

Flexible Services

EDEN

D4.3.4 Architectural alternatives for service ecosystem

Document Name	Architectural alternatives for service ecosystems
Project/WP Title:	EDEN WP4 - User-driven Service Ecosystem Evolution
Document Type, Security	P (Public)

Document Title:	Deliverable 4.3.4: Architectural alternatives for service ecosystem
Agreed date of delivery	31.5.2009
Actual date of delivery	
Editor	Heidi-Maria Rissanen
Version and	Version 1.0
Date Last Change	31.5.2009
File:	FS-EDEN-D434-v10.doc

Participants	Name	e-mail
Oy LM Ericsson Ab	Heidi-Maria Rissanen	heidi-maria.rissanen@ericsson.com
TKK	Seppo Törmä	seppo.torma@tkk.fi

1. Table of contents

1. Table of contents	2
2. Figures.....	3
List of Acronyms and Abbreviations	4
3. Introduction	5
4. Messaging	8
4.1. Capabilities of message servers	8
4.2. Interfaces and Protocols.....	9
5. Message Format	10
5.1. SOAP	10
5.1.1. Features of SOAP	10
5.2. REST	10
5.2.1. Features of RESTful Services.....	11
5.2.2. REST Anti-patterns	11
5.3. SOAP versus REST?.....	12
6. Description Languages	13
6.1. WSDL.....	13
6.2. WADL	14
7. Data Interchange Format	14
7.1. XML	14
7.2. JSON (JavaScript Object Notation)	14
7.3. Comparing XML and JSON.....	15
8. Discussion	15
9. References.....	15

2. Figures

List of figures

Figure 1: Concept map of messaging, formats, and languages..... 7

List of Acronyms and Abbreviations

AJAX	Asynchronous JavaScript with XML
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
B2B	Business-to-Business
EAI	Enterprise Application Integration
IETF	Internet Engineering Task Force
JMS	Java Message Service
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
NGO	Non-governmental organization
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SOAP	was: Simple Object Access Protocol, now: just SOAP
WADL	Web Application Description Language
WSDL	Web Service Description Language
W3C	The World Wide Web Consortium
XML	eXtensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol

3. Introduction

Distributed systems consist of *multiple components* that communicate with each other and coordinate actions by sending messages (Coulouris, 2005). Distributed systems are characterized by the independence of the components: they run concurrently, have no global clock, and can fail independently.

The Flexible Services program develops technologies for a specific type of distributed system: a service ecosystem. *Service ecosystem* is an environment for service provision and consumption consisting of different kinds of interacting participants. In the minimum it provides the means for the participants to communicate about services, i.e., their content, context, and terms of delivery. The participants may include individuals, companies from local businesses to large enterprises, governmental organizations and NGOs, as well as automatic agents such as crawlers, brokers, and mediators. The structure is not just a flat network of service providers and service requestors but there can be multiple layers of services, including monitoring, evaluation, recommendation, dispute management, and so on. There can be multiple motivational mechanisms, such as money, reputation, social interaction and environmental concerns.

The open, flexible, and self-organizing nature of service ecosystems poses demands for the information systems that support them. In EDEN Deliverable 1.2.1 the following principles have been defined for the Flexible Services Architecture:

1. *Modularity* - System is composed of clearly separable components that can be understood and developed independently.
2. *Transparency* - The interfaces exposed by components and the operations exposed in the interfaces are clear, well defined, and preferably machine-inspectable.
3. *Loose coupling* - There are no unnecessary dependencies between components. While there are real useful dependencies between components (otherwise they would not be part of a same distributed system), in practice many of the dependencies are artificial and unnecessary; for example, the need to use same language or platform, particular API, synchronous communication when asynchronous would suffice, and so on.
4. *Composability* - The components are recombinant and can be selected and assembled in different configurations to satisfy specific external requirements. Composability deals with the way connections between components are established and specified.

5. *Late binding* - The creation of connections between components is postponed to run time, which makes it possible to use components available in particular situations, and make the binding decision with maximum information.
6. *Reflection* - The running system can inspect its structure and state, and modify the composition and the state at run time through generic interfaces.

The question behind this deliverable is the following: *What kinds of technologies should be used when developing systems that conform to these principles?* Since the focus is on distributed systems - consisting of multiple components that communicate by sending messages - this question translates into more concrete questions: How should the messaging between components happen? What is the format and content of messages and how can they be described? How are components and their interfaces described?

