

# Flexible Services

## Ecosystem Design and Evolution (EDEN) D1.2.1 Flexible Services Ecosystem Architecture

<b>Document Name</b>	FSE Architecture Document
<b>Project/WP Title:</b>	EDEN WP1: Architecture and Technology
<b>Document Type, Security</b>	P (Public)

<b>Document Title:</b>	Flexible Services Ecosystem Architecture
<b>Agreed date of delivery</b>	5/31/2009
<b>Actual date of delivery</b>	
<b>Editor</b>	Jaakko Kangasharju/TKK
<b>Version</b>	version 0.5
<b>Date Last Change</b>	8/28/2009
<b>File:</b>	

Participants	Name	e-mail
Elisa	Olavi Karasti	
Ericsson	Tomas Mecklin	
NSN	Janne Parantainen	
TeliaSonera	Olli Jussila	
	Mikko Laukkanen	
TKK/CSE	Jaakko Kangasharju	
	Sanna Suoranta	
	Tancred Lindholm	

## Table of Contents

Table of Contents .....	2
List of Acronyms and Abbreviations .....	5
Executive Summary .....	6
1. Introduction .....	7
2. Definitions .....	7
3. Requirements Analysis .....	7
3.1. Requirements from Use Cases .....	8
3.1.1. Adaptation .....	8
3.1.2. Security .....	8
3.1.3. Information .....	9
3.1.4. User .....	9
3.1.5. Service .....	9
3.1.6. Compensation .....	10
3.2. Other Requirements .....	10
4. Principles of Architecture Design .....	10
4.1. Modularity .....	10
4.2. Transparency .....	11
4.3. Loose Coupling .....	11
4.4. Composability .....	11
4.5. Late Binding .....	11
4.6. Reflection .....	12
5. Technical Enablers for Flexible Service Architectures .....	12
6. Functional Architecture .....	13
6.1. Top Level of the Functional Architecture .....	13
6.2. Components of the Functional Architecture .....	14
6.2.1. Information .....	14
6.2.2. Service .....	15
6.2.3. Compensation .....	16
6.2.4. Security and Trust and Privacy .....	16
6.2.5. Authentication .....	16
6.2.6. Authorization and Access Rights Management .....	16
6.2.7. User Management .....	17
6.2.8. Media management .....	17
6.2.9. User Tools .....	18
7. Technical Architecture .....	18
7.1. Service and Process Lifecycle Management .....	18
7.2. Media and Content Management .....	19
7.3. Context-aware Services .....	19
7.4. Identity Management Services .....	19
7.5. Payment Services .....	19
8. Information Architecture .....	19
8.1. Users .....	20
8.2. Services .....	20
8.3. Access Control .....	20

8.4. Combination .....21  
9. Next Steps ..... 21  
10. Conclusion ..... 21

## List of Figures

Figure 1. Bootstrapping through established infrastructure .....	13
Figure 1. The main level of the functional architecture .....	16
Figure 2. The information component and its subcomponents.....	17
Figure 3. The Service component and its subcomponents .....	17
Figure 4. The Security component and its subcomponets.. <b>Virhe. Kirjanmerkkiä ei ole määritetty.</b>	
Figure 5. User management and its subcomponents.....	19
Figure 6. Media Management and its subcomponents .....	19

## List of Acronyms and Abbreviations

EDEN	Ecosystem Design and Evolution
FS	Flexible Services
FSE	Flexible Services Ecosystem

## **Executive Summary**

This document presents the Flexible Services Ecosystem architecture. Architectural requirements and design principles are described, followed by a view of the architecture from functional, technical, and information perspectives.

## 1. Introduction

This document describes the Flexible Services Ecosystem (FSE) architecture defined in the EDEN project. The requirements extracted from FS project use cases are presented, as well as architectural design principles that promote flexibility. The architecture itself is covered from three different aspects: functional, technical, and information.

The architecture presented here is expected to undergo some revisions during the second project year. Information acquired from pilot cases, as well as new requirements from new use cases, may necessitate changes in some parts. However, the overall design, and especially the principles followed, are not expected to change.

## 2. Definitions

A *service* is modeled here as a collection of *operations*, each having a specified set of *parameters* and *results*. A *user* interacts with a service by *invoking* its operations. An *execution* consists of a user invoking a series of operations on a number of services, with later invocations in the series potentially depending on the results of earlier ones, to accomplish some goal.

The user-service distinction here is to be interpreted as a distinction of roles in a specific service invocation. A service may act as a user to some other service, as is the case when an invoked operation requires some other service to complete.

A *service ecosystem* is fundamentally just a collection of services supplied by a number of service providers, with two characteristic features:

1. For any given functionality, there may be (and often are) multiple services providing it.
2. Users have minimal constraints when selecting a service for a particular functionality.

These two features ensure that there is competition among service providers and that services better suited to users acquire more of them, enabling a selection process where good services succeed and bad services fade away.

## 3. Requirements Analysis

Requirements for an architecture are commonly found in use cases that the architecture is expected to support. Analysis of the use cases provides a set of general requirements that each include a more detailed set of sub-requirements. The actual use case documents and their analyses are not public documents, but the end results of the full analysis are provided here as a set of requirements for the EDEN architecture.

It is not always necessary to have a use case for a particular requirement, though such requirements should be brought in only sparingly, as a requirement without use case support might be something that does not actually materialize in practice. Still, some requirements may be identified as necessary for a flexible services architecture independently of whether they appear in use case analysis.

### ***3.1. Requirements from Use Cases***

The requirements extracted from the use cases have been divided into six classes: Adaptation, Security, Information, User, Service, and Compensation. Individual sub-requirements in these classes are not necessarily presented, as combining the requirements has in some cases resulted in more generality than is specified by any particular use case.

