Introduction

The application architecture document serves two purposes:

- Define a high level architectural blueprint for e-government applications
- Describe a set of guidelines and best practices that should be taken into account when developing an e-government application.

The first section of the document elaborated on the overall architectural blueprint. Most e-government application share a common set of non-functional requirements. By using an application framework the developers can focus on implementing functionality and delivering business value. These frameworks typically offer support for transaction management, security, concurrency, remoting,… Applications expose their functionality to end-user not only directly but potentially also through the portal, and as a set of services to be used by other applications. This requires an additional set of framework services. These are services typically offered by a portal and service bus. The first section gives an overview of the support framework for building, exposing, using and integrating services. The concept of a service is key in this regard. Based on the concepts of a Service Oriented Architecture (SOA) we elaborate on the different parts of the overall e-government framework. Figure 1 gives a high level overview of the framework.

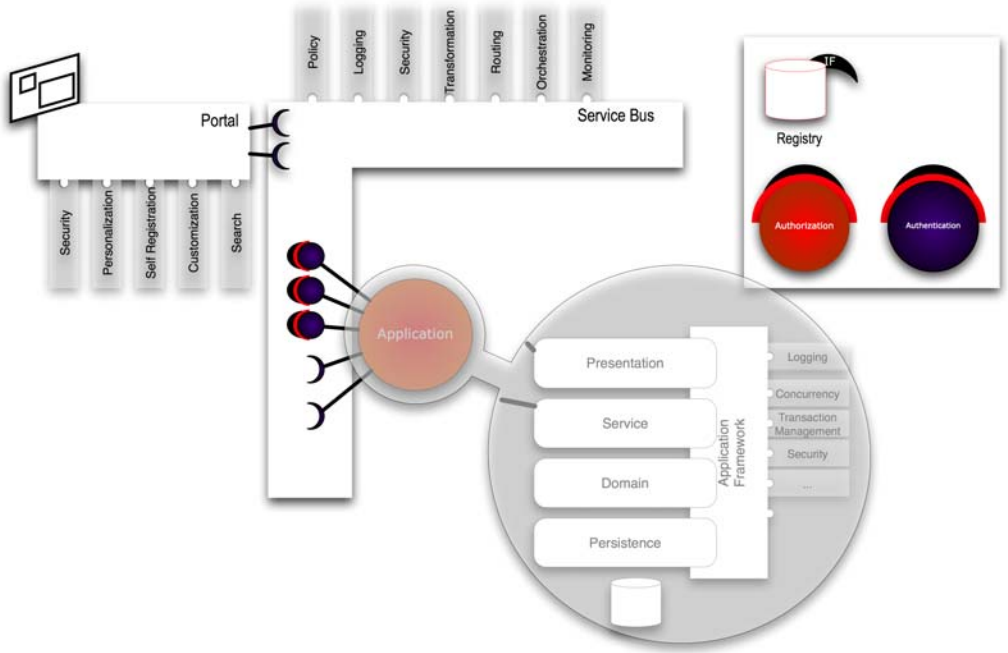

Figure 1 High-level overview of the architectural blueprint

This first section defines a set of common building blocks, their role and requirements. Most applications depend, to some extent on the services these common building blocks offer. In this regard one could question whether it is still appropriate that each application builds and maintains its own implementation of these services, and whether it wouldn't be beneficial if

these building bocks where offered by the framework, ready to be leveraged by others to build their applications. This becomes more and more important as integration between applications gains prevalence. In this context the potential evolution path of the e-government framework will be covered as well.

After establishing a common terminology, the second part of the document will go further in detail on how to develop applications by proposing a set of guidelines based on best practices. These guidelines should help implementers develop applications and services that are usable by others, and that meet the general standards for an e-government application. Typical questions that will be addressed in this section are:

- How to design a service interface?
- What information needs to be published with regard to the services?
- How should authentication and authorization be handled by an application?
- How to develop an application ready for deployment in the Portal?
- What principles and practices should be taken into account when developing an application?
- ...

Architectural Blueprint

Introduction

The section gives an overview of the different building blocks of fedict's e-government reference architecture. The primary goal of the section is to define the various building blocks of the architecture, and to describe how they work together. This will form the basis for further elaboration on the building blocks in guidelines and best practices.

The architecture adheres to Service Oriented Architecture.

Service Oriented Architecture

"Service-Oriented Architecture (SOA) is an IT strategy that organizes the discrete functions contained in enterprise applications into interoperable, standards-based services that can be combined and reused quickly to meet business needs".

SOA builds on the concept of interoperable, standards-based services. A *service* is an exposed piece of functionality with the following properties:

- The contract of, or interface to the service must be platform independent and standards based. The service consumer and producer can be implemented on any platform that adheres to these open standards. The use of open standard is key. It enables reuse of services across enterprise applications. The service produces and consumer communicate via the exchange of messages. The messages the service accepts and returns are defines in the contract of interface. The structure and design of the messages should be well over thought as it completely defines the usability of a service.
- The service can be dynamically located and invoked. This is often referred to as the *find-bind-execute* cycle (see Figure 2). Dynamic discovery suggest that a central registry is available where service consumers can query for services based on criteria (e.g. interface specification, quality of service attributes, location, cost…). Once a service has been found, the service consumer binds and invokes the service. When invoking a service a set of rules or policies need to be enforced in order to prevent unauthorized access.
- The service must be self-contained. That is, the service maintains its own state. This requirement mandates a particular interface design that will cover in a subsequent section, and is a prerequisite for easy integration with other services.

It is worth noting that the concept of a service is not limited to just faceless programming artefacts. An application that exposes its functions directly via a user interface in an easy integrable and interoperable manner can also be considered a service in the context of SOA. We will elaborate on this concept in a subsequent section.

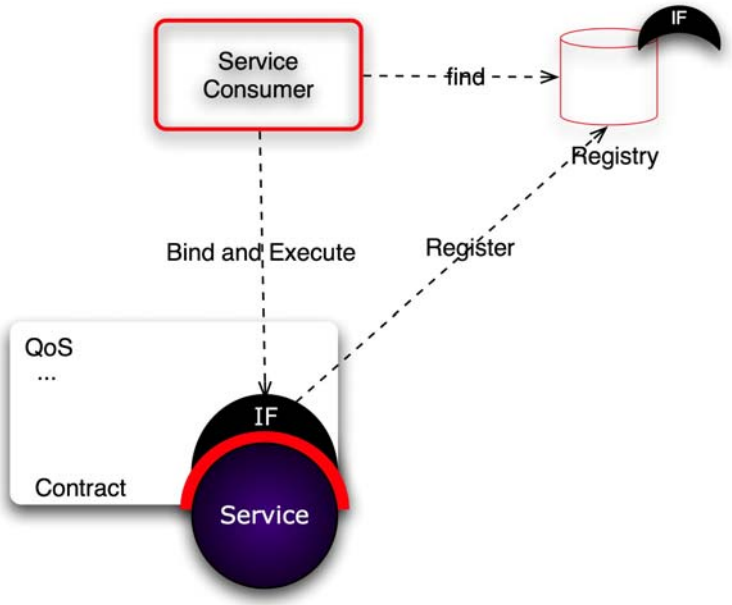

Figure 2 Find-Bind-Execute cycle for services

As the aforementioned definition states, there is more to a service-oriented architecture then just exposing functionality through a set of services. Actually services only form the starting point. Services can be leveraged to expose even richer services, and automate entire business processes. This will yield the true value of a service-oriented architecture:

- *Improving Efficiency* by leveraging existing investments to increase productivity. Reuse of existing functionality not only shortens the development cycle but also reduces the overall maintenance cost.
- *Improving Responsiveness* to stakeholders that support the business. The system needs to overall improve information gathering and fascilitate the use of the information flowing through the business.
- *Improving Agility*: Needing to rapidly adapt the business as the business changes, and avoiding having to begin from scratch with new applications and infrastructure as business requirements change. By publishing services , they can be discovered and reused more easily. This reuse forms the basis for improved agility.

The impact and success of an SOA depends only partially on pure technical aspects. Nevertheless a solid and well over thought architecture is the starting point. The architecture should be an enabler by providing a *platform* for exposing functionality through services. As such, the platform should offer a set of value-added services or building blocks that help in developing services, and addresses problems that service implementers all face, such as:

- How should users authenticate to the system?
- How will authorization or access control be handled?
- How and where should services be published and discovered?
- What standards and rules do services have to comply with in order to be reusable?
- How can the application chain be monitored?
- What about auditing of individual services, and the service composition?
- How can services be reused and integrated with each other?

...

In the subsequent sections a set of building blocks will be introduced. Each individual building block addresses one or more of the aforementioned questions. Their combination results in a high level architectural blueprint for a service oriented architecture.

Security

Protecting information, and ensuring access to the services is only granted to those permitted is a key requirement. Establishing identity is a critical prerequisite in determining the legitimate actions that a subject may perform. The term *Subject* is used to refer to an entity that is asserting its identity. A subject is person, organization, software program, machine, or other thing making a request to a resource. To gain access to the resource, the subject lays claim to an identity. *Identity* refers to an individual or an entity – such as a machine – that might represent itself to an organization.

Credentials are asserted as assertions: a claim that may be challenged before being believed, when the subject identity who possesses them wants to perform an action. Authentication is an assertion that a subject is who he claims to be, and it can be proven by verifying that the presented credentials are legitimately in the possession of the subject. Authorization is an assertion that the subject is allowed to perform a specific action, and it must be proven before the requested action can take place.

