Diplomarbeit

Concept and Implementation of a Service Transformation Gateway

for the secure and accountable provision of enterprise services

Tobias Rothe

Dresden University of Technology
Faculty of Computer Science
Chair for Computer Networks

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Betreuender Mitarbeiter:	Dr.-Ing. Thomas Springer
Betreuender Amadeus Mitarbeiter:	Nathalie Evan

Professur Rechnernetze

amadeus
Your technology partner

One of the major advantages of service orientation is the possibility to combine services to form new applications. Emerging composition technologies focus on business to business service integration. They lack in supporting the specific needs of client integration.

In this document a generic middle-ware will be introduced which enables flexible service composition based on an abstract architecture specification and emphasizes on the incorporation of security and accounting facilities. The result is a common platform that fosters reuse and allows rapid software development in a secure and accountable manner.

Contents

1 Introduction **1**
1.1 General Context . . . 1
1.2 Problem Background . . . 1
1.3 Purpose . . . 2
1.4 Directions for the Reader . . . 2

2 Foundations **4**
2.1 Service Orientation . . . 4
2.1.1 Fundamental Principles . . . 4
2.1.2 Contemporary SOA . . . 6
2.2 The Web-Service Technology . . . 7
2.2.1 Introduction . . . 7
2.2.2 Core Standards . . . 7
2.2.3 Second Generation Web Service Specifications . . . 9
2.3 Summary . . . 11

3 Requirement Analysis **12**
3.1 Use-Case: Mobile Travel Wallet . . . 12
3.1.1 Functional Overview . . . 12
3.1.2 Information Retrieval . . . 14
3.1.3 Access Control . . . 15
3.2 Service Composition Requirements . . . 16
3.2.1 Composition Type . . . 16
3.2.2 Process Control Flow . . . 16
3.2.3 Interface Adaptation and Data-type Conversion . . . 17
3.3 Security and Accounting . . . 19
3.3.1 Authentication and User Session . . . 19
3.3.2 Service Authentication . . . 19
3.3.3 Accounting . . . 20
3.4 Context . . . 20
3.4.1 Stateful Service Access . . . 21
3.4.2 Adaptation of Composition Processes . . . 21
3.5 Non-Functional Requirements . . . 22
3.5.1 Tool Support . . . 22
3.5.2 Composition Engines . . . 23

3.5.3 Performance and Quality of Service . . . 23
3.6 Summary . . . 23

4 Related Work 24
4.1 Service Composition Techniques . . . 24
4.1.1 Web Service Composition Languages . . . 24
4.1.2 Web Service Business Process Execution Language . . . 25
4.1.3 The Web Ontology Language for Services (OWL-S) . . . 31
4.2 Selected WS-* Technologies . . . 37
4.2.1 WS-Addressing . . . 37
4.2.2 WS-Security . . . 40
4.3 Summary . . . 41

5 Concept 43
5.1 Introduction . . . 43
5.1.1 Directions of Approach . . . 43
5.1.2 Overview . . . 44
5.2 Components . . . 45
5.2.1 The Composition Engine . . . 45
5.2.2 Client Connector . . . 45
5.2.3 Message Router . . . 46
5.2.4 Service Connectors . . . 47
5.2.5 Communication Methods . . . 48
5.3 Credential Management . . . 49
5.3.1 Overview . . . 49
5.3.2 Client Credential Sources . . . 50
5.3.3 Service Credential Store . . . 51
5.4 Context Management . . . 51
5.4.1 Context Lifecycle . . . 52
5.4.2 Context Data Structure . . . 54
5.4.3 Composition Engine Integration . . . 54
5.5 Summary . . . 59

6 Implementation 60
6.1 Foundations . . . 60
6.1.1 Design Principles . . . 60
6.1.2 The Spring Framework . . . 63
6.1.3 Acegi Authentication Framework . . . 66
6.2 CSA Architecture . . . 69
6.2.1 Overview . . . 69
6.2.2 Client Connector . . . 69
6.2.3 Message Routing . . . 72
6.2.4 BPEL Engine . . . 74

6.2.5 Service Connectors 74
6.2.6 Authentication and Security 76
6.2.7 Session Context Manager 77
6.3 Use-Case Implementation 79
6.3.1 Overview 80
6.3.2 Server Side Implementation 80
6.3.3 Client Application 91
6.3.4 Tests 95
6.4 Summary 98

7 Conclusions and Outlook **100**
7.1 Overview 100
7.2 Requirement Assessment 100
7.2.1 Service Composition Requirements 100
7.2.2 Security and Accounting 101
7.2.3 Context Management 102
7.2.4 Non-Functional Requirements 102
7.3 Future Improvements 103
7.3.1 Service Composition 103
7.3.2 Security and Accounting 104
7.3.3 Context Management 104
7.3.4 Other Improvements 104
7.4 Summary 105

Bibliography **110**

1 Introduction

1.1 General Context

The Web Service framework builds up on the paradigm of service-orientation. The key goal is to overcome tightly coupled integration of software systems, vendor dependencies and isolation; to foster reuse and the development of dynamic trans-boundary applications. Services become fundamental building blocks of component based architectures.

Service aggregation makes software development less dependent on the nature of the individual service. Instead the developer can weave multiple services together, modelling composition and data-type conversion in dedicated languages. Present solutions target large-scale business to business interactions in rather homogeneous environments.

Client application development and integration into existing business infrastructures demands a solution which provides sophisticated mechanism to integrate authentication, authorization and accounting with service composition. The benefit of such a generic approach lies in the reduction of development costs and the improved reusability.

1.2 Problem Background

The motivation for this work is coming from actual practical requirements of the research department of Amadeus.

Amadeus was founded in 1987 by Air France, Lufthansa and Iberia with the core business of operating a Global Distribution System (GDS). It connects travel providers (about 775 airlines, over 70.000 hotel properties, 35.000 car rental locations, rail, tour, insurance, ferry and cruise operators) with almost 83.700 travel agencies and it is also used as an internal sales system by more than 145 airlines[AMA07].

During the last ten years, the market importance of Global Distribution Systems has been decreasing. In consequence Amadeus started to expand beyond its core business towards developing software and complete IT-solutions for travel and tourism industry.

In this context service orientation plays a key role because of the distributed, dynamic and heterogeneous nature of the actors. Web Services are allowing third parties to integrate the Amadeus services in their systems and to be integrated into the Amadeus system (access and reverse access).

In the research and innovation department future travel solutions are investigated. This requires a flexible platform for the integration of client and especially mobile applications with different services from internal and external providers. Currently no existing solution is satisfying the specific needs of such a platform.

1.3 Purpose

This document proposes a generic and open standard based approach to integrate client applications with business service compositions. It will act as intermediate gateway that exposes the required interface to the client acting as a single service provider who is mediating security and session context and provides basic accounting facilities. This Credential and Service Aggregator (CSA) is especially optimized for mobile and Web 2.0 platform integration.

The solution will address the following points:

- No single service delivers the information in the expected completeness
- The format of the information does not match the requirements
- Transparent mediation between service contexts and client context to realize cross composition transactions
- Centralized security and accounting facilities
- Focus on the specific requirements of mobile and Web applications

The key to achieve these goals are statefull conversations and lifecycle management beyond the scope of a single composition.

1.4 Directions for the Reader

The next chapter will give a general overview to the foundations of service-orientation and the Web Services technology. Furthermore second generation standards are introduced. These fundamentals are necessary but can be skipped by the experienced reader.

Chapter 3 will analyze a concrete use-case to derive general requirements which are used to develop the central concept and to validate the solution.

In the following chapter related work and existing technologies will be shown. They address some of the requirements and will be reused in the concept for a Credential and Service Aggregator which will be developed in Chapter 5.

Chapter 6 proposes a reference implementation and a prototype of the use-case to test evaluate it. Finally a results analysis and outlook is given in chapter 7.

2 Foundations

This chapter gives a brief overview of the principles of service-orientation and the fundamentals of the Web Service technology. They provide the foundations for the following chapters.

2.1 Service Orientation

Service orientation has become one of the driving forces of information technology as well as business process automation. Its actual impact is hard to quantify; some call it simply a multi purpose "buzzword" others see in it an upcoming world application computing platform. Remarkable is the fact that service-orientation is going far beyond the boarders of software-engineering, it is used in different contexts by different groups who each have own explanations what it stands for. This is why the actual meaning can hardly be broken down to a simple definition[Erl05].

It is important to be aware of the fact that the term Service Oriented Architecture (SOA) is often used outside architectural boundaries and has become a synonym for service-orientation in general.

2.1.1 Fundamental Principles

Independent from the granularity level or viewpoint, there is one communality wherever the term *service-oriented* is utilized. It is the concept of separation of concerns. It means that the processing logic to fulfill a large scale task is decomposed in smaller units specialized to treat certain aspects of the task. Service-orientation realizes this generic well proven approach in a unique and comprehensive way.

The principle entities that form a service-oriented architecture (SOA) are[Erl05]:

services Autonomous but not isolated units of logic. Commonality is ensured by contracts and standardization that does still allow these units to evolve independently. The granularity of logic can be different, services can also encapsulate other services (Figure 2.1).

service-descriptions In order to ensure a loose coupling of services and yet to enable relationships, services provide a description containing their name, location and interface definition. It has to be based on a standardized description language.

messages After being emitted by a service, a message becomes an autonomous unit of communication. It therefore needs to contain a limited ability of self-governance.

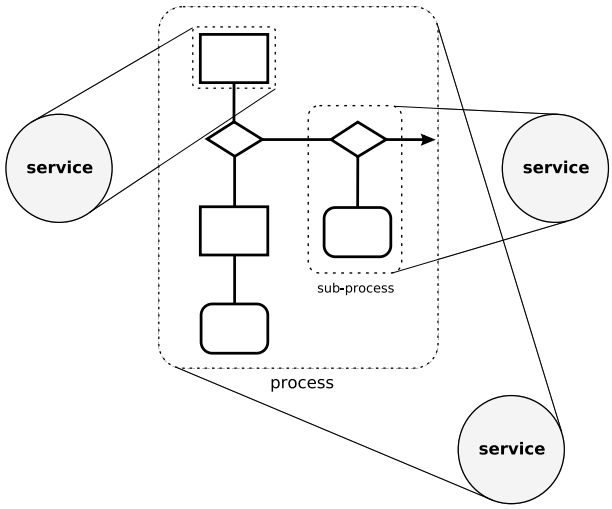


Figure 2.1: Services can encapsulate varying amounts of logic.[Erl05]

The following principals distinguish this architecture from former distributed architectures:

- **Loose Coupling** – the dependencies between services are to be minimized
- **Service Contract** – services comply to a common agreement
- **Autonomy** – services have control over the logic they encapsulate
- **Abstraction** – apart from contract and description the complexity of the encapsulated logic is hidden to the outside.
- **Reusability** – the breakdown of logic into service units is intended to enable reuse of these units
- **Composability** – services can be aggregated to larger units and recombined for different use cases.
- **Statelessness** – the dependency on state information is minimized
- **Discoverability** – the search and invocation of services is supported by an appropriate infrastructure

As service-orientation is still evolving this list should be considered as its founding principles and common baseline of all technologies that are labeled with SOA.

2.1.2 Contemporary SOA

The fundamental concepts of service-orientation are technology independent. In theory any suitable distributed computation environment could be used to realize a service oriented architecture. In practice SOA and Web Services have become bundled terms because the Web-Service technology is the most advanced and most suitable way to realize a service-oriented architecture. The availability of this technology has paved the way for the success of service-orientation and it has also led to an evolution of the founding principles.

Thomas Erl uses the term Contemporary SOA to clearly distinguish between the fundamentals and to acknowledge the important role of the Web-Service technology.

He defines Contemporary SOA as:

> *SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise[Erl05].*

Contemporary SOA is constituted by the following characteristics which are extensions and enrichments the fundamental principles discussed above:

Autonomy

Minimization of dependencies should not only be assured for basic entities but also larger aggregation up to complex solutions or systems.

Quality and Security

An important business requirement is to ensure a safety and reliability of a system. Furthermore performance criteria have to be met and transaction facilities be provided. Not only the principle SOA but also first generation Web-Services where missing concepts and specifications to address these aspects. Many of the Web-Service extensions have been provided to fill this gap.

Open Standards & Vendor Diversity

The use of open standards like WSDL, SOAP, XML and XML Schema is one of the reasons for the success of the Web Service technology[Erl04]. This allows technology neutral data exchange and supports loosely coupling.

Discovery

First generation Web Service technology is often to realize point-to-point communication. Architectures of this kind can only partly be considered as service-oriented. A contemporary SOA is based on dynamic discovery and invocation strategies.

Unity & Federation

A core feature of service-orientation is the ability to bridge different environments. In this sense it is possible to integrate existing solutions and to encapsulate legacy system.

Composability and Reusability

Open standards, autonomy, design and specification must aim to enable composability on different level of granularity. The inherent reuse of services and blocks of services must be ensured.

2.2 The Web-Service Technology

2.2.1 Introduction

The core of the Web-Service technology is formed by only a few simple elements. It consists of three building blocks:

- **Service Provider** is exposing a service and a corresponding description
- **Service Requester** is invoking a service after discovering it
- **Service Broker** is offering facilities to search and discover services

The inside of these blocks is hidden but their relationship is well specified by open standards which are all based on the Extensible Markup Language (XML).

The service contract is defined through the Web Service Description Language (WSDL). The Simple Object Access Protocol (SOAP) provides a platform independent message transfer layer. And the Universal Description Discovery and Integration (UDDI) Standard specifies a Web Service registries publish and inquiry.

Figure 2.2 gives an overview of the Web Service entities, their relations and the use of the according standards. Before moving to extended Web Services a brief overview of the core specifications shall be given. Detailed information are given on the official web sites furthermore[Erl04] gives very practical description of how to integrate them.

2.2.2 Core Standards

2.2.2.1 Service Description: The Web Service Description Language

The Web Service Description Language (WSDL) is a W3C XML specification to describe all aspects necessary to interface with a Web Service[CMRW07]. It is a commonly accepted standard that has been integrated in many tools and frameworks.

