in Model storage and management API chapter.
Perspectives for multiple Datacores include :
- checking Resources referencing another's Datacore Resource (for now it is only checked locally)
- true Proxy behaviour, where each Datacore can provide a Resource stored in any other Datacore, however beware : it's less Web-style scalable, rather ask directly the right Datacore.
- implement multi-datacore queries (the only case where Proxy-like behaviour can't be avoided), by building on the join algorithm described in the query chapter.

4.17.2 More flexible data type constraints

In addition to supported data field types (see Model chapter), the following extended constraints on field values could be supported :
- **range**, on number field types (or also alphanumeric ones)
- **regular expression**, on string field type
- **list of external values** i.e. from specified RDF Model type, or even from (cached) Datacore LDP query, or embedded ; on primitive field types
- script, on any field type. This opens up the door on fully custom checking of Resource property values. However, it would have to be finely though out, in order not to risk hampering Datacore performance because of badly written or designed scripts.

4.17.3 Allow incomplete data, but set lower quality

Contributed data will often be incomplete. In this case they should still be accepted, because any data is better than no data at all.
This is possible by defining Models with all of their fields being optional (with a required=false attribute).
However, even if it's accepted, it would be nice if it could still be remembered that the contributed data was not complete. Here is a possible solution.

if Resources are not compliant to their Model, don't fail anymore, but **fill dedicated warning technical properties** (such as dataQuality, completenessPercent, addressComplete...). Not compliant means :
- "missing field value" => fills missingFieldValue=true technical property of the Mixin.
- but also why not "bad field value type" => fills badFieldValue=true technical property of the Mixin.
- or any other failed constraint, if we were to implement an extensible, pluggable constraint framework, which is another Datacore perspective.
Note that additional attributes will still not be allowed, because all provided data must be described.
Said dedicated technical properties about (often business) aspects of data quality can then be used as criteria in queries to get all complete (or not) data. They could also be used in an endpoint dedicated to analyzing them, recomputing them and listing data quality details such as the list of missing or non compliant fields.

4.17.4 Internationalization (18n)

Datacore Resource properties are flexible enough to support internationalization. For instance, an internalized field can be defined as a map of languages to translations of its original text in said language, such as in JSON : { “FR”:”île”, “EN”:”island” }.
Further features could be developped, however beware, our experience (for instance in Alfresco ECM) is that an i18n field type that transparently stores a single text value in the user default language are more a problem than a solution, if only because it makes it harder to look up information because it is sparsely distributed among languages.
Therefore, further features might be developed, but they will be according to the requirements that

will come out along use cases development, such as :
- allow true Map field with any number of string-to-some type entries (just like Datacore lists)
- default, reusable i18n field type, with the exact list of languages that are wanted to be supported in OASIS
- fallback language : if asked language is not available for a property, default to said fallback language

4.17.5 Optimized queries

Here are what could be done in the future according to use case requirements :
- provide query execution introspection API returning information on how query is actually executed (ex. which index is used), in order to help users do better queries. Indeed, otherwise users can't optimize their queries. This is done first and foremost by returning the result of a mongo explain() command, but also which exact mongo query has been executed, the parsed LDP query, request priority…
- index on compound fields
- text index. In MongoDB, a text index drops language stop words and does language-specific suffix stemming, ~20 languages supported)
- make configurable on DCModel and use it : default sort, ex. modifiedDate (useful for hot data ex. news/posts) rather than URI (useful for cold / rarer data)
- automated join algorithm queries
- query timeout
- batch asynchronous query execution, as (prioritizable) task

4.17.6 Polymorphic models

When several Models that belong to the same business domain are rather (in compared amount of queries) queried together than alone, it is far more efficient to allow them to store their Resources all in the same MongoDB collection (rather than one per Model).

Typical use case is Citizen Kin (though for Citizen Kin the problem lays rather in not having its own business storage, it is still a valid example) wanting to retrieve all latest procedures to be handled by agents, with those procedures being of various types (because they have differents fields). Without this feature, the page that lists all latest procedures that an agent has to handle would required n+1 requests to Datacore, where n is the number of types of procedures.

This can be done by
- letting DCModel.rootType be specified and override default DCModel.getCollectionName() (modelName),
- adding a MongoDB criteria on DCModel.modelName to all queries to a DCModel where DCModel.isPolymorphic() (computed from above fields), and indexing it (only once per collection and not per Model, in DatacoreSample.createCollectionAndGenerateIndices() and #23 Optimized queries).

- ensuring that all features using persistence (EntityService, LdpEntityQueryService, #18 History, #19 Contributions...) use this DCModel.getCollectionName() for their collections rather than modelName, and also index their modelName (_mdln).
- allowing resource reference to a polymorphic root type. DCResourceField.rootType replaces DCResourceField.modelType if defined, allows to look up target Resources in a multiple polymorphic types MongoDB collection : in ResourceService.createOrUpdate() to check it, in LdpQueryEntityService.findInData(InType ?) to look up Resources of this rootType, but not in get/delete because it doesn't bring anything more and is not REST.
- Note that everything else stays the same at Resource level, such as RDF representation.

4.17.7 More technical perspectives

Historization could still see some improvements :
- Time until deletion of the data,
- maximum history size...

Contribution API has been thought out to be able to go way further than mere approvable changes, up to allowing merges derived Resources (where one was copied at some point from the other, with one's Model extending another's), in a way that mirrors Git's pull request. Additional API features could also be provided to help client applications do such merges, such as a Resource Diff API.

Support of RDF and SPARQL through the RDF / SPARQL facade may be enhanced if the need appears : full RDF / SPARQL compatibility test kit, using the jsonld-java library to add additional representation kinds for RDF (XML, Turtle or n3) or JSON-LD (flattened, nested, framed) as specified in HTTP request using content negociation, additional SPARQL features (which require to first design and implement the corresponding features of LDP native query language if they don't exist yet)...

Consistent update of a set of Resources can be developed by implementing a full proper 2 Phase Commit (2PC) algorithm. This mean defining locks (or transactions) in a dedicated Model, and referencing them from Resources (which is well modelized by a Mixin type) that ResourceService has to check prior to any update. However it must be used parcimoniously and REST principles such as optimistic locking should be preferred to avoid breaking up scalability of collaboration, which is why such feature is not a priority.

5 Kernel Implementation

5.1 Kernel general description

5.1.1 Deployment Architecture

The software architecture of the OASIS Kernel has been thinking with the following constraints:

- Each component could be deployed in an elastic way (management of scalability)
- Each component could be deployed in a redundant way.
- Each component could be installed and provisioned with an automatic system.

The management of hardware resources, according to the scalability of the system, is insured by the mechanism in an IAAS: for the exceeding of certain thresholds, some nodes are automatically appended or deleted, each component will be automatically reconfigured.
The multi sites repartition (on several Data Center) is under control of IAAS underlying functionalities.

OpenStack and Puppet are used to manage the provisioning part. A cluster of host OpenStack is the heart of a hybrid cloud extensible if necessary with the appending of VM on demand on the platform on public cloud.

5.1.2 Software Architecture

OASI kernel is developed as a java standalone application (with no needs of deployment on an application server), integrating a HTTP server based on netty, giving some request managed with the implementation of JAX-RS Resteasy.

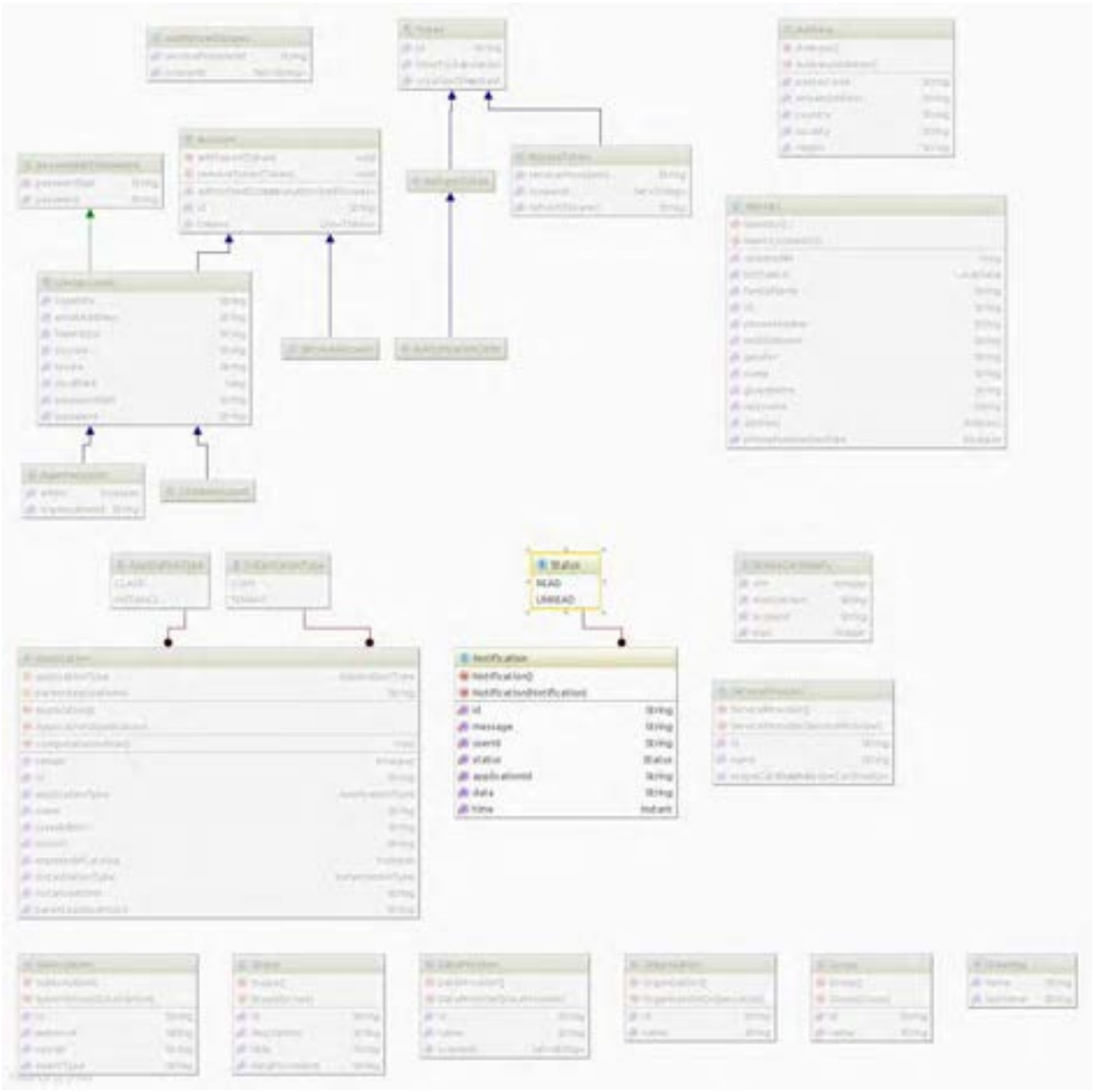
5.1.2.1 Dependencies Injection

Each component is developed in a unitary way and indicate components which it needs.
Components are set up and injected during the execution by Guice.