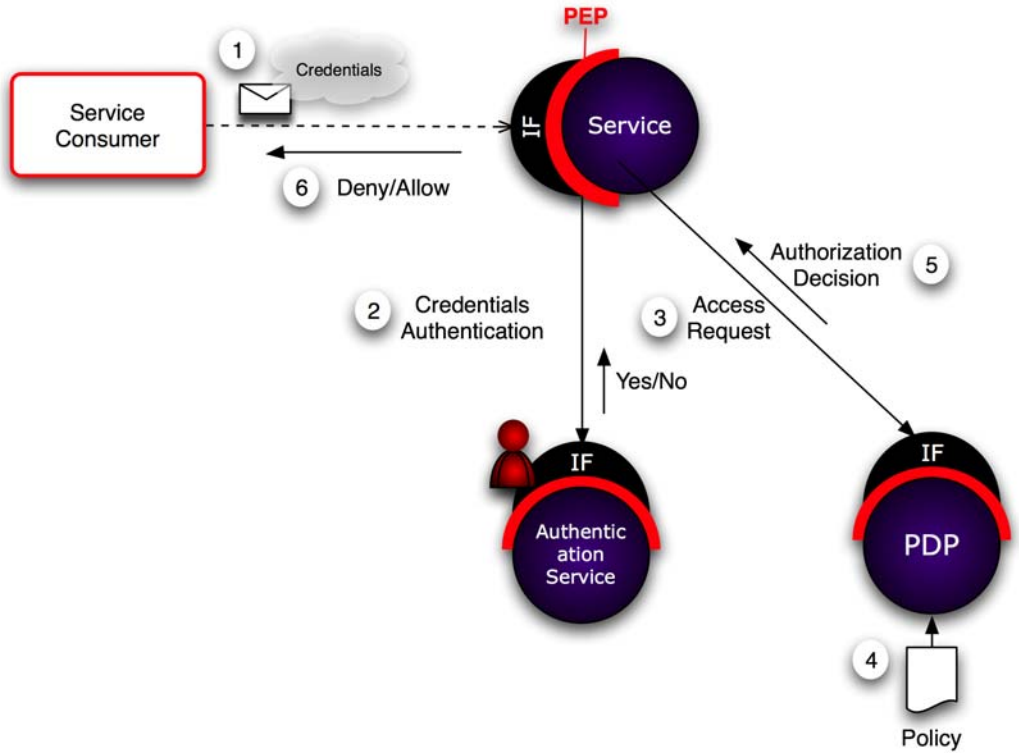

Figure 3 Overview of the typical security process

Figure 3 shows the typical steps in the security process:

1. In order to use a particular resource the user must have the appropriate identity. The user claims the identity by sending a set of credentials along with the request. The

credentials are then used as proof that the subject has the right to assert that the identity belongs to him.
2. When credentials are presented to the Policy Enforcement Point (PEP), they are verified by the authentication service. Credentials can be authenticated using many different methods, including simple username and password, an X509 certificate… The level of authentication required, should be proportional to the risk associated with accessing the resource.
3. Once authenticated the PEP asks the Policy Decision Point (PDP) whether access is allowed for the asserted identity.
4. The PDP uses a policy to determine the entitlements and permissions associated with the resource for the asserted identity. Entitlements refer to all the services and resources the identity is allowed access (e.g. disk, KBO, …). Permissions refer to the actions that the user can perform on the resource (e.g. update a record,…).
5. The PDP transfers the decision back to the PEP. This is often referred to as an *authorization assertion*.
6. Based on this information the PEP allows or denies access to the service.

A subject's identity can initially be established and verified in one trust domain and used in another trust domain. This identity is then said to be *portable*. Without portability, each organization has to acts as its own identification authority for its users. The organization has to know who its users are, and establish identity information about them, issue credentials (such as username and password), and manage the entire lifecycle of the user. In other words the organization is responsible of maintaining its entire user base. This model is no longer feasible:

- It prevents user from having an integrated user experience. Information does not cross the organization boundary, so the user has to re-authenticate, and present often different credentials. In the end, it is the user who pays the burden of crossing an artificial organization boundary. This is no longer considered acceptable.
- It limits the reusability of services. Services are used as the primary point of integration, and the user's identity must be passed to make valid authorization decisions. When a service calls a service in another organization, the user's identity may be passed to the other organization but it is not necessary valid there. The organization has the responsibility for authenticating and knowing all incoming foreign individuals. The latter would be very costly and very difficult to maintain.

The security services need to provide a solution that is flexible enough to allow individual organizations to retain their own authentication and authorization systems while solving the aforementioned integration problem. SAML (Security Assertion Markup Language) specifically addresses this problem, and plays an important role in the proposed security architecture.

There is more to security then authentication and authorization alone. Sensitive information should be appropriately secured when passed over the wire. In this regard one should make a distinction between protocol and content/message based security:

- Protocol based security, as its name suggests, works op the protocol level. SSL or TLS is the most common example of protocol security. It provides a secure communication channel between two parties, and does not interfere with the actual messages being passed between them. In order words, protocol based security is only concerned with the actual physical communication. After that it is up to the receiver to make secure the content is securely treated.
- Content or Message based security secures the actual content of the message being passed. This allows for a much finer granularity, for example only sensitive information in the message can be encrypted. Furthermore the contents of the message remain

secured. The owner of the private key can only decrypt the actual message. Message based security is not limited to encryption. It can also be used to digitally sign the message. The latter is important where messages should be non-repudiateable. The WS-Security standard focuses on message based security, and should be consider for securing sensitive services.

Authentication

Definition A Authentication[1]

The process of verifying an identity claimed by or for a system entity.
An authentication process consists of two steps:

- *Identification step: Presenting an identifier to the security system. (Identifiers should be assigned carefully, because authenticated identities are the basis for other security services, such as access control service.)*
- *Verification step: Presenting or generating authentication information that corroborates (confirms) the binding between the entity and the identifier.*

[1] IETF RFC 2828 Internet Security Glossary
http://www.ietf.org/rfc/rfc2828.txt

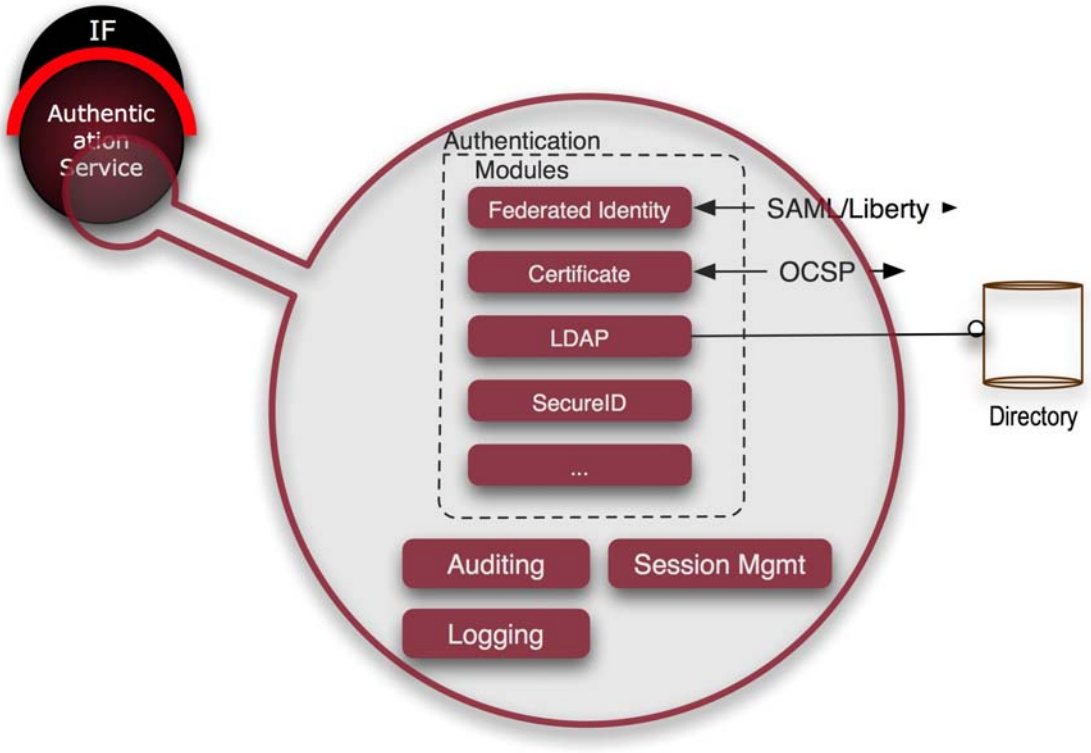

Figure 4 High level overview of the authentication service

Authentication plays a central role in the overall security architecture. Almost every application requires a subject to identify. An authentication services is as such a generic and reusable building block in the e-government architecture. The following requirements should be met by the authentication service:

- *Support for multiple authentication modules*: the authentication module should support multiple authentication mechanisms including eID, token based authentication, and SAML.
- *Support for SSO*: provides users with the ability to login one time, getting authenticated access to all their applications and resources. The user must not re-authenticate if he/she has already authenticated with an appropriate authentication level.
- *Federation*. The authentication system should support federated identity, and the common open standard in this area (e.g. Liberty, SAML). The latter guarantees interoperability between different implementations.

- *Auditability*; the authentication service is an important security boundary and must support auditing of authentication related events. Nevertheless the auditing features should take the end user's privacy into account.
- *Privacy*: the authentication service must be designed with end-user privacy in mind as such there must be no central audit log of which users accessed which applications.
- *Performance*; authentication is critical and as such the service may easily become a bottleneck. The service should be designed with performance in mind, and meet the highest performance requirements.
- *Fault tolerant*; the security service should be highly available. It is not acceptable that people cannot access the system because of a failure of the security service.

Figure 4 gives a high level overview of the authentication service.

Authorization

Definition B Authorization
An "authorization" is a right or a permission that is granted to a system entity to access a system resource. An "authorization process" is a procedure for granting such rights. To "authorize" means to grant such a right or permission.