There are many existing technologies developed recently for service-oriented and event-driven systems that address these questions and have at least some of the desired properties.

In this deliverable the focus is on technology alternatives that concern the following topics:

- Messaging - Direct messaging or using a message server
- Message formats - Architectural styles of SOAP vs. REST
- Data interchange formats - The payload of messages: XML vs. JSON
- Description languages - Description of the interfaces of components: WSDL vs. WADL

Figure 1: Concept map of messaging, formats, and languages

The choices in these areas are not independent of each other; neither are they totally dependent. There is the realm of SOAP, XML, WSDL, and WS-* standards that are in one package, while REST, JSON and WADL are in another. A concept map illustrating this is shown in. REST can, however, be also used together with XML and WSDL. SOAP can be used on message queues (e.g., through WS-ReliableMessaging) and publish/subscribe (e.g., through WS-Eventing). There are also REST-based messaging systems, like RestMS¹.

¹ <http://www.restms.org/>

This document is the first version of EDEN Deliverable 4.3.4. Its role is to introduce the alternatives that will be studied during the second project year. The second version of this document - due 5/2010 - will analyze the alternatives and document the experiences of different approaches used in pilots and tools developed in the Flexible Services programme.

4. Messaging

The first question concerning the communication in a distributed system is whether it is based on *direct messages* between components or a *message service* acting as an intermediary between the components. In the physical world an analogy would be whether people communicate directly with each other or through a postal service, bulleting board, or other such mediating mechanism.

Direct messaging is naturally a more straightforward approach; it is simpler and often more efficient in small service systems. Message services - or more generally *message-oriented middleware* - introduce additional intermediary components to the system but in larger service systems they may lead into simpler overall architecture as well as simpler programming model for the system.

4.1. Capabilities of message servers

Message services can support a wider range of *messaging patterns* than direct messaging that is based on *one-to-one* communication. They are suitable for *one-to-many* communication, in which sender wants the same message to reach multiple recipients. It is also possible to have *one-to-one-of-many* communication, which is pattern encountered for example in cloud computing.

In general, message services can promote *looser coupling* between the components of a system. This can happen in several different dimensions depending on the capabilities of the message server.

- If the message server maintains a *message queue*, the interaction between the components can happen in an *asynchronous* manner. The recipient of a message does not have to be available at the same time when other component sends the message but the message service takes care that the message is delivered next time the recipient is available. In other words, the *temporal coupling* between the components is reduced. This can also improve the overall performance of the system, as the sender does not need to wait for a response to its message, that is, to *block* its execution. There is much communication that either does not require immediate response or any response at all, for example, when updating log databases.
- If a message server that supports *publish/subscribe*-interaction style - or *pubsub* for short - the communicating components do not even need to know each other's identities, let alone addresses or interfaces. The routing

of messages from a sender to recipients is based on the content of messages. The recipients make subscriptions to particular content-related channels, topics, or query expressions. The sender publishes a message and all recipients with matching subscriptions are automatically notified about it. Using pubsub the components are loosely coupled with respect to each other's identities and the topology of the service system.

- If the message server supports *mediation* between components - such as so-called *service busses* do - the messages or the data communicated does not need to be in a format that both parties understand. The message server can do transformations to the messages sent based on knowledge of the formats that the recipients support.

4.2. Interfaces and Protocols

There are different interfaces and protocols available to support message servers:

- JMS (Java Message Service)² - The messaging API of J2EE platform specified by Sun. JMS has been implemented in numerous commercial and open-source software packages. JMS supports asynchronous and reliable communication. There are two communication models: *point-to-point* for messaging between two parties, and *publish/subscribe* for topic based one-to-many communication.
- XMPP (Extensible Messaging and Presence Protocol)³ - Originally an XML-based instant messaging protocol of Jabber system. XMPP is standardized by IETF. In addition to messaging, XMPP is meant for transferring presence information between components - i.e. information about the availability of components - for example, who are on-line in an instant messaging system. In its basic form XMPP supports neither message queuing nor publish/subscribe interaction pattern, but they are available as XMPP Extensions. In general, XMPP has been developing into a general protocol for messaging, not just instant messaging.
- AMQP (Advanced Message Queuing Protocol)⁴ - A wire-level protocol for high-performance messaging, origination from banking sector. AMQP is being developed by an industry-driven AMQP Working Group with the intention produce an open and platform agnostic messaging protocol broadly applicable for enterprise use. The protocol is still to reach the version 1.0. For performance reasons AMPQ is designed as a binary protocol. Basic concepts of AMPQ are *exchanges*, *message queues*, and *bindings*.