5.1.2.2 Model

In the center of development, some concepts were describe in the following way:

UML Diagram of the model:

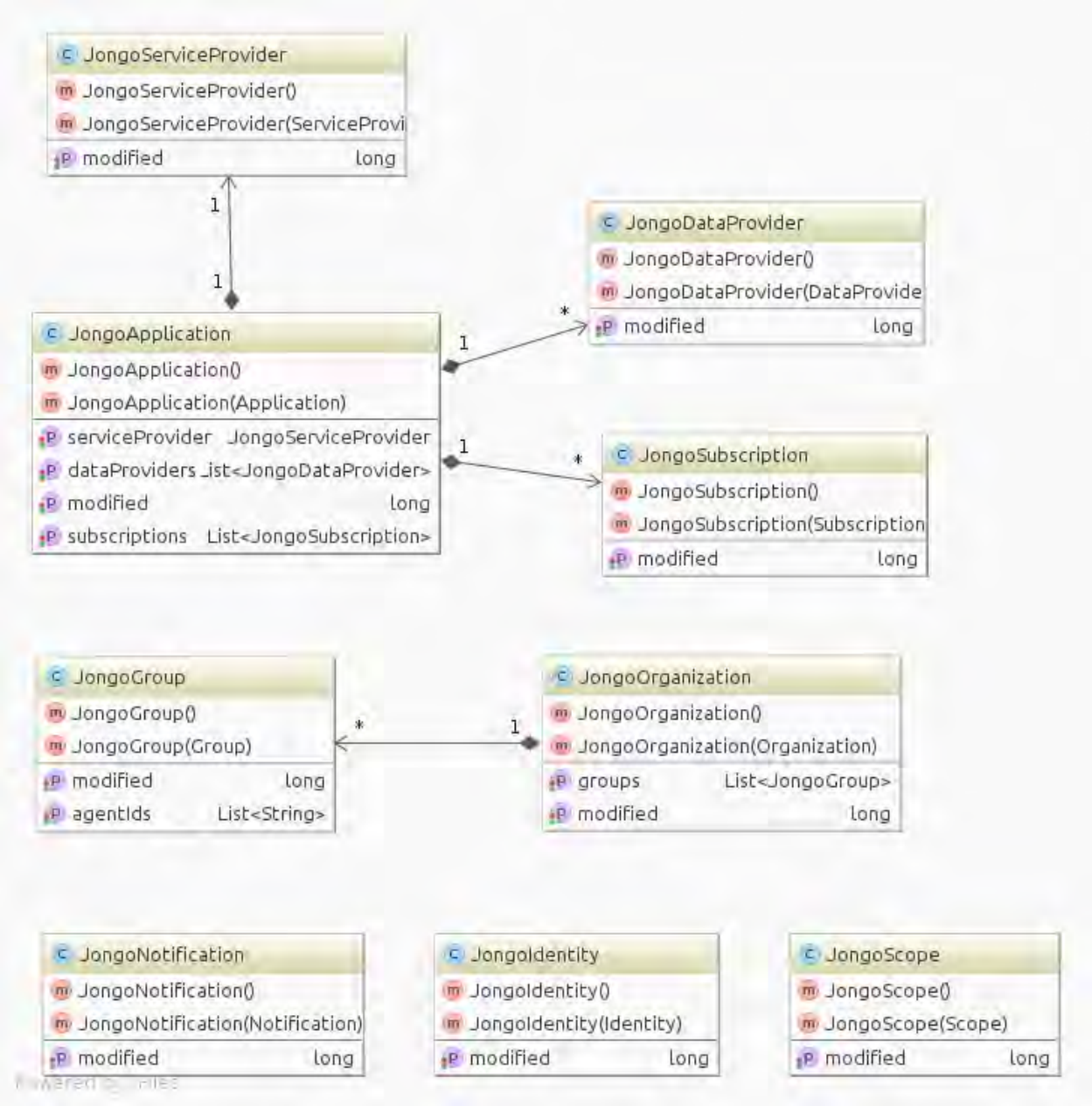


For each classes, a repository interface was defined, showing necessary operations for creating, reading, searching, modification, deleting, etc…

5.1.2.3 Storage

An implementation of these classes storage in a MongoDB was developed according to this model. Technical fields which are used only for the storage are manage at this level.
An implementation of each repository is also provided resting on the Jongo library.

UML diagram of classes for storage: each classes JongoBla inherited of Bla classes of the previous model introduced :



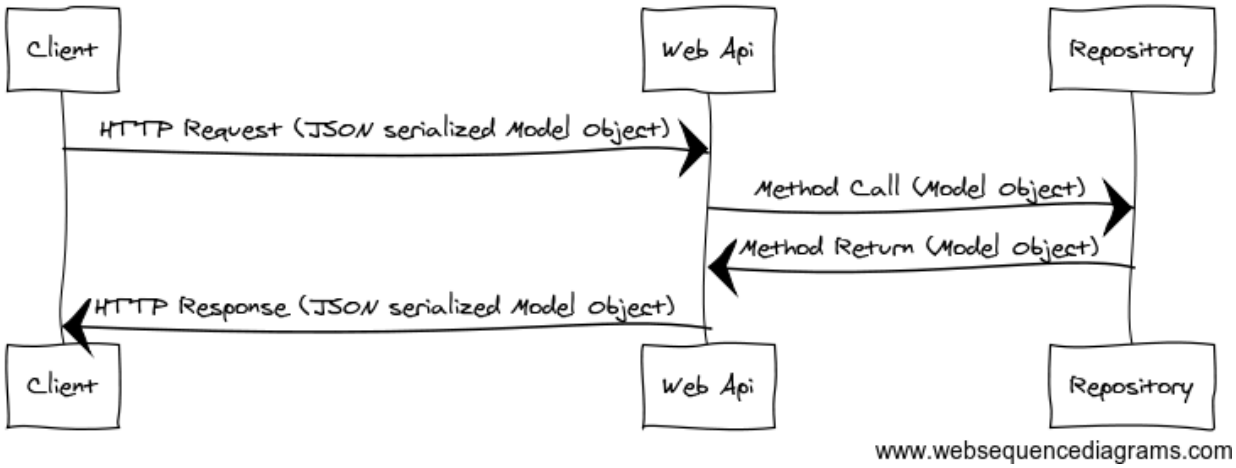
MongoDB do not purpose mechanism to manage the referential integrity between two distinct collections, we chose to store the entity dependant from each other in a same collection.
So the description of DataProviders of ServiceProvider and subscriptions of application are stored in under objects of Application collection

A DataProvider or a ServiceProvider, couldn't exist without an Application. In the same way, groups are registered as under objects of organisation. Each under Object is managed in autonomy for its login or its version.

5.1.2.4 Web API

Each identified API in the architecture is defined as a JAX-RS resources. Classes of models are used as Web API parameter. This APIs are based on Repository defined with the model.

Web Api Request :
Client->Web Api: HTTP Request (JSON serialized Model Object)
Web Api->Repository: Method Call (Model Object)
Repository->Web Api: Method Return (Model Object)
Web Api->Client: HTTP Response (JSON serialized Model Object)



5.1.2.5 Configuration

Each OASIS Kernel parameter have a default value hard standing defined in the software which could be overladen by a configuration file or by a system property. The configuration file allowed to add some parameters, this file is given as argument to the started processus. Management of this argument is done in the unix server of the OASIS Kernel.

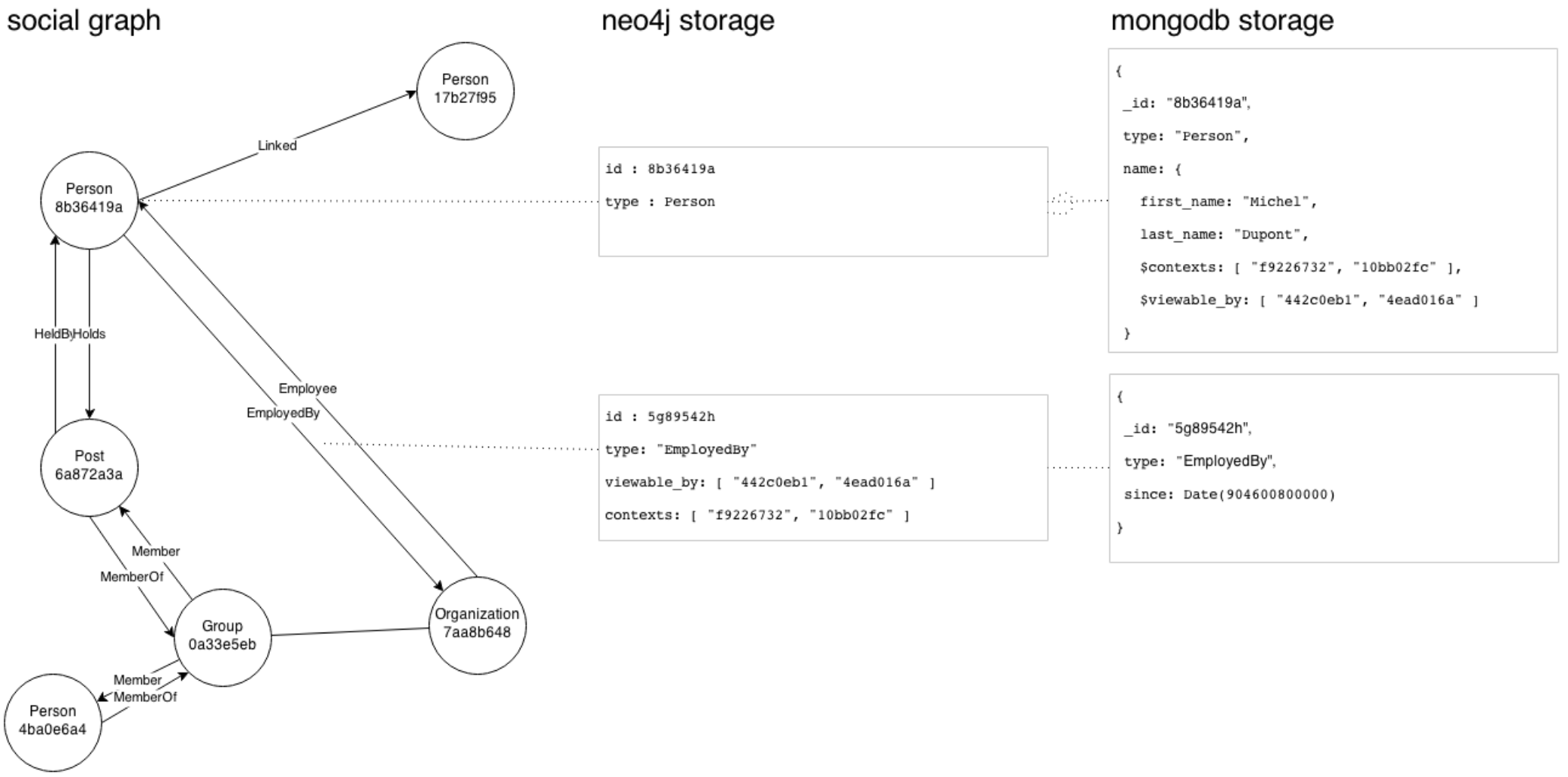
5.2 Social Graph

In the social graph we store some nodes and relationships between these nodes.
Description of each nodes could be a tree structure of properties store in Mongo DB with a unique ID.
Relationship between these nodes are stored in Neo4j database, where nodes refer informations stored in Mongo with this ID, and relationships were qualified with some properties (most of time links and ACL).