Authorization is first and foremost a policy question. The policy is a reflection of the functional requirements and security objectives in the organization. Access control will automatically enforce these policies.
In the introduction to security, a distinction was made between the PEP (place in the system where to user requests access to a resource) and the PDP (place in the system where the actual authorization decision is made). In many cases the PEP and PDP are all part of the same system, but it is possible to separate the PEP from the PDP. The application could for example ask an external system to verify whether a user is allowed access, and get back a 'yes' or 'no'.
The separation of the PEP and the PDP allows for the creation of central PDPs that can be reused across applications. This would allow for services such as s *Central Delegation Service:* that manages a person's delegates, and has the responsibility to take access control decisions.

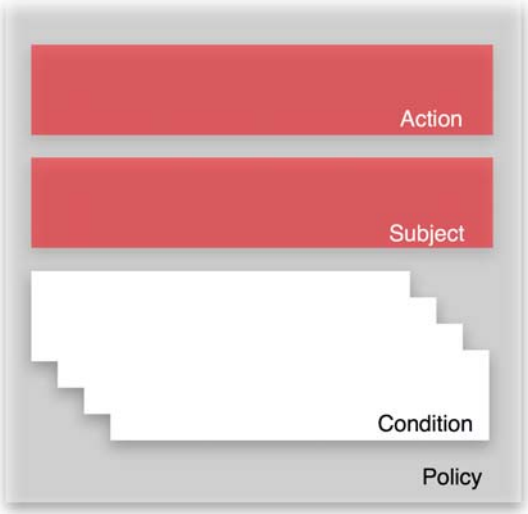

Figure 5 Generic Policy outline

Figure 5 shows an outline of a *Policy*. The Policy defines who (*Subject*) has the permission to execute a certain *Action* under certain *Condition*s. The subject itself could be an individual user, a static or dynamic group. Whether the user is part of a group can depend on assertions made by an attribute authority. An example of such a policy would be: doctors (*subject*) can view medical records of a patient (*action*) if they are affiliated with the patient (*condition*).

Integration

Services form the main integration point as they expose functionality ready to be used by others. Reusing existing services is one of the main drivers behind the adoption of a SOA. However exposing services should be done with care. The following should be taken into consideration:

- *Service interface*. The design of the service interface greatly impacts the usability of the service. Therefore the design should adhere to a set of principles outlined in section
- *Service versioning*. Once a service is exposed, others will depend on the service and its interface. Changing the service interface can impact all service consumers. Therefore the services must be designed to take service evolution into account.
- *Service Policy*. Every automated business task is subject to a set of rules and constraints. The policy expresses various characteristics and preferences that should be followed by the service consumer. This could for example restrict who can access the service, what type of security restrictions apply. This metadata is key in establishing good interoperability between services. It is important for the caller not only to know the interface the service exposes, but also what policies apply.
- *Manageability*. It is the responsibility of the service provider to make sure the exposed service is properly managed. The service level agreement is also part of the contact with the service consumer. As others depend on the exposed service, it should be properly managed and monitored.

It is very tempting to establish a direct connection between the service consumers and the service endpoints. However each service endpoint requires some form of security, management capabilities… This places a large burden on every service provider. Therefore

it is generally considered bad practice to use point-to-point integrations. An intermediary really helps the service provider by standardizing and offering these crosscutting services. This intermediary is often referred to as a *service bus*.

Service Bus

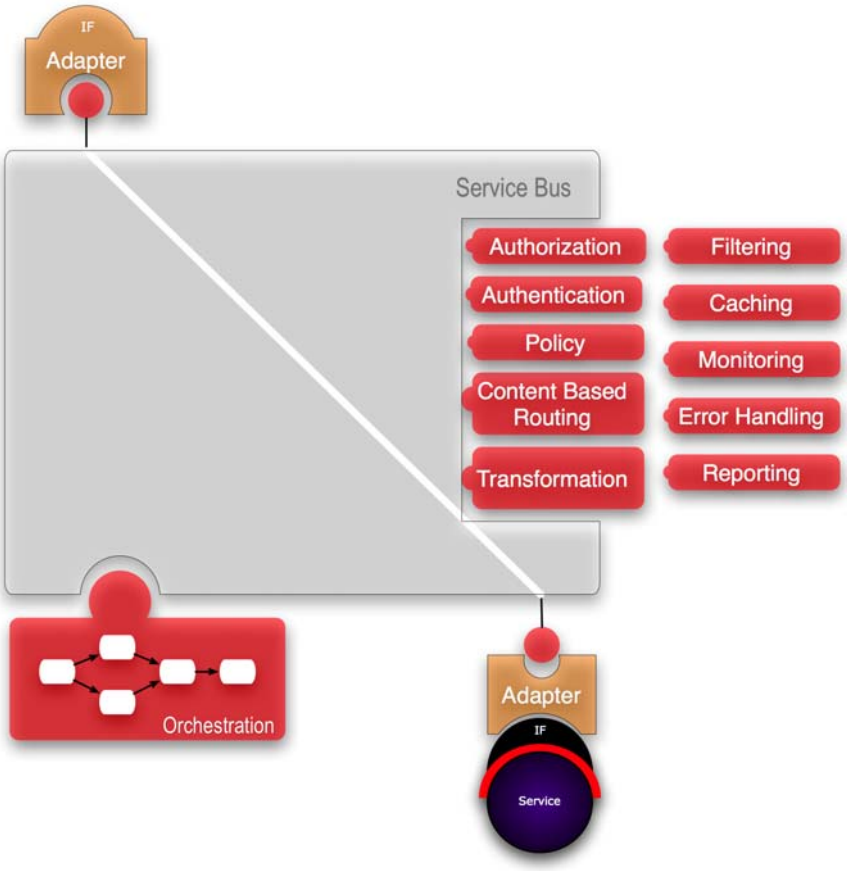

Figure 6 Overview of the Service Bus

Figure 6 gives an overview of the services the bus could offer to both the service consumer and producer:

- *Policy*. A policy service restricts valid message transmissions to those that conform to the policy rules and requirements for the service. The bus could enforce these policies.
- Authentication
- Authorization
- *Routing* is responsible to deliver the message to the appropriate endpoint. The service producer no longer has to know the exact endpoint address. Furthermore, this enables the bus to route the message to the most appropriate endpoint. The endpoint selection could be based for example; on the content of the message itself (content-based routing), or on any other metric such as physical location of the service, current load, etc. Routing makes the actual location completely transparent to the service consumer. Changing the address of the service only impacts the configuration of the service bus, and no longer impacts all service consumers.

- *Transformation*: the service transforms one message format to another. The transformation service can be used to mitigate changes to the service interface to some extent. Changes to the service interface can be handled by transformation as long as the new version of the interface exposes more information than the previous one.
- *Monitoring*: the bus can easily monitor all services it hosts, and can collect metrics about both service consumer and provider. This information can be used to verify the service level agreements, and send events when a certain exceptional condition occurs. In this context it is important that metrics can be collected based on per-consumer basis.
- *Error Handling and Exception Management*: often a set of actions needs to be taken when an exception is raised. The bus can offer the standard error handling, and trigger appropriate actions such as resending the message, informing administrators, without changing or intervening in the actual service implementation.
- *Reporting*: based on the metrics collected by the monitoring service reports can be produced. This exposes valuable information about the usage patterns of the different services, security violations, etc.
- *Caching*: the bus can optimize performance by using a cache. Message for certain often solicited services can be stored in cache and used to provide faster response times. This behaviour is completely transparent to the service consumer.
- *Filtering and data enrichment*: filters could protect the privacy of users by removing certain content from the message. The aforementioned transformation service could be used to filter the messages. It is important to note that the filtered messages should still comply with the original message interface. Data enrichment could be used to add additional information to the message.
- *Client abstraction*: the integration with the service bus must be as easy as possible, hiding the technical complexity of the integration/interaction with the service bus from the service consumer.

Orchestration

The introduction of a controller layer on top of the services would allow us to establish a centralized location for implementing composition logic related to the sequence in which services are executed. *Orchestration* is designed specifically for this purpose. It introduces a service capable of composing other services to a complete business process. Table 1 gives an overview of the characteristics of the process of workflow logic.

Table 1 Characteristics of process logic

Characteristics of process logic
Longer running spanning from minutes, week to months. Consequences: • State must be maintained and safe stored in a database for the duration of the process. • It is impossible to lock resources for the entire duration of the process, and changes are committed after each activity. As such it is no longer possible to rollback these long running transactions, and typically activities are rolled back by performing undo activities. The rollback or compensation handling can become complex.
Often asynchronous and synchronous in nature. Users or external systems may be triggered to perform certain activities or tasks (asynchronous). Services may be

triggered both synchronously and asynchronously. This results in the longer running nature. Coordinate work between different participants (both users and services).