#### **3.1.1. Adaptation**

The general requirement for adaptation refers to a number of different concepts. *Context awareness* is about the services adapting to a client's current situation. *Information adaptation* refers to transforming the available information into a form most suitable at the time. And *UI adaptation* refers to the ability of a user interface to adapt both to the client's context and the available information.

To enable proper adaptation, the infrastructure needs to support describing client context to allow services to adapt to it. The most important part of the context is location, as it is becoming widely available and location-aware services are expanding in scope and user population. For information adaptation, device capabilities similarly need to be known, but generic transformations are also possible, unlike in context awareness where adaptation is typically more service-specific. UI adaptation is a requirement on services in that information presented to the user should be designed in a manner that does not hinder its use on a variety of devices, but the actual mechanisms of adaptation have less of a relationship with the actual service architecture.

#### **3.1.2. Security**

Security-related requirements come from all use cases. The most common security requirements are *authentication*, certification of a user's identity, and *authorization*, controlling the access to services and information. Related to authorization is also the concept of *trust*, the ability to rely on other parties to carry out their obligations.

The granularity of these functionalities is also a consideration. In authorization, not all users are able to invoke exactly the same operations on each service, so fine-grained authorization and access-control mechanisms are needed. Similarly, users will not want to provide all of their information to every service, so finer granularity in the authenticatable information is also required.

Security is not an on/off property, but something with gradations. Not all services require the highest amount of security available, and with the well-known tradeoffs between security and usability, this might even be counter-productive to the service

provider. Therefore the system needs to support varying levels of security, allowing users and services to select the most appropriate level for each situation.

### 3.1.3.Information

Management of information is an essential part of services, and even though services often manage all of their information themselves, there is much common information management functionality that can be provided by the infrastructure.

There are various kinds of information, e.g., text, pictures, and video, and each kind of information also has a number of other properties associated with it. One common functionality would therefore be a *metadata facility* that can be used to associate defined properties to each piece of information. Such metadata could also include semantic descriptions of the data.

Tying into the security requirement, access to information needs to be controlled. Often in this case, there are various levels of access, called *roles*, that determine what operations each user is allowed to perform for the information. Related to this is that some permissions stick to the information, e.g., intellectual property rights that determine how and with whom the information can be shared. Still relating to the security requirement, methods for ensuring the integrity and validity of information need to be supported.

### 3.1.4.User

The central concept from the user perspective is *user identity*. The identity concept needs to be flexible enough to support fine-grained user attributes, which also allow individual preferences to be expressed. Different situations may also call for different identities to be used by a single user. This all translates into a requirement for *identity management*.

Access to a user's personal information must always be controlled by the user. This requirement is called *privacy*. Therefore the information architecture cannot only be the purview of services but must also let users access it and set permissions. However, as services will need to share user information, the permission setting must not put an onerous burden for the user.

### 3.1.5.Service

The requirements on services relate to *service life-cycle management*. This has certain phases: service provisioning, service discovery, and service usage. In addition, the service needs to be managed throughout its life cycle, and the functioning of the service must be monitored.

Services need to be able to communicate with each other, to provide more functionality than any of them individually. Thus there needs to be communication infrastructure, including methods for services to come to agreement on different matters relating to their interaction. This will, in some cases, require multi-party communication.

### 3.1.6.Compensation

Not all services are free and open to everyone, so some method for compensating the service provider is needed. Charging users for services requires accurate identification, and tying the identification to billing information. And vice versa, users also need to feel like they are getting something out of the services, which in some cases means compensation to the user.

There are a number of compensation models for services. A *subscription-based* system works for services used continuously, but does not work well for occasional uses. Thus, there needs to be some form of support for *micropayments*. In addition, currency-like instruments like various *membership points* or an explicit *virtual currency* can exist in some services, which can also be used for compensating the users. An explicit virtual currency would also enable easier user-to-user compensation without needing out-of-band exchanges.

### 3.2.Other Requirements

Other requirements usually surface after an implementation of a service exists and there is sufficient experience to determine what properties are still needed. So far, no other requirements have been specified.

## 4.Principles of Architecture Design

In Flexible Service Architectures, the design principles need to be chosen in support of that flexibility. Conversely, design principles that define the relationships of the components too rigidly should be avoided. The principles listed here have been chosen according to these requirements. Similar principles are evident in the design of, e.g., Service-Oriented Architecture.

To some degree, there is overlap between the different principles. This is understandable as the principles have been chosen with the uniform goal of flexibility in mind. However, despite any apparent similarities and overlap, they usually have at least a different focus, so some may be applicable where others are not.

### 4.1.Modularity

A common architectural design principle in many systems is *modularity*. Modularity refers to a system design where the system is composed of independent components, called modules, such that each module provides a well-defined piece of functionality.

The main benefit of modularity is to make an architecture more comprehensible and also comprehensible only partially. When each module is responsible for a well-defined piece of functionality and all such functionality is included in that module, the high-level architecture becomes simpler as it is necessary to understand only the rather coarse-grained module division. In addition, it is possible to study only parts of the architecture, as certain functionality is always in a single place.

As modularity is such a common design principle, many of the other principles presuppose or imply some form of modularity as well. Modularity can in some cases be

traded off for increased performance (e.g., by taking shortcuts that break the chain of responsibilities), but in the vast majority of cases the increased obscurity makes it unadvisable.

#### **4.2. Transparency**

In a distributed architecture with potentially many actors involved, it is not sufficient to simply have modularity. It is also necessary that the interfaces exposed by the modules are clear and the effects of any operations in the interfaces are well-defined. In addition, for maximal flexibility, the interfaces and communication should be machine-inspectable. Collectively, this is called *transparency*.