These 2 kinds of information are separated into two ways of distinct storage and the graph stored in neo4j is completely anonymous (see picture below). Administration of databases are managed by different administrators.
Each relationship own an ACL, ie. list of group identifier from source of relationship allowed to read this relationship and the list of contexts giving information on this relationship for the application.

Neo4j allows to make advanced request describing browsing rules in the graph. So we could recursively get group members, company employees, etc…

```
START g=node:Group(id={id})
MATCH (g)-[:Member*]->(m:person)
RETURN m
```

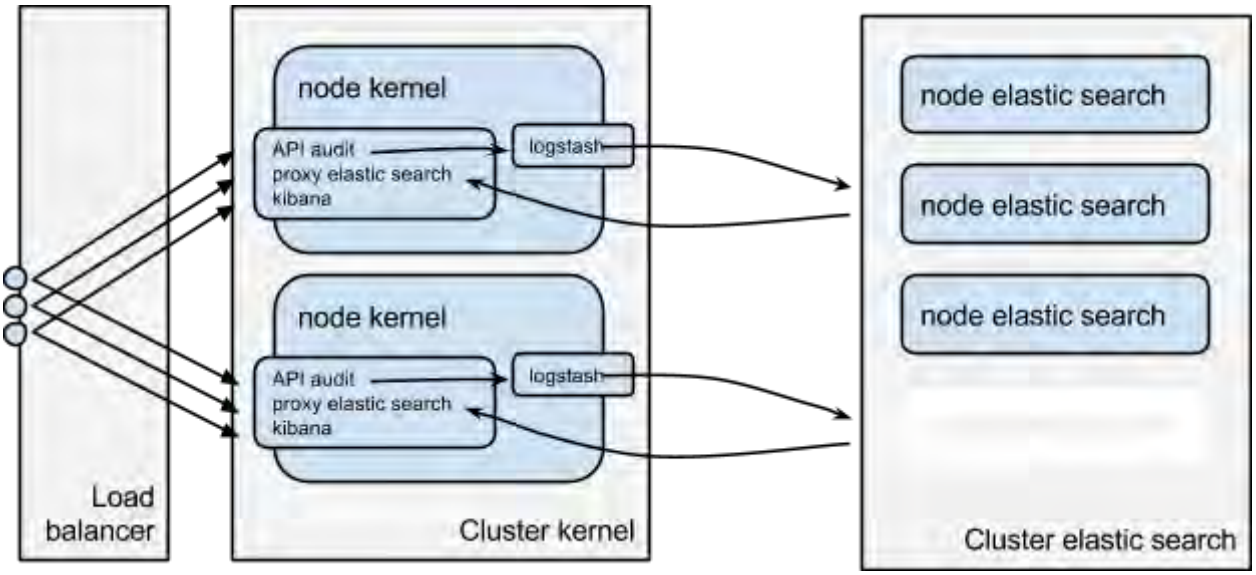


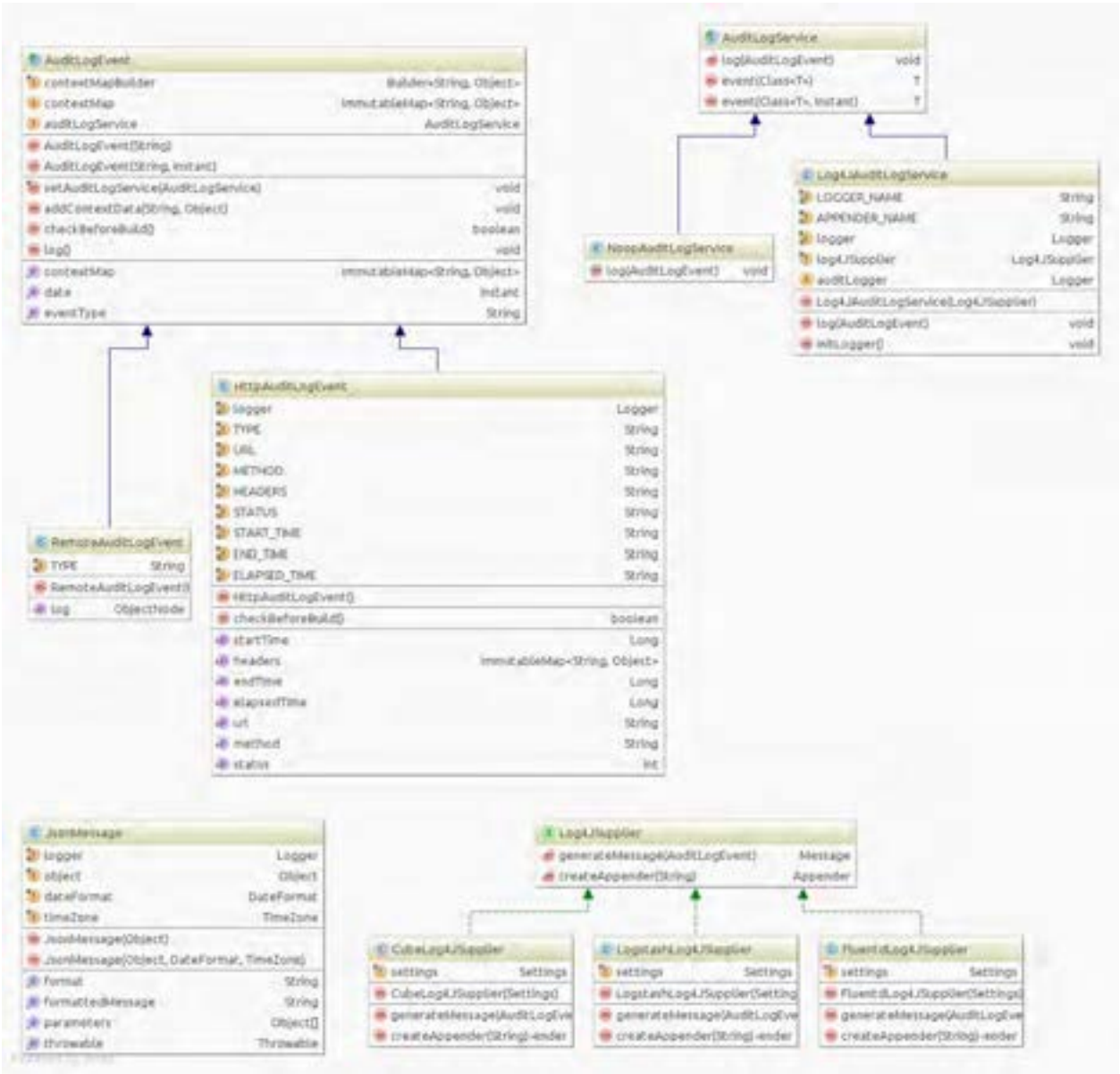
Neo4j and mongodb storage of the social graph

5.3 Log module

The audit mechanism allows to keep some events which took place on the OASIS platform. These events are a set of dated metadata.
Data are stored in elastic search, an indexation engine open source based on lucene, it had some Web API and a mechanism of elastic deployment.
Integration of Kibana allows exploitation of data with a powerful and customizable dashboard system. Data are stored under Kibana format using Logstash as a relay of audit events.
Logstash will allow to integrate different metrics, which come from different technical component already support by the same audit event management tool and to extract the good KPI's.

Kibana is integrated in the web interface of Kernel, and already configure to access to the elastic nodes search by a proxy located in the Kernel. This integration allows to manage the access rights of the application and data in same way.





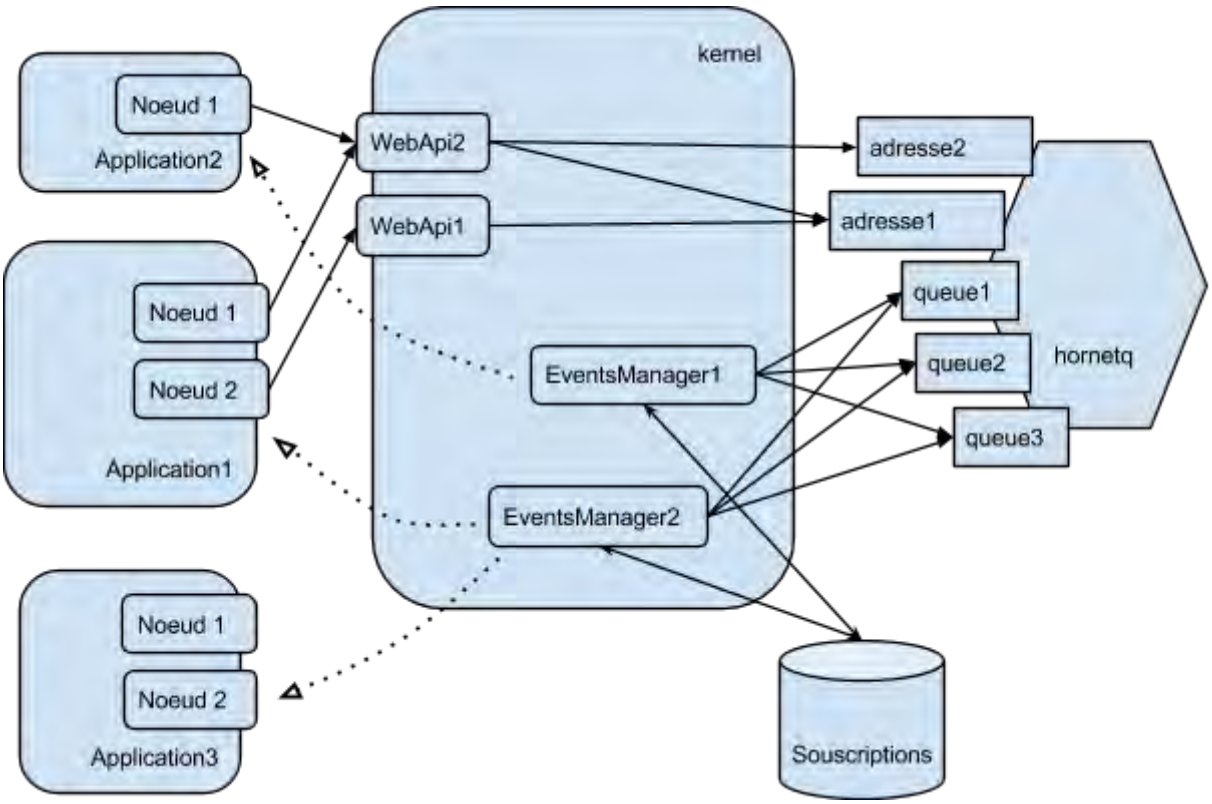
Different events which are logged by the audit mechanism, belong to a hierarchy which have the AuditLogEvent class as a parent. Web API receive events from providers managed by RemoteAuditLogEvent class, which contains a JSON structure which could contains all sender message application wishes.
Others classes are used to manage Kernel generated events (logins, errors, authentication…)
Technical events registry service, is abstract with AuditLogService. Several implementations are possible, two are giving, the first one doing nothing (allow to deactivate this functionality), and the other one based on log4j2 to use asynchronous logging mechanism from this library.
This implementation could be setup to store JsonMessage in logstash, fluentd or cube (with changing Log4JSupplier).