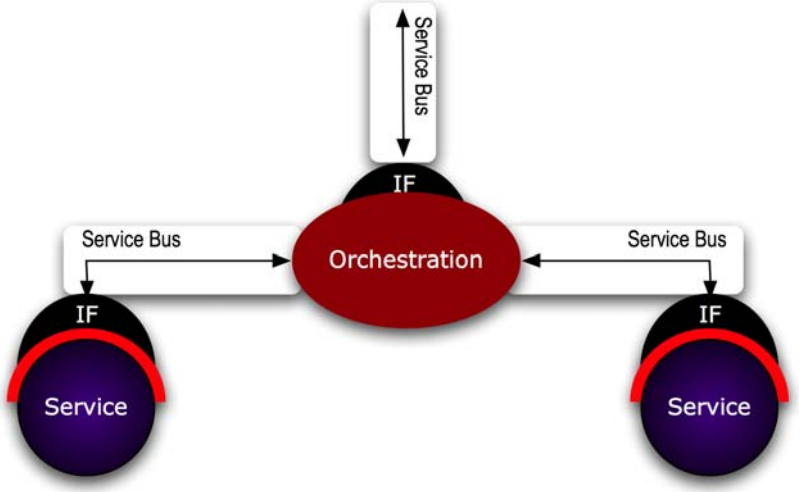

Figure 7 Conceptual overview of the orchestration service

As shown in Figure 7 the orchestration service makes use of the services the Service Bus offers, abstracting away some of the specifics of service invocation. In order to allow for the implementation of the process logic; the orchestration service must meet the following requirements:

- *Asynchronous invocation.* Most processes will be exposed as asynchronous services (one-way MEPs). Because of the typical long running characteristic of the processes the service consumer should not be waiting for the process to reply or complete as it can take minutes, days or weeks before the process finishes.
- *Flexible Correlation support*. As the process contains state, a mechanism is needed to correlate, for example an asynchronous response from a business partner with the appropriate process instance. Using a unique identifier for each process would be a trivial way of correlating messages. However, this would require the service interface to be adapted for integration into the process. The latter is not acceptable, and therefore a more flexible correlation mechanism is needed. This will allow correlating messages based on business data and communication protocol headers, and avoids the use of implementation-specific tokens for instance routing whenever possible.
- *Conditional branching*. Some activities are only mandatory when a certain condition is true while other need to be executed when the same condition is not met. Both Iterative (e.g. while / for) and conditional support (if) is a requirement.
- *Parallel execution of activities*. Activities that do not dependent on each other can be executed in parallel. Some processes wait for response while other activities can be active concurrently. Although it is always possible to reduce parallel activities to a sequence of activities this is considered bad practice as it has a considerable impact on the performance on the process. Moreover the performance reduction has nothing to do with the process itself but with the limited technical capabilities of the solution used to automate the process. Therefore the support for parallel execution of activities is a key requirement. So, forks and joins must be supported.

- *Wait conditions*. Process must wait until some condition becomes true. This is closely related to the asynchronous invocation requirement. Some processes may for example only continue when a response from a partner is received. The response times may vary heavily based on the service that is requested. A simple information retrieval service may respond almost instantaneous while other services require manual intervention on the part of the partner. Manual approval processes such as the decision on the validity of a request is an example of the latter.
- *Timers and invalidation of activities when triggers fire*. Some activities must be trigger automatically when timers expire. The automatic generation of reminders after a certain amount of time elapsed without response is an example. However when a response is received timely the timer should be invalidated automatically and no reminder should be sent.
- *Persistence*. The state of the process should be maintained in case of failure of the system. The process contains valuable information not only during the execution itself, but also after completion. The data can be used to calculate various metrics for auditing purposes.
- *Support for a large amount of in-flight processes*. The system will have to handle a lot of concurrent active processes. Typically this constraints the amount of processes that can be active in memory. Some processes should be kept in memory while others should be swapped to disk.
- *Version management*. Processes tend to change over time. Rules tend to change over time; processes should keep the process definition that was current at creation time. This requires the system to be able to handle multiple versions of the same process. Besides version management of running processes, the process definitions itself must also be versioned.
- *Monitoring*. The process coordinates different activities. It can be advantageous if the status of an individual process can be monitored. This would allow fast diagnosing in case of problems or failures. If the process blocks for example waiting for a response from a partner, the problem would surface immediately if adequate monitoring support is available.
- *Compensation support*. Because of the long running nature of processes, traditional ACID transaction cannot be used to execute the entire process as one unit of work. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and isolation cannot be maintained for the long periods during which errors and faults can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed during its progress, and the partial work done must be undone as best as possible. When things go wrong a set of application-specific activities that attempt to reverse the effects of a previous activity that was carried out, should be executed. The latter is known as *compensation*.

Discovery

Discovery related functionality is a key enabler in a successful adoption of an SOA. When services become available in the organization, they need to be discoverable by others in other be reused. However, for services to be meaningful to others, there is a need to provide information about them beyond the pure technical specifications of the service. Central in the concept of a registry is the representation of data and metadata about services. A registry offers a standard mechanism to classify, catalogue and manage services, so that they can be discovered and consumed.

The registry offers a standard way to represent information about services, and allows this information to be queried by interested parties. This allows for discovery of services both at design-time and at run-time. Typically the following scenarios apply:

- Lookup services implementations that are based on a common abstract interface definition.
- Find services providers that are classified according to a known classification scheme or identifier system.
- Determine the connectivity requirements for a given service such as supported security schemes, transport protocols, etc.
- Determine the policy for use of the service.
- Query for services based on a general keyword, or other metadata such as performance metrics, quality of service and cost.

UDDI (Universal Description Discovery and Integration) is one of the most important standards in this area. UDDI defines of a set of services supporting the description and discovery of services (Web Services) they make available, and the interfaces, which may be used to access those services. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable infrastructure for a Web services-based software environment for both publicly available services and services only exposed internally within an organization.

The aforementioned scenarios are supported by UDDI. Moreover, the specification has been written to be flexible so that it can absorb a diverse set of services and not be tied to any one particular technology. UDDI exposes its information as a web service, but this does not restrict the technologies of the services about which it stores information or the ways in which that information is decorated with metadata. UDDI has evolved over the years, and with release 3 has become very complete; supporting advanced yet essential features such as federation of registries.

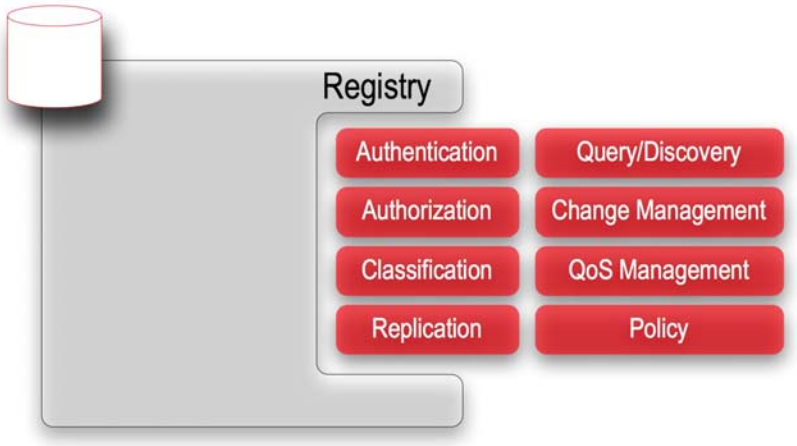

Figure 8 Overview of the registry services

In order to be reusable a service provider should be able to publish metadata about the service in a registry for discovery by others. The architecture should facilitate this by providing a registry as a building block, together with a set of guidelines on what information should be published. Figure 8 gives an overview of different services the registry adds.

Monitoring

The monitoring service really crosscuts all layers of the architecture. A problem at the application level, or even OS level or network level can cause a service to fail. Without a well integrated monitoring and management solution, managing the complete SOA becomes very cumbersome. Establishing a complete end-to-end monitoring framework is probably one bridge too far, as it requires hooks in the entire runtime infrastructure (network, database, application servers…). Nevertheless the service bus provides an excellent entry point for monitoring, as it handles all service requests. By gathering metrics about the various services it hosts and about the bus itself, bottlenecks can be pinpointed. In order to find the root-cause, the application that provides the service should be examined. This is the responsibility of the application owner.

End-to-End monitoring is considered to be the ultimate yet costly solution. We believe an appropriate service level can be guaranteed by offering monitoring services in the service bus, together with application specific monitoring support.

Auditing and Logging

Auditing is an essential part of any security design. The auditing information allows auditors to reconcile events that have taken place in the application. In this matter the audit log provides a trace of events that have taken place in the application that can be used for forensic purposes after a security breach. As the events for auditing are often not completely clear prior to development of the application, it is key to support easy addition or changes to auditing events being recorded. Because of the critical nature of the information being recorded the right security measure must be taken to make sure only authorized people can access the audit logs.

Where auditing traps security related events, logging is more focused on application and runtime related events. It allows system administrators, and developers to diagnose a problem that occurred in the application. The requirements with regard to the events being logged tend to change over time. Therefore the logging framework should be flexible enough to accommodate evolving requirements.

The auditing and logging requirement cross cuts all applications. Therefore a reusable component can be provided to facilitate, and standardize processing of auditing and logging events.

Content syndication and content aggregation

The definition of a service is not limited to a pure technical concept; a service can implement the interaction with an end user as well. This allows for the creation of a service delivery platform for the end user; a Portal that aggregates the individual services to create a uniform and integrated end user experience.

A Portal can be defined as: ‘a web based application that –commonly- provides personalization, single sign on, content aggregation from different sources and hosts the presentation layer of Information Systems. Aggregation is the action of integrating content from different sources within a web page. A portal may have sophisticated personalization features to provide customized content to users. Portal pages may have different set of Portlets creating content for different users.’

As such it is an entry point to a set of resources that are made available to targeted user communities. For most corporate portals, the set of resources include information, applications and other resources that are specific to the relationship between user and corporation. More in particular, the portal provides a point of entry to customers to access a controlled content aggregation service. Portals enable the identity-aware delivery of content and services based on a user's identity. Users are assigned roles within the context of the organization, and the content and services they see in their portal views reflects the usage policy associated with their identities. Users can customize their views within the portal as well. Traditionally, portals have been broken out into three types:

- *Government-to-Employee* portals, which provide information and services to its employees, both internally and externally.
- *Government-to-Business/Government* portals, which provide relevant internal applications and information to external partners, for example a supply chain management portal.
- *Government-to-Citizen* portals, which typically provide information about services and offerings to a target customer audience. The content of the portal is not limited to information distribution but may also provide access to self-service applications. Examples of such applications include self-customer care, self-registration, etc.