Transparency is of benefit in many different cases. For one, when developing an application using a number of services, it is important that the service interfaces are easy to program against. Moreover, a transparent interface and protocol promote understanding an existing system, both statically and dynamically.

A non-transparent approach is sometimes used as a poor man's substitute for security, i.e., access to a component is hindered by taking an undocumented and opaque approach. This is however to be avoided, as "security by obscurity" has been found to offer very weak protection.

#### **4.3. Loose Coupling**

Transparency, when properly expressed, is an essential part in providing *loose coupling*. Coupling refers to the degree to which the architectural components are dependent on each other. In a loosely-coupled system, the dependence is low, so it is easier to change components independently without the changes propagating throughout the system.

Standard enablers of loose coupling include extensible data formats and protocols. A properly-designed extensible data format allows communication to happen even in cases where both sides do not understand the exact same versions of the format, and similarly for protocols. This increases the space of potential communication partners, making the system less coupled than a system with rigidly-specified interactions.

A disadvantage of non-rigid communications and data exchange is that undocumented conventions may arise, which can only be understood by studying real-world implementations and interactions. As an example, consider the evolution of HTML.

#### **4.4. Composability**

Modularity, transparency, and loose coupling are principles that promote architectural flexibility from the component point of view. To also attain flexibility in the connections, the principle of *composability* is a useful one. Composability refers to being able to alter the relationships between components dynamically, giving the ability to compose new concepts into being from existing ones.

#### ***4.5.Late Binding***

*Late binding* is a principle enabled by composability. In late binding, connection formation between components is delayed to as late as possible. This promotes flexibility in that the selection of communication partners can be made with as much information as possible.

Late binding requires loose coupling in the components, as a rigid architecture cannot support the kind of dynamic connection formation that late binding requires. Even more, while transparency and loose coupling promise an understandable system composed of independent pieces, to achieve late binding, this all needs to be machine-understandable as well, to allow the running system to make the binding decisions on its own without needing human guidance.

Late binding typically implies some additional dynamicity in the system which can introduce behavior that is not easy to model a priori. For instance, consider the difference between strictly and loosely typed programming languages. While loose typing is generally more flexibility, the late binding it offers sometimes introduces flaws that are not detectable until at run time.

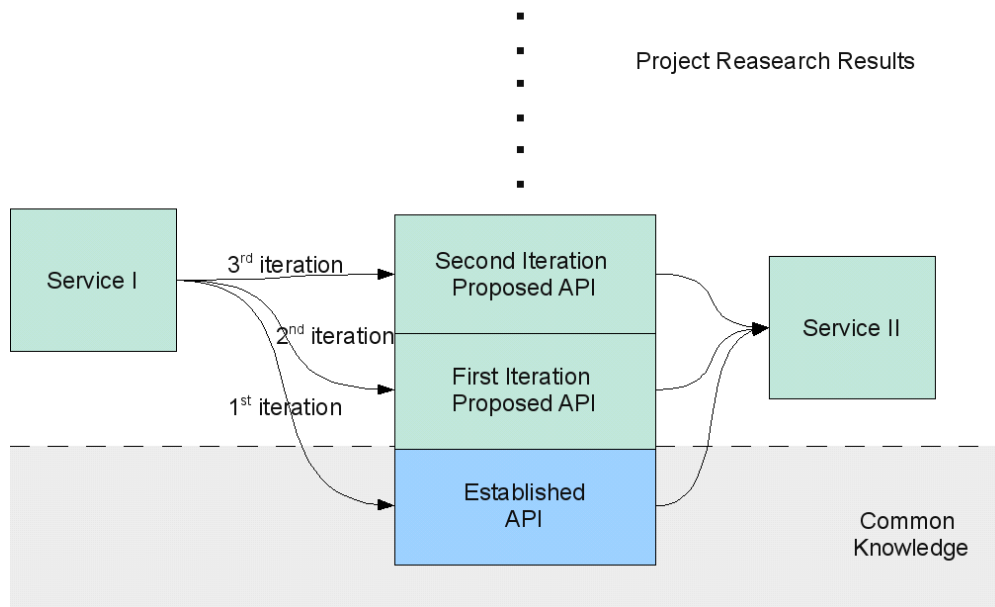
#### ***4.6.Reflection***

*Reflection* is the ability of a running system to inspect itself and its complete state and to modify its state and composition at run time through generic interfaces. In a way, reflection is an end run around the restrictions of the system, restrictions that may have good reasons, but properly used, reflection can provide needed flexibility in the system. Namely, a flexible system is often put to uses that were not even envisioned when the system was designed, so it may be necessary to go around inappropriate limitations. In a potential next iteration of the design, the limitations exposed by reflective use can also be taken into account.

Reflection, as well as late binding and loose coupling may introduce some additional computational complexity in terms of e.g. marshalling logic and fault-tolerant processing of exchanged data. Furthermore, some optimizations may not be possible in such a system (compare compiled vs. interpreted languages).

#### ***4.7.Bootstrapping through Established Infrastructure***

To manage the complexity of concurrent research on multiple services, and to guard against premature commitment to functionality that is in flux and needless coupling of research deliverables, we propose that interaction between services are bootstrapped through a reasonably well-established service invocation interface (API). In following iterations, services can proceed to communicate through evolved APIs that have emerged as research results, provided that this is beneficial for the involved services. In particular, a switch to a proposed API should only be made if that enables some key functionality that would otherwise be lacking. This is illustrated in Figure 1 below.



