Deliverable D2.2

Guidelines for designing and developing Pilot B services

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_D2.2
Status	Final

Dissemination level[1]	PU
Nature[2]	R

Due delivery date (project month)	M10
Actual delivery date	16/02/2014

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

Revision History

Revision	Date	Author and Organisation	Description[3]
0.1	19/12/2013	Sylvain Chambon (Open Wide)	Creation
vf	17/02/2014	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This deliverable contains original unpublished work except where clearly indicated otherwise.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

This document aims to give all necessary and sufficient information to service providers willing to participate to the OASIS system, in particular with regard to what effort is necessary on the providers' part to adapt or integrate their (possibly pre-existing) services to the OASIS federation.

It is, deliberately, a very concrete document based on the current state of OASIS kernel development, so that service providers are provided with an accurate, up to date view of what "integrating into OASIS" entails. As such, this is a living document: all kernel services that are slated for inclusion in the target OASIS platform are not described, and all services described are not necessarily the definitive ones. Only the OASIS platform "as-is" at the date of authoring this document is described – this means that some APIs described here may be eventually phased out as they are replaced by more sophisticated APIs as the latter are implemented.

This document will be updated on a regular basis to reflect the evolution of the OASIS platform. However the most up-to-date information is always the auto-documenting APIs on the kernel test servers. The kernel and data core APIs use Swagger to provide a continuously up-to-date snapshot of the API documentation.

For these reasons, this document focuses more on explaining the concepts underlying the APIs than describing these APIs. In any event, the online Swagger documentation should always be any developer's reference for implementation of services.

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 Scope of OASIS

OASIS is :

- a federation of services
- an interoperable database, which is a common good of data
- an advanced social network
- a smart and user friendly graphical user interface

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

OASIS provides applications joining the federation with advanced « à la carte » services so that they can share data (and then contribute to the creation of a common good and benefit from it), improve user experience (portal, SSO, centralized notifications), and improve the application with the help of the services supplied by the OASIS kernel or made available by the federation.

1.2 Description of the document

This document describes how to use the services supplied by the kernel, the datacore and the OASIS portal so as to adapt an OASIS existing application or to develop a new application based on OASIS.

This document is for the pilot site's providers who then take part in the definition of governance. It corresponds to the version corresponding to the pilot where the functions concerning the providers themselves (as the catalogues management and the implementation of governance rules) are still to be evaluated and are then mainly manual.

This document mainly presents the software related aspects of the integration of one service into OASIS.

This document presents exclusively the software aspects of integration of a services into OASIS, including the following parts:

- a presentation of services provided by OASIS to federated applications (chapter 2)
- an approach of the ameliorations that OASIS services provide to applications (chapter 3)
- a description of different services' API to allow their utilisation (chapter 4)

2 Services supplied by OASIS

This chapter presents the services supplied by OASIS for the providers services so as to develop or adapt applications.

2.1 The portal

The portal does not provide API for the federated applications. But its role is particularly important for no implementation in the applications is required for the functionalities it provides to the user.

- The portal allows the creation of accounts and the directory function. (management of groups via the social graph : see next paragraph).
- The portal gives the user the possibility of setting his environment, managing his personal data and managing his preferences for the reception of messages and various notifications.
- For the software suites integrating several software, they can be divided, the portal then acts as the main menu
- The portal gives the user the possibility of consulting the history of his use of data and services, revoking access rights and managing the security of data.

To offer a unified and fluent user experience, it is particularly recommended to take into account the fact that they have access to application through the portal, in order to assure the best continuity between the portal and the federated applications.

2.2 The authentication

The authentication module assures a unique authentication of users, and a management of the access (with a mechanism of authorisation tickets) to different resources of OASIS' federation: web services, data source, social graph and personal data.
This is a unique and simple mechanism of security management for all applications.

2.3 The social graph

The social graph is an important innovation of OASIS.
It allows each user to organize its environment, its data and the sharing of data, around its social relationships with other people or organisations (companies, administrations …).

The social graph is, in the same time:

- The OASIS history (in sense of records centralized, used in information system of organisations, but focused on each user and extended beyond the strict relations in an organisation).
- Data allows to create functionalities based on social networks
- The storage service and the security management of personal data for users

In this respect, the social graph provides API for:

- accessing to directory's information (particularly for management of users and organisations' groups)
- accessing to relations between each user and the rest of the world (with organisations, legal or natural persons), with a qualification for those information to be able to exploit them (for example, a company has a different relation with its employees and its customers).
- accessing to users' personal data (if this one gave before the authorisation)

According to the level of the application integration, the social graph can allow the total management of users' profiles in services, centralized, for all user applications.

2.4 Datacore

Datacores are data sharing services between users and between services themselves.
It is, with the Social Graph, an important part of OASIS: datacores allow to define precise governance rules on data, and to guarantee their respect, inciting users to share data, and so building a common good of data.
Even if the access is limited, the fact of stocking data into the social graph gives the possibility to open them (when it will be accessible) and also to share them between applications.

Datacores offer mechanisms for the management of advanced access rights (to make the collaboration easier), mechanisms for the quality management of data, and mechanisms of mediation on data (mediation a priori or a posteriori)

2.5 The communication module

The communication module offers a unique API to communicate with the user, the OASIS kernel dealing with the management of user's preferences for its messages reception (by displaying on the portal, by sending an e-mail as they happen or as a daily report, by sending by SMS, etc..).

This module avoids applications to ask the user's e-mail address, which is an information considered as sensitive. It allows a personalised and simplified with users, in absolute respect of personal data, avoiding the manipulation of address files, avoiding distribution mistakes and the establishment of unsubscribe modules.
The utilisation of this module is strongly recommended because it enables the user to control its messages (making them more efficient) and avoids asking an e-mail address (which can stop a

user or incite it to give a temporary address).

2.6 Notifications and status

To improve the user experience and the fluidity between the different applications in a same functional process, the OASIS kernel provides a notification service (allowing the notification of an event to another application, without knowing it) and to notify status, which could be watched by applications and by the portal and for which changes could be notified to concerned users.