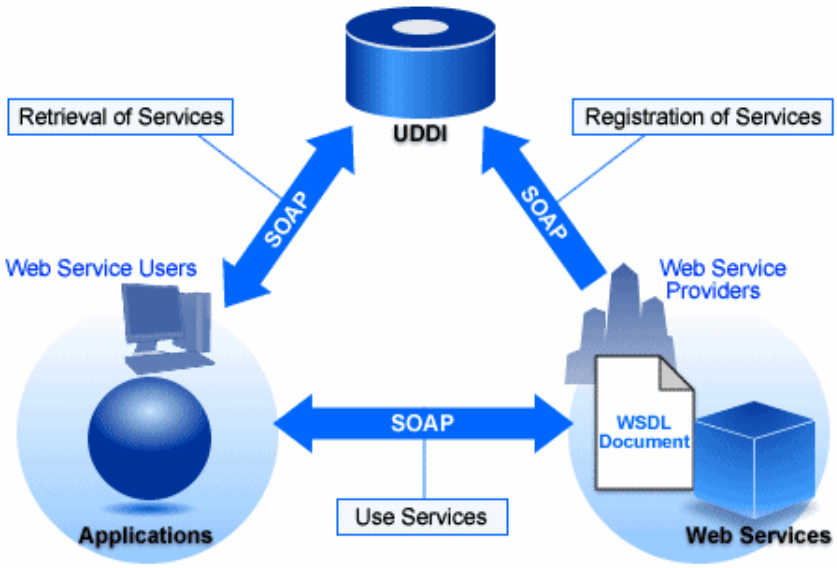


Figure 2.2: first generation Web Service components[fujitsu]

A Web Service provider must convoy the WSDL document to the consumer (Ideally through a registry). This document roughly consists of two sections:

Abstract service interface definition

The atomic construct of a service is a message. Messages are exchanged between service consumers and service producers. An operation represents an action or function; it associates message parts according to their direction as incoming or outgoing. Finally logically related operations are grouped in interface elements.

Service implementation definition

This part can be understood as a implementation centric enrichment of the logical interface definition. So it provides the "glue" to the messaging protocol (mostly SOAP over HTTP) and it also specifies the endpoint(s) where the Web Service can be reached.

This division fosters platform independence of the service interface definition. Furthermore, different implementation specific definitions can be build upon it.

2.2.2.2 Messaging: SOAP

SOAP was originally an Acronym for Simple Object Access Protocol and does specify a flexible and platform independent messaging mechanism[GHM+07]. It covers message exchange, structure and encoding.

The message exchange can synchronous as well as asynchronous. The transport protocol (so called binding) is not specified in the standard, it can be chosen by the developer. Most common is HTTP binding which also enables bypassing cooperate firewalls.

A SOAP message is represented by a XML document which is called "envelope" it contains a header and a body section.

Header is an optional section that can contain additional metadata about the message contents or it can be used for extended functionality. One example could be special processing instructions or context information like a session identifier. Second generation Web Service specification strongly depend on the SOAP header.

Body contains the payload application data. The content or structure can be freely chosen by the application (but is defined in the WSDL file).

It is principally possible to convoy binary data within the body section. Due to the overhead of encoding it is better to use SOAP attachments instead.

2.2.2.3 Universal Description Discovery and Integration (UDDI)

Besides interface description and messaging service discovery a broker infrastructure is the third essential part of a service-oriented architecture. The Universal Description Discovery and Integration (UDDI) specification[CHvRR04] defines such an infrastructure. It provides two functionalities:

Registration - providers can publish their service

Discovery - enable the inquiry and invocation of services

A public UDDI registry can act as open business service marketplace while internal registries enable companies centralize the management of internal and third party services.

Interaction with the registry is done using SOAP and service descriptions are exchanged in the WSDL format.

2.2.3 Second Generation Web Service Specifications

The first set of Web Service specifications where not covering a wide range of core business concerns. This must not be seen as an insufficiency but is explained by the modular

and extensible nature of Web Services.

A set of extensions evolved which is commonly referred to as second generation Web Service- or simply WS-* specifications. Their range of features is wide and overlapping and concurrencies occur.

From a high level point of view, these specifications can be categorized by the problems they address[Erl04]:

Context Management and Transactions

As Web Services where missing facilities to maintain a common context throughout the service activity, it was not possible to realize certain essential functionalities like transactions. This gap was closed by WS-Coordination, WS-Transaction and other specifications.

Security

Available mechanisms like SSL are transport layer specific and do not allow advanced security concepts and different message exchange pattern. To provide SOAP level confidentiality, integrity and authentication the WS-Security framework and related specifications where developed (WS-Trust, WS-Authentication etc.). It allows the utilization of established XML security standards.

Reliable Messaging and Policies

WS-ReliableMessaging provides instruments to ensure failsafe message delivery or at least to react to delivery problems. WS-Policy allows defining high-level business and security rules. Both specifications help ensure and enforce quality of service criteria.

Attachments

SOAP messages are not convenient for all types of content. Especially the encoding of binary data leads to an unwanted overhead. This problem was known from the Internet mail protocol where the solution of binary attachments was developed. In the WS-Attachment specification this idea was adopted by defining to use of Direct Internet Message Encapsulation (DIME) within SOAP.

Service Combination and Workflow

The first possibility of combining Web Services is so called port-type inheritance that allows grouping of multiple services (WSDL 1.2). It is very limited, does not provide control flow mechanism or ways to incorporate incompatible interfaces and data semantics.

In consequence workflow and composition standards like WS-Orchestration, the Business Process Execution Language (BPEL) or OWL-S have evolved. They provide high level composition and process control facilities for Web Services. Details about this topic will be given in the next section.

2.3 Summary

The beauty of service-orientation is certainly the unique way to overcome long existing problems of software engineering by providing an approach which seems rather simple. In practice this often leads to misinterpretation that it is as easy to achieve.

A Gartner study states:

> *Through 2008, 70 percent of IT organizations will fail to successfully select and implement an SOA strategy on the first try (0.7 probability)* [Daryl C. Plummer, Gartner Research].

So it is important to point out, that a SOA does not come "out of the box" it can not be implemented over night and no matter how good a tool is, it can not just transform a system to become service-oriented[Car07].

The design of such a system needs skills and a fundamental understanding for the paradigm. Technologies have to be chosen carefully, because the number of specifications and their complexity is growing which inheres the risk to restrict interoperability.

Finally service-orientation is not the best solution for any problem.

3 Requirement Analysis

Integrating an application in a service oriented environment mostly means combining multiple services from possibly different providers. This task can be complex and should be performed by a dedicated and generic middle-ware component.

The benefits of such an approach are:

- simplified client development
- improved reusability
- Separation of concerns
- Establishment of security boundaries
- Centralization of control

There are existing service composition solutions but they focus on business to business integration and lack in support for the specific needs of client applications especially when targeting mobile platforms. Starting from a motivating example this chapter defines the necessary characteristics of such a generic solution.

3.1 Use-Case: Mobile Travel Wallet

The Mobile Travel Wallet (MTW) is an application to be executed on a mobile device so it can be taken along by the traveller to get personalized trip information. In this section the scenario will be described as well as the requirements in terms of service aggregation and authentication.

3.1.1 Functional Overview

A fictive story:

Alice is preparing her business trip to visit a customer in Moscow. She books the flight, car rental and hotel on the Internet. After finalizing the payment she gets informed about the possibility to download a mobile and personalized travel information software which is called Mobile Travel Wallet.

On the next day Alice goes to the Charles de Gaulle airport. On the way she starts the application to get information about her trip.

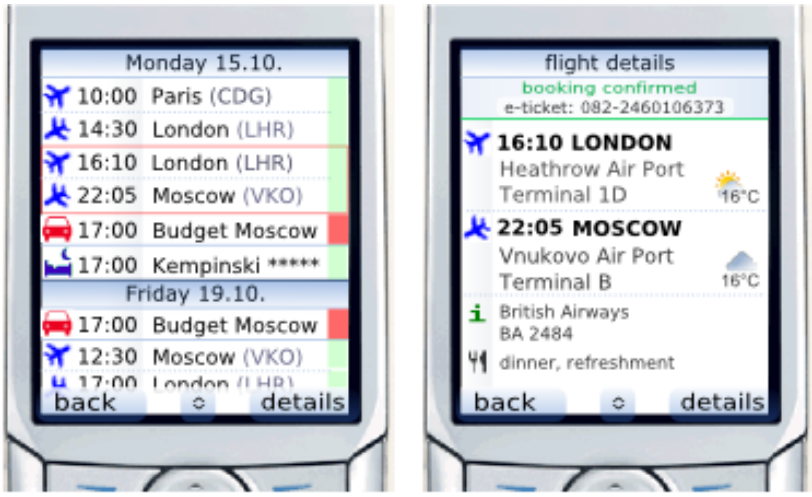


Figure 3.1: Travel Wallet - Trip Overview and Itinerary Detail

From the main menu she navigates to a scrollable overview of her journey (figure 3.1 left).

For each item, details can be displayed. This way Alice can see terminal and flight numbers as well as other information like the weather at the time of arrival (figure 3.1 right).

Figure 3.2: Travel Wallet - Destination Information

The flights are all confirmed and on time, so she is happy. Then she discovers that the car rental is marked red (figure 3.2 left). From the detail view she can easily place a

phone call to the hot-line. The problem can be resolved quickly because Alice is able to provide the reference number directly from the mobile application.

After the check-in, the phone rings and displays an alert message. Due to a terror warning the connecting flight will be delayed one hour.

Arrived in Moscow, Alice goes to the car rental office. She authenticates by simply swapping the phone at the reception desk and receives the keys.

During her stay in Moscow the travel wallet is able to provide Alice with additional destination information like a tube map, a restaurant guide on an event schedule (Figure 3.2 right).

3.1.2 Information Retrieval

The use-case includes not only the client interface but also the provisioning of the appropriate information. To achieve this a composition of different services has to be performed. The mobile device has limited resources which makes it desirable to do the service aggregation and data preprocessing on server side and provide a custom service to the client.

For the use-case a combined combined information service could be defined like the one shown in figure 3.3:

> In the first step a Web Service is invoked(S_1) that provides the booking record based on the reservation number and name which were provided by the user. The following call (S_2) retrieves information containing flight, hotel and other itinerary details. This record does not contain up to date information about the flights (delay, cancel etc.) so a schedule has to be requested for each booked segment (S_3). In parallel additional services are invoked to provide short weather and destination information(S_4, S_5). Finally all information are assembled and conveyed back to the client.

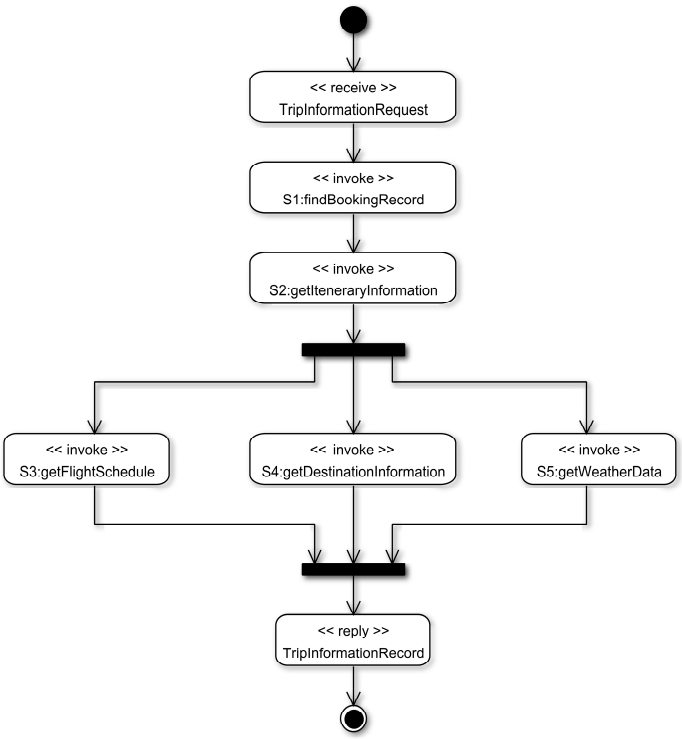


Figure 3.3: Composite Trip Information Service Example

3.1.3 Access Control

The access to the Mobile Travel Wallet should be controlled by an appropriate authorization mechanism. There should be possibilities to make content specific to user groups through a role-based access control mechanism. As an example there could be three different types of users (see figure 3.4):

Anonymous guests will only get very basic information about their booked trip. They are encouraged to register to receive more details like for instance weather information or destination guides. In addition the fact of being registered makes it also possible to adapt the composition to certain properties which are stored in the user profile (e.g. the language). Finally premium users will be granted access to an extended service offer (e.g. to search and book hotels).

To provide authorization the user must be identified through an authentication process.

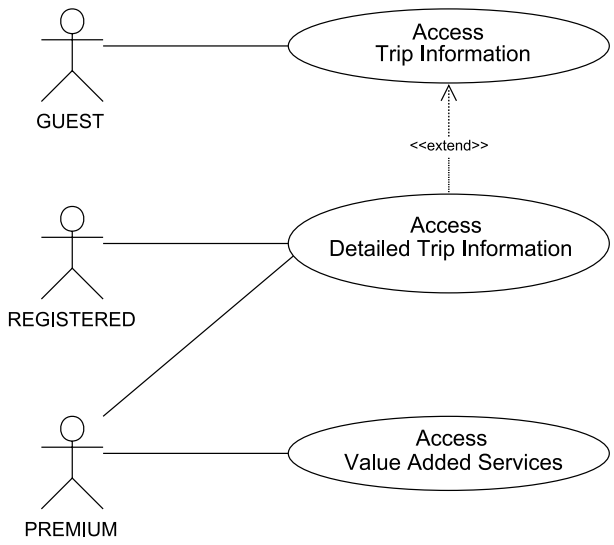


Figure 3.4: Use-Case: User Groups

3.2 Service Composition Requirements

The composition that was shown in the use-case 3.3 requires complex information that need to be retrieved from different sources, aggregated and processed. This predefined task should not be done on client side but rather by an intermediate Web Service composition component which exposes the composite as a single service to the client.

3.2.1 Composition Type

The central goal of the composition is to combine the functionality provided by multiple services from possibly different providers to solve a task which can not be satisfied by a single service. For the client itself, this composition should be transparent.

The requirements can be restricted to support only preemptive composition. That means that all services are known at design time and are statically defined. The description should be done in a standard language which serves as configuration for the composition engine.

3.2.2 Process Control Flow

Because of the directed data and control flow, Web Service compositions are processes[HB03]. Figure 3.3 shows such a composition process and two possible control flows (parallel

and sequential). Also some services have to be called multiple times (e.g. the flight schedule for each segment of the journey).

Formally the requirements for process control can be defined as followed:

> The most basic component is an atomic service X which is not decomposable. A Service S can be atomic or composite. Services can be executed in a sequence $S_1 \odot S_2$ or in parallel $S_1 \| S_2$. It can also be necessary to execute a service a certain number of times πS. Furthermore, upon a condition there can be a choice to execute one of two services $S_1 \otimes S_2$. Finally the execution of a service might need to wait for the completion of all previous services $(S_1|S_2) \rightsquigarrow S_3$
>
> In summary, the process algebra of the CSA can be defined by the following BNF-like notation:
>
> $$S ::= X \mid S_1 \odot S_2 \mid S_1 \otimes S_2 \mid S_1 \| S_2 \mid \pi S \mid (S_1|S_2) \rightsquigarrow S_3$$

3.2.3 Interface Adaptation and Data-type Conversion

Every service S has a set of input and a set of output parameters $S^{in} = \{I_1, I_2, ..I_n\}$ and $S^{out} = \{O_1, O_2, ..O_n\}$. If two Web Service operations are connected, the output parameters of the first S_1^{out} must match the input parameters of the subsequent S_2^{in}one. In many cases this can not be guaranteed, so there must be functionality to provide an interface mapping.