*Figure 1. Bootstrapping through established infrastructure*

Taking a slightly pessimistic, and perhaps therefore realistic, view of the project as a whole, it may be that some service research will be less successful. In this case, migration to a proposed API will not happen due to lack of utility to other services, and the research impact of these other services will not be negatively affected, since they will rely on the baseline well-established API, which is familiar to the research and user communities.

For instance, consider the interaction between a media management service and an identity verification service. To authenticate access to media, the media manager uses the identification verifier. Initially, the media manager relies on some established identification method, e.g., OpenID. However, it turns out that group management is also required by the media manager, and furthermore, that research on an in-house identification service provides such functionality. Then, in the second iteration, it makes sense for the media manager to move off the established API and start utilizing the group management functionality provided by the in-house identification service API.

Anchoring initial efforts in a well-established API may introduce some initial overhead in cases where the design of a more evolved interface is obvious. However, it seems this is rarely the case.

#### **4.8. Easy Ecosystem Entry**

From all possible architectures conforming to the requirements, we should choose one that entices service developers to enter the ecosystem, both from a technical as well as economical perspective. There is no competitive ecosystem of services, unless there are enough services to choose from, meaning we need as many developers as possible that write services.

While it is hard to quantify just what makes an ecosystem developer and business-friendly, some guidelines may be given

- Strong focus on simple and unsurprising design
- Provide a path from an initial single person effort with 10 users to a world-wide service with 100M+ users
- Use mainstream development platforms and languages where possible
- Minimize amount of fees, registrations, and other paperwork needed for initial entry, e.g., by providing small-scale test accounts etc.

## 5. Technical Enablers for Flexible Service Architectures

Security is usually required to some extent. Security protects information, resources, and services from threats. Security has several components that provide different aspects of security. *Confidentiality* means that only legitimate users can access information, and it is usually achieved by encrypting information. *Integrity* means that the information is intact and not changed by unauthorized parties, and it can be achieved by digitally signing the information. *Availability* means that the information, resources, and/or services are available for legitimate users when they want them. Denial-of-service (DoS) attacks are hard to prevent but systems can be replicated or distributed which makes the attack harder to implement. In *authentication*, the identity of a party is verified, and in *authorization*, a party's right to perform an action is checked. *Access control* is used to separate legitimate users from others. *Non-repudiation* means that an action cannot be repudiated, or denied, afterwards.

Technologies for achieving confidentiality and integrity are widely available, and they are typically bundled into protocols designed by experts to avoid compromises by faulty design. Such protocols are typically based on *asymmetric encryption*, necessitating some form of a *public-key infrastructure* to allow the participants to authenticate each other and to be able to protect the confidentiality and integrity of the communication.

Separating authentication from authorization is beneficial, as it permits users to use established identity providers instead of acquiring a new identity for each service, which further leads into *single sign-on* where a user authenticates only once and is able to use multiple services. Well-known standard protocols for single sign-on include OpenID and SAML. Such protocols do not concern themselves with matters of trust, such as whether a service should trust a particular identity provider.

Service life cycle begins with the *provisioning* of the service, namely making it available for use. Provisioning by requiring registration in some centralized service registry is somewhat inflexible, but is better for *discovery*, when a user attempts to locate the service to use it. A chaotic model where services are not registered at all needs an easy way to identify the services so that they can be shared or collected into

directories. In some cases, a *federated* registry, such as the Internet's Domain Name Service, can function well.

There are a number of options for communicating between users and services, as well as among services. Depending on the type of service, user-service communication is usually either *request-response*, such as on the WWW, or *subscribe-notify*, such as with news feeds. Communication among services, especially among the components of a composed service, is a different matter. Here, it is more common to need *one-to-many* communication, as the messages from one component are often of interest to many of the other components. Depending on whether the communication is seen as message-like or shared-memory-like, a *publish/subscribe* or a *tuple space* architecture provides a suitable one-to-many communication abstraction.

## 6. Functional Architecture

Functional architecture provides generalizations of the physical components. It presents functional elements and their relationships to each other needed for the proper working of a system. Functional architecture omits descriptions of where and how these functions are realised. Also detailed description is not given how the actual system works. Functional architecture is a starting point to move towards implementation and system design. This functional architecture is based on use cases developed in Flexible Services projects.

The use case analysis has found several architectural enablers from which grouping and generalisations provided following items: **Information, Service, Compensation, Authentication, Authorization and access right management, User management, Media management and User tools.**

### 6.1. Top Level of the Functional Architecture

In Figure 1 these architectural components are illustrated and their relationships to each other are shown. Arrows show how components use other components. The security issue as a own component is not shown in this figure because security deals with all instances in this functional architecture and it is dealt in chapter 5.