5.4 Event Bus and communication between applications

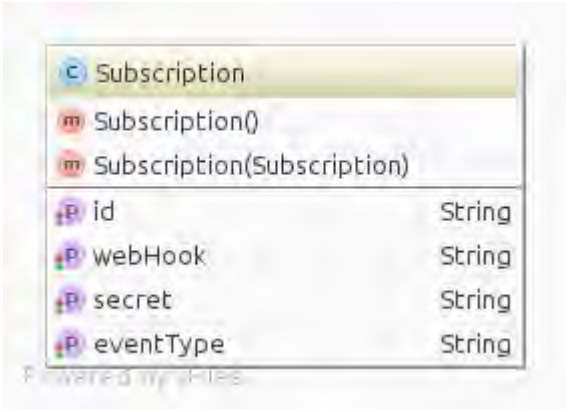
The event Bus between applications is in charge to allow to decoupling some processing: an application indicate that an event has encountered (data created, deleted or state changed…), and applications which want to react on this kind of event are notified to permit them to processing.
It's an event bus kind of publish/subscribe. In an elastic software architecture a same component couldn't deployed several times. Main goal is to allow a processing repartition and not a duplication of works already done. So it's very important that an event is consume only once by a listener. To implement this events bus, tool of HornetQ from JBoss was chose. It's a high performance messaging tool, all written in Java, purposing modes working on network or on intern memory communications of the JVM. Beside, HornetQ could works in autonomy in its own process or embedded in an application server.

In OASIS scope, an autonomy version of HornetQ is deployed on a nodes group dedicated to this event bus. Applications servers storing HTTP servers of Web API of Kernel communicate with the nodes cluster under TCP connexions.
HonetQ manage two notions: addresses and waiting queues. Queues are associated to Adresses, events are publish on addresses and get on queues. The routing messages between addresses and queues could be setup thanks to filters.

Here routing is simple: each message published on an address is redirected on all associated queues. So we create one address by kind of event to manage, and we associate as queues as there is applications interesting by this kind of event.
At the beginning of a Kernel node, as ClientConsumer are created as number of subscription in database, who will connect to HornetQ in the wait of event. Each ClientConsumer delegate to its MessageHandler the processing of the event. This MessageHandler in charge of calling the subscription WebHook which concern it.
A sharding subscription processing is possible when the number of subscription to process would begin too important to be manage by a lonely node.



Application publish via the event API of the Kernel on addresses which corespond to the published event type. EventManager of each kernel node have the same number of Clientconsumer as subscriptions in database. When the message is received by a consumer, the associated WebHook is called and application that have subscribed to this type of event could process it.

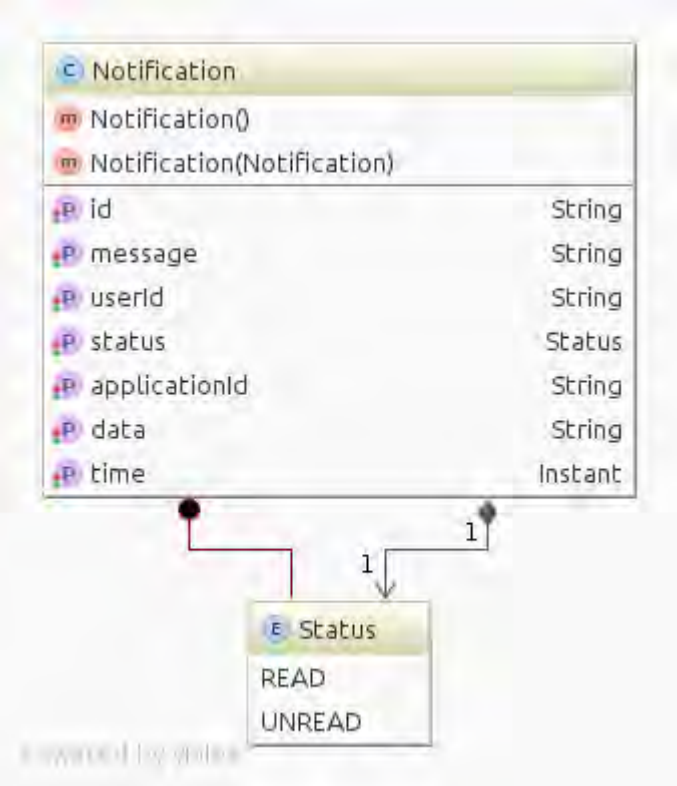


5.5 User Notifications

The objectif of this component is to centralize the notification management from applications to user. User can configure in one area the way he want to be notify, and it's possible to see all notifications in a same area.

System of notification management is designed to be consult by several clients (web portal, mobile application…). Applications send notifications to a list of users or groups. Each user interested by a notification will have a copy of it with the state “unread”. Each created notification have an associated timestamp which allow to keep order of their integration in the system.
A client which receive some notifications could be sure that new notifications are olders than the most recent he already know.
OASIS user could mark a notification “unread” as “read”, with a client application (even if the Web API stay accessible). He also could mark a “read” notification as “unread”.
It’s also possible to mark several notifications as “read”.
Catching or marking notifications as “read”, could be done on one specified application.

Model



5.6 Identity management

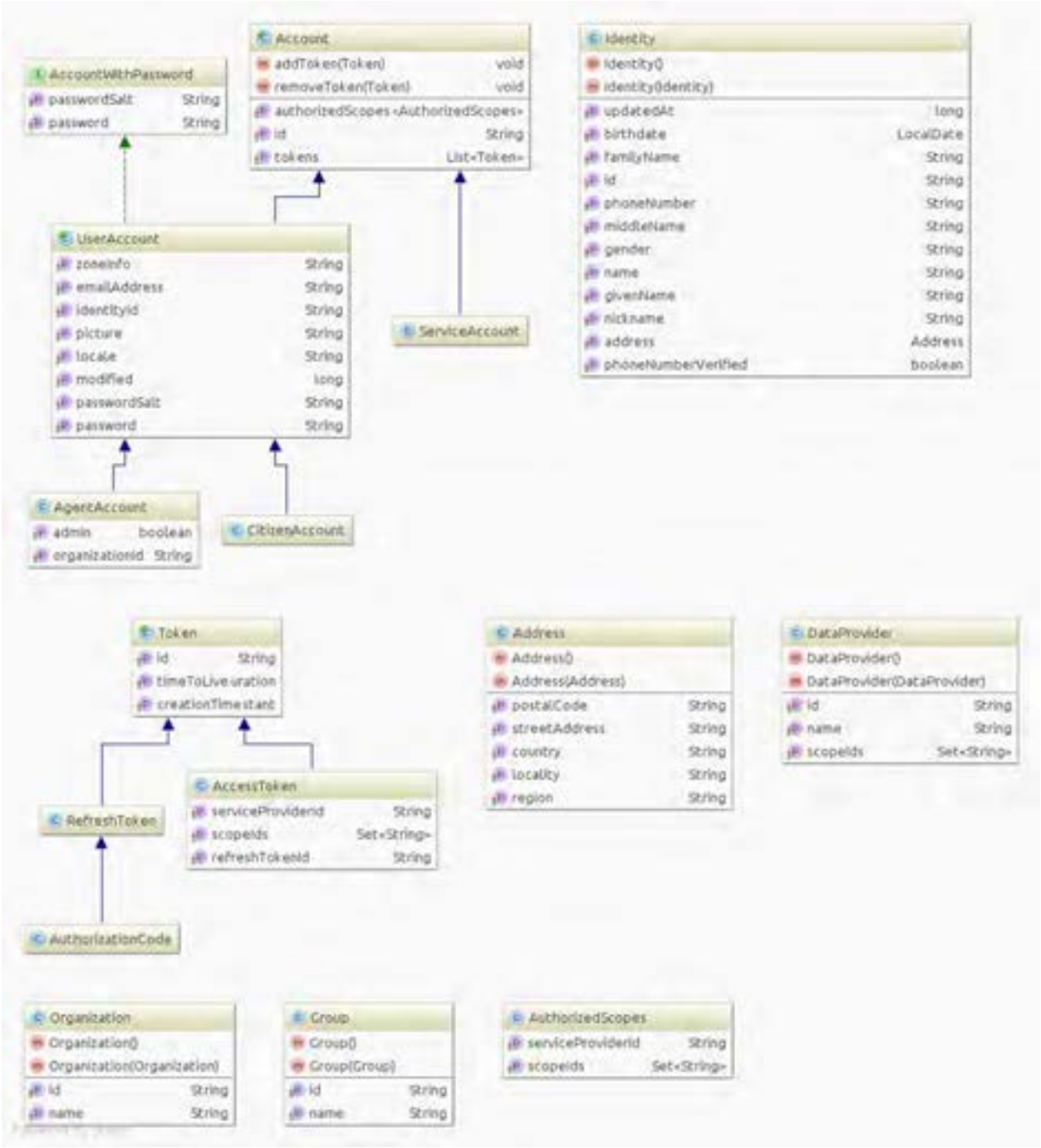
A person coul have several accounts, each one get its own credentials in different organization. He also could have one or several citizen account. However there is no Kernel control on authenticity of informations given in profils, each application have to insure of it, with asking necessary justificatives it needs.
Grouping of all these accounts are a unique identity, eventually checked, with uniques credentials could be considered in future, but no implemented in this first version, in order to be easier and clear for users. Because even the different account are technically well separated it's not sure that user would trust in a system who could link his private and professional life.
A user account is identified by his email address used to create it. This email is used as a login account and have to be unique in OASIS.

5.6.1 Accounts type

A user could initiate the account creation which is not linked to any organization. This kind of account is called CitizenAccount.
Account created by organization are called AgentAccount.
It's also possible to create service accounts for applications which need to procees without anyuser, it's called ServiceAccount.

5.6.2 Creation of an entity

An organization is created in the same time as a user account which is automatically the administrator of it. Organization is created for a specific domain and all the users emails of this organization have to be in this domain. Several users could be declare administrators for one organization.
Administrators of organization could append users or group to the organization. A synchronize users functionality from a LDAP directory allows to import users who already exists in a database.