Three mapping aspects can be distinguished, Parameter Assignment, Data-type Conversion and Interface Transformation:

Parameter Assignment

Assignment of parameters means that it can be specified which output parameter of a service S_1^{out}should be taken as input parameter of the subsequent service S_2^{in}(figure 3.5 left). Such assignments are essential if the input of a service is composed from the result of multiple precedent services.

Data-type Conversion

Especially in business contexts and compositions with multiple providers, there can be a problem of mismatching data semantics. As a simple example it could assumed that a service X returns the geographic coordinates for a given airport code. It provides as

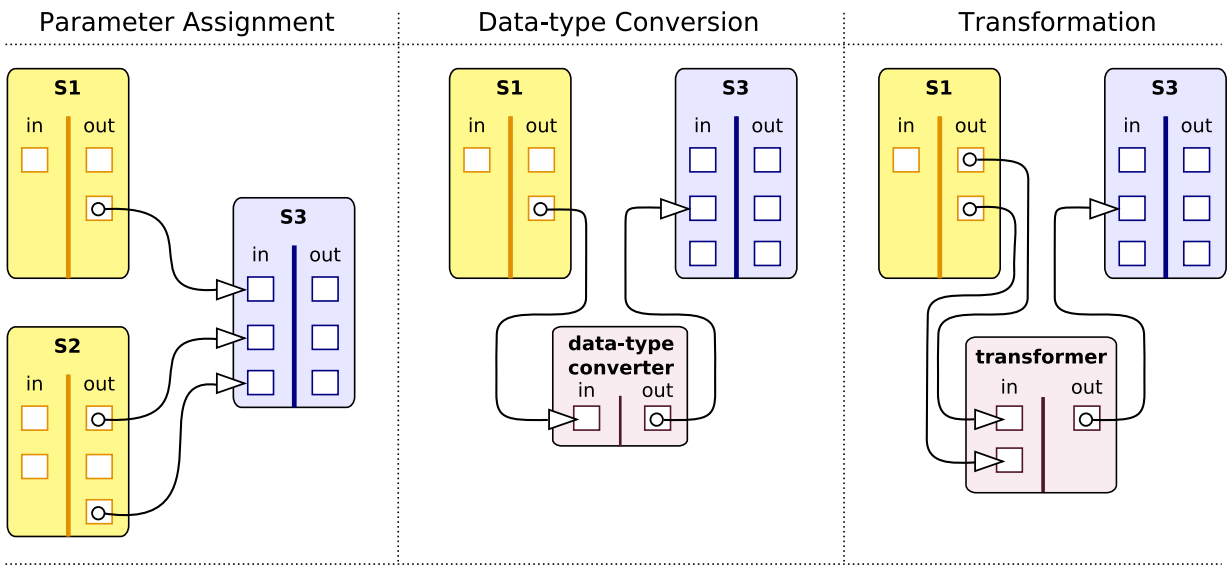


Figure 3.5: Categories of Interface Adaptation (schematic samples)

output the latitude and longitude $X^{out} = \{lat, lon\}$. Now the following weather service Y expects as input parameter geo-coordinates $Y^{in} = \{lat, lon\}$.

At first glance there is no issue to connect these two service interfaces. But if the two services are operating on basis of disjoint coordinate systems the direct mapping would lead to wrong results.

Other common examples are date formats or currency values. In such cases data-type conversion can be used to successfully connect the interfaces (figure 3.5 middle).

Interface Transformation

The task of matching complex business interfaces can easily exceed the possibilities of parameter assignments and data-type conversions. In such cases interfaces need to be transformed based on a transformation definition that is part of the composition description, an intermediate transformer can then fulfill the task of connecting the mismatching interfaces (figure 3.5 right).

The Credential and Service Aggregator must provide parameter assignment, data-type conversion and interface adaption capabilities.

3.3 Security and Accounting

The CSA clearly aims business use cases with a specific need for security and authentication. The use-case shown in 3.1 can be used to illustrate the authentication and authorization requirements.

3.3.1 Authentication and User Session

The access to the CSA has to be controlled by appropriate authentication mechanism. This authentication is required to initialize the stateful client connection and the user context which will play a key role in the CSA (see 3.4.1).

Furthermore unauthorized access has to be prevented and data security and integrity needs to be guaranteed. Ideally a Web Service standard should be implemented (WS-Security, WS-Authentication, WS-Session) with the constraint of possibly limited capabilities of client devices (encryption etc.).

3.3.2 Service Authentication

If a composition consumes other services it is usually faced with a prior authentication. Just recently second generation WS-* standards are filling the gap of missing security and authentication mechanisms in the basic Web Service standards. But in the meantime a heterogeneous often proprietary landscape has established. They have to be supported by the CSA to enable a successfully integration of such services.

Three of the most common approaches are:

HTTP-Authentication

The HTTP/1.1 standard defines basic authentication which is widely used. User-name and password are transferred in clear text so that additional transport layer encryption is needed.

This method was adopted by many Web Service providers. But neither SOAP nor WSDL are bound to a concrete transport layer so they lack in a native support for this type of authentication.

Sign-In and Session

It is quite common to combine a structure authentication message with a stateful conversation. So a sign-in request is answered with a reply that contains a token which needs to be added to following functional calls. Finally a sign-out message terminates the session.

Mostly all parts of this process are proprietary, the structure of the message as well as the token in the header (sometimes even transport layer session handling is used). To consume such services a custom implementation has to be done.

Certificate Exchange

In long-term business to business relationships it is no unusual to exchange certificates (X.509) or to establish trusted networks to overcome authentication overheads.

It is important to be able to map the state of services to the client session. For this reason the concept of a composition context is necessary. Such a context must be based on a secure and stateful connection to the client application. These aspects are described in the next sections.

3.3.3 Accounting

An economic requirement is the ability to track the use of services down to the granularity of individual users. This can enable later access based accounting. To enable such features, the service architecture must include a infrastructure to record the services accessed by users. This can be done on different level:

- **Client endpoint access** log recording each call made by a client including a unique client identification
- **Service access** log recording the usage of services that are invoked to complete a composition task

Recording service access is especially necessary to validate claims from third party providers.

3.4 Context

One of the principles of service-orientation is the loose coupling between the services which is why the concept of a session or a context was not included in the basic stan-

dards. To a certain extend it is possible to pass all information from one service call to another without having to tightly couple them by a context. Complex business processes depend on contextual information. This can be for simple access control or to hide information and complexity from the client.

A common context is the central element of the CSA because it enables personalization and the integration of authorization and accounting aspects into the composition process. Concretely it will fulfill the following tasks:

3.4.1 Stateful Service Access

Especially in transactional environments or where services need to share information, it is common to use a session context (e.g. flight booking, e-commerce etc.).

To visualize the requirement of such a stateful service access, an example can be taken from the *TripInformationService* (see figure 3.6):

> There are two services: S_1 to get the booking record (called PNR - Passenger Name Record) and the subsequent service S_2 which retrieves itinerary information. They share a common session which needs to be established by a prior authentication request S_a. The authentication process does also initialize the session. Without this session the services can not operate because they exchange information indirectly through the session context.

A conversation as shown in the example is user specific. A service aggregation middleware must be able to store and to associate it with a client user context so that a follow-up call can reuse a session that was established before.

3.4.2 Adaptation of Composition Processes

In the use-case examples of different user roles where given (3.1.3). For each group the service composition can contain different facets. For instance the guest user process excludes the invocation of a weather service.

There are two ways to implement such facets of a service composition:

- A separate composition for each user group
- Adaptation of a common composition depending on the user's context

The first case is trivial but can lead to a significant overhead especially if only slight adaptations are necessary. It also implies a replication of common parts which complicates updates and is a potential source of errors.

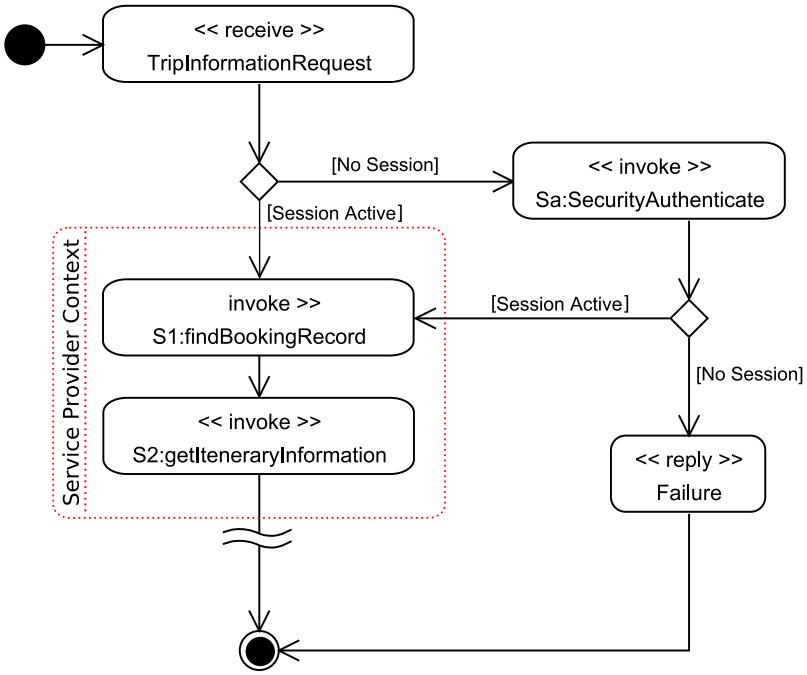


Figure 3.6: Example: Schematic Flow of a Service Provider Context

The second option of adapting a common composition requires an awareness of the user context. That means that within a composition decisions can be made by taking individual user parameters in account. For the example given above it could be defined that a weather service is only invoked if the current user belongs to the group of premium users.

In future extensions it can be imagined to build even more context sensitive composition processes where parameters like device capabilities, location or individual preferences are taken in account.

3.5 Non-Functional Requirements

3.5.1 Tool Support

Even non technical staff should be able to define service composition processes. This is only possible if a visual tool supports the chosen workflow language to hide the complexity and add features like debugging and verification. Such a tool should be extensible to enable CSA specific definitions.

3.5.2 Composition Engines

Composition engines are run-time environments to execute composite process definitions. Such a composition engine must be integrated into the CSA with the following features:

- Extensible, ideally open source
- based on open standards
- Stable and performant in a production environment

There is a mutual dependency between a composition language and the engine which has to be taken in account when a language is chosen.

3.5.3 Performance and Quality of Service

The solution will have to meet criteria which allow the production use:

- Continuous availability including the possibility to change compositions and configurations on a running system
- Parallel access of at least ten clients without performance loss
- Robustness in case of failure (e.g. partial execution)

In addition the concept should include points of future extensions especially to support load balancing and dynamic switch to a backup system in case of failure.

3.6 Summary

The use-case of a mobile application for travellers was given as motivating example. It showed that client software development can require specific information which need to be provided by a service composition. This composition should be performed by an intermediary component which incorporates client authorization and service authentication mechanism. To provide a maximum transparency and advanced features like personalization a stateful context is needed.

A solution that provides these features can be reused for a wider range of application and accelerate their implementation. To make this possible it has to be ensured that the approach is generic and customizable. It should make use of existing standards and tools to facilitate the adaptation.

Finally to allow a production use, performance- and quality of service criteria have to be taken in account.

4 Related Work

The Credential and Service Aggregator shall be based on existing technologies and standards which will be introduced in this chapter. The first part focuses on service composition techniques and the selection of the most suitable language. In the following assorted WS-* standards for security and addressing are discussed. They will play a major role to ensure that the connection with the server will not be restriced by aplying a proprietary solution.

4.1 Service Composition Techniques

The basic layers of Web Service technology were shown in the introduction. They provide a simple but powerful model to enable inter-service communication in a standardized, platform-independent way. The challenge lies in implementing larger scaled business tasks by combining multiple services while upholding the principles of service orientation. In the past such integrations were mostly realized manually using general purpose programming languages. The increasing demand of productivity and integration of composite services requires new standards for the definition and coordination of complex service aggregations[QP06].

This section will introduce formal modeling techniques for composite Web Service architectures and it will provide an overview of technical approaches to implement them.

4.1.1 Web Service Composition Languages

Formally a composition layer makes Web Services become part of the family of Architecture Systems. These systems go one step further than classical component systems (e.g. Corba or EJB)[Ass03]. They clearly distinguish between:

- Programming in the small: components encapsulate service knowledge
- Programming in the large: architectural level (orchestration, aggregation, compositions)

This is realized by the connector and ports technique and relies on a separate composition language also called the Architecture Description Language (ADL).

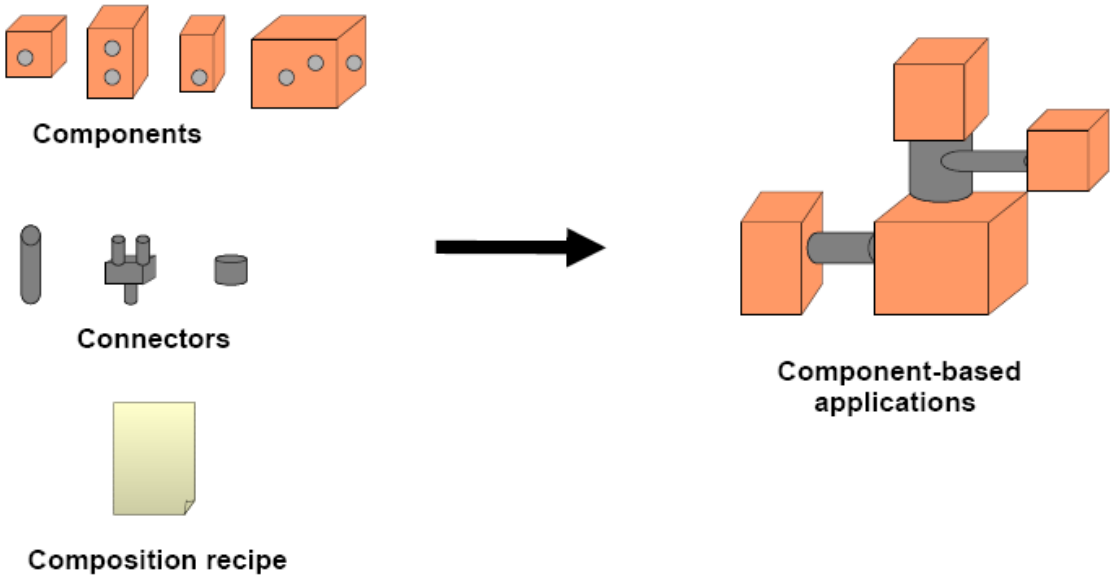


Figure 4.1: Black-Box Composition [Ass03]

The key advantages are:

- Separation of Architecture from Application specific functionality
- Mismatching interfaces are bridged by automatically generated code (glue-code)
- Connectors and communications can be varied without changing the component
- Two dimensional reuse of components and architecture.

For Web Services composition systems there exist several specifications. They differ in their roots, target group and level of industry acceptance. In the following the two most important will be introduced.