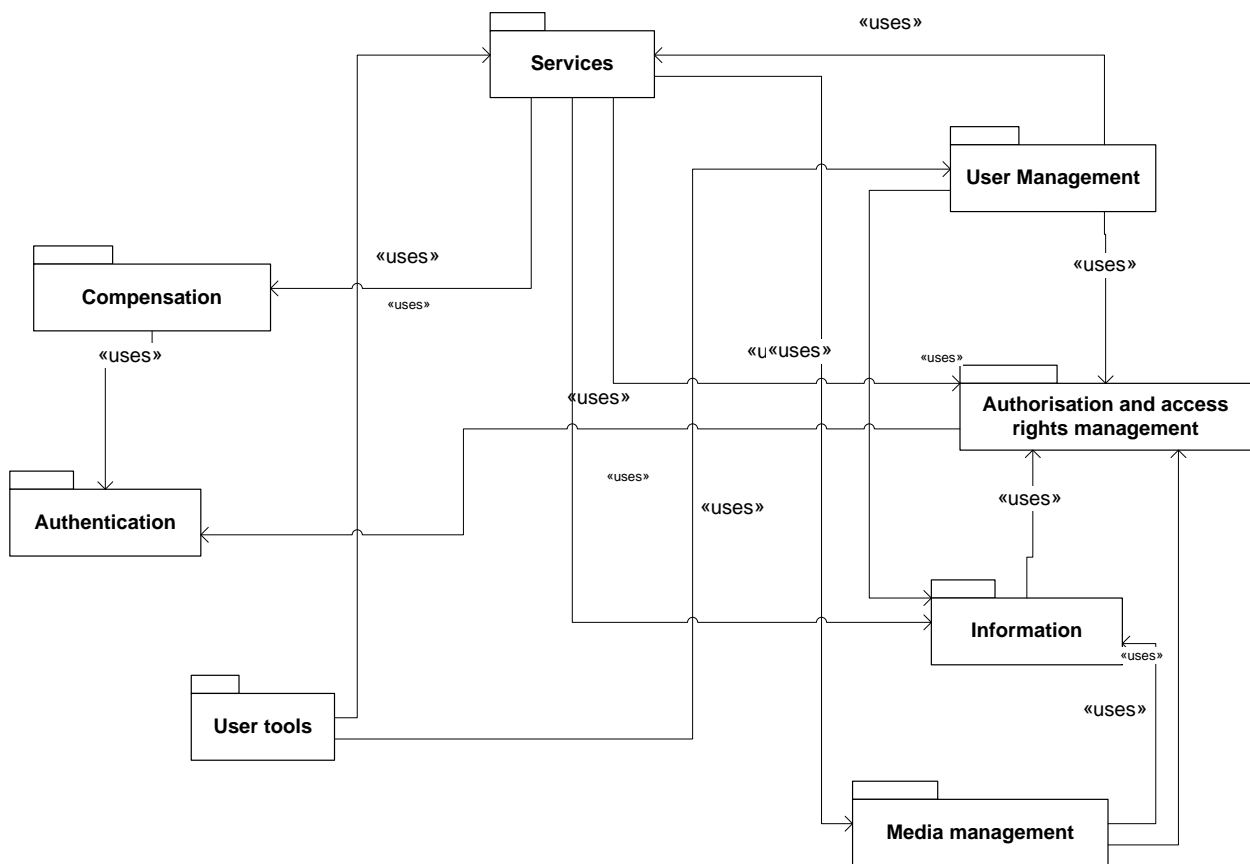


Figure 1. The main level of the functional architecture

## 6.2. Components of the Functional Architecture

### 6.2.1. Information

The Information entity consists of several subcomponents the most important of which are:

- Collection and processing of information,
- Access and validity, and
- Metadata handling.

These subcomponents are presented in Figure 2. Collection means all the functions needed to gather information from producers to be available for further use. Collection needs authorization to ensure that the proper access rights of the collection are followed. Processing handles gathered information when needed. Access is connected to collection and provides communication between actors. Validity of data has to be supported and is connected to information. Metadata is collected partly at the same time as other information and partly from other sources. Metadata is an important part of information processing, and it is also of great significance for the service. Context is any information that can be used to characterize the situation of an entity, and context awareness is the ability to use this information in processing. For example, a service can provide personalized news for the user.

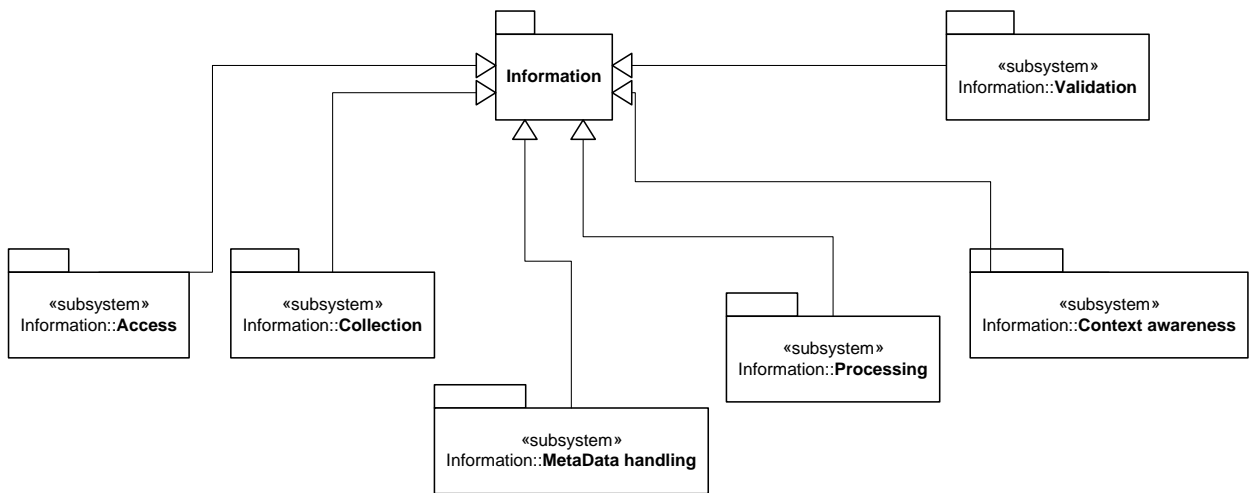


Figure 2. The information component and its subcomponents

### 6.2.2. Service

The Service component provides services to be consumed by everyone who needs services. This component consists of Search, Locate, Delivery, Management, Monitoring, and Provisioning subsystems. The Search subsystem supports searching of different services, information, users, or any other entity. Locate refers to showing where a service or entity is. Delivery is to use a service or to provide the actual location of a person. Management means running, handling, and supporting a required service. Management is not the same for all services, and in some cases management is not needed at all. Monitoring is about measuring and calculating used services. Monitoring is not only the technical details of performance but also the satisfaction of the end users of the service.