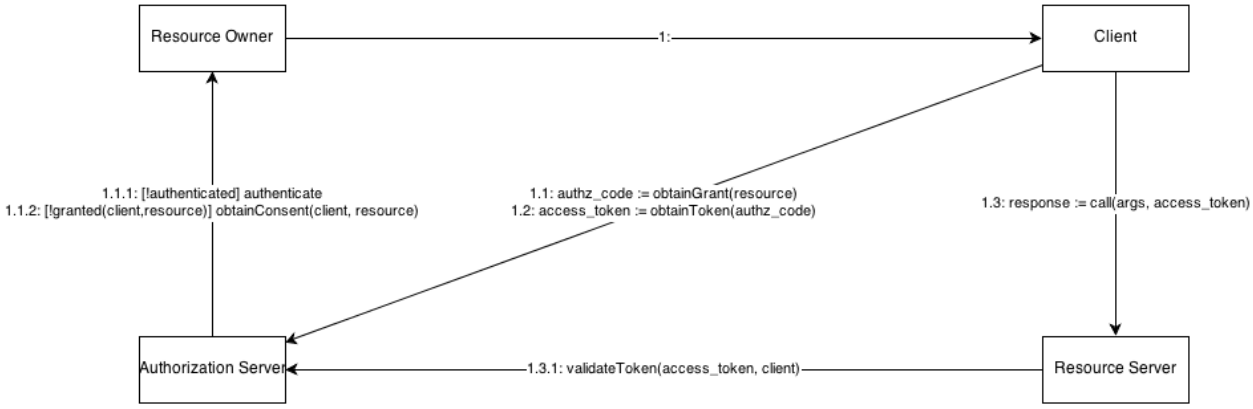


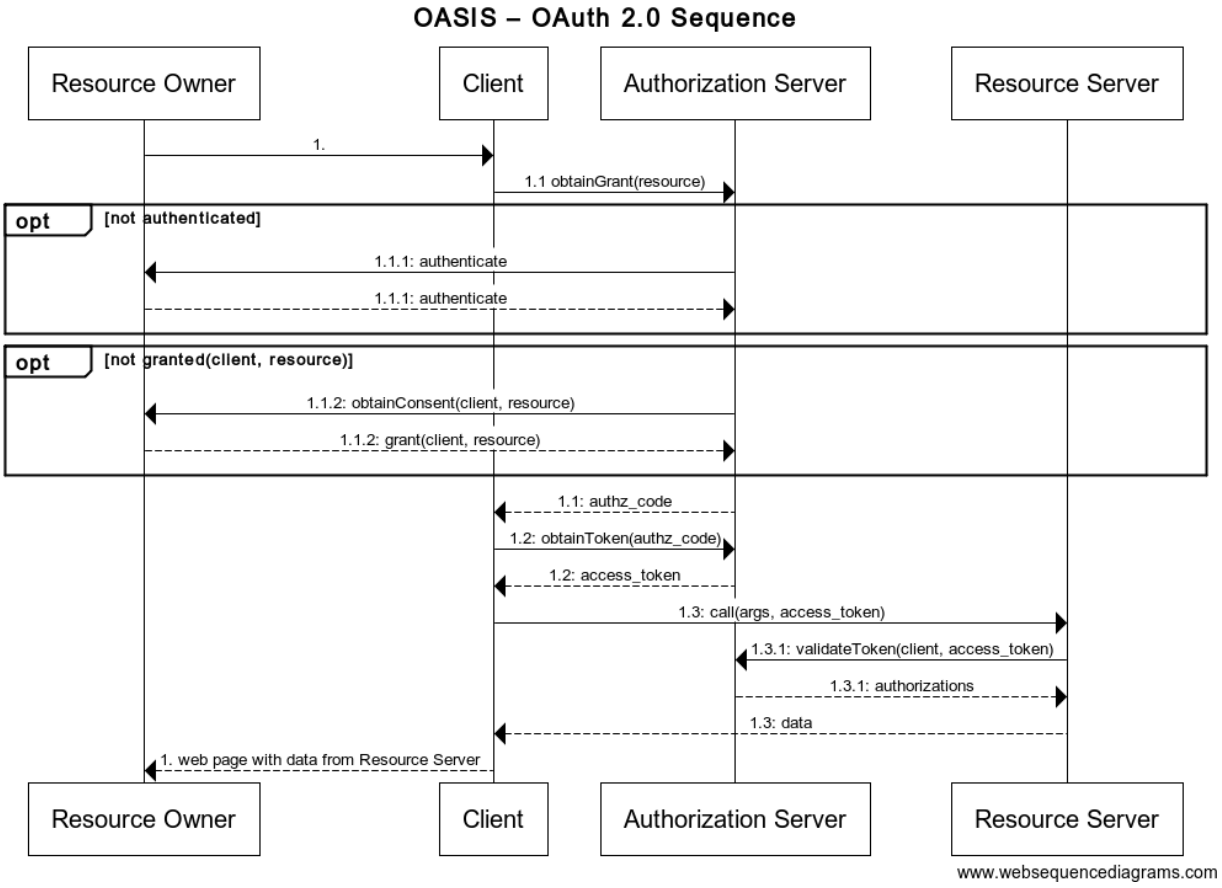
5.6.3 Authorisation (OAuth2)

The followings schemes introduce interactions Open ID Connect 1.0 and OAuth 2.0. Terminology used is OAuth 2.0.

- A ressource is given from user; Resource Owner is the user and Resource Server is a Data Provider
- Client is Service Provider which would like to access to user resource.

- Authorization Server is the security server which is guarantor of access permissions.





5.7 Catalog

Catalog of registered applications to the plateform lists componants and technical paramerters of known applications but also user's subscription to them.
Two levels of catalog were introduce, describing thes two kinds of usage and informations.

These two catalogs are managed in the same collection Application with a complement of data which have to be defined another time in objects inherited from parent's objects which correspond to a model.
Two properties allows to know which kind of object it is, in order to recompose informations necessary to describe an application: ApplicationType and InstantiationType.
Beside an application could reference another one with the parentApplicationId property.

5.7.1 Static catalog

List some applications to show in an application which is a kind of "store", in this one we can chose application to append to an organization, which allows organization's members who have rights to access this application.
Contain a technical description of purposed componants and necessary componants (Data providers, Event listeners, etc.) and some generic descriptive information.

ApplicationType = CLASS
InstantiationType non utilisé
parentApplicationId = null

5.7.2 Dynamic catalog

List of applications chosen by organizations, used for access rights or traceability actions.

5.7.3 Case of single-instance applications

The purposed service is the same for all users, access rights at the data level are enough.
Exemple: safety box

ApplicationType = INSTANCE
InstantiationType = TENANT
parentApplicationId = null

5.7.4 Case of multi-tenant applications

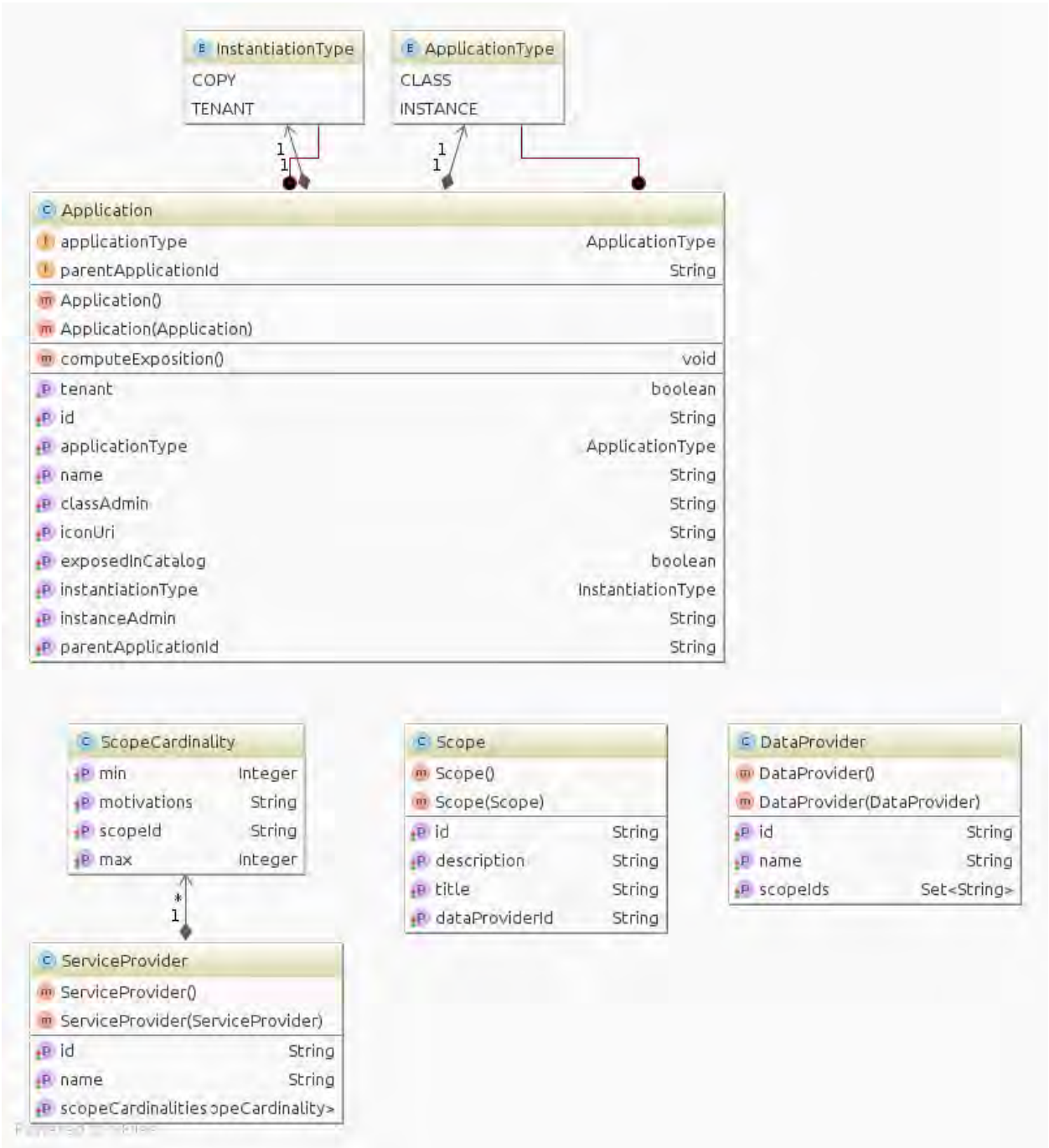
Each organization have its own version of application in a logical view, but the same technical infrastructure is used in practice
An access authorization to organization instance don’t give rights to instance of other application.

ApplicationType = INSTANCE
InstantiationType = TENANT
parentApplicationId = id de l’application parente, appartenant forcément au catalogue statique

5.7.5 Case of multi-instance applications

Each instance is technicaly deployed on a different infrastructure, with differents technical parameters.

ApplicationType = INSTANCE
InstantiationType = COPY
parentApplicationId = id de l’application parente, appartenant forcément au catalogue statique


InstantiationType
COPY
TENANT
ApplicationType
CLASS
INSTANCE
1
1
1
1
Application
applicationType ApplicationType
parentApplicationId String
Application()
Application(Application)
computeExposition() void
tenant boolean
id String
applicationType ApplicationType
name String
classAdmin String
iconUri String
exposedInCatalog boolean
instantiationType InstantiationType
instanceAdmin String
parentApplicationId String
ScopeCardinality
min Integer
motivations String
scopeId String
max Integer
*
1
ServiceProvider
ServiceProvider()
ServiceProvider(ServiceProvider)
id String
name String
scopeCardinalities opeCardinality>
Scope
Scope()
Scope(Scope)
id String
description String
title String
dataProviderId String
DataProvider
DataProvider()
DataProvider(DataProvider)
id String
name String
scopeIds Set<String>

6 Implementation of the portal

The portal is the OASIS sub-system which will the most develop during the pilot phase : indeed, it is the user interface and this is then with the users feedback that this interface will be refined, or even strongly modified for certain aspects. Indeed, some functionalities are totally new for the users, it was then not possible to consult them upstream : they will have to use these interfaces to make a consistent feedback.