While classification of portals based on the user types holds true, this should not lead to multiple portal deployments within an organization. A single portal infrastructure can be used to provide multiple customized portal views that can focus on a single user community and aggregate and deliver content and applications required by the community. The same portal deployment should be able to use the same services (application, content, etc.) that provide external access to employees and additionally provide business partners with access to a limited set of resources. The primary responsibility of the portal service itself is the presentation of all information, applications and services to end-users. While presenting these resources, portals typically personalize the content so that end users interact with the resources in a meaningful, tailored manner.

Personalization is closely related to identity management, as it requires some type of identity management to register, classify, and recognize particular users before applying personalization rules to the content that these users see. Personalization includes the ability to automatically choose and modify applications that users are able to access or use, based on their roles. Personalization enables aggregation and delivery of personalized content to multiple communities of users simultaneously.

Support for syndication is essential. Without good support for syndication, content and applications are closely linked to the *portal provider* itself, and neither content nor services can be shared among different portals. The responsibilities of the *portal provider* should be limited as much as possible to Portal container itself. The container should offer standard support allowing others, more in particular the *Portlet providers,* to add their content and applications to the Portal in a standard way. The Portlet provider remains the owner of the services they publish.
As Figure 9 shows, this results in an interaction between the Portal and Portlet provider that is similar to the *find-bind-execute* schema covered in the Service Oriented Architecture section. The Portlet provider publishes its Portlets (services) in the registry. The Portal provider can query the registry for available Portlets, and add them to the Portal.

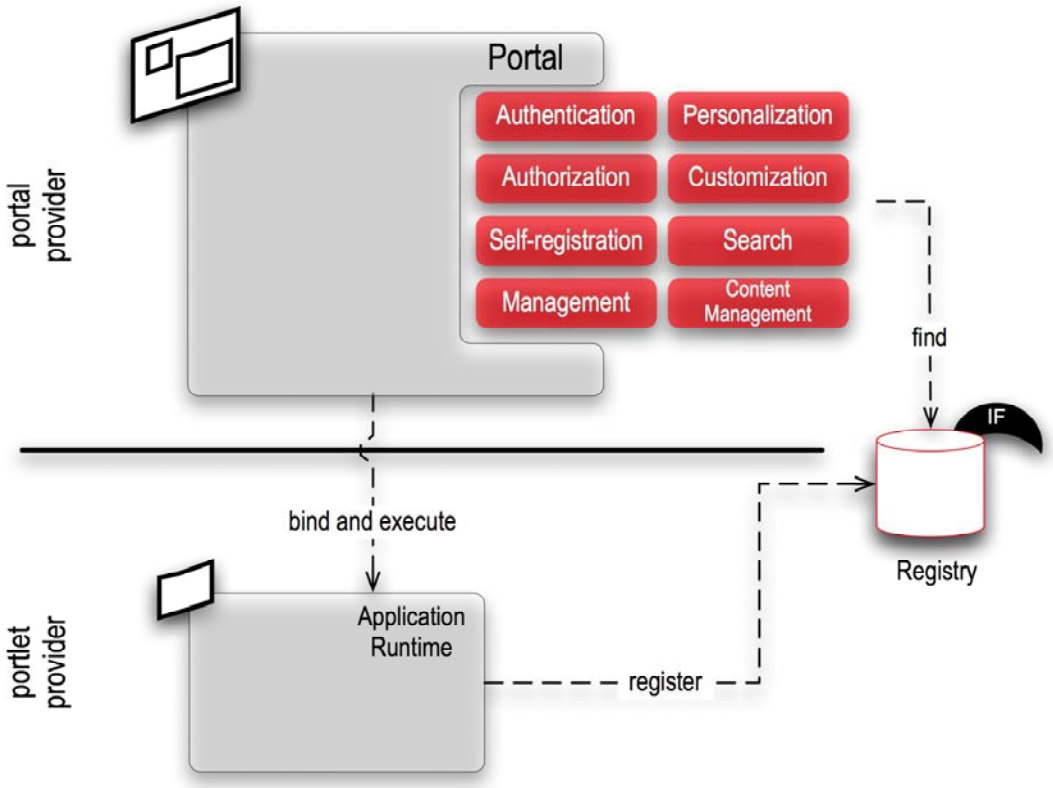

Figure 9 Overview of the Portal services

Use Cases

This section describes the role of the different architectural building blocks in a set of common use cases. This section focuses on the benefits the building blocks provide for all stakeholders.

Access an authoritative source

An authoritative source can be considered as a read-intensive data service. Accessing the service can be decomposed in the following steps:

1. The consumer looks up the requested service in the Service Registry. The registry contains besides the actual interface of the service, information on the applicable policy, quality of service, cost, etc. By querying the Service Registry the consumer may find additional providers.
2. After selecting the service based on the consumer's requirements, the consumer generates a stub for this service. The generated stub encapsulates the specifics of the interaction with the service. This approach aims at freeing the service consumer from the actual technical details of the interaction as much as possible.
3. The consumer accesses the data service using the FSB. The FSB will enforce the applicable policy with regard to security by collaborating with the *authentication* and *authorization* service. The FSB routes the request to the appropriate service. The FSB makes sure that all request are properly logged, and may use its internal caching service to speed up the request processing. The latter is completely transparent to the service consumer.

The proposed architecture holds a set of benefits for the service consumer:

- It simplifies the interaction with the service by hiding the technical specifics of the invocation as much as possible.
- By providing a central repository of services, it makes service easy to find and triggers the reuse of services.
- It standardizes the documentation (metadata) available for the services.
- It allows the service consumer to make abstraction from the actual location of the service. The FSB supports graceful evolution of the service interface.
- The central management infrastructure allows faster diagnostics in case of problems.

Expose an authoritative source

When exposing an authoritative source:

1. Metadata should be provided
2. Configuration on the ESB
3. Data aggregation/enrichment, transformation rules,…

Authenticate a user

Often a user has to be authenticated before accessing an application. The architecture proposes the use of a central and reusable authentication service. This will free the individual applications from handling authentication themselves. In this perspective authentication can be seen from a user's perspective and application developer's perspective.

For the application developer authentication is delegated to the central security server. The application needs to be integrated with the central security server. This can be done on an application level (using a Filter for example), or infrastructure/container level. The latter is

probably the easiest way, as it requires only the appropriate changes to the configuration of the infrastructure/container. In a J2EE context a custom Realm could be provided that handles authentication. As such the actual integration with the security service becomes transparent for the application developer.

The end user will benefit from a uniform authentication experience:

1. When a user tries to access an application, the system checks whether the user has already authenticated.
2. If the user has not authenticated yet:
    a. The user is redirected to the central authentication service and prompted to enter her/his credentials.
    b. The system validates the credentials of the user.
3. If the user is successfully authenticated he/she is redirected after the authorization decision to the actual application.

The use of a central authentications service holds a set of benefits for both the end-customer as the application owner:

- Offers the end user gets a uniform authentication experience.
- Supports Single Sign-on between different applications.
- Establishes single point of contact for user management.
- Minimizes the effort to integrate with the existing authentication mechanisms such as token and eID.
- Establishes a loose coupling between the authentication service and the actual applications. The authentication service itself can evolve with a minimal impact on the applications.
- Support for identity propagation when invoking internal and external services.

Authorize a user

Only authorized users are allowed access to applications. The rules or policies that apply should be enforced at all times. The decisions to allow of deny the execution of an action such as access to a certain URL, update an object… is made in the policy decision point (PDP). The PDP can be local to or embedded in the application, can be a global service or a combination of all three. Delegate management is an example of such a global service, as it implements the delegate management business processes cross application.

When a user accesses an application the following steps take place:

1. The policy enforcement point (PEP) accesses the policy decision point (PDP) in order to verify whether the user is allowed access. All information required to validate the policy should be provided.
2. The PDP evaluates the policy, and allows of denies access to the resource.
3. The PEP enforces the decision of the PDP and allows or denies the user access to the resource.

A central PDP has benefits, but also drawbacks:

- It establishes central source of information with regard to the policies that apply. It becomes possible to get an overview of the actions a user can perform.
- It provides a central framework for policy management.
- The PDP can become a bottleneck, as all decisions are made by the PDP.
- May require additional yet application specific information to be made available to the PDP in order to make the proper decision.

Establishing a single central PDP is probably not feasible due to the complexity introduced to support all policy needs for a widely heterogeneous set of applications. Nevertheless some policies can be enforced on a central level, while others remain the responsibility of the application itself.

Exposing a service

One of the most important things that need to be done when thinking about publishing a service is to design an interface for that service.

The public interface is the exterior view of our service, and is the primary interface that our clients will have to deal with. Therefore, some special attention needs to be given to the design of that interface, making sure it complies with the service interface design guidelines.

This part will outline the steps that need to be performed by a service producer before he is able to publish his service to the registry, allowing consumers to make use of it. In order to expose a service, the service producer will need to perform the following tasks:

- Design the service interface
- Provide an implementation of that interface
- Deploy the service
- Publish the service

As far as designing the service goes, please refer to the service interface specifications guidelines.

Design the service interface

In order to ensure interoperability between services, the service producer should be WSI Basic Profile compliant. The guidelines for service interface design are covered in some extent in the interface specifications guidelines document.

Provide an implementation of the interface

The implementer should only focus on the business logic and compliance with the interface. He shouldn't be aware of the fact that his service will be consumed as a webservice. In many cases, existing service logic can potentially be reused, and exposing that functionality usually involves exposing the service façade interface, or writing a web service façade around existing services within the application.

Deploy the service

The actual deployment of the service doesn't differ from traditional web application deployments. The service implementation will be deployed on an application server as is the case right now.

Publish the service

Services in a Service Oriented Architecture follow the Publish-Discover-Invoke model where a particular service is published to a Web Services Directory (UDDI). That service, once published, can be discovered by a service consumer. After discovery, the service consumer can invoke the service.