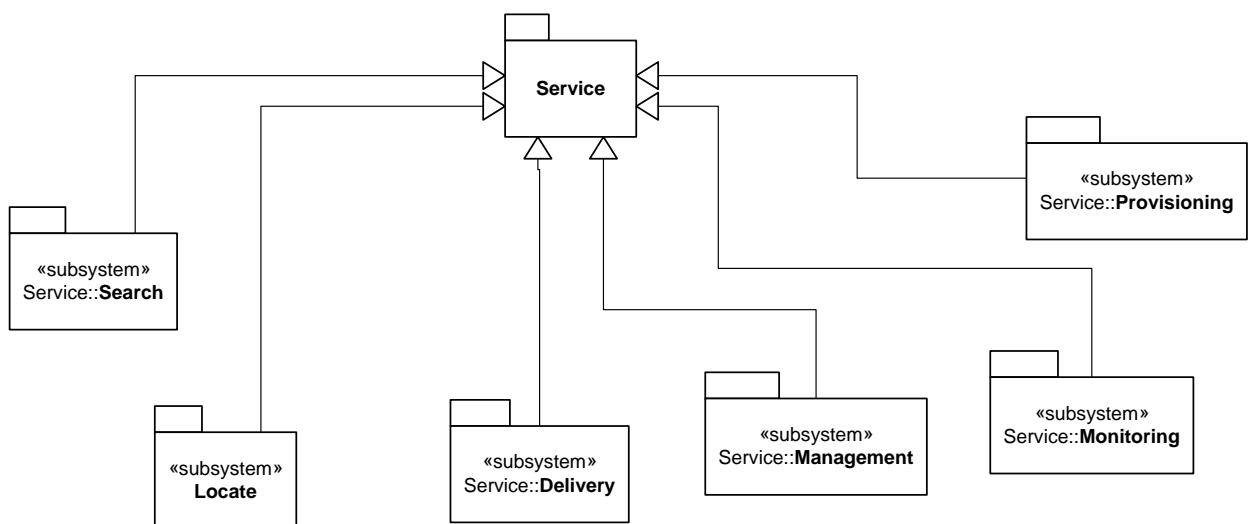


Figure 3. The Service component and its subcomponents

### **6.2.3.Compensation**

Compensation is about money (or some other valuable thing) flow between actors in the Flexible Service Ecosystem. Compensation ensures that it is possible to make business profitable and in its part that makes an ecosystem feasible. Compensation includes dynamic agreements, i.e., it is possible to make an agreement on the fly on how resources are used and how this usage is paid for. Compensation can be monetary or non-monetary, e.g., credits for different types.

### **6.2.4.Security functionality and Trust and Privacy**

Security functions are needed in every component of an ecosystem, and e.g. communication between components may need to be encrypted and at least its integrity must be guaranteed. Functions for protecting all components of security must be supported. From users perspective, security is manifested in two important subsystems: Privacy and Trust. Privacy has been defined as the right to be left alone. This means that user must be able to control her own information. Trust is harder to define simply. Here, trust is that parties can rely that the other parties act as has been agreed or is expected from them (explicit and implicit trust), and the ecosystem is protected from outsiders that are not trusted.

### **6.2.5.Authentication**

The Authentication system provides evidence, with selected means, that claims made by or about a subject are true. A typical case is verifying the identity of a person or a service. Authentication is often needed before authorization is given to a party.

### **6.2.6.Authorization and Access Rights Management**

Users and services must be authorized to perform certain actions, e.g., a service permits an operation to a party that has real or virtual currency to pay for the service. Authorization can use authentication to verify the identity of a party if identity verification is needed for access. Authorization can also be given based on some token (e.g., micropayment) without specifically identifying the party.

### **6.2.7.User Management**

User management is about taking care of user. User has preferences which has to be taken account if user wishes so. These preferences must be user controllable. Depending on service there is needed user interface adaptation from service side and also from user terminal side.

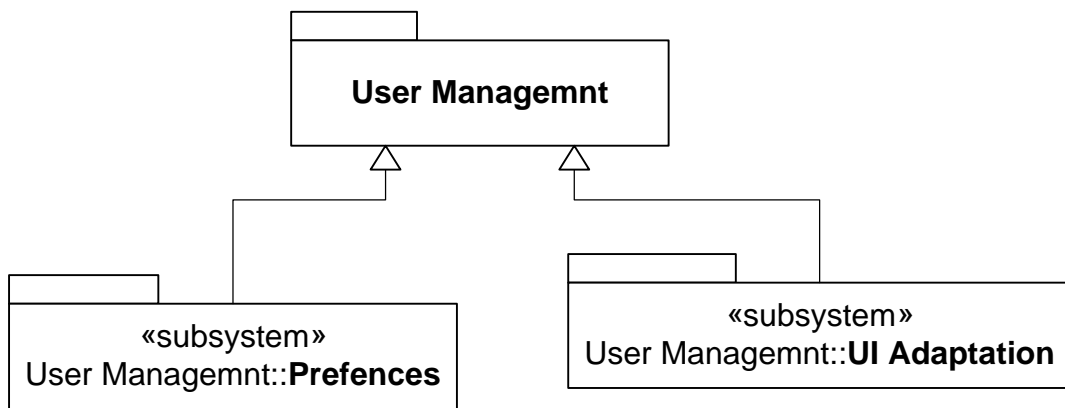


Figure 4. User management and its subcomponents

### 6.2.8. Media management

Media management consists of creating, publicising, distributing, and commenting of media. Actors of ecosystem need media management for handling of different type of media. This media can be text, pictures, voice or video and different actors need different tools. The ecosystem should support an easy management for different media types in terms of providing these basic building blocks.