2.7 The open resources (broker)

The broker, kernel module, allows the consultation of OASIS catalogue, which means all the resources federated by OASIS, and particularly the data sources, the data scope (including governance rules), the web services and the applications.

2.8 The store

The OASIS portal includes a store, which allows to broadcast easily the applications, giving the possibility to the user to find them quickly and to subscribe to it.

3 Improve applications with OASIS

OASIS gives a lot of functionalities which allow the improvement of services to the benefit of the user (beyond the simple federation and the management of the services by the portal).

Especially, OASIS guides the process to what the users expect, both in personal use and professional use (IT or functional manager).

Indeed, heavy and confirmed trends of IT system are:

- the proliferation of information, which no longer allows a top-down distribution of the information, require that everyone be, at the same time, producer and consumer of the information, that the information be shared directly between the people concerned, and that everyone has the tools to search and to pick out the pertinent information for him.
- the massive arrival of mobile phone equipments, which reduce strongly the borders between personal and professional tools (which is also a challenge for the security).
- the social and collaborative dimension of all activities: CRM utilisation, professional social networks, forum, wiki, social networks (like LinkedIn, Facebook, Twitter) for company jobs (communication, recruitment, search of suppliers, search of customers, monitoring).
- a strategic importance of data, which give a quick access for everybody, becomes more important than the absolute exactitude, et for whom the update isn't possible anymore at the level of one organisation.

Those trends are complex to manage.
The jobs process are still based on data validation trips, an isolation of the information (mostly for confidentiality and integrity reasons)

OASIS gives all tools to advance the services towards news trends, in security and easily.
And, it is the fact that services advance towards those trends that give to OASIS all its sense for services providers.

3.1 The social graph instead of the directory

The common organisations' directory is replaced by the social graph (which has a lot of other functions).
The principal difference is that a directory (type LDAP) has more a hierarchical vision of organisations, when the social graph allows to anyone to have a vision focused on him.

The groups' management allows to write an eventual hierarchical model, but also extensive models (superimposition of organisations, eternal partners integration, project groups, etc...)

The API presentation of different relations, in the form of groups, allows to simplify the use of the graph for the services which don't need to exploit the nature of the links.

The management of contexts by the user allows to services not to worry about what the user evaluates pertinent in his work context.

The management of personal data rights, including habitual data of a directory, confided to each user, allows also to use exactly and effectively the access rights to its data (when that is complicated with a directory which has a centralised management)

For a good integration to OASIS, we advise the applications to strongly rely on the OASIS social graph, without searching to import data in a local directory.

3.2 The social graph for social functions

The social graph allows to centralise all information on users, organisations and relationships between them.
This data is not in an application owner format of a data social network.

That permits to build social and collaborative specialised applications, which have, as soon as they start, all relationships described by OASIS users, without having to create a lot of import API and researches.
The social graph offers all functions giving to users the possibility to master their data and to secure them: the services which rely on the graph can therefore focus on functionalities with added value they want to give, respecting the legislation and people rights.

Of course, for that, that's imperative not to stock other data bases managed by the graph.

3.3 The data kernel for the data sharing

The OASIS data kernel is at the disposal of all services.
It allows to stock data, guaranteeing the governance rules, which will be defined by those data producers. By giving this guarantee, it improves the confidence and makes easy to everyone the opening and the data sharing.

The linked and reusable models allow to converge toward an efficient data sharing, without a heavy hand in hand interface, and without any connexions to interoperability hubs.
Services can evaluate, giving to users a way to have available more data, and to search a lot of data linked to the current work.

The data quality mechanisms and of data mediation allow to manage imperfect data and to improve them progressively.

3.4 Tools to communicate

The notifications mechanisms, the status management and the communication modules gives to services simple tools to communicate, without managing bridges of communication and without having to organize configuration interfaces, more or less complexes.
Those tools being used by different services; they are still updated. Changes of other applications are open, the way to communicate with the user are still updated.

4 Participating in the OASIS ecosystem

4.1 Prerequisites

"Participation" (or "integration") in the OASIS ecosystem is a generic phrase that maps to a number of actual use cases. Application developers are free to use any or all of the various services that OASIS provides, and that we shall expose in this document; however there is a minimal integration level that is absolutely mandatory for any other uses. This is:

- The service provider and the application are declared in the OASIS broker;
- The application is configured to use the OASIS authentication and authorization framework.

In this respect, the absolute minimum prerequisites for OASIS integration are:

- Have (or create) a web application;
- Register this application and its service provider in the OASIS catalogues, in order to have an application identifier, a service provider identifier and a service provider password (or "client secret") for authentication.

A note on terminology

The OASIS kernel uses the following definitions, and so shall we in this document:

- an *application* is your service's overall container entity, it is what your users think of;
- a *service provider* is your application's main entry point; think of it as the user-facing part of your application;
- a *data provider* is a REST interface provided by your service; think of it as the rest-of-OASIS-facing part of your application.

An application may have no service provider (if it is a purely technical service, for instance exposing raw data that other OASIS applications consume[4]), but it cannot have more than one. Also, an application may have as many data providers as it needs (from zero, if the service is purely consumer-facing or store shared data in a OASIS datacore, to many).

Also note that deployment of an application is not particularly restricted. In particular, an "application" in the OASIS kernel sense is not particularly constrained to be deployed as one instance on one server. Applications may be "multi-tenant" − i.e. a single server deployment serves multiple logical application contexts.

[4] Note however that a better-behaved way of sharing data is to model it and export it to the data core.

4.2 Collaborating with the data core

4.2.1 Overview: data models

“Collaborating” with data means either or all of the following use cases:

- Discover and query shared data stored in the OASIS data core;
- Enrich that data with additional information;
- Create and publish new data.

We shall see how to do all of this; but first one needs an understanding of the data core modelling principles.
First off, it is important to note the OASIS data core is not a relational database - it is a web-friendly REST API. Do not expect there to be typical relational features such as complex queries involving several entities (“joins”). Instead, the data core offers a great level of flexibility in data modelling as well as high-level features such as automatic versioning, auditing, and security constraints.