4.1.2 Web Service Business Process Execution Language

The Web Service Business Process Execution Language (WS-BPEL)[1][AAB+07, **?**] is a high level workflow language that enables the specification of business processes. It is a product of a joint development from Microsoft and IBM and was standardized by the Organization for the Advancement of Structured Information Standards[2] (OASIS). This fact and the participation of other major vendors like BEA and SAP illustrates the general importance of Web Service composition for business use.

[1]formally known as BPEL4WS or simply BPEL
[2]OASIS: `http://www.oasis-open.org`

WS-BPEL provides the facilities for a high level and implementation neutral description of business processes which can be synchronous or asynchronous. To handle the complexity and foster reuse there is the concept of *Abstract Processes* which provide core grain descriptions for usually more than one concrete use-case. They can then be implemented by *Executable Processes* which are fully detailed and can be processed by a run-time environment (BPEL Engine).

In the scope of this document only executable synchronous processes will be described but it is important to mention that the standard provides more flexibility which can be useful for future extension.

4.1.2.1 Basic Concepts

WS-BPEL is designed to fit into the stack of the existing Web Service standards. It is directly based on WSDL which provides the connectivity to the process itself and for all the services invoked in the composition. These connections are defined as `PartnerLink` which are bound to a WSDL port by a `partnerLinkType` (see figure 4.2).

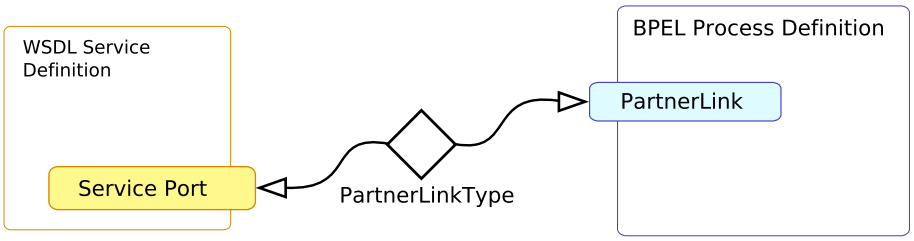


Figure 4.2: BPEL Partner Link with Web Services

Since the entry point of the BPEL process itself is described by a WSDL definition it can be accessed like any other Web Service.

4.1.2.2 Service Compositions

A Service Composition is defined as `process` which consists of *activities*. There are basic activities like:

- `receive` - wait for a client call
- `invoke` - call to a participating Web Service
- `reply` - send response to client
- `wait` - interrupt the execution until an event occurs

The control and data flow between basic activities is defined by structured activities which can be nested to form more complex aggregations. Example elements are:

- `sequence` - for executing a row of steps ($S_1 \odot S_2$)
- `flow` - to run activities in parallel ($S_1 \| S_2$)
- `while` - to define loops (πS)
- `switch`, `pick` - to choose between alternatives ($S_1 \otimes S_2$)

With these elements the CSA process algebra can be represented (shown in brackets).

Based on the `TripInformationProcess` the following example will illustrate the structure and the task of the sections of a WS-BPEL definition[3]:

The process definition starts with a header-like section. The root tag with the process name is followed by WSDL file `import` statements:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <process name="TripInformationProcess">
3   <import namespace="http://xml.amadeus.com"
4           location="AmadeusWS.wsdl"
5           importType="http://schemas.xmlsoap.org/wsdl/"/>
6   <import ... />
```

Next a partner link section defines references to the external Web Service ports (as shown in figure 4.2). `AmadeusWebServices` is a concrete example:

```
1   <partnerLinks>
2      <partnerLink name="AmadeusWebServices"
3               partnerLinkType="ns2:AmadeusWebServicesPTLinkType"
4               partnerRole="AmadeusWebServicesPTRole"/>
5     ...
6   </partnerLinks>
```

After that all variables that will be used throughout the process are defined. As example the variables for the incoming request and the outgoing response message are shown below:

```
1   <variables>
2      <ns:variable messageType="ns1:TripInformationRequest"
3                   name="TripInformationRequest"/>
4      <ns:variable messageType="ns1:TripInformationReply"
5                   name="TripInformationReply"/>
6      ...
7   </variables>
```

[3]simplified example with regards to readability

Now the actual process definition starts with a structured activity (`sequence`). The first child is a `receive` activity which is connected to the partner link that represents the BPEL process interface. As soon as there is a incoming call, the process will be initiated:

```
1    <sequence>
2       <ns:receive createInstance="yes" name="ReceiveRequest"
3                   operation="GetTripInformation"
4                   partnerLink="TripInformationServicePartnerLink"
5                   portType="ns1:TripInformationServicePortType"
6                   variable="TripInformationRequest"/>
```

In the next step an `assign` statement copies the booking number from the received request to the input of following service call.

```
1     <ns:assign name="AssignPNRRequest">
2       <ns:copy>
3        <ns:from>$TripInformationRequest/recordLocator</ns:from>
4        <ns:to>$PNR_RetrieveIn/…/reservation/controlNumber</ns:to>
5       </ns:copy>
6     </ns:assign>
```

With the booking number it is then possible to invoke the Amadeus Web Service and retrieve the complete booking information record:

```
1      <ns:invoke inputVariable="PNR_RetrieveIn"
2             name="InvokePNR_Retrieve"
3             operation="PNR_Retrieve"
4             outputVariable="PNR_RetrieveOut"
5             partnerLink="AmadeusWSLink"
6             portType="ns3:AmadeusWebServicesPT"/>
```

After all other steps are processed (not shown) the result is send back by the reply statement which terminates the process definition:

```
1        <reply name="ReceiveTripInformationReply"
2               partnerLink="TripInformationLink"
3               variable="TripInformationRecord" .. />
4    </sequence>
5  </process>
```

WS-BPEL contains other features that are not explained at this point. Detailed descriptions can be found in the specification[AAB+07].

4.1.2.3 Data-type Conversion

The WS-BPEL specification is based on the eXtensible Markup Language (XML). It allows utilizing the XML standard specifications like the XML Path Language (XPath 1.0[4]) and XSL Transformations (XSLT 1.0[5]) to perform data mapping and data type manipulation.

XPath can be used whenever variables are defined or referenced. Generally this is in the `assign` element. The lines below show the use of such an expression within the `from` element of a `copy` assignment:

```
1 <assign>
2     <copy>
3         <from xmlns:b="urn.xyz">$FooVar/b:bar/text()</from>
4         <to variable="B"/>
5     </copy>
6 </assign>
```

Besides node-tree accessibility, XPath also provides an important number of functions to manipulate data or aggregate new information. Details can be taken from the XPath reference. Some examples are:

- String functions: `concat(str1,str2,[..]),contains(str,str),string-length(str)`
- Number functions: `sum(node-set), round(number)`
- Boolean functions: `boolean(object), not(boolean), true()`
- Node-Set functions: `last(), count(node-set), position()`

More complex data type transformations can be delegated to an external XSLT-Stylesheet by using the `doXslTransform` method:

```
1 <assign>
2  <copy>
3    <from>bpel:doXslTransform("urn:stylesheets:AB.xsl", $A)</from>
4    <to variable="B"/>
5  </copy>
6 </assign>
```

In summary it can be stated that WS-BPEL offers the full data manipulation power of XPath and XSLT. This is a big advantage especially for developers who are familiar with these standards.

[4]XML Path Language: `http://www.w3.org/TR/xpath`
[5]XSL Transformation: `http://www.w3.org/TR/xslt`

4.1.2.4 Tool Support and BPEL engines

The fact that there are at least 30 software products[6] for designing and executing BPEL shows the wide acceptance of this language. There are commercial as well as open source solutions.

The tool support is crucial because workflow definitions become easily complex. Besides visualization features like validation and debuging facilitate the creation of WS-BPEL workflows. An example of such an IDE is given in figure 4.3 which shows a screenshot of the Active Endpoints[7] BPEL Designer.

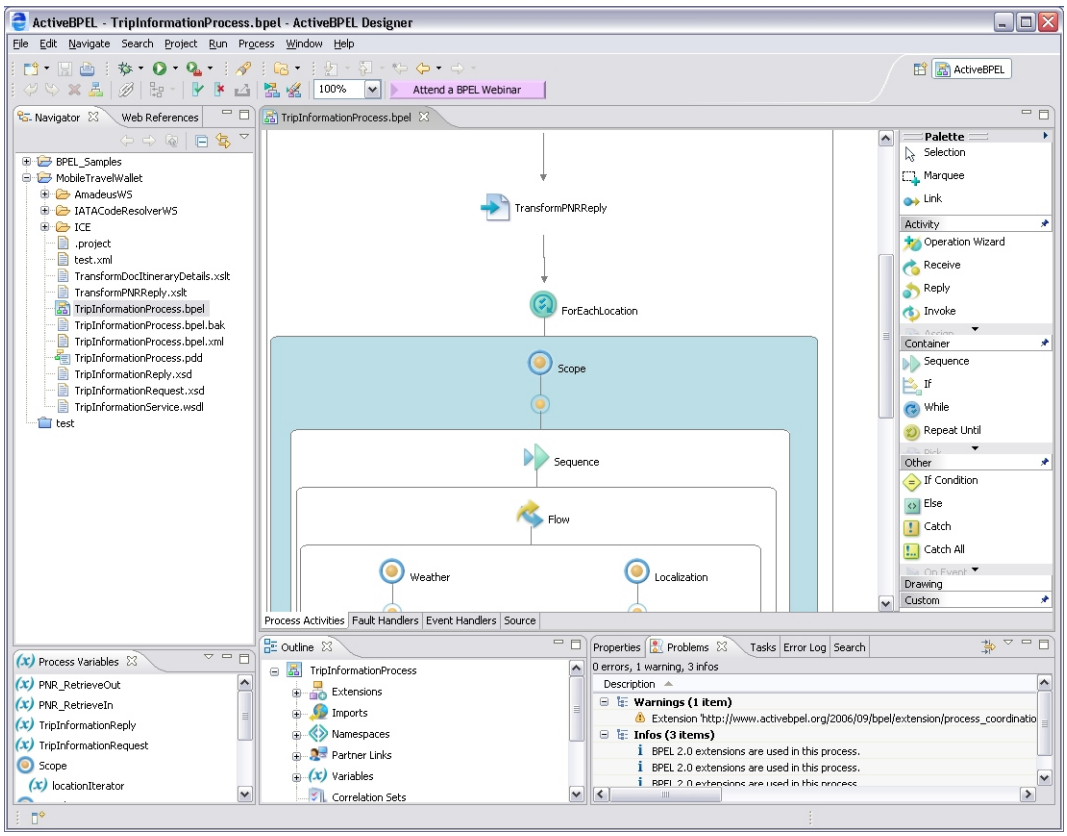


Figure 4.3: Screenshot of a WS-BPEL development tool

The variety of tools and engines to choose from make WS-BPEL very attractive. The interoperability is slightly limited through custom extensions and the fact that there is no uniform graphical notation.

[6]List of BPEL engines: `http://www.answers.com/topic/list-of-bpel-engines`
[7]Active Endpoints: `http://www.active-endpoints.com/`

4.1.3 The Web Ontology Language for Services (OWL-S)

The Semantic Web[8] is an initiative of the World Wide Web Consortium (W3C) that aims to extend the existing World Wide Web with structured semantic information[Her07]. This machine readable metadata enables software agents to overcome key-word based analysis and gain actual knowledge about the contents and their correlation. In consequence it will be possible to provide a more precise and complete access with more powerful tools.

To enable this vision, a new generation of web markup languages is needed. As such, the Web Ontology Language (OWL) allows software-interpretable knowledge representations[BvHH+03]. In addition to content description and classification it allows defining logical coherence which can be processed by reasoning application.

The importance of services within the Internet is constantly increasing. A semantic description is needed to enable dynamic discovery, invocation and composition in an automated way. OWL for services (OWL-S) is an ontology that makes such functionalities possible[BHL+04]. It provides a semantic layer to the existing standards, it relies on WSDL and extends UDDI.

Even if the vision of the semantic web is still away from becoming reality, OWL-S provides concrete facilities to model service compositions and processes. Before going into details a short overview of the basic concepts will be given.

4.1.3.1 Basic concepts

The service class forms the central entity of the ontology. Figure 4.4 depicts its relations. The *ServiceGrounding* provides information about the transport protocol needed to access the service.

A service can have multiple profiles and groundings.

Service Profile The *ServiceProfile* gives a description of the service to enable discovery. There are 28 predefined attributes which provide the following information:

- What organization is providing the service
- Functional details like input and outputs
- Metadata like what type of service which category etc.

The goal of the profile is to build up a registry and enable automatic discovery. In extension to UDDI it becomes possible to look for a service based on its capabilities.

[8] W3C Semantic Web Initiative: `http://www.w3.org/2001/sw/`

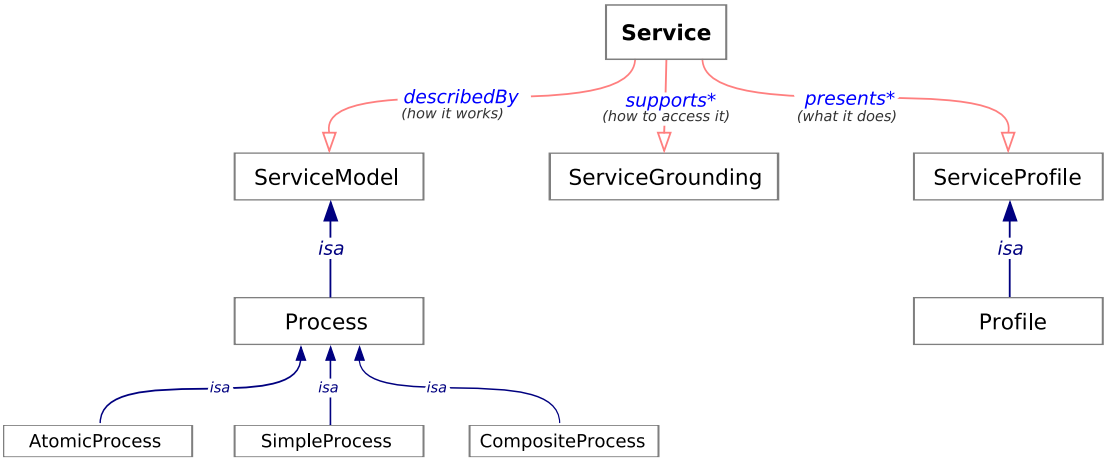


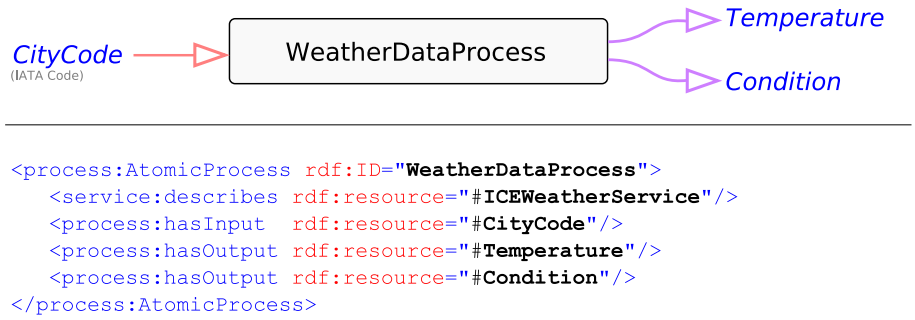
Figure 4.4: OWL-S Service Class and Relations

Service Model Once the service is discovered the *ServiceModel* describes the concrete usage by defining a *Process*. The most basic atomic process takes an input message, processes it and returns the result as output. They can be assembled to form composite processes which describe service compositions.

Service Grounding It is important to understand that the Service Model and the Service Profile are abstract representations which are not bound to concrete protocols or messages. This connection is done by the *ServiceGrounding* which provides a mapping from the input and output of the service model processes and the concrete operations of a service (usually the output and input parts are defined in WSDL definition).

4.1.3.2 Atomic Processes

Figure 4.5 shows as example the atomic `WeatherDataProcess` which is part of the use-case composition described on page 15. It refers to a concrete weather service provider (called ICE). This service requires an IATA city code as input and will provide temperature and condition as output.