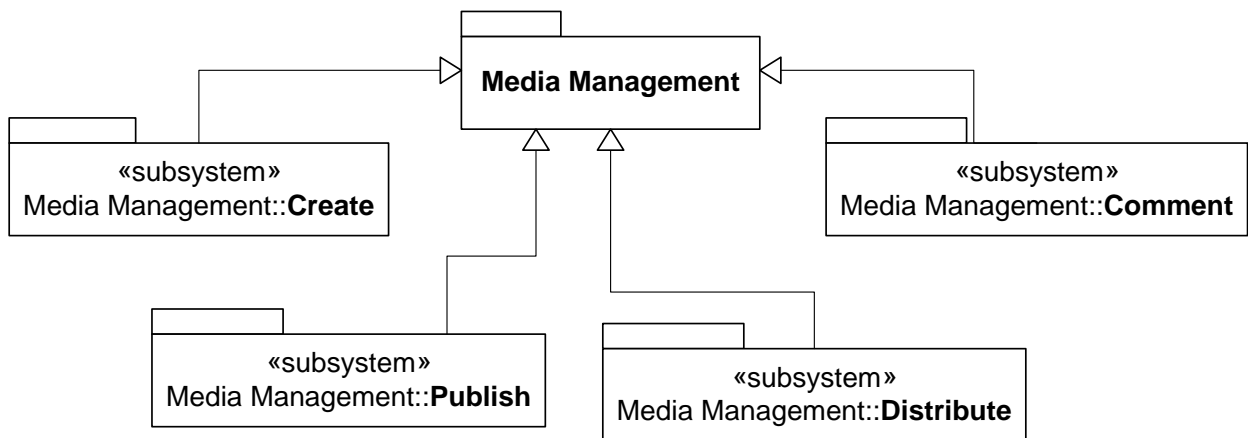
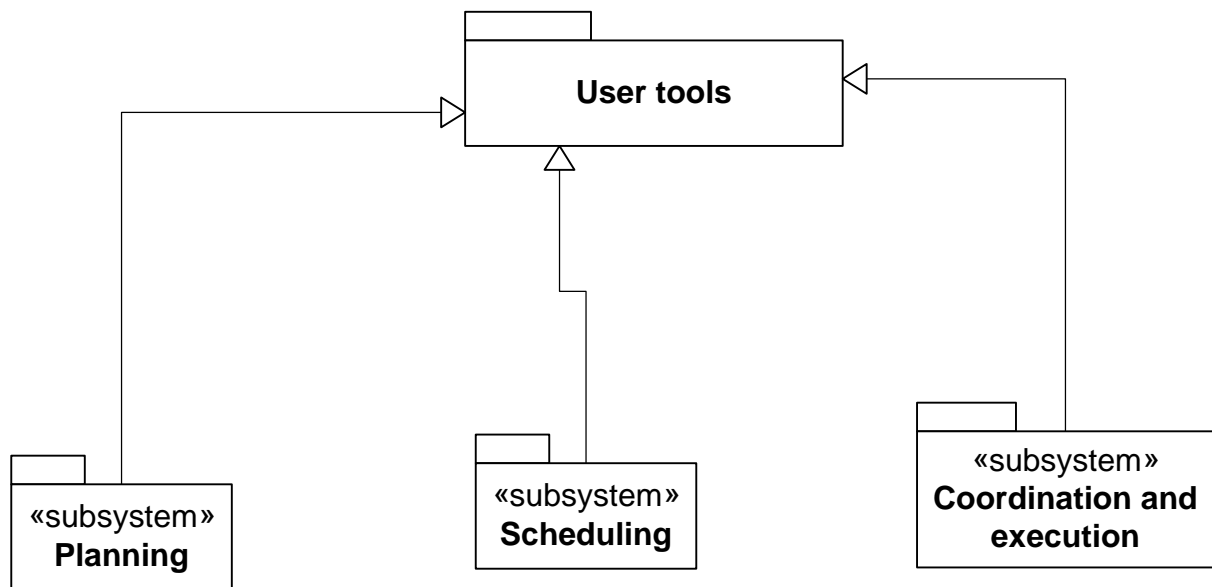


Figure 5. Media Management and its subcomponents

### 6.2.9. User Tools

Planning occasions needs certain type of tools and functions to be possible and feasible. After planning there is needed scheduling and proper coordination and execution of planned occasions. Therefore for the user there is needed planning, scheduling, coordination and execution functions in ecosystems.



## 7. Technical Architecture

There exist a number of technologies already that can be used to build the architecture. These are presented here with short descriptions and links to more information.

### 7.1. Service and Process Lifecycle Management

- Intel MashMaker, a Web browser extension to create mashups during browsing (<http://mashmaker.intel.com/web/>)
- Mozilla Ubiquity, an extensible command-based mashup creator (<http://labs.mozilla.com/projects/ubiquity/>)
- Karma, a mashup tool for integrating data from multiple Web source (<http://digarc.usc.edu/search/controller/view/usctheses-m1634.html?x=1242293718927>)
- Vegemite, a mashup tool for Firefox that allows storing and sharing the created mashups (<http://www.jeffreynichols.com/papers/p97-lin.pdf>)
- Microsoft Popfly, a service for creating Web pages, mashup, and games (<http://www.popfly.com/>)
- Yahoo Pipes, a composition tool for aggregating and manipulating Web content (<http://pipes.yahoo.com/pipes/>)
- iGoogle, a personal portal with the possibility of adding Web feeds and widgets (<http://en.wikipedia.org/wiki/Igoogle>)
- Netvibes, an AJAX-based portal similar to iGoogle (<http://www.netvibes.com/>)