In any service oriented architecture, we need some kind of registry that is used to acts as a container for the various services. UDDI (Universal Description, Recovery and Integration) provides this registry mechanism so that clients can easily find the services they're interested in.It basically provides a registry API (based on WSDL / SOAP) for registering and discovering web services.

UDDI registries can be classified as:

- *Public* - UDDI registries can be accessed by anyone, while typically
- *Private* - UDDI registries will only permit access to authorized users.

Most UDDI registries provide some kind of integration with external authentication/authorization systems.

Consuming a service

In order to expose a service, the service producer will need to perform the following tasks

- Discover the service interface via UDDI
- Retrieve a reference to the service interface
- Invoke methods on the interface

Discover the service

The first thing a service consumer has to do is discover the service he's interested in. This is usually done via a UDDI registry.

The main piece of information a service consumer needs is the WSDL definition of the service. The WSDL describes the public interfaces associated with the service (public methods, parameter types, return types…)

Based on the WSDL file, we can generate the necessary artifacts that are needed to interact with the webservice.

Retrieve a reference to the service interface

Most web service toolkits allow for the generation of stubs/proxies (based on the WSDL file provided by UDDI), in order to hide the lower level plumbing code associated with accessing a webservice (connecting to the endpoint, marshalling & un-marshalling XML messages…).

The amount of work needed for a developer to interact with a webservice should be reduced to a minimum.

A first step in achieving this is to have the service consumer use a webservice toolkit that has been approved by Fedict. That way, development teams should make use of a standardized way of interacting with Webservices that will streamline the overall process of connecting to a Webservice endpoint.

The service interface API is the API that the client will use to consume the service. This API is typically provided by the webservices toolkit supported by the platform.

Should it prove necessary, Fedict could provide an API that acts as a wrapper for these webservice toolkits that could provide additional services (ex: guarantying the secured exchange of messages between parties), making it easier for development teams to correctly implement a secure end-to-end scenario using webservices.

Expose an application through the portal

The enterprise portal is increasingly becoming one of the basic building blocks in any service oriented architecture. All major vendors are offering a portal solution, and the industry is trying to consolidate on all the development efforts surrounding it by creating standards such as JSR-168 (Portlet Specification) and WSRP (Web Services for Remote Portlets).

A typical portal offering has the following features:

- a development platform for writing portlets
- an execution platform for running portlets
- an integration platform that can be used to incorporate a set of existing applications into the corporate portal.

The primary goal of the enterprise portal is to provide a uniform view to a set of different applications. Some effort needs to be put in integrating those applications within the portal environment. There are several integration types available that allow for existing applications to be plugged into the portal environment.

Different integration types

Link

The easiest way to integrate an existing application is by providing a simple link on the portal. The link will bring the user to the target application while leaving the portal.
The downside of this approach is that the application won't be integrated in the portal (upon clicking on the link, the user will enter the application, leaving the portal).

Single Sign On can still be achieved with this type of integration, providing that the necessary portal infrastructure has been put into place (this usually involves setting up Agents in front of the application in order to propagate the identity and achieve single sign on).

Recommended usage:

- When the target application is beyond your control and no customization is possible in order to integrate it with the portal.
- When no tight integration is required
- Provides the easiest and quickest way to add applications to the portal.
- No real requirements in terms of security need to be fulfilled.

This can be an integration strategy that will allow application that are remotely deployed to be plugged into the portal.

Web Clippings/URL Scraping

Web Clipping is all about capturing a subset of screen data coming from an external source. For example, an employee website might have a page that is able to show the employees working for a particular department.

Most Web Clipping portlets will have some kind of user interface, enabling the visual selection of screen portions by business users in order to integrate that external content into the portal.
In the screenshot below, you can see how a portal administrator has visually selected portions of a particular screen.

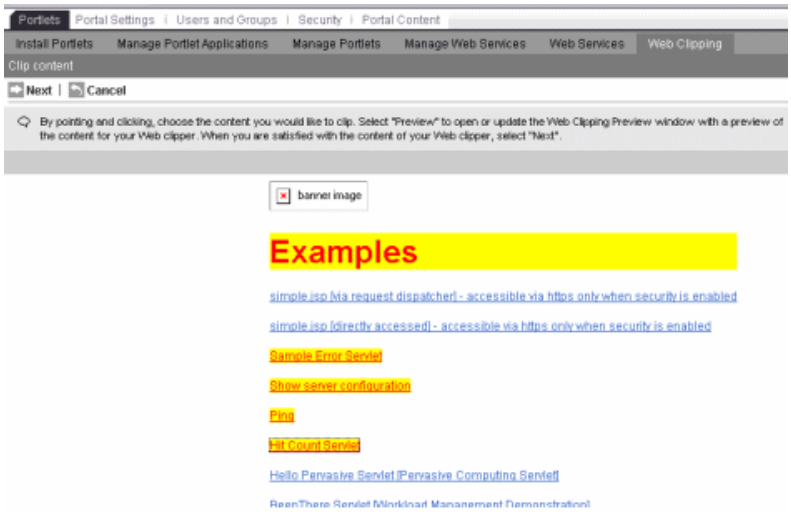

Figure 10 Clipping pieces of HTML

After selecting the required pieces of text, the portal administrator can preview his selection before placing it on the portal.

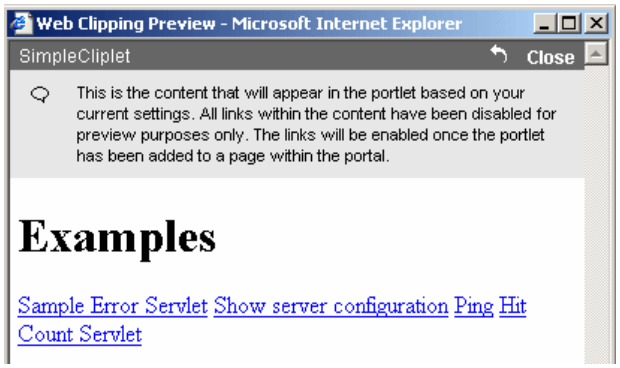

Figure 11 Web clipping preview

Once the administrator is happy with his clipping efforts, the resulting portlet can be placed on the screen. Updates to the original page will automatically be propagated to the new portal.

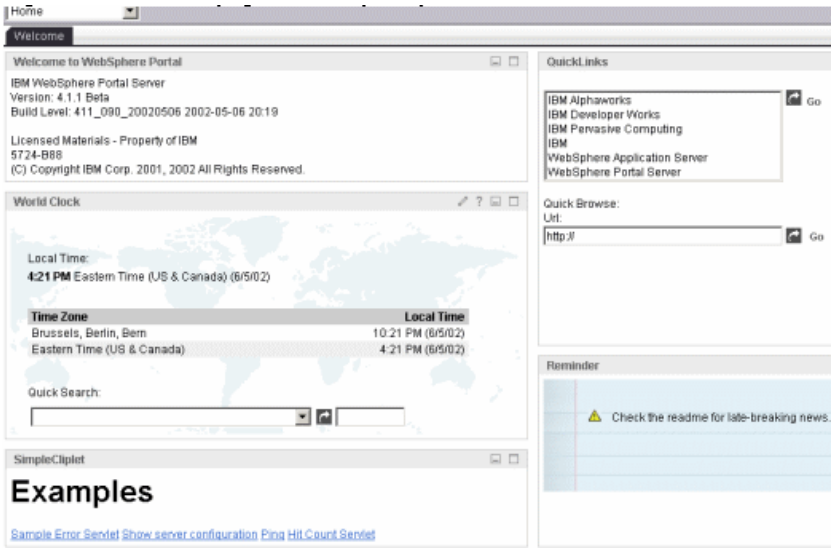

Figure 12 Web clipping portlet on the portal page

Recommended usage scenarios:

- Web clipping shouldn't be taken seriously when trying to integrate fully fledged business applications into the portal.
- It can be used however to show specific pieces of information that has been pulled from remote sites.
- This integration type isn't really suited for real application integration.

Display

Sometimes, we only need to portalize a small part of our application. Imagine a portal page that displays a portlet containing news items. We could write a simple portlet that displays simply the list of news items. Upon clicking on an item, the portal would redirect the user to the news application (full screen).

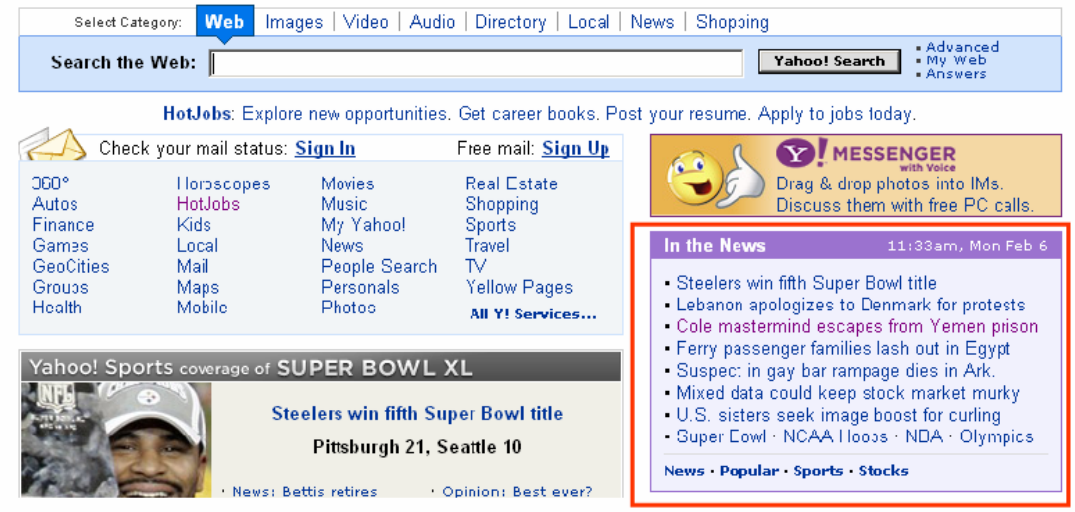

Figure 13 News Portlet integrated in the portal

Upon clicking on a news item, a new page is opened and the user leaves the portal desktop (as opposed to showing the news item in the portal page itself)

Figure 14 External news site

This goes to show that it's not always necessary to integrate the entire application to the portal. Provided that the application is well designed (using a multi-layered approach, with a clear separation of presentation and business logic), the business logic of the application can be reused to build the portlet. That way, the business logic of retrieving the list of news items could be reused inside the portlet implementation.