Secondly, in the OASIS data core, individual items (analogous to a relational “row”, though richer) are accessed (created, queried, etc.) through their main model. This is not to say that data cannot have additional facets or are bound to a single type; but in OASIS version 1, all data manipulation is performed in the context of a model.

So, what is a model? a model is simply a data “type”, that is to say a description of that data in an interoperable format (encoded in JSON). For instance the following is a sample model shipped with the data core:


```
  "sample.city.city" : {
    "name" : "sample.city.city",
    "documentation" : "{ \"uri\": \"http://localhost:8080/dc/type/city/France/Lyon\", \"inCountry\":
\"http://localhost:8080/dc/type/country/France\", \"name\": \"Lyon\" }",
    "fieldMap" : {
      "founded" : {
        "name" : "founded",
        "type" : "date",
        "required" : false,
        "queryLimit" : 0
      },
      "pointsOfInterest" : {
        "name" : "pointsOfInterest",
        "type" : "list",
        "required" : false,
        "queryLimit" : 0,
        "listElementField" : {
          "name" : "zzz",
          "type" : "resource",
          "required" : false,
          "queryLimit" : 0
        }
```

```
    },
    "inCountry" : {
      "name" : "inCountry",
      "type" : "resource",
      "required" : true,
      "queryLimit" : 100,
      "resourceType" : "sample.city.country"
    },
    "name" : {
      "name" : "name",
      "type" : "string",
      "required" : true,
      "queryLimit" : 100
    },
    "populationCount" : {
      "name" : "populationCount",
      "type" : "int",
      "required" : false,
      "queryLimit" : 50
    },
  },
  "mixins" : [ ],
  "security" : {
    "authentifiedCreatable" : true,
    "authentifiedReadable" : true,
    "guestReadable" : true,
    "authentifiedWritable" : true
  }
}
```


Note: we have voluntarily hidden some technical (non-user-facing) properties of this model.

You can find out about existing models by calling the `/dc/model` API on a data core instance.

So, what can we see in this model?

- it has a **name**. Model names are important; they must be globally unique, and cannot be changed once created. Pick one correctly! While the data core currently does not enforce any particular policies on model naming (aside from the unicity constraint), a good practice is to have explicit names based on several components separated by dots (“.”) The components of a name should be its business domain description, from more general to more specific. In our example we have: “`sample.city.city`” which indicates that the business domain is the city modelling sample, and that the actual model is about cities (other models in the same business domain are countries, points of interest, etc.)
  Note that if your data model is thoroughly bound to your application, it may make sense to include your application name in the model name. However, we discourage this approach, as this tends to preclude collaboration on the data.
- it has **documentation** − free text that can be used to describe and query data models.

- it has **fields**, the data actually stored (more on this later).
- it has **mixins** : these are facets, groups of fields that tend to stick together and the definition of which can be reused for multiple models − for instance, an address could be well-suited to being a mixin, since it is (schematically) made of several fields, inclunding a post code, and a city name.
  Note: there are specific API endpoints for finding out and manipulating mixins.
- Finally, there is **security information**. OASIS places a strong focus on data safety; both personal data (as stored in the social graph) and interoperable data (as stored in the data core). While the more common use case is to restrict access based on the resource itself, it is possible to set some basic security information on the data type, such as: is this type meant to be public (i.e. world-readable)? Who can create new items of this type?

4.2.2 Linked data strategy

Note : For the datacore, we use the term linked data, but it doesn't mean than data are linked with external data. It's a notion of links inside the datacore.

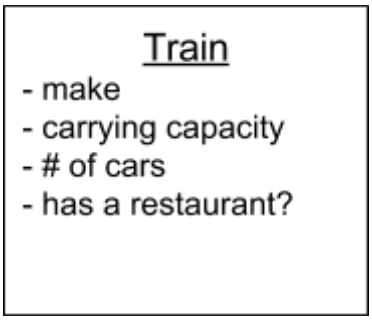
OASIS aims at building a *consolidated* "common good" of data − that is to say, avoid duplication as much as possible. Reuse of data, and especially data models (which together form a conceptual subset of a RDF ontology) is thoroughly recommended.

So whenever you are tempted to "just create" a new model or mixin, first−ask yourself if some existing model cannot be reused instead. That model may not be sufficient for your needs, but that's fine−you can add additional mixins and fields, or better: if what you really need is a derivation (that is to say, an application) of a model to a specific use case, then what you should do is create a new model containing just those few extra fields that you need, then *link to* an instance of the more generic model.

In a slightly contrived example, imagine you are working on an application mapping out the railroads of Europe; for your application you need to model a train. A simplified version of your business model is the following:

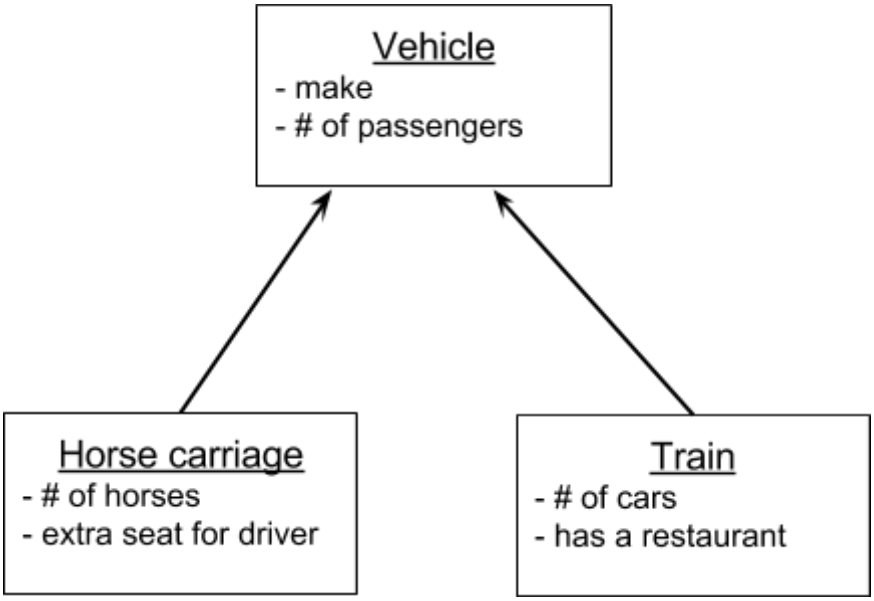
- A train has a make / commercial brand (e.g. ICE, Eurostar)
- It is important to know how many passengers the train carries and in how many cars, as well as whether there is a bar.