We retained the use of a content management software (CMS) to develop this portal.
The development was given to David Holding who knows Joomla and uses it for all its developments. This choice was then retained.

6.1 Implementation of the Technology

The technology platform on which the portal is being developed is CMS Joomla version 3.X released on November 06, 2013.
Joomla is written in PHP, uses object-oriented programming (OOP) techniques (since version 1.5) and software design patterns, stores data in a MySQL, MS SQL (since version 2.5), or PostgreSQL (since version 3.0) database, and includes features such as page caching, RSS feeds, printable versions of pages, news flashes, blogs, polls, search, and support for language internationalization.
The Joomla! Framework is a framework for writing web and command line applications in PHP. It is a free and open source software, distributed under the GNU General Public License version 2 or later.
A Joomla template is a multifaceted Joomla extension which is responsible for the layout, design and structure of a Joomla powered website. While the CMS itself manages the content, a template manages the look and feel of the content elements and the overall design of a Joomla driven website. The content and design of a Joomla template are separate and can be edited, changed and deleted separately.
Joomla extensions help to extend the Joomla websites' ability. There are five types of extensions for Joomla: Components, Modules, Plug-ins, Templates, and Languages. Each of these extensions handles a specific functionality.

6.2 Oasis Interface

Joomla is a multi-interface environment and allows embedding of different types of interfaces. The interface of the Oasis's portal is developed and implemented as a template and will be based on the selected communication methods like XML JSON RPC и REST.

6.3 Interaction with other parts of Oasis kernel

6.3.1 Authentication system

Authentication in Oasis is based on Oauth2/ID mechanism. In Joomla an appropriate framework to use the Oauth2/Open ID authentication is implemented. In fact Joomla can be associated with arbitrary external systems like Google, Facebook and more to check the user's rights. When the user logs to the Web portal, OASIS kernel will check the validity of the user and his rights and the result will return to the portal.

6.3.2 Data transfer

Joomla saves the data in the SQL databases, but Oasis works with No SQL database. For this reason, the Web portal will work with Oasis's data through the kernel mechanism of Web services. For Joomla developed framework with the appropriate API functions that implement XML JSON RPC. The communication with the data will be done with the kernel of these mechanisms.

7 Infrastructure

7.1 Infrastructure implemented for the pilot

From the start of the pilot, the infrastructure in place will test the target architecture, even if the load at startup would be satisfied with a lighter infrastructure.

We implement two datacores:

- A specific datacore for the data of the Italian communities, hosted and managed by CSI, in Italy
- A datacore for other data, under the responsibility of Pôle Numérique, and hosted by IPgarde, a French operator of private cloud

The kernel, the portal, and some services will be hosted on the private cloud provided by IPgarde.

We use three datacenters:

- CSI datacenter
- Two datacenters of IPgarde, in France (Lyon and Bonneville)

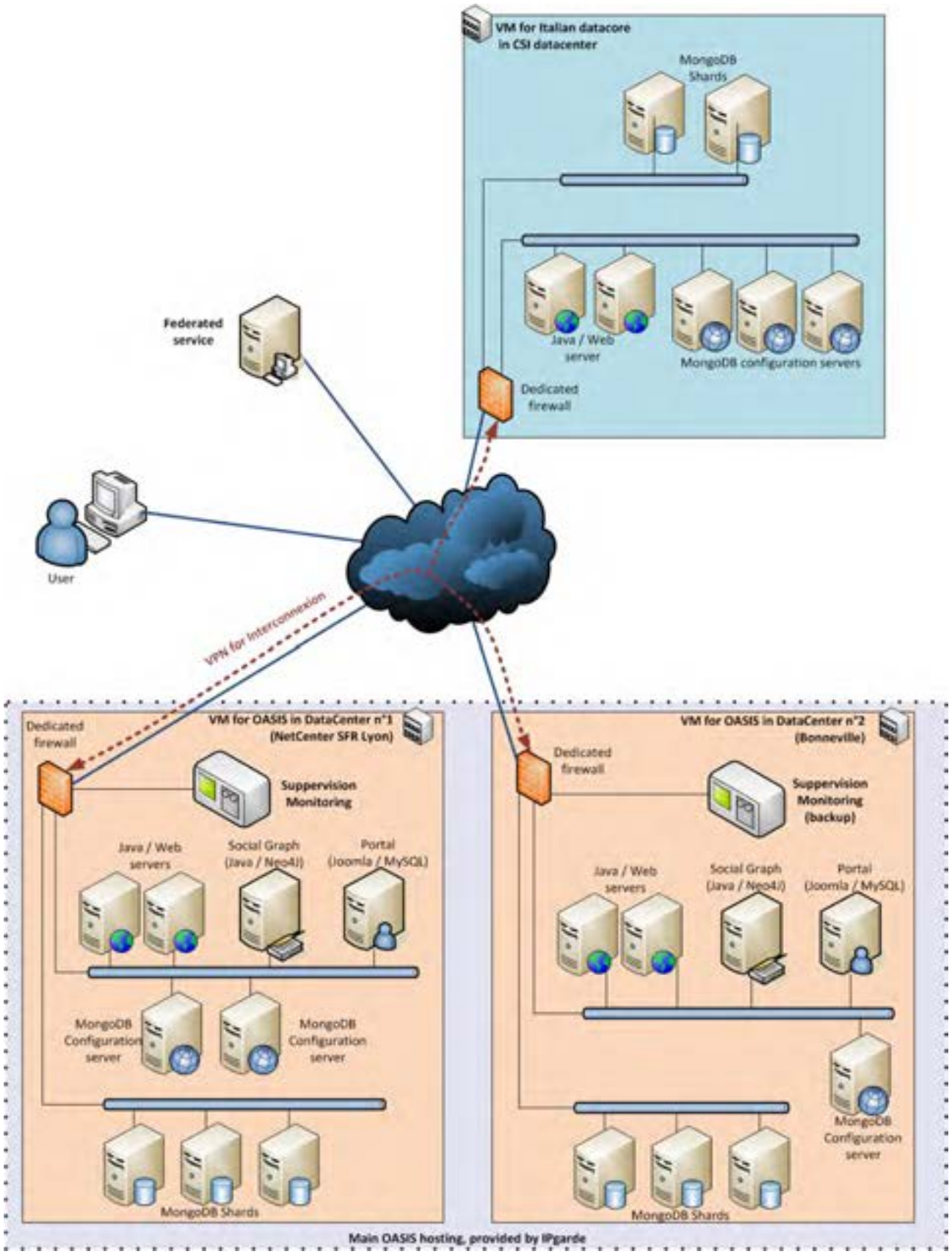
The infrastructure hosted in France allow redundancy and load balancing between the two datacenters.

All servers are virtualized (see below the description of virtualization infrastructure of each provider).

We use a Linux operating system (64-bit Debian stable distribution) for each server.

The diagram on the next page represents the virtual servers we implements on the cloud architecture provide by CSI and IPgarde.

We describe further the physical architecture used in each datacenter to support these virtual servers.



Virtual servers for OASIS

Firewall:

In each datacenter, OASIS infrastructure is protected by a firewall. We entrust this firewall to cloud provider, and therefore implement its usual solutions (see below).

The communication protocol used is HTTPS: for reasons of safety and reliability, we do not want to implement decoding of encrypted packets in firewalls. Therefore the filters cannot be at level 7 (application level).

Server configuration, at the operating system level and at the web server's level, will be especially secure.

The program code is written so as to avoid gaps (especially code injection).

Backup:

A pilot phase is particularly sensitive: beyond data replication mechanisms integrated into the software architecture, and recovery mechanisms provided by virtualization middleware, a daily backup of servers will be provided by each cloud provider.

Load balancing:

We implement a load balancing between the two data centers, using geoDNS features : DNS servers (that is to say the servers that convert URLs into IP addresses) use the client IP address to determine which server address it will give in response (usually a server in the fastest to reach data center). In case of partial or total failure of a data center, the calculation rules are updated to provide only operational addresses.

The chosen solution is an active / active mode (all data centers are used) with performance optimization.

This solution implements N active datacenters: clients are supported by the nearest.

In case of failure of one of the datacenters, clients are automatically redirected to another (second closest). If the servers from the nearest datacenter are overloaded, the client may be referred to a more remote datacenter, if the solution "judge" it necessary, according to several criteria. This ensures a good level of performance.

The performance assessment takes into account several criteria:

- Geographic location of the source (the client)
- Servers load
- Network latency

At the entrance to each datacentre, load balancing built into the infrastructure of the cloud provider can distribute requests to different servers.

7.2 IPgarde Infrastructure

We rely on IPgarde, a provider of private cloud, a French company, using its own infrastructure, hosted in datacenters in France.

IPgarde provided us a private cloud in IAAS mode (Infrastructure As A Service).

IPGarde has its own technical infrastructure, including:

- More than 450 physical servers
- 12 SAN for more than 150 To of data
- 30 42U bays
- a backbone with a capacity of 40 Gbps

Computer, networks and telecommunications equipments are present in 4 datacenters:

- Paris
- Lyon
- Geneva
- Bonneville (with dark fiber to Geneva)

We will use for the OASIS project, Lyon and Bonneville datacenters, located in the Rhone-Alpes region, France.

IPgarde uses 7 transit operators to ensure the best connection to Internet of its facilities in all circumstances:

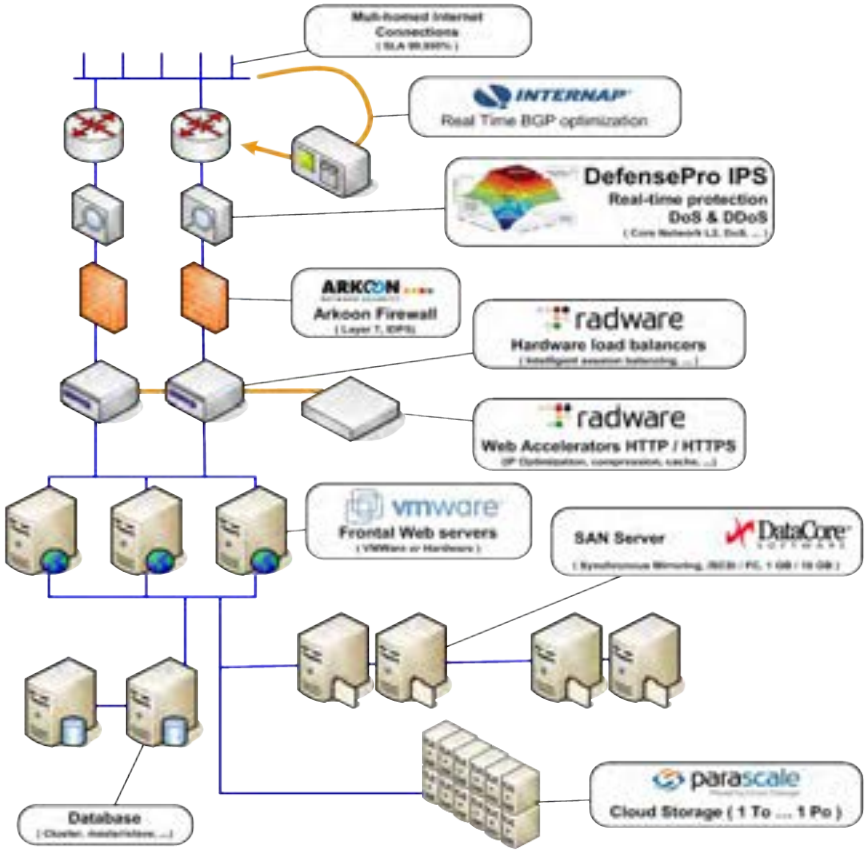
- Level3
- Interoute
- 9Telecom - SFR
- Cogent
- GlobalCrossing
- Highwinds
- Telecom Italia

The traffic is optimized, and switched automatically in the event of failure of an operator.