- Yahoo Widgets, a service for creating widgets for Windows or Mac OS that run directly on the desktop (<http://widgets.yahoo.com/>)
- Facebook, a community site that includes a development environment for creating applications (<http://en.wikipedia.org/wiki/Facebook>)

### **7.2. Media and Content Management**

- Helsinki Testbed, a weather observation and forecasting system (<http://testbed.fmi.fi/>)
- A user observation database and air quality prediction model (<http://fmi.fi/products/air.html>)
- A broad range of environmental information covering the whole Finland, provided by the Finnish Environment Institute (<http://www.ymparisto.fi/syke>)
- XML, the de-facto standard data format for various data-driven services, with facilities to represent metadata (<http://www.w3.org/XML/>)
- XForms, an XML language and data processing model for XML and user interfaces to XML (<http://www.w3.org/MarkUp/Forms/>)

### **7.3. Context-aware Services**

- Elisa Zircle, a positioning and messaging service for mobile users (<http://www.zircle.com>)
- The Hubi platform, a map-based information service including events, public transportation, and weather (<http://owela.vtt.fi/owela/2009/04/07/hubi-palvelu-kokeilukaytossa/>)
- Google Earth, a global 3-dimensional map of the Earth and space (<http://earth.google.com/>)
- Mobile phones with near-field communication (NFC) capabilities (<http://www.nfc-forum.org/aboutnfc/>)

### **7.4. Identity Management Services**

- OpenID, an open decentralized framework for user-centric digital identity (<http://openid.net/>)

### **7.5. Payment Services**

- Etuile!, a Web service by Tieto for Web payments

## **8. Information Architecture**

The information needs in the system are related to two concepts: users and services. Technically, as users are also expected to act as service providers, the real-world difference between the two is perhaps not as clear-cut as one would imagine, but as architectural concepts there are distinct benefits in separating the two.

### **8.1. Users**

The information on a user divides into a hierarchy of levels. Each user conceptually has a single *physical identity* that is unique to that user. However, this physical identity is not visible in the system. Instead, *digital identities* are used to identify

users. As identity management is federated in the architecture, there is no specific central component for creation of new identities. Instead, users use whatever method their identity provider deems acceptable.

Each user can have several digital identities. A digital identity consists of a set of *attributes*, each of which has a specific value in that identity. *Identification* consists of a user presenting a subset of the attributes of one of her digital identities. Identities may be *correlated*, meaning that two identities can be revealed to refer to the same physical identity. Such correlation must always be the choice of the user possessing the identities.

In most cases, identification is further supplemented by *authentication*. Authentication is a process whereby some entity ensures that the attributes presented by a user during identification are correct. Conceptually, there exist authenticating entities that convert the provided attributes into *certified attributes*.

Users can also be members of groups, which can restrict or expand a user's access privileges. For each group, there is some rule that determines which users are members of that group. This can be as simple as a static list prepared by someone, e.g., the friends of a specific user, but it can also be complex or dynamic. An example of the latter would be all the users in a specific location, where group membership changes continuously, and there is no "administrator" who can grant membership.

## **8.2.Services**

On the information plane, the most important service-related part is metadata about the services. Metadata primarily describes what a service does and how it is invoked. This metadata needs to be in some language understandable by the invoker. If the invoker is a machine, the language needs to have well-defined semantics that define precisely the invocation pattern. Semantic descriptions of the service can also be a part of the metadata.

## **8.3.Access Control**

Controlling the access to information is one of the most important tasks of the information plane. Two different kinds of access control methods exist:

1. Access-control lists (ACLs): Each piece of information includes associated with it a list of principals to whom access is possible. Commonly, these principals can also be groups to make the ACLs of a manageable size.
2. Capabilities: Access is granted based on unforgeable tokens that identify both the piece of information being accessed and the access rights that are available.

Capabilities are useful when composing services, as the composed service uses only the capabilities that its invoker provides to it, avoiding a privilege escalation attack that is possible when ACLs are used. Capabilities also support delegation and can be made user-specific by including a user's identity inside the capability.

Like authentication, access control can sometimes be federated. In such cases, a single entity takes responsibility for determining whether to grant access. Such an entity is called a Policy Decision Point (PDP), and it works on behalf of a Policy Enforcement Point (PEP).

#### ***8.4. Combination***

Information is not just individual pieces of data but is linked with other pieces. These linkages can be explicit or implicit, public, private or revealed in time. Access control for any combined information is restricted by the access allowed for any individual part of the combination, but it is possible to set stricter control for the combined information, e.g., in cases where the combination method is not self-evident.

To properly combine pieces of information, it is necessary to have the syntax and semantics of that information known. It is not needed to have all the information follow the same rules, but as the combined information essentially needs to follow a superset of all its parts, there need to be some transformations into a common form from all the kinds of information that are being combined.

Combination of information is also needed when composing services. Service metadata will describe the semantics of invoking a service, which determines the ways in which it can be composed with other services.

### **9. Next Steps**

The architecture as described here is a first attempt at a feasible flexible services ecosystem architecture. Future work, carried out during the second year of the EDEN project, will refine the architecture further. Experience from pilot cases will inform the future work, as will any new use cases that may appear. The ecosystem as a whole will also influence further development, even if precise requirements do not appear in the future use cases.

### **10. Conclusion**

This document defined the Flexible Services architecture based on the requirements gathered from the use cases and on design principles that have been found to promote flexibility. A key enabler of flexibility is separation of concerns into different components, which permits separation of the components themselves and therefore allows outsourcing of generic functionality from services.