² <http://java.sun.com/products/jms/>

³ <http://xmpp.org/>

⁴ <http://amqp.org/>

Exchanges route messages to particular queues according to their mutual bindings. These concepts make it possible to emulate the behavior of a publish/subscribe interaction.

- RestMS (Rest Message Service) - An AMQP compatible protocol for messaging that aims to separate the control protocol from the data protocol. The control protocol is implemented in a RESTful manner, using text commands and synchronous interactions. The data protocol is asynchronous binary protocol as in AMQP. The designers are same as those behind original versions of AMQP; the work on RestMS is still very much in progress.

5. Message Format

SOAP and REST (Representational State Transfer) are two different approaches of developing web services. SOAP is a heavily standardized protocol whereas REST is a lightweight, non-standardized, developer-led design approach. There has been - and there still is - a big debate between these two approaches in the web service community. In the following, we will describe the features of these two approaches.

5.1. SOAP

SOAP is a messaging protocol used for accessing Web services. It is standardized by W3C. SOAP originally stood for Simple Object Access Protocol, but since the current 1.2 version only SOAP is used. SOAP 1.2 defines XML-based information that can be used for exchanging structured and typed information between peers in a decentralized and distributed environment. SOAP messages are usually sent over HTTP, but other protocols such as SMTP or TCP can be used as well. To secure the communication, HTTP basic authentication, HTTPS or SOAP signatures can be used. SOAP is used especially in Business-to-Business (B2B) and Enterprise Application Integration (EAI) communication. SOAP suits well for this kind of use as it is interoperable and based on standards.

5.1.1. Features of SOAP

One of the advantages of using SOAP is its interoperability. SOAP has been the easiest mechanism to integrate and interoperate between enterprises and different platforms (Siddiqui, 2002). Another advantage of SOAP is its extensibility that comes from XML's extensibility. Extensions can be used to provide for example security in the web service.

One of the main security problems of SOAP is that it goes through firewalls as HTTP. Thus, it is quite impossible to filter SOAP in firewalls. In addition, as SOAP typically runs over HTTP, it is vulnerable to same security issues as HTTP.

5.2. REST

REST (Representational State Transfer) is a design approach, in which web services are resources and each resource can be identified by a unique URI. For example, most of the Web services provided by Yahoo! are RESTful.

REST was first introduced in the doctoral dissertation of Roy Fielding. In REST, standard HTTP methods (GET, POST, PUT and DELETE) are used to communicate with the resource. GET method is used to get a representation of a resource. A GET request must not have any effects on the server. POST method is used to create or update a representation of a resource. PUT method updates a representation of a resource. DELETE method removes a representation of a resource from the server.

To implement a RESTful service no special tools are needed. Thus, the barrier for adoption is very low (Pautasso et al., 2008). For example, normal web browsers can be used for testing.

5.2.1. Features of RESTful Services

The advantages of REST include for example that it is easy to code and there is less code to maintain than for example in SOAP.

REST takes the advantage of caching of information on both the service side and the client side. By using conditional HTTP GET, a client can refresh a representation if the data has not changed. Truly RESTful services are stateless. This means that each interaction is independent of the others and each request contains enough information to complete an action. Thus, the server never stores any application state. REST resources themselves are allowed to have a state - and they often have. Caching and stateless approach improve for example the scalability of RESTful web services. For example, no additional messaging layer is needed when using REST.

REST services are sometimes categorized into two categories based on the level of “RESTfulness”. These categories are Hi-REST (high REST) and Lo-REST (low REST). In a Hi-REST API, all main HTTP standards methods are used to communicate with the resource. In Hi-REST services, XML documents are used as the message payload, metadata is put into HTTP headers and URIs are used meaningfully. In a Lo-REST API, in turn, only HTTP GET requests with no side effects are used.