7.2.1 Global infrastructure

IPGarde has built its platform by selecting components and binding agreements with companies recognized for the quality of their products:

- Intel severs and processors
- Datacore storage solutions
- VMWare for virtualization
- RadWare for intrusion detection and load balancing
- Juniper for IP infrastructure
- AhSay and Veeam for backup



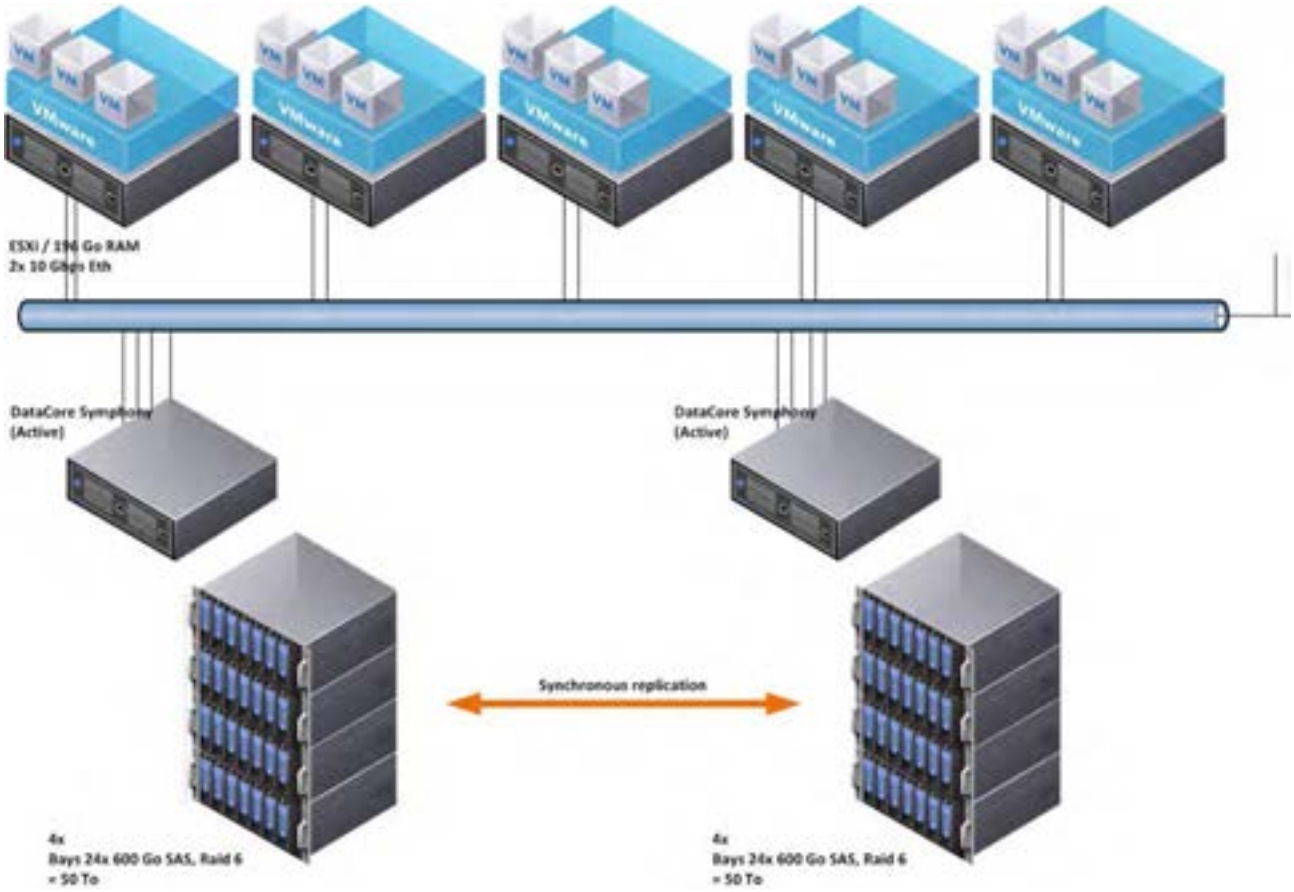
7.2.2 Load balancing

Load balancing is managed by AppDirector of Radware. They allow to distribute requests across multiple servers, focusing on the least loaded server in order to ensure an optimal response time. In case of failure or loss of a server, the AppDirectors redirect requests to available servers, playing a high-availability service.

7.2.3 Storage

Storage is a sensitive issue for cloud infrastructure requiring high performance.

In each data center, storage is supported by two “DataCore Symphony” SAN, using LSI bays. SANs are configured in RAID 6, active / active synchronous replication.



Ultrafast caching

- Accelerates disk I/O
- Uses a x86-64 processor and memory from the storage nodes as powerful mega-caches. Up to 1TB of caching memory can be configured on a node, allowing it to run at the speed of an SSD disk
- Anticipates reading of the following blocks and groups writings to avoid waiting on disks

Interleaving

- Bypass disk failures
- Distributes the I/O across multiple arrays
- Discharge RAID 6

Load balancing

- Overcomes the usual bottlenecks due to storage
- Distributes the load on physical devices using different channels for different virtual disks
- Automatically bypasses defective or offline channels

Performance Analysis

- Performs monitoring and analyzes the behavior of the SAN in real time
- Helps to decide for setting and development
- Indicates potential bottlenecks

Synchronous mirroring

- I/O Replication in real time for high availability
- Removes a SAN or storage as a single point of failure when combined with multipath I/O drivers
- Enhances survivability using physically separate nodes
- simultaneously maintains two distributed copies
- Mirrored virtual disks behave as a single shared disk with several ports

Continuous protection and restoration of data

- Rewind at any time without explicit backup
- Log and set timestamp for all I/O on the selected virtual disks;

7.2.1 Description of the backup solution

Solution # 1: disk-level

IPGarde used for this backup Veeam technology that doesn't require any agent installed on the virtual server. Data deduplication and compression are implemented to reduce the consumption of disk space.

The "image mode" backup allows recovery VM faster and less error-prone, while restoring files can quickly instant recover a single file directly from a image, regardless of operating system or system files.

Moreover, the vPower technology enables running a VM directly from a backup file compressed and deduplicated stored on remote backup system.

Solution # 2 : file-level

A second backup is proposed using an agent (Ahsay) installed in the server. It allows to select the candidate files for backup.

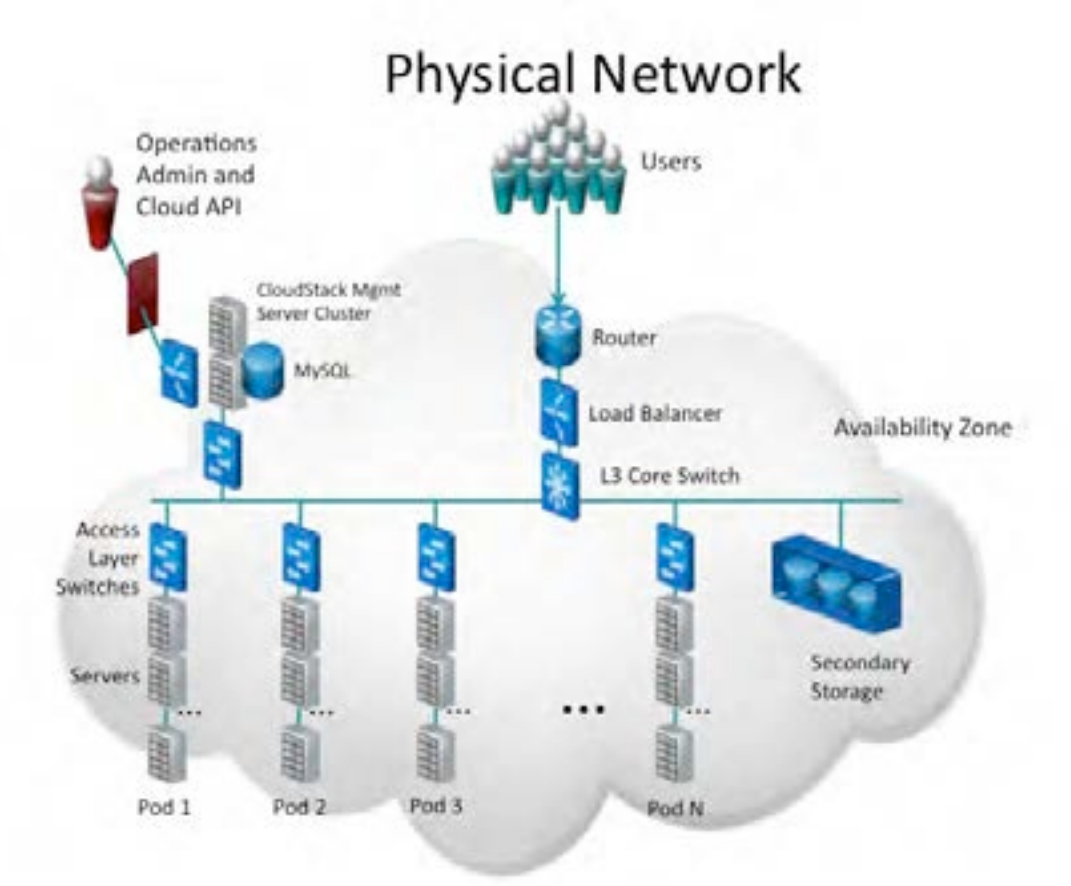
7.2.1 Security

IPGarde has implemented all of the following solutions to ensure the security of applications and data:

- Firewall dedicated to the platform (Fortinet)
- IDPS Radware DefensePro with signature detection and behavior analysis
- Automated weekly scan weekly of vulnerabilities and generating a report
- Encrypted VPN (with personal certificate) for the administration
- Load balancer with protection against attacks (Sync ...)

7.3 Infrastructure CSI

The hosting infrastructure is CSI Cloud Laboratory's Cloud IaaS (Infrastructure as a Service) based upon Citrix Cloud Platform open source solution. The virtualization hosts, within the Cloud, are CentOS 6.3 Linux servers with KVM.