```
<process:AtomicProcess rdf:ID="WeatherDataProcess">
   <service:describes rdf:resource="#ICEWeatherService"/>
   <process:hasInput  rdf:resource="#CityCode"/>
   <process:hasOutput rdf:resource="#Temperature"/>
   <process:hasOutput rdf:resource="#Condition"/>
</process:AtomicProcess>
```

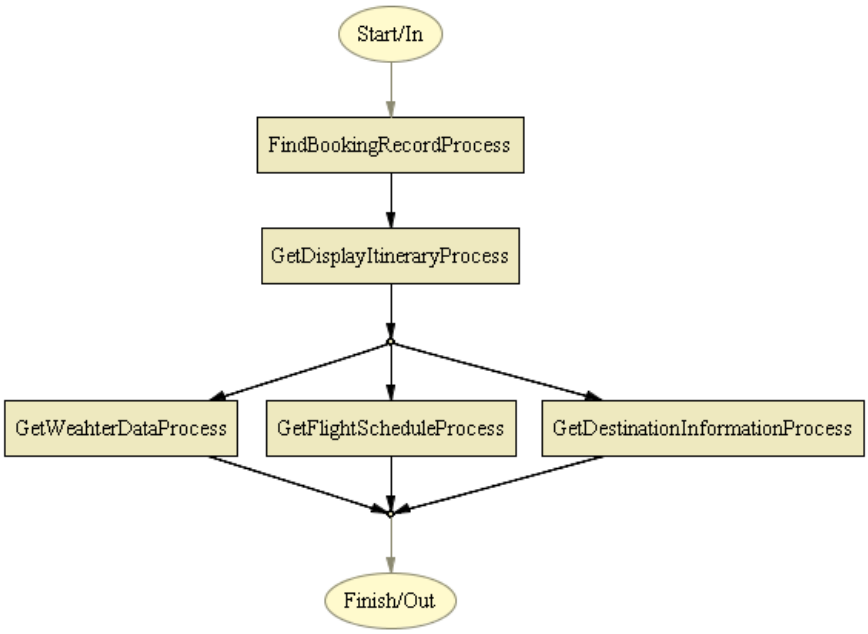
Figure 4.5: OWL-S Atomic Process Example(schematic/notation)

The code snipped shows the process definition in the OWL/RDF-XML syntax. It defines an instance of the atomic process class with the identifier `WeatherDataProcess`. This class is described by relationships like `hasInput` or `hasOutput` which point to other class instances which e.g. define what `CityCode` or `Condition` means.

4.1.3.3 Composite Processes

Composite processes are composed from composite or atomic processes. The composition is done by control constructs like: `Sequence`, `Split`, `Any-Order`, `Choice`, `Iterate`, `Repeat-Until` and others. OWL-S provides the expressiveness that is required by the CSA composition process algebra.

As an example, figure 4.6 depicts the trip information process. It consists of a `Sequence` with three elements. After two atomic process calls a `Split-Wait` construct enables the parallel execution of three other atomic processes. The composition process terminates after the execution of all atomic processes and returns the result. The XML-RDF representation can get very complex (figure 4.6 only contains a snippet).




```
<process:CompositeProcess rdf:ID="TripInformationProcess">
 <process:composedOf>
   <process:Sequence rdf:ID="Sequence_12">
      <process:ControlConstructList rdf:ID="ControlConstructList_13">
      <list:first>
        <process:Perform rdf:ID="Perform_6">
          <process:process>
            <process:AtomicProcess rdf:ID="FindBookingRecordProcess"/>
          </process:process>
        </process:Perform>
      </list:first>
      <list:rest rdf:resource="#ControlConstructList_15"/>
    </process:ControlConstructList>
   </process:Sequence>
 </process:composedOf>
</process:CompositeProcess>
```

Figure 4.6: Example: OWL-S Composite Process Example (schematic/notation)

OWL-S offers various ways to specify the data flow within the composite process. Basically the output is mapped in between the processes. This can be done in two ways:

consumer-pull means that a process is specifying where the input is taken from. It is defined either by usage of the `hasDataFrom` property or an `InputBinding` object.

producer-push means that the input of a following process is specified at the output of a predecessor. This can for instance be necessary when the predecessor depends on a condition like in the following example $((S_1 \otimes S_2) \odot S_3)$. In this case it is much more convenient to define that the output of $S_1$and S_2serve as input for service S_3. To achieve this a special control construct called `Produce` needs to be inserted. The target is defined by a `produceBinding` property.

From this brief description it can be stated that OWL-S also provides the necessary functionalities to define data flows. The actual mapping between concrete services as well as data type transformations can be defined in the process groundings which will be described next.

4.1.3.4 Data-type Conversion through Process Groundings

It is important to understand that OWL-S process definitions are abstractions from real services. Output and input values are actually disconnected from the underlying services. In this way the process definition can be seen as an ideal model of the reality. The concrete implementation is then defined by a grounding typically an instance of the *WsdlAtomicProcessGrounding* class.

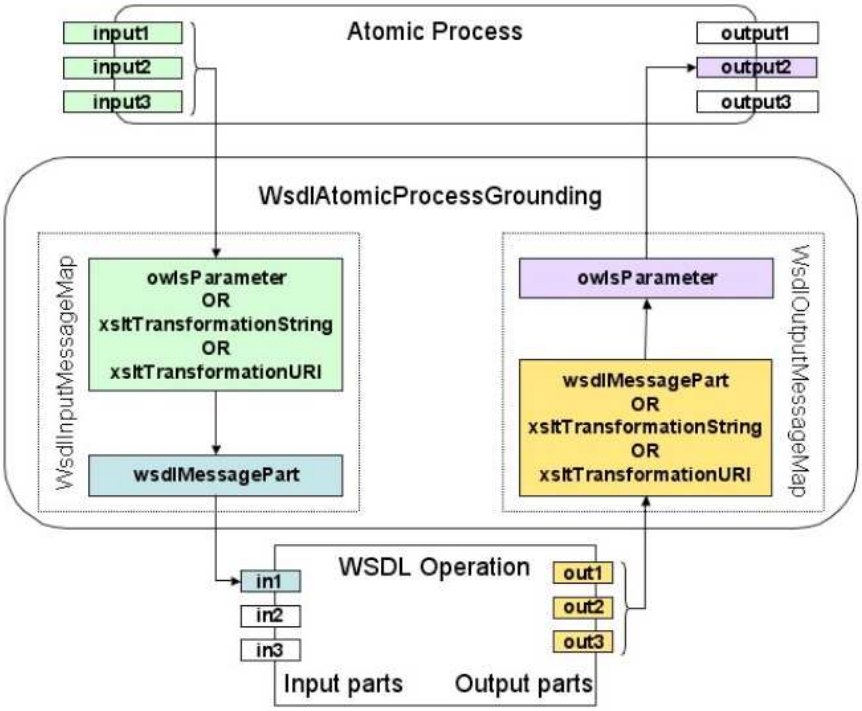


Figure 4.7: OWL-S Service Grounding [SD05]

As depicted in figure 4.7 the grounding provides the adapter between WSDL operations and OWL processes. If the interfaces match 1:1 the grounding just forwards the values. If the interfaces do not match XSL transformations can be defined. This allows even complex interface and data type conversions.

Processes can have multiple groundings which fosters the exchangeability of services and providers.

4.1.3.5 Tool Support

One of the few but very good tools is the *OWL-S Editor*[9][SD05]. It is a plug-in for the popular Protégé ontology and knowledge-base manipulation framework (see figure 4.8).

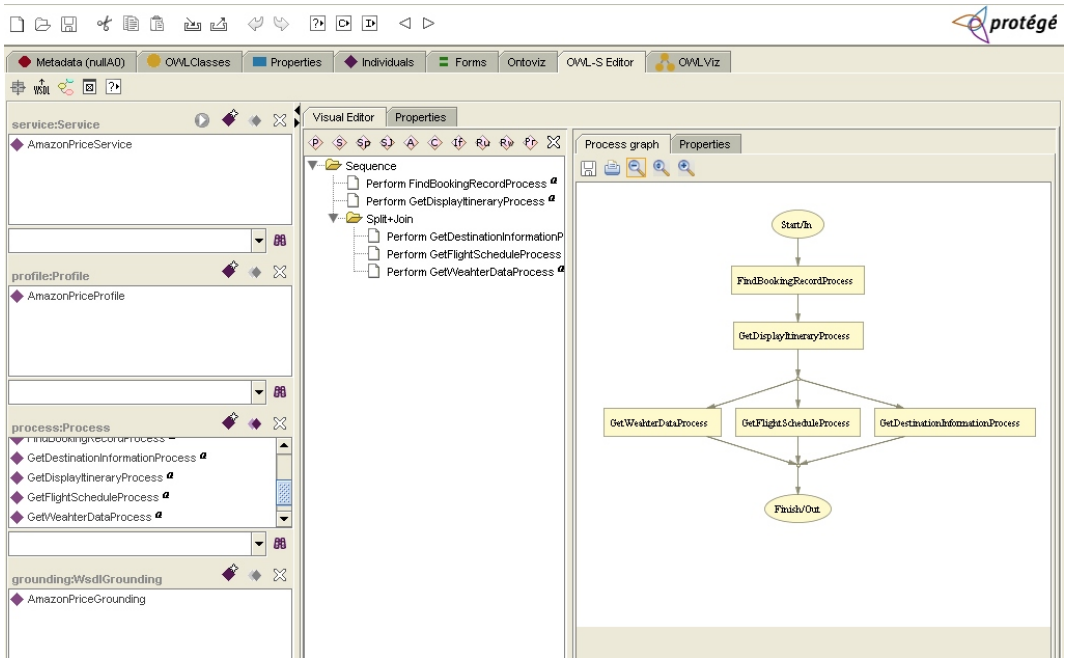


Figure 4.8: OWL-Editor Screenshot

The editor provides the following features:

- Create and edit services, processes, profiles and WSDL groundings
- Visual tool for composite processes (see also figure 4.6)
- Consistency check
- WSDL import
- Input/Output/Precondition/Result Manager
- Overview graph generation
- Execution of the service with generated interface for input and output values

The editor provides an intuitive tool to work with OWL-S but still requires the user to be familiar with the concepts of OWL, Protégé and of course OWL-S.

[9]OWL-S Editor: `http://owlseditor.semwebcentral.org/`

4.1.3.6 OWL-S Engines

The environment to execute OWL-S definitions is called OWL-S Virtual Machine (OVM). It can be used in combination with an OWL-S Matchmaker which serves as registry to discover definitions[**?**].

After the client has obtained a definition that matches the ontological request description, it invokes the OVM. The virtual machine executes the process and uses the WSDL grounding to invoke the neccessary Web Services. When a response is received the result is transformed and sent back to the client.

Current implementations of virtual machines are only able to execute static compositions. They have mainly be implemented in proof of concept projects and must still be considered as immature for production use.

4.2 Selected WS-* Technologies

Second Generation Web Service technologies try to close gaps or to provide a way to integrate additional features. It is important to use these extensions with care because they are sometimes still in the process of specification or only supported by a subset of frameworks (an overview is shown in table 4.1).

WS-Security and WS-Addressing are two of the most common and widely supported SOAP extensions which is why they were selected to implement the necessary addressing and security features of the CSA. The will be shortly introduced in this section.

4.2.1 WS-Addressing

A severe leak of the SOAP standard is the absence of a uniform addressing method. In practice it is either delegated to the underlying transport protocol (e.g. SOAP-Action attribute in HTTP header) or achieved by adding proprietary information to the SOAP header. Both approaches are not sufficient to provide advanced cross platform message exchange pattern (e.g. asynchronous call back, error handling etc.).

In consequence the W3C established the WS-Addressing working group to develop recommendations which extend the Web Service framework for providing uniform addressing mechanism at SOAP message layer. These mechanism exceed the simple declaration of an endpoint. For instance they allow to declare a sender, a recipient, message properties such as a correlation identifier and finally error handling directives.

The specification is divided in three W3C recommendations which treat: the core concepts[**?**], a concrete SOAP-binding[**?**] and the integration into the service description

Feature	Axis 1.x	Axis 2	Celtix	JBossWS	XFire	JAXWS 2.1
WS-Addressing	•	•	•	•	•	•
WS-Eventing	•			•		
WS-Notification	•				○[1]	
WS-ReliableMessaging	•	•	•		○[2]	○[3]
WS-Policy		•				○[3]
WS-Security Policy						○[3]
WS-Security	•	•		•	○[2]	○[3]
WS-Trust						○[3]
WS-Secure Conversation						○[3]
WS-Atomic Transaction						○[3]
WS-Coordination						○[3]
WS-Metadata Exchange						○[3]

[1]via ServiceMix [2]in version 2.0 [3]via WSIT

Table 4.1: Support for selected WS-* Specification in Popular Web Service Stacks [Source: `http://xfire.codehaus.org/Stack+Comparison`]

(WSDL, WS-Policy)[**?**]. Many other WS-* extensions rely on WS-Addressing.

The core specification provides two header type constructs, *Endpoint References* and *Message Information Headers*.

Message Information Headers

The Message Information Headers consist of a loose collection of elements. Mandatory is only the `<Action>` element which allows replacing the SOAP-Action attribute in the HTTP header. This is already a big achievement that ensures transport layer independence. But there are other properties that are actually the basis to realize asynchronous messaging and different message exchange pattern (`<MessageID>`,`<RelatesTo>`, `<ReplyTo>`). An example is shown in figure 4.9.

To detail the information about the endpoint and the service there is the complex type `EndpointReference`.

Endpoint References

The endpoint reference does not only enable the identification of static service endpoints but also specific instances. It contains an mandatory Addressing element and two extensible sections which can carry service specific data:

`EdpointProperties`: help to identify and instance of a service (e.g. by version)[10]

`EndpointParameters`: allow to define service specific parameters (e.g. a customer record number)

In figure 4.9 lines 9 - 14 depict an example for a endpoint reference definition.