So that your model looks like this:



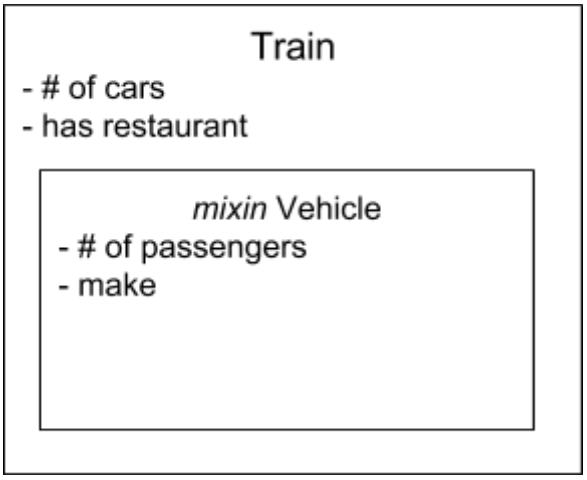
Being a good OASIS citizen, before creating your own model you query the data core to find out about existing models. There is no model for a train, but there is a more general model for a passenger vehicle, defined as something which has a make and a number of passengers.
In an object-oriented world, one would simply *extend* the Vehicle "class" to create the Train "class". However the data core is not objected but rather, it is *link-oriented* (or aggregation-based). So the correct approach is to create a Train model containing a link to a Vehicle (that is to say a field of type "resource" and target type "vehicle"). In effect, one train entity in the application is mapped to two entities in the data core: one Vehicle, and one Train.

In the future, some other developer working on a tourism application may want to add support for "horse carriage rides around the city"-type services. So they need to model horse carriages−no problem, they also derive-by-aggregation the Vehicle model, and voilà.



4.2.3 Mixin strategy

Mixins are another way of reusing data definitions. In our example above, we could have simply used a "vehicle" mixin to add the vehicle fields directly to the train object:



This however is a fundamentally different type of reuse:

- A mixin cannot be instantiated − there cannot be an object which is simply “a vehicle”
- A “train” is not a “vehicle”, rather it has all the “vehicle” properties. Data core objects are not polymorphic; in particular one *cannot* query “vehicles” regardless of their actual type. While you can express a query such as “find me all the trains of make ‘TGV’ that can carry more than 500 passengers” there is no way in the data core to express a query such as “find me all the *vehicles* that can carry more than 5 passengers”. This is the most important restriction to have in mind with respect to mixins.
- Mixins are however more performant than linked data, for two reasons:
    - mixin data is transmitted within the scope of its owning object; so getting the full definition with all fields of one specific train takes just one HTTP request rather than two in the case of linked data.
    - it is possible to cumulate query criteria across mixin boundaries within the same type. In other words it is possible to query for “all trains that can carry more than 500 passengers and have a restaurant” in one query. This makes up for the lack of “join” semantics in the data core APIs.

4.2.4 Modeling best practices and pitfalls

When working with models, keep the following rules of thumb in mind:

- avoid duplicating data or models that already exist elsewhere (“don’t repeat yourself” or rather “don’t repeat everyone else either”),
- link to existing data. If you need a closed list with generic values, for instance for personal titles (“Mr”, “Mrs”, “Dr”, “Prof”, etc.) then use the datacore `model` API to find out whether such a list does not already exist (chances are that it does),
- use mixins for query performance while promoting reuse. However remember that the data core querying API is not polymorphic; if you need polymorphism, use linked data.

4.3 OASIS APIs

4.3.1 Overview - commonalities

There are two basic sets of OASIS APIs:

- the Kernel APIs (authentication, social graph, event bus, etc.)
- the Data core APIs (data access)

Both of these API sets are REST- and JSON-based; the extensive documentation (auto-generated with Swagger) is available on the test servers
Since all OASIS APIs use Swagger self-description, you can use Swagger-codegen (https://github.com/wordnik/swagger-codegen) to bootstrap a client implementation of the API in your language of choice (Swagger-codegen supports Java, Python, Scala, Ruby, PHP, and Flash). However, some additional work on these client implementations must be done to support the security features of OASIS. In this chapter, we shall describe the APIs at a high level and focus on the specificities and pitfalls of using these APIs. We shall strive to provide concrete code examples, though obviously we cannot write examples in all languages for all APIs.

Most importantly, all OASIS APIs are *self-documenting*, that is to say their documentation is auto-generated and always up-to-date. This is the reason why this document is more a guide about the philosophy of each API than an exhaustive API documentation: that can be found on the test servers that can be found at:

- for the Kernel services: https://oasis-demo.atolcd.com
- for the Data core: http://srv2-polenum.fingerprint-technologies.net

4.3.2 Authentication and authorization

4.3.2.1 Overview

The OASIS authentication system is based on the OpenId-Connect protocol, itself an authentication wrapper over the OAuth 2.0 authorization protocol.
Warning: despite its name, OpenId-Connect has little to do with OpenId. Do not try to use the OASIS authentication system with an OpenId client library.

A simplistic view is that OpenId-Connect uses the OAuth 2.0 authorization mechanism to access the user identifier (the “openid” scope) in order to perform authentication. In turn, the application gets an access token that lets it identify it on behalf of the user, to third party services.

- The application still is responsible for managing its own user session (if only to persist the access token).