Recommended usage

- This integration type should be used to if an application has one or more screens that act as an entry point to the application, and provide an added value when being placed on the portal.
- A list of news items is a good example here, as it will give the users an immediate view of news item, well integrated in the portal.

Integrated

In order to have let the user perform its tasks without ever leaving the portal, we need to integrate existing applications by portalizing them completely. This involves the execution of all use cases via the portlet, allowing the user to stay on the portal.
This means that all screens that are provided by the application need to be able to run in a portlet.
The following screenshots shows an application that is spread out across 3 different portlets. The user has a uniform view into the application, and never has to leave the portal.

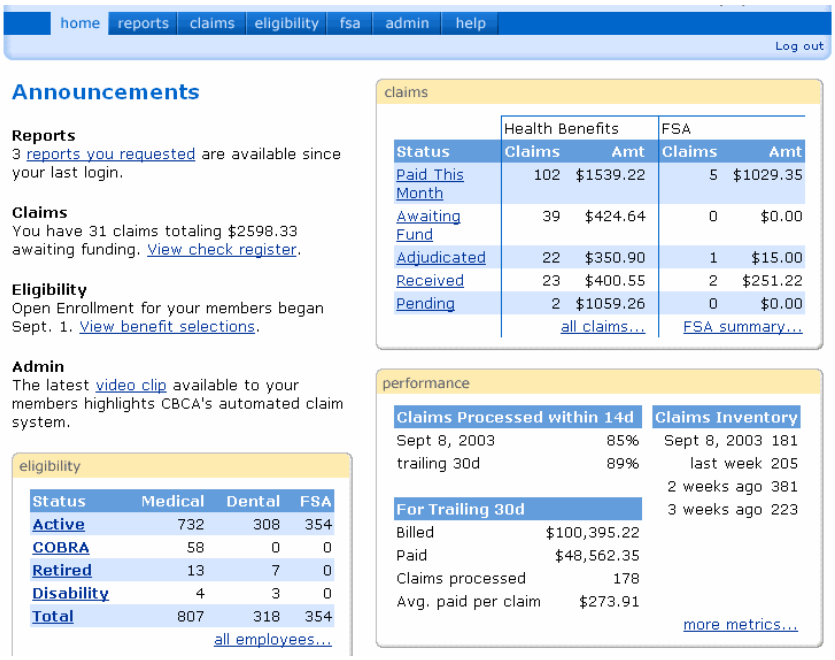

Figure 15 Application integrated in the portal

This solution involves writing presentation logic code in order to integrate the application into the portal. Service and business logic can potentially be reused, provided that the application that needs to be integrated has been developed using a layered approach, where the presentation logic is clearly separated from the service/business logic.
Obviously, this is the tightest integration pattern possible, and allows you application to make full use of services provided by the portal (single sign on, personalization…)

Recommended usage:

- If you have full control over the application, and the application has an added value when being run in the portal.
- When developing new applications that will only run in the portal.
- When developing applications that rely heavily on the services offered by the portal.

Remote Portlets

Another way to integrate existing applications into a portal is by using WSRP. WSRP (Web Services For Remote Portlets) is an OASIS web service standard that allows for presentation oriented webservices. These webservices do not only provide business logic, but also handle presentation logic associated with the service. By adding the presentation logic to the service, integrating existing applications into the portal becomes a lot easier, as no presentation logic (portlet code) needs to be developed in order for the application to integrate into the portal.

Without programming effort, a portal administrator can select various 'services' (remote portlets), and plug them into the portal.

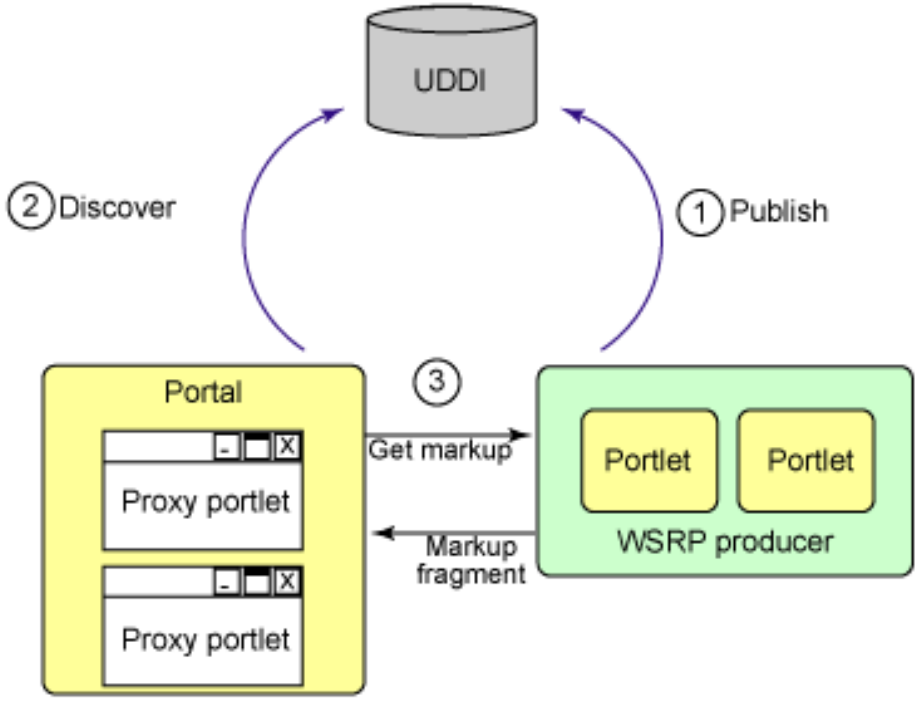

Figure 16 WSRP interaction schema

In the image above, we can see that

1. A WSRP producer has published a couple of services (portlets) into a UDDI registry.
2. A WSRP consumer has the ability to discover those services through the UDDI registry
3. The administrator for the WSRP consumer (typically also a portal server) can request the markup that is generated by the remote portlet, and aggregate that markup on the local portal.

Proxy portlets are used to represent the remote portlets offered by the WSRP producer on the local portal. These proxy portlets will simply proxy the user requests through to the WSRP producer, who in turn sends back markup fragments to the proxy portlet running in the local portal. In other words, WSRP allows you to reuse existing user interfaces when integrating applications into the portal.
WSRP uses SOAP over HTML for exchanging messages between the WSRP consumer and WSRP producer.

This is a fairly loosely coupled way of integrating remote applications into the portal. WSRP allows for security information to be propagated from the consumer to the producer.

Recommended usage :

- WSRP is great for integrating applications you have no control over. It's up to the WSRP producer to expose the portlet as a webservice.
- The WSRP consumer just needs a proxy portlet to connect to the actual portal instance

Automate a business process

Role of the BPM. Orchestration on service level.

Transition Guidelines

This section proposes a transition path from the current, as-is architecture to the to-be architecture. In every phase in the transition path a set of guidelines apply.

This is work in progress.