5.2.2. REST Anti-patterns

Some of the services that claim to be RESTful fail to adapt the REST ideas. Some of the most common REST anti-patterns are:

- *Tunnelling through GET* means that parameters needed to perform some action are encoded into the URI. In this case the used HTTP method does not match with the action caused at the server.
- Example: *GET* <http://example.com/some-api?method=deleteCustomer&id=13>

- *Tunneling through POST* is similar to tunneling through POST, except for that the POST method is used. A typical anti-pattern is to use different payloads in messages to perform different actions on the resource.
- Example: *POST http://example.com/CustomMgmt*

```
<soap:Envelope>
<xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<deleteCustomer xmlns="http://example.com/ns1">
<customerId>13</customerId>
</ns:deleteCustomer>
</soap:Body>
</soap:Envelope>
```
- *Ignoring caching* is against one of the advantages of using REST. Usually this is because a default setting specified in the web framework.
- *Misusing cookies*. Even in RESTful applications, cookies can be used to store a resource or client state. However, they should not be used for storing information that the server can validate without the session state, for example authentication token. Also, cookies should not be used to encode information that can be transferred for example in the URI or in the standard headers.
- *Absence of links in representations and forgetting hypermedia*. Resource representations could be communicated as links.
- *Not using HTTP status codes* correctly. Even if HTTP provides a rich set of status codes, many applications use only 200 (OK) and 500 (Internal Server Error).
- *Ignoring MIME types*. Different formats can be used to represent a resource for different needs.

5.3. SOAP versus REST?

There are a lot of presentations available describing the consideration for making the right choice between SOAP and REST. Neither one of the protocols is the right choice in all cases. In the following, a couple of things and arguments to consider when making the choice are listed (Little, 2008):

- The requirements of the API methods and their arguments. If there are complex methods with multiple levels of arguments or nested argument structures in the API, SOAP may a good choice. Otherwise, REST may a good choice.
- The expected volume of traffic. SOAP has greater overhead than REST.
- Possible limitations in the environment or host.

REST may be appropriate choice if (Tyagi, 2008):

- Web services are completely stateless.
- Caching infrastructure can be leveraged.
- There is a mutual understanding between the service producer and consumer of the context and content being passed.
- If the bandwidth of the end devices is limited like in mobile phones.
- If the service is going to be integrated with existing web sites that use for example AJAX.

On the other hand, SOAP may be an appropriate choice if (Tyagi, 2008):

- If a formal contract (to be provided using a description language) is needed for the service.
- If the architecture must address complex, nonfunctional requirements. Such requirements are for example security.
- If the architecture must handle asynchronous processing and invocation.
- If describing resources of the service would get cumbersome using REST, it is probably better to use SOAP

At the moment (05/2009), REST seems to be much more popular protocol to develop web services. For example, in ProgrammableWeb⁵ 65 % of the APIs are REST. SOAP is the second popular protocol with 21 %. However, SOAP is still very popular in enterprise IT.

6. Description Languages

One important part of a Web service is the way how the provided interface is described for the client. There are some requirements how a description language should be like (Takase, 2008). First of all, there should be some level of automation to ease the development. In addition, there should be some kind of a link between the actual service and its description. If that is not the case, it is common that the two will get inconsistent very fast.

6.1. WSDL

WSDL (Web Service Definition Language) is an XML language for describing interfaces of a Web service standardized by W3C. WSDL is commonly used together with SOAP. It can also be used for describing Web services using any other protocol than SOAP. WSDL 1.1 is at the moment the mostly used technology to describe Web services. For example businesses can exchange WSDL files to understand the different services. WSDL can be used for describing service interfaces using any protocol. The latest version 2.0 defines a HTTP binding extension. Still, it is more interface-centric and provides no true resource-centric model.

⁵

<http://www.programmableweb.com/apis#topa>

By using WSDL tools it is possible to generate the client stub code automatically. Automatically generated stub hides the complexity of interacting with the remote end from the user.

6.2. WADL

The Web Application Description Language (WADL) is a description language that can be used for describing REST services (Hadley, 2006). Whereas WSDL can be used to describe services using any protocol, WADL is designed for describing HTTP services. Thus, it is much simpler than WSDL. WADL is a resource-centric alternative to WSDL. A WADL file describes a set of resources and the operations that can be performed on that resource. The advantage of WADL compared to WSDL is that it abstracts the details of HTTP requests and responses without adding any new abstraction on top. The downside is that WADL does not address features such as security.

The WADL is not mandatory when describing REST services, thus not many Web services today provide official WADL files. It is still not clear if WADL will be widely adopted. This far, WADL is not standardized even though standardizing it might make the adaptation faster.

As most REST services today provide only a human-readable, textual description, some people think that even WADL is kind of overkill.

7. Data Interchange Format

Another ongoing debate between two technologies is JSON versus XML. A data interchange format technology allows one language to pass data along to another language in a predetermined format. XML and JSON are two widely used data interchange formats.