```
1  <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
2  xmlns:wsa="http://www.w3.org/2005/08/addressing">
3       <S:Header>
4            <wsa:MessageID>http://example.com/someuniquestring</wsa:MessageID>
5            <wsa:ReplyTo>
6                 <wsa:Address>http://example.com/business/client1</wsa:Address>
7            </wsa:ReplyTo>
8            <wsa:Action>http://example.com/fabrikam/mail/Delete</wsa:Action>
9            <wsa:EndpointReference>
10                <wsa:Address>http://example.com/fabrikam/acct</wsa:Address>
11                <wsa:ReferenceParameters>
12                     <tns:SessionID>25324523</tns:SessionID>
13                </wsa:ReferenceParameters>
14           </wsa:EndpointReference>
15      </S:Header>
16      <S:Body>. . .  </S:Body>
17 </S:Envelope>
```

Figure 4.9: Example of WS-Addressing SOAP Header Elements[**?**]

Modeling State in Web Services

In principle Web Services are stateless but in practice it is often necessary to map stateful resources to services. There is a controversial discussion about this topic and different solutions and standards exist (e.g. WS-Session, WS-Context, WS-ReliableMessaging).

The concept that will be introduced in the next chapter does also require a stateful conversation. The reference-based session model in WS-Addressing will be used to realize it. It was chosen for convenience because it will be used for addressing anyhow and it is one of the most supported Web Service extensions so many implementations exist (see table 4.1).

The realization is rather simple. The server will put a session identifier token in the extensible reference parameter element of the WS-Addressing header. And example is shown in figure 4.9 on line 12. For the next request the client will need to repeat this section so that the server can use the token to reload the session.

[10] EndpointProperties have been removed from the current WS-Addressing Recommendation[**?**]

4.2.2 WS-Security

WS-Security was initiated by Microsoft in October 2001. It has undergone changes and is now an OASIS standard[LKN+06] which is complemented by other specifications such as WS-Trust or WS-Policy.

Overview

The idea of this specification is to introduce a security abstraction layer that allows using different existing implementations like X.509 or Kerberos in Web Services interactions.

Confidentiality, Integrity and Authentication are covered at SOAP message level:

Confidentiality

Today the most common way to ensure confidentiality is to use transport layer encryption. This method is only sufficient when there is a direct communication between client and server. WS-Security introduces much more flexibility. It allows to use symmetric or asymmetric encryption and also to encrypt only confident parts of documents. This way true end-to-end confidentiality can be ensured.

Integrity

XML signature mechanism can be used to verify that a message has not been modified and also to be sure that the originator is the one it claims to be. Various algorithm and enforcement level are possible.

Authentication

WS-Security incorporates authentication methods. It ranges from plain username password token up to certificate exchange. The advantage compared to HTTP authentication lies again in the fact that it is done at message level and is therefore transport layer agnostic.

Elements and Usage

All security definitions are conveyed under a `<Security>` element in the SOAP header. Each security information is represented by a security token which can be signed or unsigned.

The least complex token are `<UsernameToken>`. In the simplest case they contain an element for the name and an element for the password of the user. A bit more sophisticated