4.3.2.2 Integration libraries

As the protocol is rooted in OAuth2.0, using OpenId-Connect is really about using an OAuth2.0 client such as:

- Google OAuth Client Library for Java (https://code.google.com/p/google-oauth-java-client/)
- The MITREId Connect project includes a Spring Security plugin for OpenId-Connect (https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server) which is useable with OASIS with a few minor tweaks that we will describe in this document
- The Google API client library for PHP includes an OAuth2 implementation (https://code.google.com/p/google-api-php-client/wiki/OAuth2)
- OpenID Connect Client Library for PHP (https://github.com/ivan-novakov/php-openid-connect-client)

4.3.2.3 The Authentication Process

In this chapter we assume:

- that a Service Provider ID and Application ID exist in the system catalogues

- that the OASIS security server is deployed at {auth server} (for instance the test server is at oasis-demo.atolcd.com)

The REST services (“endpoints”) that you need to know are:

- the authorization endpoint: https://{auth server}/a/auth
- the token endpoint: https://{auth server}/a/token
- the keys endpoint: https://{auth server}/a/keys

The authentication process generally requires the following steps:

1. The application redirects the user to the authorization service
2. The authorization service “does its thing” such as checking for an existing session or requesting the user type in their login and password
3. The authorization service redirects to the application, embedding an authorization code
4. The application gets the authorization code, uses it to call the authorization server and get back two tokens: an access token and an identity token. The identity token is a signed JSON Web Token (JWT[5]) that encodes the user’s identifier
5. The application fetches the keys from the service, and uses it to validate the identity token’s signature (optional, but strongly recommended)
6. The application parses the identity token to get the “Subject” claim − that is the user identifier
7. The application uses some mechanism (such as a cookie) to establish a user session containing at least the subject and the authorization token
8. The application calls the User Info service to get the user information.

In the following paragraphs, we shall expand on the most important of these steps with code samples where relevant.

4.3.2.3.1 Redirect to the authorization service

When?

When your application needs to know who’s the connected user. In other words, when your application wants to start an authenticated session.

How?

Issue a Web Redirect (HTTP 302) to the Authorization endpoint (https://{auth server}/a/auth) with the following URL parameters:

- response_type=code
- client_id={service_provider_id}
- scope=at least “openid”
- redirect_uri={your callback endpoint}
- state={state}
- nonce={nonce}

Where:

[5] http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-14

[5]

- {your callback endpoint} is a Web URL that you serve. There are no constraints on what the callback endpoint is like or what technology it is built with.
- {state} and {nonce} are random strings that shall be sent back to you with, respectively, the authorization code response and the id token response. You **must** save these values in some sort of session management system (e.g. server-side session or encrypted cookie) in order to check them. These strings protect you from replay attacks.

Note about scopes
With the redirect you send a list of "scopes" containing at least the string "openid". Scopes are strings identifying resources; generally speaking, they are the "permissions" users are asked to grant. Scopes are listed in the access tokens and checked by resource servers. For instance, the "profile" scope allows you to get basic information about the connected user, such as their name. The "openid" scope is a special case. You can think of it as granting access to a resource containing the user's identity (i.e. their user id). The "openid" scope **must** be requested in order to perform authentication.
In general you should be careful with your user's privacy and only ask them to grant the minimal set of scopes that your application requires. An ideal case is asking only for the "openid" scope. Some special scopes: "profile" (user information), "email" (email address), "datacore" (read and write shared data under the user's identity).

Example: send the following response headers to your user:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
Location: https://oasis-demo.atolcd.com/a/auth?response_type=code&client_id=41184194-
d40b-4720-87a9-
284d2fa9d5ed&scope=openid+datacore&redirect_uri=http%3A%2F%2Flocalhost%3A9090%2Ffro
nt%2Fopenid_connect_login&nonce=1e081accb799&state=3d0b36a66585
Content-Length: 0
Date: Tue, 21 Jan 2014 14:05:08 GMT
```

4.3.2.3.2 Verify authorization – your callback

The interaction between the security server and the user is something that you do not manage. Generally speaking, either there is no interaction (i.e. the user is already logged in to OASIS and has already granted your application the set of scopes you are asking for), or the security server asks the user to confirm granting the scopes you have asked.
In any case, when the interaction is complete, the security server issues a redirect to your callback − the URI you specified in redirect_uri in the previous call. This is where you will do the heavy lifting.
Your callback gets two URL arguments with the redirect:

- state − what you sent in the first place
- code − an opaque string that you check against the token endpoint.

Example callback URI with arguments:

```
http://localhost:9090/front/openid_connect_login?state=3d0b36a66585&code=eyJpZCI6IjAwZTJhMDFjLWYxYTQtNDA
1NC1iMGQyLTc5NDAyODA4MWU0NCIsImlhdCI6MTM5MDMxMzQ3NTAxOSwiZXhwIjoxMzkwMzEzNTM1MDE5fQ&
```

(Notice how the *state* value is the same here and in the previous example).

What you **must** do with these values:

- check the value of the "state" argument against what you sent to the authorization endpoint
- directly from your application (i.e. without going back to the browser) POST a form (in application/x-www-form-urlencoded format) to the token endpoint (https://{auth server}/a/token).

This form must contain the following arguments:

- grant_type: should be "authorization_code"
- code: must contain the code you received from the authorization endpoint redirect
- redirect_uri: your callback, again; must be identical to the string you passed to the authorization endpoint (checked by the security server).

Your POST request must include an "Authorization: Basic {auth}" header, where {auth} is the Base64 encoding of the string "client_id:client_secret" (excluding quotation marks).

In response you will get a JSON string containing the following fields:

- access_token: the actual token you will use throughout the system, to access resources, etc.,
- expires_in: the time (in seconds) the access_token is valid for,
- id_token: an ID Token,
- scope: the space-separated list of scopes your access token is good for,
- token_type: the string "Bearer"

The ID Token is a JSON Web Token embedded in a JSON Web Signature, that is, a signed JSON string. You **should** decode it. Documentation about how to decode JWS and JWT can be found at:
JSON Web Signature: http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-20
JSON Web Token: http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-14
JOSE (Javascript Object Signing and Encryption, of which JWS is a part):
http://datatracker.ietf.org/wg/jose/

The keys to validate the JWS are to be found at the keys endpoint (https://{auth server}/a/keys).

Warning: the keys endpoint is not public! It requires the same Basic auth as the token endpoint, i.e. you **must** send an "Authorization: Basic base64(client_id:client_secret)" header to the keys endpoint. This is often a difficulty with pre-existing OpenID Connect client libraries such as MITREId Connect in Java.

Explaining how to decode and validate the ID Token is beyond the scope of this document, however here are some pointers:

- in Java you can use the Nimbus JOSE (com.nimbusds:nimbus-jose-jwt) library or the Google HTTP Client Library (https://code.google.com/p/google-http-java-client/)
- in PHP you can use the PHP JOSE library at: https://github.com/gree/jose
- in Ruby you can use the json-jwt gem

You should obtain and check the following claims from the JWT:

- Issuer (iss) should be present and equal to {auth server} (e.g. "https://oasis-demo.atolcd.com/")
- Expiration time (exp) should be present and not in the past

- Not Before time (nbf) should be present and not in the future
- Issued At time (iat) should be present and not in the future
- Audience (aud) should be present and equal to your client_id (service provider id)
- Nonce (nonce) should be present and equal to the nonce you sent the authorization endpoint at the start of the process

The Subject (sub) claim contains the user identifier. *At this stage the authentication process per se is finished!* You should store the ID Token, Subject, and (crucially) Access Token in some session storage.

4.3.2.4 Authorization – accessing resources

Now you have your access token, valid for one hour, that lets you access resources such as user information, the data core, the various kernel APIs (e.g. send notifications). To use them, simply add an "Authorization: Bearer {access_token}" header to your HTTP calls.

Example: get user info by calling https://{auth server}/a/userinfo. Depending on the Accept: header, the result will be sent back as a JWT (application/jwt) or a JSON string (application/json). Interestingly, the content of the userinfo will vary depending on the scopes bound to the access token:

- if scopes contain only "openid": userinfo contains only "sub" (for Subject), the user identifier,
- if scopes contain "profile": userinfo contains "name", "given_name", "family_name", "picture", "website", etc. (according to what the user has set in their profile),
- if scopes contain "email": userinfo contains "email" as well.

Additionally, if the user is an agent of an organization (e.g. a civil servant, or a member of staff) then the userinfo endpoint will return the following two additional attributes:

- organization_id: identifier of the organization
- organization_admin: a boolean flag that marks an agent as having "admin rights" on the organization, that is to say, they are allowed to act *on behalf of* the organization with regard to the OASIS platform (such as buying an application in the app market, adding new agents, etc.)

4.3.2.5 Authorization – request additional scopes

This case arises when your application's workflow requires additional privileges to continue operation. For example up until that point, you could operate with only the user id, but the user has activated a function that requires access to their profile information.

Note : it is tempting to require access to all of the possible scopes right from the beginning of the authentication workflow. However, that is not a good practice: to begin with, your users may be put off by a long list of privileges they have to grant just to start using your application. It is much better to ask them to grant additional privileges as and when they have actively triggered an action that requires those privileges.

Also, in case of a security breach, it is much better to leak access tokens with low privileges rather than ones with high privileges.
Finally, authorizations are persistent − i.e. if you require additional scopes that the user has already granted your application, the exchange between your application and the OASIS security server will be transparent. In particular, requesting additional scopes will not require the user to grant additional access every time (unless you specify it so by using prompt=consent). In that respect, there are no reasons on UI grounds not to request scopes as needed only.

Requesting additional scopes is performed by using the very same mechanics as initiating authentication in the first place:

- save the user interaction context in some way (e.g. cookies, server-side session…)
- generate a state, redirect to /a/auth specifying all the scopes you need (openid, profile, etc.) and a callback
- implement the callback like before
- save the new access_token − it is now valid for the extended scopes you have required

- restore the user interaction context

4.3.2.6 Token expiration

Access tokens are only valid for a given amount of time after being issued (by default, one hour), after which they silently stop being usable to access resources and you **must** refresh your tokens (i.e. get a new one). The mechanics are the same as for initiating the authentication protocol and requesting additional scopes, however this can be transparent with respect to the user as the OASIS session is eternal and the required scopes have (by definition) already been granted. In that respect, although the browser does issue requests to the kernel, the user never sees a Kernel page.
In order for the process to be totally transparent though, the same precautions have to be taken as for requesting additional scopes in an ad-hoc fashion: you must save the interaction context before initiating the process, and restore it afterwards.
Depending on your application's implementation, it may be awkward to recover "optimistic authorization" errors (i.e. only finding out that a token is expired by trying to use it). For instance, if you try to access a resource as part of handling a POSTed HTML form, you have to:

1. save the form in session storage
2. try to access the resource, get a "Token expired" exception
3. redirect the user to /a/auth
4. receive a GET request to a callback URL, obtain the access token
5. restore the form
6. resume handling the form

This can add unwanted complications to your application code. For this reason, we recommend you perform token verification proactively, at times when your application workflow can easily deal with the interruption due to the token-refreshment process.

4.3.2.7 Logging out

"Logging out" is a process involving either or all of three steps:

- log out of your *application*,
- *revoke* the access token so it cannot be used any longer,
- log out of *OASIS*.

These steps are independent. To truly log out a user, you should do all three.

4.3.2.7.1 Log out of your application

This is the easy step−do what your security framework normally does (this usually entails removing cookies and freeing up server-side-held resources).

4.3.2.7.2 Revoke the access token

In addition to logging out of your application, you should notify the security server that the access token should never be used again; for this POST an application/x-www-form-urlencoded form to https://{auth server}/a/revoke containing a single "token" field containing your access token. Also, add an "Authorization: Basic base64(client_id:client_secret)" header.
You should receive a 200 OK status code in return.

4.3.2.7.3 Log out of OASIS