7.1. XML

XML is a widely used data interchange format standardized by W3C. For example, SOAP-based web services use XML to format the data in the message payload. The biggest advantages of XML are that it is very flexible and extensible. If the data to be transferred is more like a document, or if the order of data matters, XML is the correct choice. On the other hand, as XML is very inefficient carrying a lot of baggage, XML is not very ideal for transferring light-weight data.

7.2. JSON (JavaScript Object Notation)

JSON is a text-based interchange format, which is commonly used in AJAX applications. Design goals of JSON were to be textual, minimal and a subset of JavaScript. JSON is data-oriented, which means that there is no need to do XML parsing in the browser. Thus, JSON suits well for AJAX web applications. JSON was standardized by the IETF (Internet Engineering Task Force) as RFC 4627 (Crockford, 2006).

As JSON is designed as the subset of JavaScript, there are also serious security issues related to executing JavaScript.

7.3. Comparing XML and JSON

The advantages to use JSON over XML are that it is light and data is typed. Data is also readily accessible as JSON objects from JavaScript. Thus, JSON fits better for AJAX Web services in which not very heavy data structures are transferred and the receiver should get the same data structure with minimal effort (Marinescu et al. 2006).

However, XML has the advantage of being widely interoperable. For transferring general-purpose data or if the order of the data matters, XML is probably the better option. Also, if the data is long-lived - that is if the data is stored somewhere in the system - XML is more likely to be the better option.

8. Discussion

This is the first version of EDEN Deliverable D4.3.4. It is focused on the infrastructure technologies of service ecosystems. Based on the principles of Flexible Services Architecture - modularity, transparency, loose coupling, composability, late binding, and reflection - we have identified the following areas for further study:

- Messaging - Whether to use direct messaging between services or message servers as intermediate components to facilitate the messaging between the components? Are there clear benefits of message queues or publish/subscribe interaction in service ecosystems?
- Message formats - How should messages be structured? What are the relative merits of SOAP and REST?
- Data interchange formats - What are the roles of XML and JSON as formats for message payloads?
- Service description languages - Should service interfaces be described using a specific description language instead of informal textual documentation. What are the merits of WSDL and WADL?

The second version of this deliverable is due in May 2010. In that version these questions will be analyzed with respect to the architectural principles of Flexible Services Architecture. In addition, it will document the experiences with different approaches used in pilot services and tools created in Flexible Services programme.

9. References

Coulouris F., et al. (2005). Distributed Systems: Concepts and Design, 4th Edition. Addison-Wesley.

Crockford D., Internet Engineering Task Force (2006). The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627.

Fielding R. (2000). Architectural Styles and the Design of Network-based Software Architectures.

Hadley M., Sun Microsystems (2006). Web Application Description Language (WADL), <https://wadl.dev.java.net/wadl20061109.pdf>.

JSON. JSON: The Fat-Free Alternative to XML <http://www.json.org/xml.html>

Little M. (2008). SOAP vs. REST API Implementation, http://www.fliquidstudios.com/2008/12/17/soap_vs_rest_api/.

Marinescu F. and Tilkov S. (2006). Debate: JSON vs. XML as a data interchange format, <http://www.infoq.com/news/2006/12/json-vs-xml-debate>. InfoQ.

Pautasso C. et al. (2008). RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision, <http://www.jopera.org/files/www2008-restws-pautasso-zimmermann-leymann.pdf>. 17th International World Wide Web Conference (WWW2008), Beijing, China.

Reindel B. (2007). Summarizing the JSON and XML data interchange format debate, <http://blog.reindel.com/2007/10/03/summarizing-the-json-and-xml-data-interchange-format-debate/>

Richardson L. and Ruby S. (2007). RESTful Web Services: Web services for the real world. O'Reilly.

Shin S. (2008). Introduction to JSON (JavaScript Object Notation), <http://www.javapassion.com/ajax/JSON.pdf>. Sun Microsystems.

Siddiqui B. (2002). Deploying Web services with WSDL, Part 2: Simple Object Access Protocol (SOAP), <http://www.ibm.com/developerworks/library/ws-intwsdl2/>

Takase T. et al. (2008). Definition Languages for RESTful Web Services: WADL vs. WSDL 2.0.

Tyagi S. (2006). RESTful Web Services, <http://java.sun.com/developer/technicalArticles/WebServices/restful/>. Sun Microsystems.