General CloudPlatform diagram

The IaaS Cloud provides plenty of elasticity and scalability, with the possibility to add nearly unlimited computing and storage resources on the go. Internal network bandwith can be allocated as needed up to 1 GB dedicated per hosted machine.

The public networks available to CSI Cloud Laboratory are directly connected to a regional Internet Exchange (Top-IX - http://www.topix.it/) with a redundant 10 GB bandwith peering link.

Both the physical and the virtual infrastructure of the Cloud is completely redundant with automatic failover systems if any of the component systems should fail.

A complete backup of all infrastructure is made weekly. All hosted systems and storage are backupped daily.

A double security layer is available.
An internal filtering and traffic control system is available on the Cloud system, composed of Virtual Routers and Firewalls.
Additionally, a traditional Firewall system and network organization (private public, public DMZ, unprotected public) has been implemented to protect the Internet-side infrastructure.
All systems of the OASIS project, both inside and outside the OASIS system are hosted on protected networks.

8 Supervision

The supervision of the OASIS platform (datacores, kernel, portal, services hosted on OASIS cloud) is based on several tools and features:

- Automatic monitoring of resources and software services
- a measure of the performance and resource utilization
- An analysis of the logs provided by the dedicated module

These tools allow:

- To analyze the functioning of the platform
- to predict load variations and incidents
- To receive alerts in case of failure of a component (hardware or software, even if the failure does not impact the provided service)
- To investigate incidents

We use the monitoring tools of the cloud provider (IPgarde), which we add specific probes to monitor the OASIS software components.

A dedicated server (with IPCheck software, based on free software PRTG) is set up to monitor the complete infrastructure (technical and application).

IPCheck saves settings network and servers settings. The data are stored in a database to build the historical reports requested.

The user interface (web-based) allows real-time monitoring the health of all servers, network equipment and software services, and the use of resources.

The kernel module “LOG” also provides specific indicators (KPI), necessary for the evaluation of the pilot, who also put together on Ipcheck (via SNMP) to be monitored more easily with integrated graphics tools.

IPCheck includes more than 100 sensor types for all common services (eg Ping, http, smtp, pop3, ftp ...) which allows to monitor all services and immediately detect a failure or slowdown.

9 Federated services

OASIS is designed to federate applications providing « à la carte » functionalities to share data, improve the applications and unify the user experience.

OASIS is based on a full web REST architecture, with a weak coupling between the applications and the services supplied by OASIS.

The federated applications then use their own technologies both for the development of the softwares and for their hosting.

Each provider's choices are linked to his internal standards and skills, to the organisation of the operation of the services and to his business model.

Within the context of the pilot, the providers wished to keep their usual hosting solutions without applying specific modifications to OASIS.

This gives the advantage of evaluating several technologies for the pilot.

We describe below the various platforms implemented by the project's partners.

The following table shows on which platform each service is hosted for each pilot site:

Service	Pilot site	Hosting
Archiving	Pôle Numérique	Axecio (Locarchives)
Citizen portal	Pôle Numérique	IPgarde (cloud OASIS France)
Citizen portal	Province de Turin	CSI (Cloud datacore Italie)
Citizen portal	Blau	IPgarde (cloud OASIS France)
Ushahidi	Pôle Numérique	IPgarde (cloud OASIS France)
Ushahidi	David Holding	David Holding
Suite OpenMairie	Pôle Numérique	Atreal / OVH
Suite OpenMairie	Blau	Atreal / OVH
INVPROM	EMDA	Microsoft Azure
Data Collection	EMDA	Microsoft Azure
City Planning	Pôle Numérique	CSI
City Planning	Province de Turin	CSI
Mapping activity	Province de Turin	CSI
Mapping activity	Blau	CSI
Opendata	Pôle Numérique	IPgarde (cloud OASIS France)
Alternative Tourism	David Holding	David Holding
Financial management	Pôle Numérique	IPgarde (cloud OASIS France)

9.1 Axecio (Locarchives)

The legal record keeping requires the use of a certified datacenter. Since this is a long process, we have decided to ask a specialized supplier for the pilot.

This provider is Axecio and his trusted archiver third party service, Locarchives.

This service is certified NF Z42-013 (trade certification) and SIAF-accredited (certification for the storage of physical and digital archives).

This company relies on security device consistent with OASIS and meets the NFZ42-013 and SIAF standard requirements (Redundancy on a secondary site (localisation of two sites : Ile de France and Picardie)

- Migration of the storage supports within the framework of the infrastructure management process.
- Guarantee of the integrity of the electronic archives.
- Securing and traceability of the access when entering and communicating
- Logical division of the containers for archive storage at the level of front and back office.
- Disk and tape back-up

The availability rate is higher than 98%.

9.2 CSI Infrastructure

The services supplied by CSI are hosted on the same infrastructure than the Italian datacore (see chapter 7).

9.3 Infrastructure de David Holding

David holding is an IT company specialized in the production of software. For his own purposes David Holding operates with two sites – public and private. On the public site is located the official sites of David Holding and product lines, and private site serves the development, testing, backup, and engineering activities. Both sites can be used for OASIS's applications.

1. Public site – collocation in Digital Systems Inc.

Server: IBM X3550M4 – Xeon E5 – 2609 / 2.4 GHz 2 processor system, 12 GB RAM, RAID 5 - 600 GB, Operating system – Linux Ubuntu Server 10.4, LAMP server

Upgrade options are:

- Internet connection - Up to 1GBps
- We don't plan a hardware upgrade on the server

2. Private site – David Holding's self-hosting system

Blade Centre H - Xeon E5 – 2609 / 2.4 GHz 4 processor system, 48 GB RAM, RAID 5 – 1TB Host OS – Microsoft Windows 2012, Hyper V – virtualization, Guests systems: Linux Ubuntu 10.4 Server, LAMP server

Upgrade options are:

- Internet connection - Up to 1GBps
- RAM – Up to 256 GB
- Disk – Up to 3 TB
- CPU – Up to 8 slots

Disaster recovery plan:

David Holding has licenses: ISO 9001:2008, ISO 27001:2005, ISO 20000-1:2011

All data management, security and recovery activities are described in internal procedures.

Protection against attacks:

1. Public site

Cisco based IDS/Firewall – from Digital System
Linux based IDS/Firewall – on Host Server

2. Private site

Mikrotik and Linux based IDS/Firewall

Supervision

Digital systems operates with 24x7x365 Network Operating Centre – core network support.

David Holding has a Help Desk for their customers, which works at 8x5x4, 8x5x6 and 8x5xNBD modes depending on the defined SLA – customer software support.

9.4 Infrastructure Microsoft Azure

EMDA work with the cloud Microsoft Azure:

The Windows Azure platform runs in multiple datacenters around the world. Customers can and are encouraged to export their data in Windows Azure SQL Database to multiple datacenters. In the event of a catastrophic failure involving an entire datacenter, a customer could deploy their application at a backup location.

Each layer of the Windows Azure platform infrastructure is designed to continue operations in the event of failure, including redundant network devices at each layer and dual Internet service providers at each datacenter. Failover is in most cases automatic (requiring no human intervention), and the network is monitored by the Network Operations Center 24x7 to detect any anomalies or potential network issues.

This hosting solution is not strictly under European legislation but the services deployed on the Turkish pilot site do not manage any personal data.

9.5 Infrastructure utilisée par Atreal (chez OVH)

OVH is the hosting company of the services provided by Atreal.

OVH detains 4 data-centers, connected each other with dedicated links.

We decided to host our servers at the Strasbourg and Roubaix data-centers.

Each data-center is highly secured. The access is strictly controlled and all the server system maintenance is done by us distantly.

Servers :

We did our own server installation and configuration. Our goal was to provide a better security process than the one provided by common hosting services. Our architecture is based on security, performance, flexibility and reliability.

Our OS is GNU/Debian. The reason of that choice are :

- the overall performance of GNU/Linux
- the stability of GNU/Debian
- the community driven governance
- the constant quality of the packaging
- the high concern on security
- the constant quality of major upgrades

For more informations about GNU/Debian : http://www.debian.org/

Each server is installed as a host of virtual machines (see next diagram). It has no publicly exposed service.

The goal of the main server is :

- to host virtual machines, that actually do the job

- to filter the traffic, between virtual machines and between the Internet and virtual machines
- to provide a maintenance access from atReal IP, with a certificate by user for authentication

Virtualization :

We have tested many virtualizations solutions and have chosen linux-vserver.

We plan to switch to its successor LXC as soon as its security issues will be completely fixed into the linux kernel.

Linux-vserver is a virtualizations based on context isolation. That implies that there is no virtualized kernel : one kernel is used for all the virtual machines. This has a benefit in performance and a better optimization of physical resources usage.

We compared the resources usage of the same virtual machines using Linux-Vserver, Xen and KVM. We got an increase of performance by a 4 factor using linux-vserver, mostly because there is no need of virtual filesystem and because a library used by several vservers is loaded only once in RAM, with an execution context per server, using linux-vserver.

Our latest benchmark invites us to switch to LXC, that is implemented on the same principles as Linux-vserver, in a more recent implement. LXC is integrated in the linux kernel, but it lacks maturity and some parts of the linux kernel are not yet properly isolated by the context jail.

Storage :

Each server has its own storage, in RAID 1 (mirror). Physical volumes are handled by LVM (Logical Volumes Manager). Each logical volume is formated in EXT4 for applications and data.

LVM main advantages are :

- allow to freeze a logical volume filesystem in order to have consistent hot backup
- allow to add physical disks to logical volumes : it is therefore possible to increase a volume size depending on needs
- performant
- robust

Each physical server has its own security hosting infrastructure.

An IP is dedicated to maintenance and only accessible from a specific IP with individual certificates authentication. Each maintainer has its own certificate.

This connexion is the only way to access the system for maintenance.

All the traffic is secured by the NetFilter firewall integrated to linux kernel.

Each public IP is secured by an anti-attack global protection infrastructure provided by OVH in its main architecture.

Only services from Vms are exposed to the web. They are reverse proxies that forward the traffic to application servers.

We may use backend servers for databases or other non exposed internal services.

The virtualization system used is linux-vserver. It is a context isolation system. It is extremely robust and well tested. Its best advantages are the flexibility and optimization of hardware resources.

It may be replaced by lxc in a close future. This is not done yet due to a lack of maturity.

Each server has its own RAID1 storage.

A backup is done daily to a server located in another data center. This backup may be restart in a 1 our delay.

The monitoring system is located outside of the production environment, in a data center.

10 Conclusion

This closes the chapter of Cloud infrastructures that have been chosen to be deployed in OASIS. Note that their deployment architecture, including hosting chocies, will be assessed at the end of the project so that they will be able to be adapted, improved and scaled up in post-project phase. This assessment will be done by the OASIS Technical Governance Committee, according to criteria and process exposed in deliverable D3.4.

The implemented architecture provides a number of tools to service providers to connect their applications to the platform. Available APIs for this adaptation ared described in the deliverable D2.2.

Authentication and data security in OASIS are dealt further in deliverable D2.3.