The simplest way of logging out the user from OASIS is to redirect the user to https://{auth server}/a/logout (the *logout endpoint*).

4.3.2.8 Best practices and pitfalls

- Remember to check that your access token is not expired! Recovering from an exception can be hard. Check proactively.
- Always ask for the minimal set of scopes. For instance, the "email" scope should almost never be required−if you need to send an email to the user, you should use the Notification API instead as this lets the user decide if they get notifications via email, SMS, or Portal notifications, etc.
- Remember that there are two sessions going around: the OASIS session and the application session. These two sessions may have different lifetimes; you should be especially concerned with the case of an OASIS session being destroyed before the application session. A good practice that we recommend is to check periodically, at least every few minutes, that the access token is still valid, for instance by calling the userinfo or token introspection endpoint, and either destroying the application session or (preferably) requesting a new access token (which will restart the authentication mechanism if the user is really logged out) if the token is invalid.
- For similar reasons, 401 errors ("Authorization required") when accessing OASIS resources should be gracefully handled, for instance by restarting the authentication mechanism.

4.3.3 Data core resources

4.3.3.1 Commonalities to use all APIs

All the datacore APIs require a valid access token, as obtained by the authorization mechanism. The access token should be valid for at least the technical scope “datacore” as well as additional scopes required by your models.
The access tokens are passed to the APIs by using Bearer tokens in the Authorization header.

4.3.3.2 Manipulating models

As mentioned earlier, proper data modelling is the important aspect of datacore interaction. Keep in mind the need for cross-context reuse (which promotes linked data strategies) on one side, and performance considerations (which promote the use of inline or mixin data structures).

The APIs for manipulating models are fairly straightforward. Please refer to the Swagger-generated auto-documentation on the development environment for complete API reference.

4.3.3.3 Manipulating data

The data manipulation APIs are straightforward. Please refer to the Swagger-generated auto-documentation on the development environment for complete API reference. In this chapter we will only mention the major features, caveats and pitfalls associated with using data core resources.

Overview
In keeping with RDF philosophy, data core resources are uniquely identified by URIs. This URI should be of type: {container URI}/dc/type/{type name}/{iri} where:

- container URI is the data core instance’s identifying URI, for instance: http://data-test.oasis-eu.org for the development environment
- type name is (obviously) your type name, for instance “altTourism.place”
- iri is the *internal resource identifier*, it should uniquely identify your resource within its type. There are no particular constraints on the iri format, we only ask that they be as human-readable as possible when it is relevant. For instance, in the type “altTourism.place” you may setup an iri scheme of the form {country}.{city}.{placename} (e.g. uk.london.tower-of-london). When the data is not meant to be shared (which should remain a limited case) a random string is valid.

Creating data is simply a matter of POSTing to /dc/type/{type name} a JSON structure that contains:

- uri: the full URI of the resource you want to create
- version: 0
- your payload (data fields)

Example of a data core resource that is POSTable:

{
"uri":"http://data-test.oasis-eu.org/dc/type/citizenkin.procedure.envelope/6666_7777_aaaa",
"version":0,
"state":"DRAFT",
"definition_name":"electoral_roll_registration",
"initiator":"person:09cf5b64-b08b-46b2-af19-5302e8b13fe0",
"administration_id":"820",
"start_date":"2014-01-02T10:54:00.000Z",
"modify_date":"2014-01-02T10:54:00.000Z"
}

Loading a resource by URI is straightforward, just GET that URI.

Querying resources is done through searching. In the data model one specifies what fields are searchable; it is currently possible to search through other fields (albeit much slower) but this ability may be removed in future versions of the data core; do not depend on it.
Searching is done through URL parameters, specifying:

- the field being searched,
- the "=" sign,
- optionally: an operator (such as ">" or "$in"),
- a value or values.

Specifying the same parameters multiple times is possible and is the way to do e.g. range searches.

There are two special URL parameters, which **may not** be used as search fields:

- start: index of the result at which to start sending data,
- limit: number of results to send.

These two parameters control paging. You do not want the data core to return *all* matching data in case of many results; this is how you control which page gets returned. If you do not provide these parameters, then they are valued at: start=0, limit=10.

Note: dates are specified using ISO-8601 format with time zones, e.g. "2014-01-01T18:04:43.287+01:00" represents the 1st of January 2014 at 18:04:43 and 287 milliseconds, Central European Time.

Query examples:
Search for "date" between July 1st 2010 and May 30th 2011, fetch results 10 through 19
GET /dc/type/{type}?date=$in["2010-07-01T00:00:00.000Z", "2011-05-30T00:00:00.000Z"]&start=10&limit=10

Search for "threshold" greater or equal than 500 and "name" equals "John Smith", fetch results 0 through 9
GET /dc/type/{type}?threshold=>=500&name="John Smith"

Note: be careful, if using a library to build URIs, that operators are not URL-encoded. Also, make sure that values are quoted as appropriate.

Caching

The GET data core APIs support client-side caching through standard HTTP (Etag) mechanisms. Datacore resources are versioned; when querying datacore resources, simply specify the latest version number that you have in cache as an Etag; if the resource is unchanged, the datacore will respond with a 304 Not Modified HTTP status. By circumventing network traffic this can dramatically speed up communication.

Versioning and optimistic locking
The data cores uses optimistic locking throughout. When modifying data, you *need* to specify the resource version you want to update − if it is not the latest version, the operation is aborted and a 409 Conflict status is returned. By the same token, deleting resources require you to specify an If-Match header containing the version you want to delete.
The general workflow for updating or deleting resources is therefore:

1. fetch the resource by URI, get the current version
2. PUT the updated resource (or DELETE the resource's URI) specifying the version number.

4.3.3.4 Managing permissions

The model lets you define a basic permission level, which is creation permission. Once objects are created though, you want to specify who may perform what operations on this particular resource. This is the goal of the Data core permission API.
We will not in this document specify the whole permission API as it is rather straightforward (please check the online documentation); rather, we will explain its philosophy.

Each resource is associated with three groups:

- its **readers**
- its **writers**
- its **owners**

In these groups you may add either:

- user ids
- user groups (as defined by the Kernel's Directory APIs)

The groups give their members specific permissions:

- readers may query and read the resource,
- writers have all the rights of readers, plus the right to update the resource,
- owners have all the rights of writers, plus the right to set permissions.

For instance: suppose your application creates a datum that is private to its creator. Simply flush all rights using `PUT /dc/r/f/{type}/{iri}/{version}` to remove all permissions except for those of the owner (a resource's owner is initialized with the identity of its creator).
Then in your application workflow, you want to add another person as reader; simply add their user id to the "readers" list.

4.3.4 Social graph resources

The Social graph APIs allow to query and update the various node types and relationships stored

in the Social graph. The APIs are rather straightforward, please check the online documentation.

Example query: fetch the children of a user
Assuming the user's ID is "09cf5b64-b08b-46b2-af19-5302e8b13fe0" and the "parent-child" relationship is "children", then the request is:
GET /s/identity/09cf5b64-b08b-46b2-af19-5302e8b13fe0/relation/children/members

4.3.5 Broker resources

4.3.5.1 Directory API

The Directory API is used to query and update agents, organizations and groups, that is to say, to manage users that are agents of organizations and not simple citizens. Generally speaking, a group is a set of people, which can be used (for instance) to
The Directory API is straightforward. Please check its online documentation.

Example query: fetch all agents in an organization
Assuming the organization's ID is 6dccdb8d-ec46-4675-9965-806ea37b73e1 and we only want the first 25 agents:
GET /d/org/6dccdb8d-ec46-4675-9965-806ea37b73e1/agents?start=0&limit=25

Example query: create an agent in an organization
Assuming the organization's ID is 6dccdb8d-ec46-4675-9965-806ea37b73e1, we can use the following request:
POST /d/org/6dccdb8d-ec46-4675-9965-806ea37b73e1/agents
with request body:

```
{
  "address": {
    "country": "France",
    "streetAddress": "1 rue du Ru",
    "locality": "Ain sur Ain",
    "postalCode": "01011"
  },
  "name": "Jean Dupont",
  "id": "jdupont",
  "locale": "fr_FR",
  "gender": "M",
  "organization_admin": "false",
  "organization_id": "6dccdb8d-ec46-4675-9965-806ea37b73e1",
  "family_name": "Dupont",
  "given_name": "Jean",
  "email_verified": "true",
  "email": "jdupont@mairie-ainsurain.fr"
}
```

4.3.5.2 Data provider, service provider, application API

These APIs let you declare a new application (within your own organization), query the data providers and service providers for a given application, etc. Please refer to the Terminology chapter above in this document for a thorough description of applications, service providers, and data providers.

Example: get your service provider's information

GET /d/serviceprovider/41184194-d40b-4720-87a9-284d2fa9d5ed

This returns:

```
{
  "id": "41184194-d40b-4720-87a9-284d2fa9d5ed",
  "name": "Citizen Kin",
  "modified": 1387380525785
}
```

Example: create a data provider for your application

Assuming your application's id is: 0a046fde-a20f-46eb-8252-48b78d89a9a2

POST /d/app/0a046fde-a20f-46eb-8252-48b78d89a9a2/dataproviders

with body:

```
{
  "name": "CitizenKin Files",
  "scopeIds": [
    "ck_files"
  ]
}
```

This returns a new data provider:

```
{
  "id": "d435070b-afd6-4227-85d9-b9e90179b95f",
  "scopeIds": [
    "ck_files"
  ],
  "name": "CitizenKin Files",
  "modified": 1390817234116
}
```

4.3.5.3 Market API

This API is meant for use by the Portal, rather than generic applications. It lets clients list all available applications and instantiate ("buy") an application in the context of an organization. Please refer to the online documentation for how to use this API.

4.3.5.4 Audit log API

This API is used to create auditability logs. The logs are centralized and can be requested and visualized through the OASIS log tool (Kibana). Note that:

- it is not meant for debugging logs! These logs are specifically for auditability, i.e. functionally important actions that users undertake and that you want to store safely, in a centralized manner, by a third party (OASIS in this case).

- Examples of events you may want to log could be financial transactions, creation of an object, deletion of an object, alteration of user permissions, etc.

Please refer to the online documentation for how to use this API.

4.3.6 Event bus API

The event bus API lets you:

- register an interest for a given type of event. You do this by providing a web hook URL and a shared secret; when an event is published for that topic, the Kernel POSTs the event payload to the web hook URL, adding a signature computed from the shared secret into the X-Webhook-Secret header.
- publish an event. You do this by POSTing an Event payload to the /publish endpoint. The kernel dispatches it to the webhooks that registered an interest for this type of event.

The objective of this event bus API is to provide an easy and decoupled way to do inter-application communication.

Please refer to the online documentation for how to use this API.

4.3.7 Universal notification

The Notification API lets you notify users of something. Rather than sending the user an email, you should send them a notification: through the Portal, users can decide to receive notifications via email, text message, or through the Portal GUI only; they can opt for a daily digest, etc.
The notification API is straightforward. Please check the online documentation.

5 Conclusion

Having read through this document, you should now have a fair grasp of the following:

- What OASIS can do for you,
- What effort is necessary on your part to integrate your services into the OASIS ecosystem,
- What the underlying concepts of the OASIS APIs are.

You should also have a good understanding of the current APIs; however it is important to note that these APIs may fluctuate. This document only aims to give an overview of the OASIS API design, rather than a definitive description of each and every API call parameter. While the overall design will not change much, it will be thoroughly enriched in certain domains; existing APIs may be progressively phased out as more comprehensive APIs are implemented and published.
With respect to the actual APIs, you should view this document as a milestone, a point-in-time snapshot of what the OASIS kernel offers to service developers.

For detailed, up-to-date API documentation, you should refer to the auto-generated online Swagger documentation at:

- https://oasis-demo.atolcd.com/swagger-ui.html#!/ (kernel services excluding data core)
- http://srv2-polenum.fingerprint-technologies.com (data core)