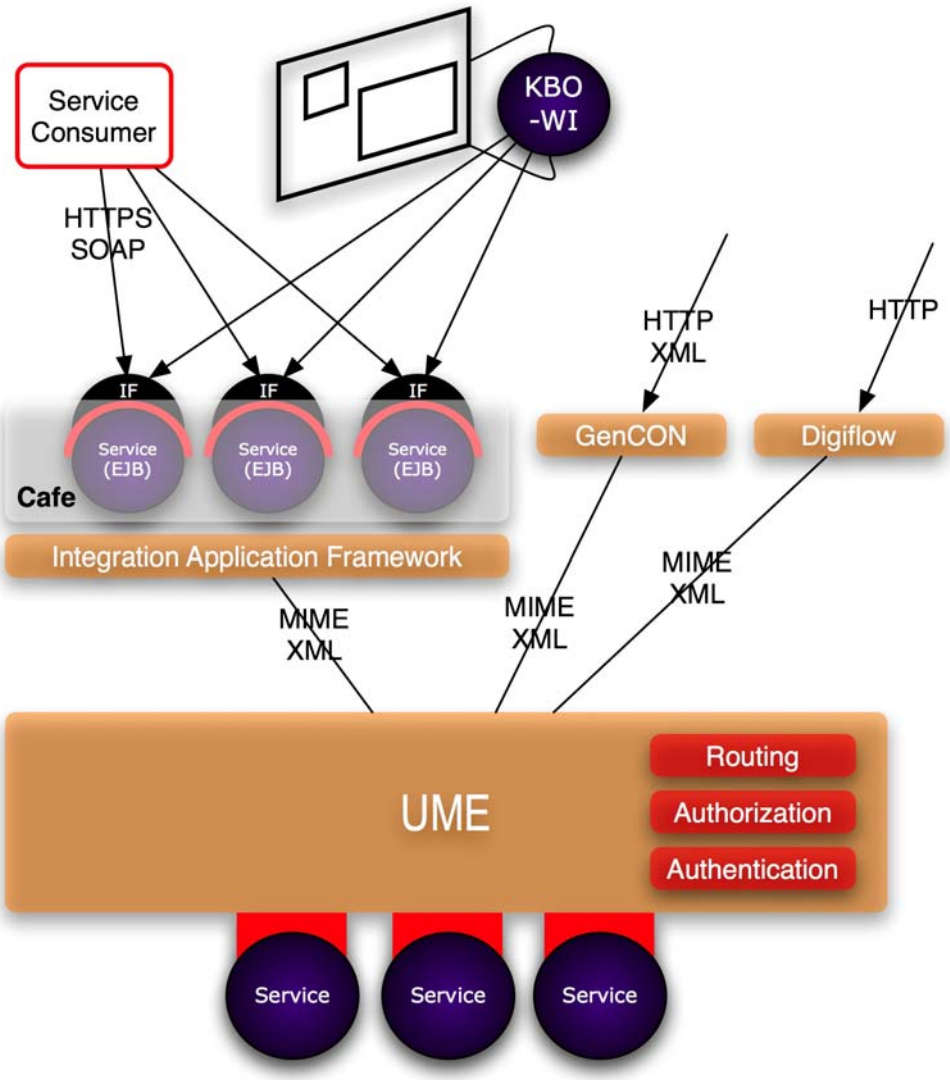

Figure 17 Current Architecture

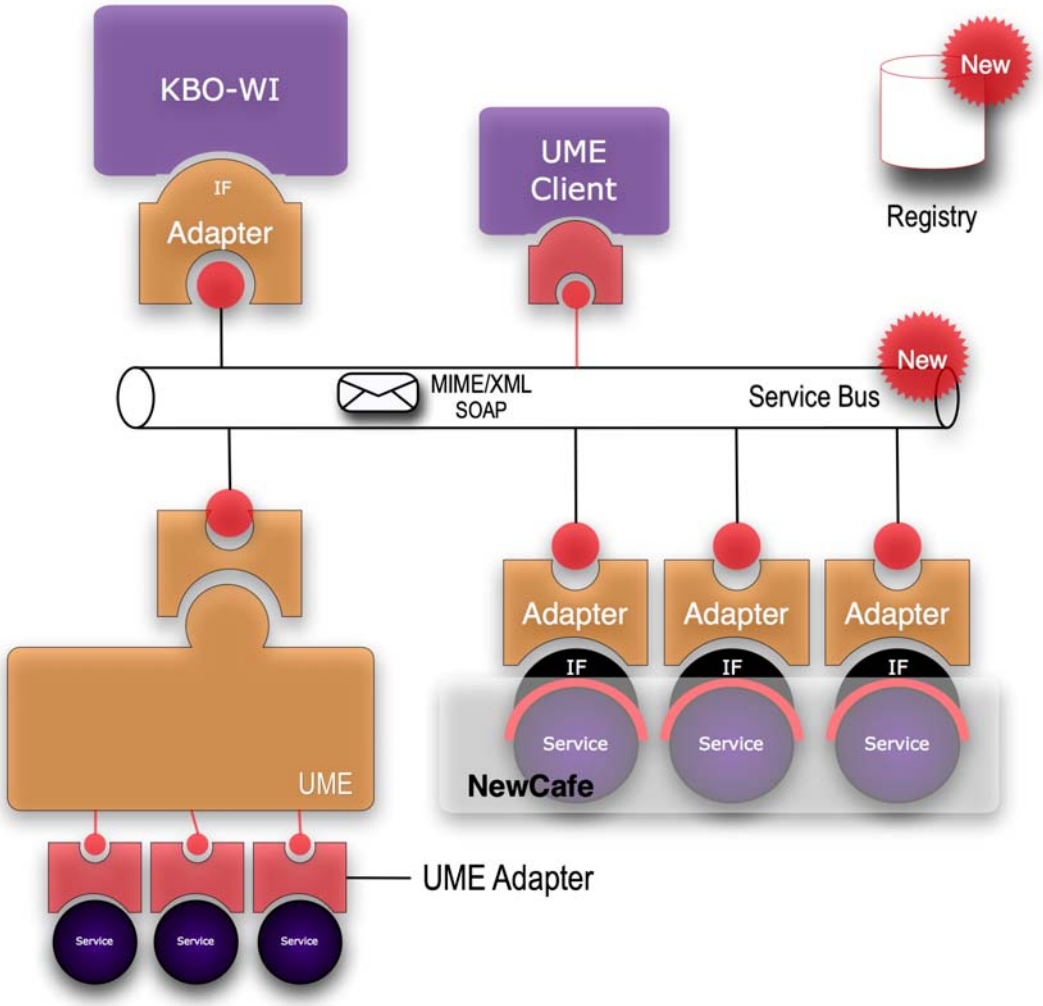

Figure 18 UME and FSB Coexistance

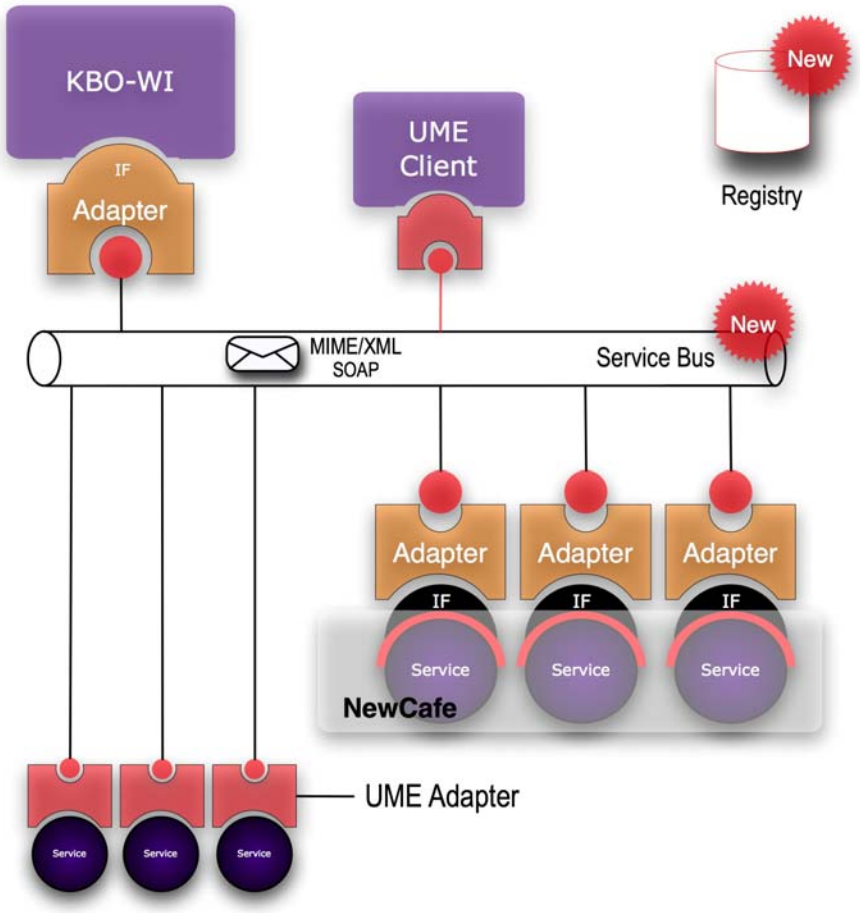

Figure 19 Phase out of UME

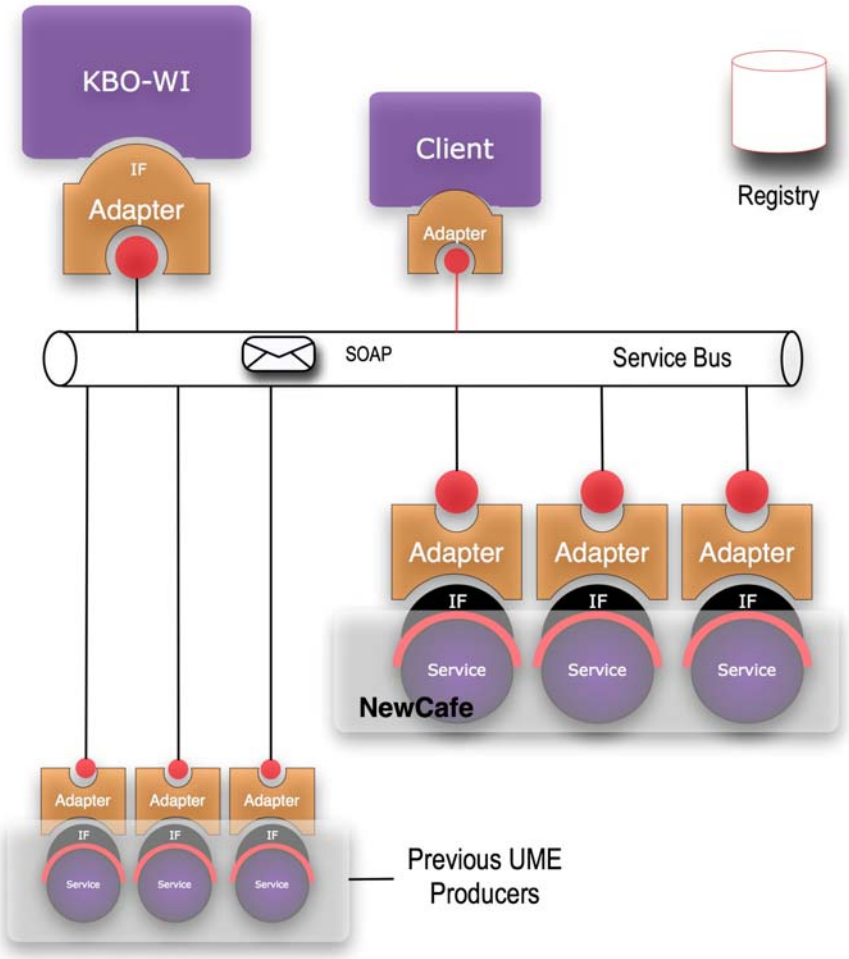

Figure 20 Web Services