```
1  <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="...">
2     <S11:Header>
3        <wsse:Security>
4           <wsse:UsernameToken>
5              <wsse:Username>NNK</wsse:Username>
6              <wsse:Password Type="...#PasswordDigest">
7                              weYI3nXd8LjMNVksCKFV8t3rgHh3Rw==</wsse:Password>
8              <wsse:Nonce>WScqanjCEAC4mQoBE07sAQ==</wsse:Nonce>
9              <wsu:Created>2003-07-16T01:24:32Z</wsu:Created>
10          </wsse:UsernameToken>
11       </wsse:Security>
12    </S11:Header>
13 </S11:Envelope>
```

Figure 4.10: WS-Security Example for a UsernameToken[AN04]

is digest authentication which is depicted in figure 4.10. In this case only a hash of the password is transmitted. The hash includes a nonce[11] value to avoid simple reuse.

To realize integrity and encryption there are binary security token which can refer to XML-Signatures. Finally a collection of standard SOAP faults is defined for different types of security failures (e.g. if a security token is invalid or a certificate is missing). More information and detailed samples can be found in the specification [LKN+06] and in [AN04].

The use of WS-Security still states a certain overhead compared to transport layer security and is only necessary if the level of security or the message exchange pattern do require it. For the scope of this paper user-name token will be sufficient to realize authentication but the general use of WS-Security ensures that at a later point more sophisticated security measures can be put in place.

4.3 Summary

OWL-S as well as WS-BPEL fulfills the central requirements to integrate them into the Credential and Service Aggregator middle-ware. The Web Ontology Language for Services is an academy driven language. It aims to support dynamic (reactive) service compositions based on semantic service descriptions. On the other hand it requires a fundamental understanding for semantic web technologies, OWL and the Resource Description Framework. Present implementations still have to be considered as immature.

WS-BPEL is clearly a proactive language that requires a concrete knowledge of all actors at composition design time. Its extensive integration of common standards like XPath and WSDL ease the adoption and understanding. Finally the wide variety of tools and production grade composition engines lead to the decision to use WS-BPEL inside the

[11] number used once: random value used to preven replay attacks

CSA infrastructure.

To handle authentication and addressing the standard SOAP extensions WS-Addressing and WS-Security have been chosen.

5 Concept

It was shown that there are existing standards and technologies which address certain aspects of the requirements without completely satisfying them. Taking this in account, a component based solution will be developed in this chapter. There is a clear emphasis on the extensibility of the overall solution.

5.1 Introduction

5.1.1 Directions of Approach

In the related work chapter was shown that there are standard languages which are able to fulfill the composition requirements defined in 3.2. As these standards focus primarily on realizing Web Service compositions, they lack in coverage for other important aspects such as security and authentication or context management.

From a conceptual point of view there are two basic directions from which these limitations can be addressed. The first is to introduce extensions to the composition language and its run-time environment. This is actually common practice of existing BPEL implementations. It has the advantage that all aspects can be addressed at a central point. But on the other hand business processes get overloaded with aspects that are not directly related to the composition itself. In addition these extensions limit the interoperability and recreate dependencies on vendors and platforms.

In clear opposition the second approach has the fundamental idea to leave the composition engine and environment untouched. This is achieved by embedding it into a complementary infrastructure.

The concept of the CSA follows this path by providing a modular architecture of components which are designed to be as independent as possible from each other. This is believed to be the optimal approach to provide the necessary extensibility for future requirements.

5.1.2 Overview

Central part of the CSA architecture is a *composition engine* which provides all service composition facilities (figure 5.1-1).

Following the principle of separation of concerns there are surrounding components which handle the following tasks:

- The *Client Connector* is the entry point of all client requests. It provides the first level of authentication and transport related facilities such as encryption and session handling (fig. 5.1-2).
- The *Message Router* delegates the request to the right target endpoint within the composition engine (fig. 5.1-3). It also provides second level authentication by validating the requested message path.

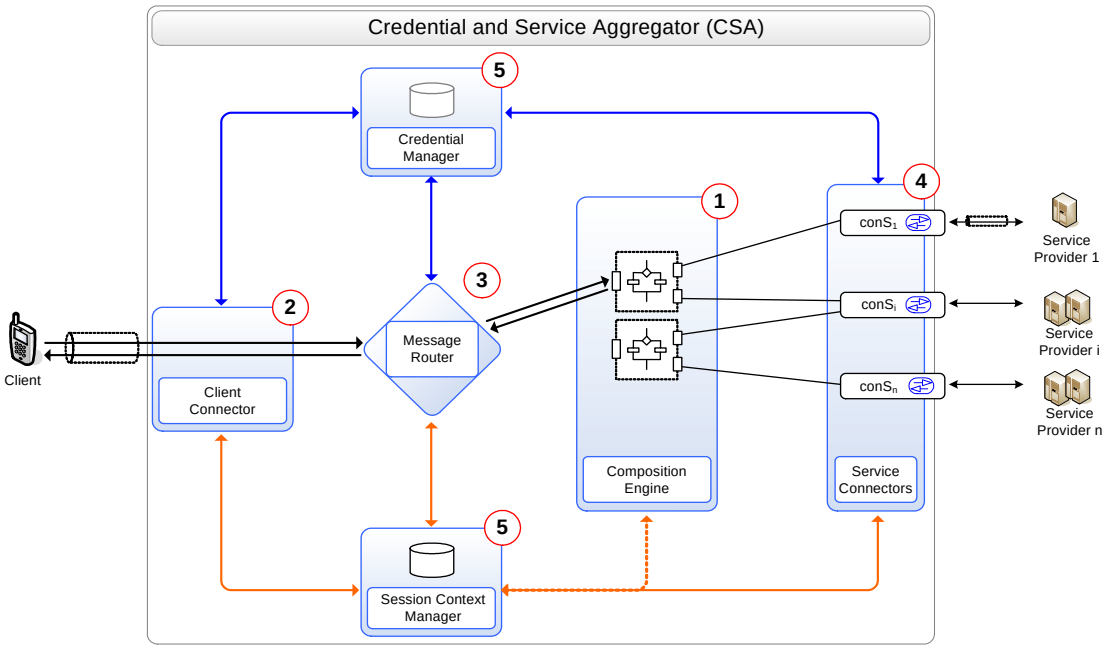


Figure 5.1: CSA Architecture Overview

- The *Service Connectors* are the gateway for the Composition Engine to external services (fig. 5.1-4). They handle service provider authentication and session handling and they also bridge to none standard Web Service implementations.
- The *Session Context-* and the *Credential Manager* provide common functionalities across all components (fig. 5.1-5). They enable a shared security and information context as well as the access to outside entities such as third party authentication sources or user preference stores.

In the following an iterative approach is used to introduce each component and explain the specific features it provides.

5.2 Components

5.2.1 The Composition Engine

A BPEL composition engine is a run-time environment that executes the business processes based on the loaded workflow specification. An engine can host and run multiple synchronous or asynchronous workflows in parallel. They can be consumed like traditional Web Services. The invocation and reply is defined in a WSDL contract. For the client the composition is transparent.

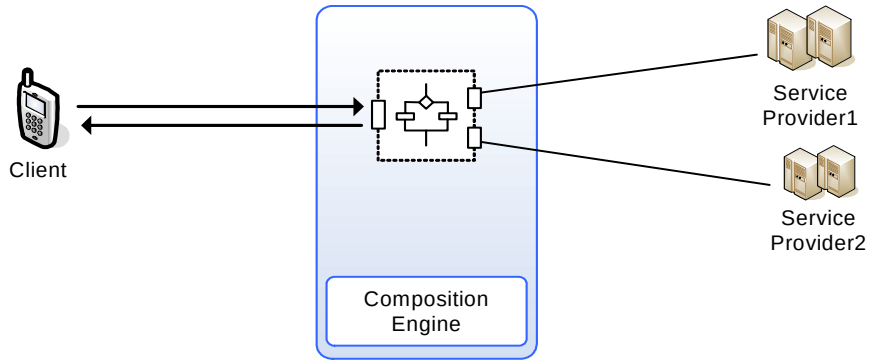


Figure 5.2: CSA Components: Composition Engine

The available engines are mainly different in the level of support for certain features of the WS-BPEL standard and in the way processes are configured and deployed. There is no standard procedure for that. A process configuration is also the most common point to introduce extensions or specific features.

The aim of the CSA is to stay agnostic to a specific implementation. It will only use the standard constructs of the language. All messages between the client and the service connectors are exchanged by applying SOAP and HTTP which are supported by all implementations (figure 5.2).

5.2.2 Client Connector

As the composition engine does not provide sophisticated authentication features it practically exposes its service without any restrictions. For this reason a client connector is put in place. It ensures first level authorization (figure 5.3).

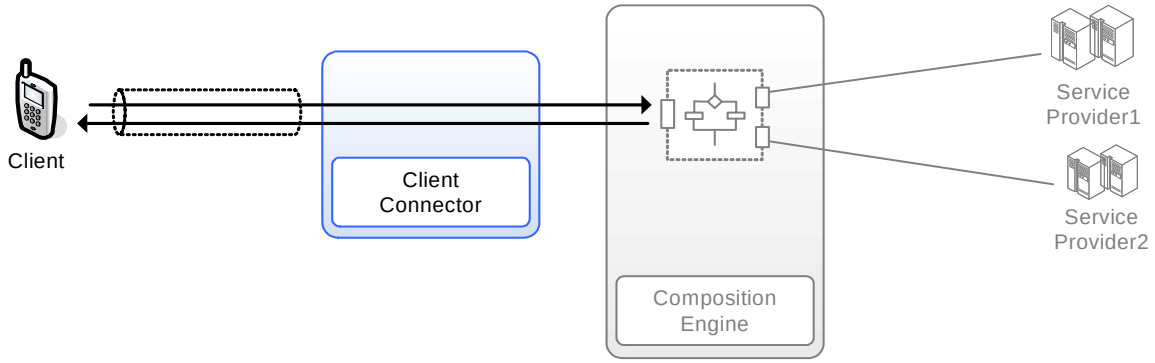


Figure 5.3: CSA Components: Client Connector

The message layer authentication will be realized by applying WS-Security SOAP header extensions. Depending on the type of the application this allows different methods ranging from plain user-name and password token up to certificate based authentication. This is a clear distinction to BPEL engine configuration extensions which usually only provide basic HTTP authentication.

After the user is authorized a session is maintained for follow-up requests. To communicate the session identifier the reference-based session model in WS-Addressing will be used (see 4.2.1).

Finally the Client Connector is the point where transport related concerns can be implemented such as compression or encryption. This is not necessary for the first version but can be a useful extension (see 7.3).

The Client Connector passes all authorized and valid requests to the sub sequent message router or replies with a SOAP fault.

5.2.3 Message Router

A composition or workflow in a WS-BPEL Engine is equivalent to one specific service method. In the use-cased which was described in chapter 3.1 the trip information would be provided by such a single composition. It is very likely that multiple workflows are hosted. For instance there could be different versions or additional service methods. In such a case the client request needs to be forwarded to the appropriate composition (BPEL partner-link).

A message router will treat this task of delegating the message to the right composition entry-point (figure 5.4). The client will encode this information in a standardized way by making use of WS-Addressing attributes.

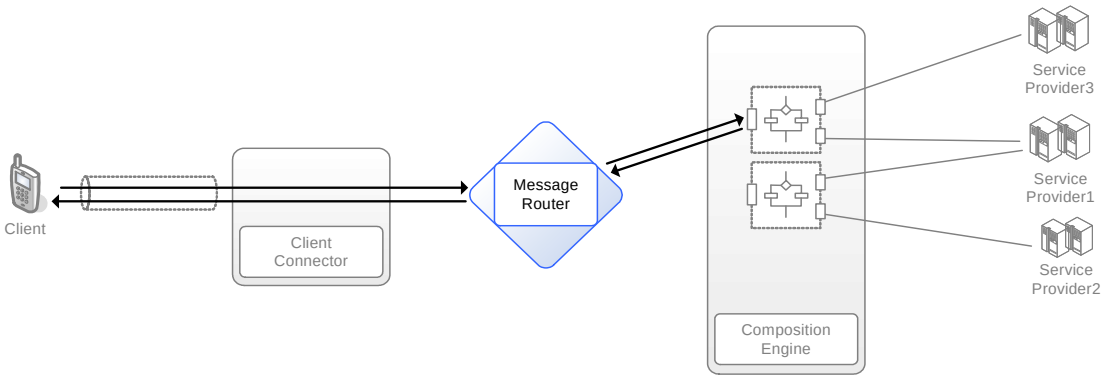


Figure 5.4: CSA Components: Message Router

Before forwarding the message, the router will also verify the user's permission to target the addressed composition. This second layer authorization enables the rejection of fraud requests which result in a fault reply to the client.

5.2.4 Service Connectors

In the Web Service stack WS-BPEL is placed on top of WSDL. It is independent from concrete implementations on lower layers where authentication and security are usually addressed. In consequence there is no native support for concepts like HTTP authentication or SOAP header security token as well as session management mechanism. Also the support of WS-* specification is still rather limited. This makes the integration of secured and stateful services difficult.

In practice these issues are resolved by proprietary extensions which are limiting vendor independency. Fore these reasons authentication and session management will be handled outside of the composition engines by dedicated service connectors (figure 5.5).

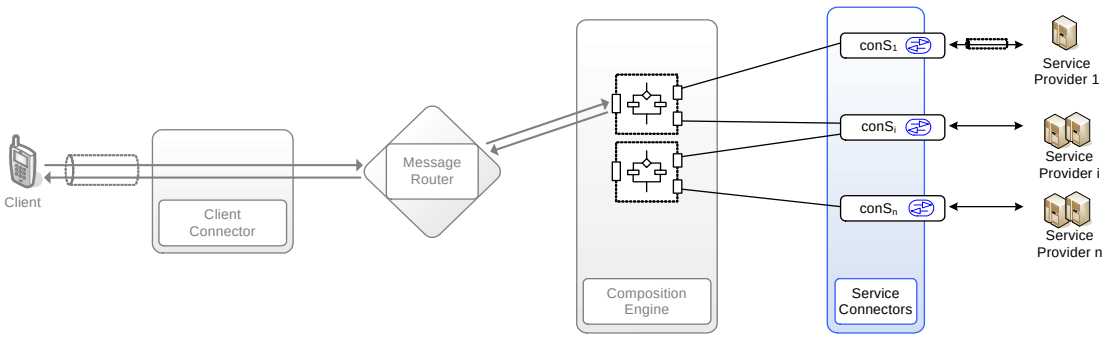


Figure 5.5: CSA Components: Service Connectors

The service connectors will also serve additional integration purposes. As an example WS-BPEL engines are very sensitive to non standard Web Services and even small issues can lead to invocation failures. In these cases the connectors can be used to normalize messages (e.g. correct wrong name-space declarations, encode characters etc.).

It must be pointed out, that such approaches are running against the idea of service orientation. Still they are often the only way to efficiently integrate required services. In this way they can be seen as the glue code in a black-box composition. Encapsulating such functionality in dedicated connectors fosters their reuse and helps to keep the composition free of non-standard BPEL extensions.

5.2.5 Communication Methods

Client Connection

Client connections to the CSA are done by exchanging SOAP messages through a Web Service interface based on a WSDL service contract. SOAP extension should be used instead of proprietary or transport layer solutions (e.g. for session management or security). Different transport protocols should be supported in future versions (e.g. asynchronous messaging).

BPEL Engine

The composition engine is connected through WSDL partner links. The client requests and the workflow reply are exchanged through SOAP messages. Furthermore all Service Connectors expose a Web Service interface that can be invoked by the business process. Service Connectors do offer the possibility to bridge to none Web Service providers.

Inter-Component Communication

For Context and Security management the container infrastructure should be used. Furthermore the Session Context Manager will provide a Web Service to enable the Composition Engine to access contextual information (Context Pull as described in 5.4.3.2).

5.3 Credential Management

5.3.1 Overview

Combining credential management with service composition is the main purpose of the CSA. The access and management of credentials is an important requirement to enable authorization and third party service invocation. Client- and Service Credentials can be distinguished:

Client Credentials

The client credentials can be distributed across heterogeneous sources such as user databases or directories. They must be normalized and provided by a central entity. Authentication and especially authorization should not be hard wired within the components but need to be configurable through the outside (fig. 5.6).

Service Credentials

The consumption of business services throughout a service composition often requires authentication. This can be simple user-name and password token, structured sign-in messages or certificates.

The concrete authentication is handled by the service connectors but the credentials are provided by the centralized credential manager (fig. 5.6-right). This fosters transparency and control and provides the possibility to correlate service with client credentials.

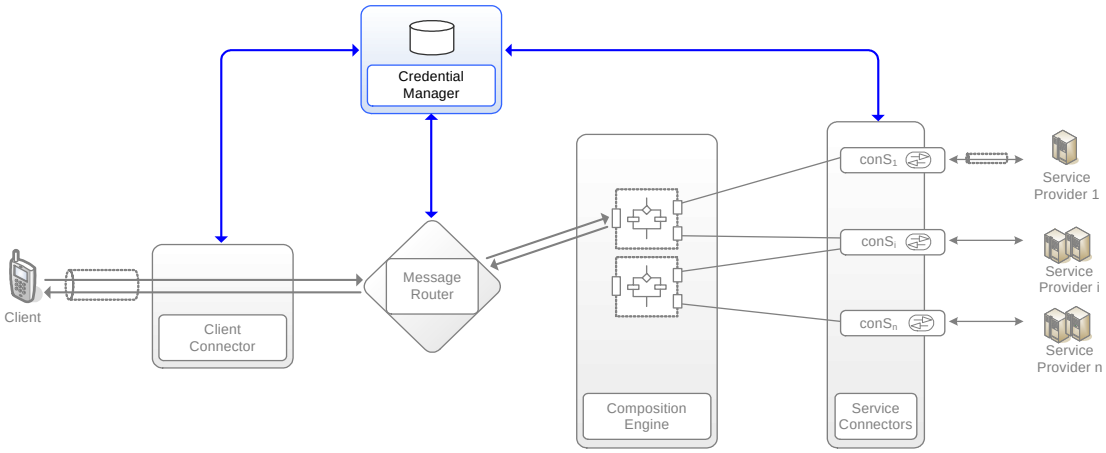


Figure 5.6: CSA Credential Manager

5.3.2 Client Credential Sources

The idea of the CSA is to offer a flexible integration in existing infrastructures and credentials sources are a typical example. Every hosted application is likely to have different requirements in regards to authentication and authorization.

To name an example it could be envisioned that the user of the Mobile Travel Wallet (see 3.1) can authenticate by using an existing frequent flyer account. In this case the CSA must be able validate these information against the external systems of the responsible providers. The Credential Manager will provide this access by making use of an Authentication Framework which can handle common protocols and technologies such as:

- Lightweight Directory Access Protocol (LDAP)[1]
- Single Sign-On (SSO) and the various implementations [2]
- HTTP Basic Access Authentication [3]

External sources will be complemented by a *local credential store* within the credential manager. It is mainly intended for validation and testing purposes but also for use-cases that do not provide existing credential facilities.

The complete data model is depicted in figure 5.7:

- The local credential store allows distinguishing between different applications. This means that each composition is related to an superior application (e.g. the TripInformationProcess belongs to the Mobile Travel Wallet application).
- Each application has a number of users (individuals) which have associated username and password credentials.
- Each individual can have one or more free parameters which make it possible to store additional data such as a telephone number or an address.

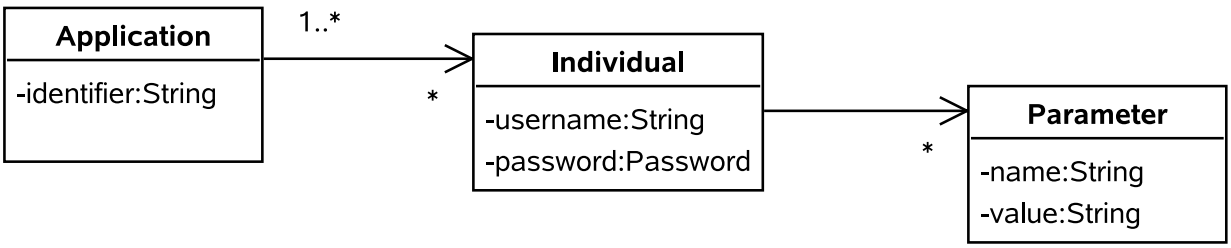


Figure 5.7: Data Model of the Local Credential Store

[1]Lightweight Directory Access Protocol (LDAP): `http://tools.ietf.org/html/rfc4510`
[2]Single Sign-On Introduction: `http://www.opengroup.org/security/sso/sso_intro.htm/`
[3]Basic Authentication in HTTP/1.0: `http://tools.ietf.org/html/rfc1945`

5.3.3 Service Credential Store

The credentials which are necessary to access external services are stored in a local datastore within the Credential Manager. It is able to hold an extensible list of different credential types.

Currently the following types are supported.

- Identifier and password token (e.g. for HTTP Basic Authorization)
- structured messages (e.g. for extended authentication process which requires additional parameter)
- Certificates (e.g. for X509 authentication)

The store is currently realized in form of a structured XML configuration file which is part of the overall application configuration and is loaded by the Credential Manager.

5.4 Context Management

The term context is two fold. It can be used to describe the environment which the client application is executed in by gathering data about the user, the device capabilities, the connection and so on.

But a context can also be a special storage through which different components are able to share data even throughout long running transactions. This is often referred to as the *session context* or simply a *session*.

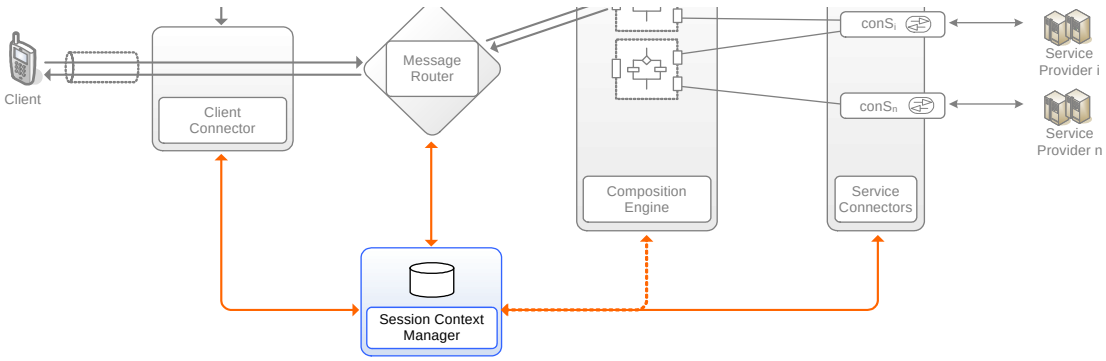


Figure 5.8: CSA Session Context Manager

The Credential and Service Aggregator is unifying both concepts in a single Session Context Manager. It forms the key integration unit that offers all components of the CSA

the possibility to share information independent from the actual message and throughout the lifetime of a client session (figure 5.8). This concept will be explained in the following sub sections.

5.4.1 Context Lifecycle

A service composition is initiated by a client request it is executed and finishes when the result is returned. A user session is a sequence of one or more such executions. The life cycle of the CSA context is identical to a user session. In the following all aspects will be explained in detail.

5.4.1.1 Context Initialization

During the initial authentication handshake the Client Connector requests the Context Manager to start a new session. Figure 5.9 gives an overview of the individual steps:

1. A Client sends an authentication request.
2. The Client Endpoint invokes the Context Manager to start a new Session.
3. The Context Manager replies with a unique Session Identifier.
4. The Session Identifier is attached to the client response (WS-Addressing correlation identifier).
5. The client session is now initialized and can be used for functional request messages.

Along with the session request, the Client Connector can store user context information in the session context. These parameters could be part of the handshake (e.g. the client explicitly sends information about itself) or they could be taken from other sources such as the credential manager (e.g. the telephone number or name of the user).

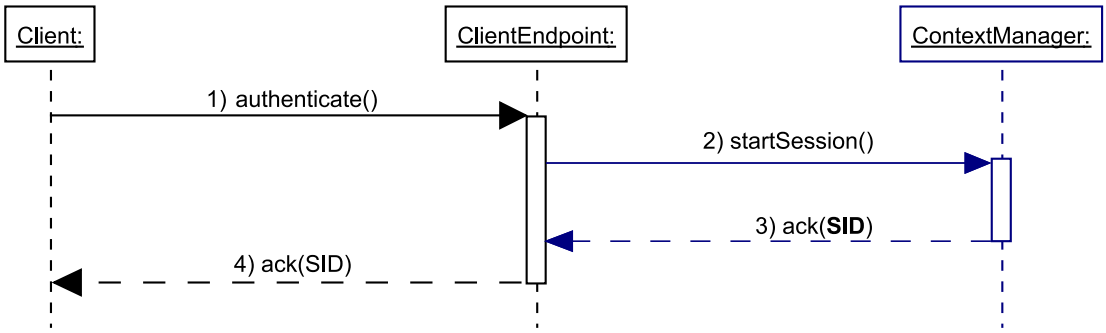


Figure 5.9: Context Lifecycle - Initialization

5.4.1.2 Context Reload

For each of the follow up calls, the client communicates the session identifier in the SOAP header by applying the WS-Addressing specification. It is used by the Client Connector to reload the context. Figure 5.10 shows a sample sequence:

1. A Client request includes the session identifier obtained through a prior initialization (as shown in figure 5.9).
2. The Client Endpoint sends a load request to the Context Manager.
3. Then the functional request can be processed by the other components of the CSA.

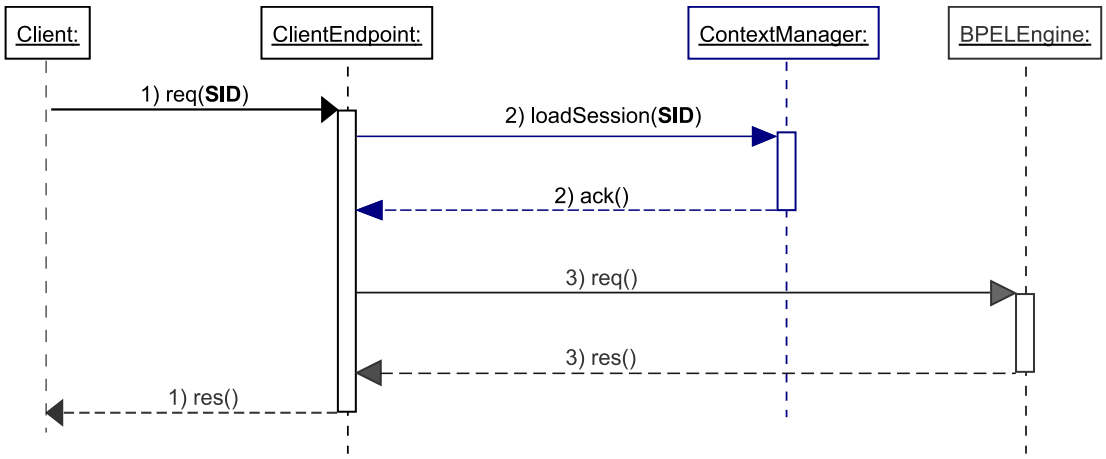


Figure 5.10: Context Lifecycle - Session Load

After the context is loaded, all components can access it and store or retrieve information to accomplish complex tasks like:

- Persist data that is needed during a follow-up request (e.g. a service session identifier)
- Provide data to another component (e.g. information not related to the message itself)
- Access context information needed to personalize the service (e.g. the user language, screen size etc.)

5.4.1.3 Context Use-Case

To visualize the usage of the Context Manager a small use-case shows how it can be used to realize a transaction with an external Service Provider (numbers as in figure 5.11):

1. The BPEL Engine requests an external service through a Service Connector.
2. The Service Provider requires an active session for authentication and transaction management. The Service Connector sends a request to the Context Manager to see if such a session already exists in the user context.
3. By returning `NULL` the Context Manager indicates that no service session is present.
4. Through an authentication the Service Connector obtains a new service session (SSID token).
5. The Connector stores the SSID token to the context for later use. Then the Service Provider functionality is invoked and the result returned to the BPEL engine.
6. At a later point in time a follow-up request from the original or a subsequent business process is done to the Service Connector.
7. This time a service session is found in the user context (SSID that was obtained in step 4.).
8. The Service Provider can be invoked without prior authentication and the transaction context with this provider stays alive.
9. The result is returned to the BPEL process.

The example has shown how the Context Manager can be used to persist the state of a Service Connection. This allows to maintain business transactions throughout one or more business processes (e.g. for booking or payment). Furthermore the overhead of service session initializations can be significantly reduced.

5.4.2 Context Data Structure

The context data structure is represented by a simple dictionary that contains parameters which are represented by a key and one single value. The key must be unique. Persisting a parameter with an existing key will overwrite the stored value.

To enable the conversion to and from a textual representation it must be required to only store values of type string.

5.4.3 Composition Engine Integration

It was said that the CSA context is accessible for all components to store and retrieve information that are relevant to a user session. It will be shown that such a context integration into the composition engine itself can be very useful and two possible ways of realizing this will be discussed.

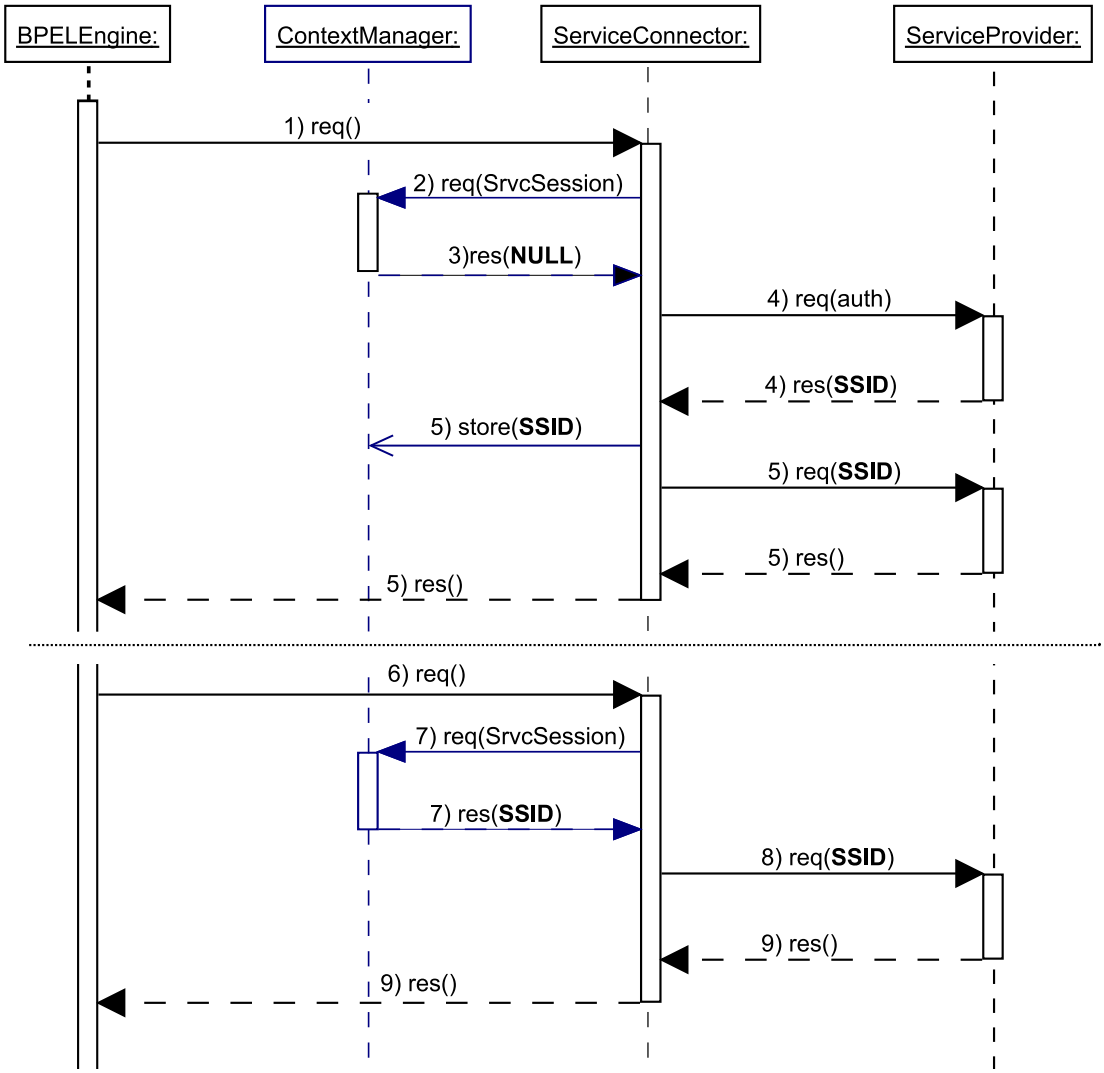


Figure 5.11: Context Lifecycle - Service Provider Transaction

5.4.3.1 Application Fields

The WS-BPEL language as well as present BPEL Engines do not know the concept of a context which covers more then the life cycle of an executed composition.

In the following two basic categories of context integration into a service composition are shown. They give a first impression about the advantages:

Control Flow Adaptation

Contextual information can be used to dynamically adapt the control flow of a service composition. This means that a parameter from the user context can be used to make decisions about the subsequent execution path.

As an example role based access is realized within the composition by using control flow adaptation. Figure 5.12 shows a schematic illustration where depending on the privileges a premium service is invoked instead of a basic service. This example also makes clear that context information should not be a direct part of the functional request. Otherwise the client would obtain an unwanted possibility to manipulate the composition.

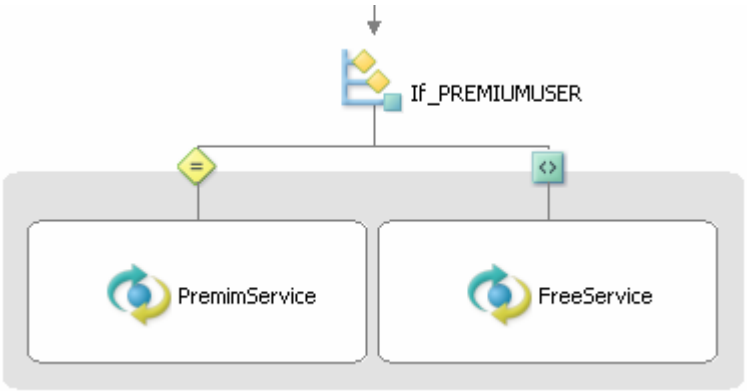


Figure 5.12: Composition Engine Integration - Control Flow Adaptation (Example)

Content Adaptation

The integration of data from the user context can be used to enrich a composition. The knowledge about things like user preferences, device capabilities or connection bandwidth can be used to personalize the outcome of a composition.

Figure 5.13 depicts abstractly how a service invocation is parameterized with the language code of the user.

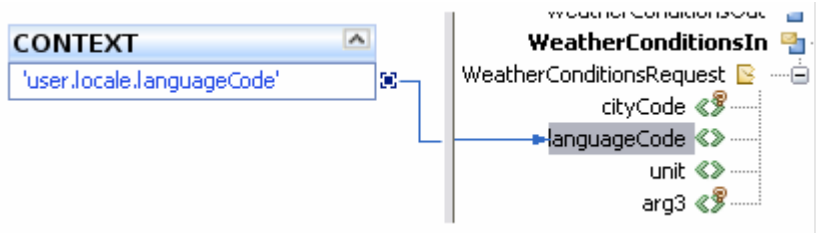


Figure 5.13: Composition Engine Integration - Content Personalization (Example)

5.4.3.2 Realization

Semantically context information is not a direct part of the request payload and should not be defined or manipulated by the client. Furthermore there are context parameter which are not known by the requester.

In the case of the CSA all context information is maintained by the Context Manager. The main question is how these information can be combined with the client request without extending the WS-BPEL standard or run-time environment. Hence a solution must make use of partner links as the only existing connection of the workflow to the outside. This constraint leads to two possible solutions:

Context Push

To isolate the context from the request message it can be added as an additional part of the request SOAP body. In the workflow it is assigned to a local variable which can be used throughout the composition. On termination the possibly modified context is then returned as part of the result.

The message router will play the key role in this scenario. It will get the context from the Session Context Manager to add it to the request message and strip it from the reply (figure 5.14).

This push mechanism is completely transparent to the client and less invasive for BPEL design. On the other hand a certain overhead is created because this procedure is independent from the actual need of the context information within the composition.

Context Pull

Values of the user context can be retrieved like any other information by invoking a Web Service through a BPEL Partner Link. To make this possible the Session Context Manager has to be extended to provide a Web Service interface (figure 5.15).

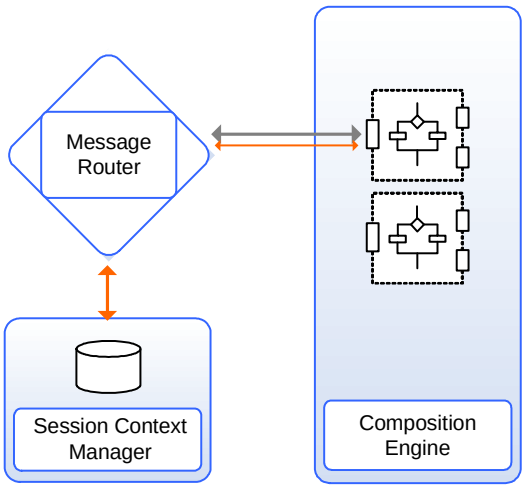


Figure 5.14: Composition Engine Integration - Context Push

In the next chapter the implementation of such an interface is described in detail (see 6.2.7.2). It basically contains one operation to store a parameter name/value pair and another operation to retrieve the value for a given parameter name.

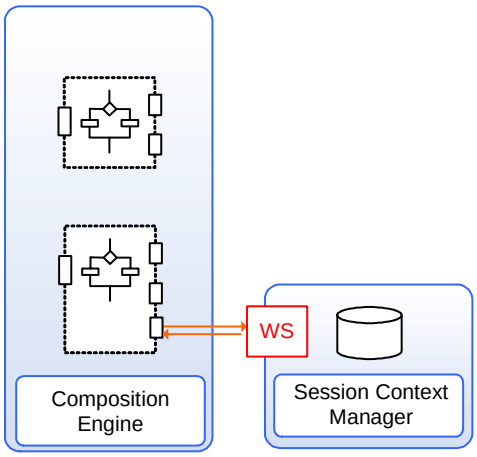


Figure 5.15: Composition Engine Integration - Context Pull

Consistency

If a context parameter is modified during the execution of a composition, it must be synchronized with the Context Manager store. In the case of a Context Push scenario this can be done transparently because all parameters are returned with the reply and can be stored automatically.

In case of pulling a context parameter, the BPEL workflow designer must pay attention to updates and explicitly write the value back through the Context Manager Service interface.

Push vs. Pull

Pulling a context parameter through a Web Service partner link is natural in the sense of WS-BPEL. It furthermore avoids the overhead of having the whole context copied to every workflow even if it is not needed.

On the other hand the context push facilitates the workflow creation. The designer must not explicitly call a service nor pay attention to consistency.

5.5 Summary

The component based approach that was shown in this chapter is a comprehensive way to fulfill the requirements of a flexible infrastructure which combines service aggregation and credential management. It is also a suitable architecture which is open for extensions and the integration complex and non standard services.

The next chapter will take this concept and suggest a possible implementation. A use-case will be shown to proof the feasibility.

6 Implementation

A prototype will be implemented to validate the concept that was described in the previous chapter. In parallel a client application that implements the use-case described in chapter 3.1 will be developed. The validation will take place in several steps up to a production environment. For this reason there are high requirements to the quality and robustness which shall be ensured by the application of enterprise development principles and frameworks. They will be described in the first section followed by an architecture description. Finally an evaluation and outlook will be done.

6.1 Foundations

Beside the core aggregation functionality, the CSA is a infrastructure component. It has a strong emphasis on providing a flexible and configurable architecture. To achieve this, two enterprise application strategies will be applied: *Inversion of Control* and *Aspect Oriented Programming*. Spring, a framework that provides this features will be used to base the actual implementation up on[**?**].

In the following a short introduction to the design principles and the benefits of this approach will be given.

6.1.1 Design Principles

6.1.1.1 Inversion of Control

The components of the CSA should not be tightly coupled to each other. This means that they are not to be hard-wired through direct references. Instead the connections should be configured and controlled from an outside entity.

There are two common ways to solve this problem: Dependency Injection and Service Locators. Both approaches require the programming against well defined interfaces and a supporting infrastructure.

Dependency Injection

To obtain flexibility, a class should not initialize a concrete instance of an interface. This is a common problem that is one of the driving factors between many well known software patterns (e.g. factory). The externalization of this initialization is referred to as dependency injection. To achieve this, an additional entity called *assembler* is needed. Based on a configuration it is able to assign a concrete instance.

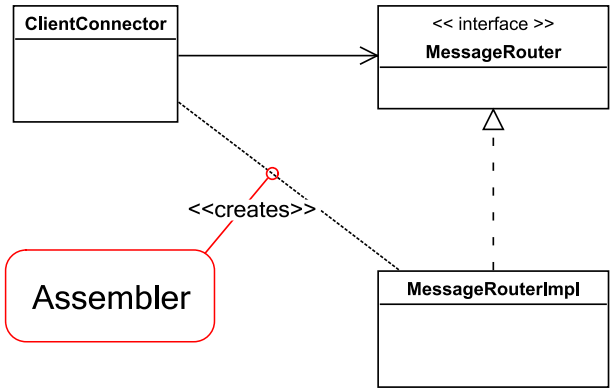


Figure 6.1: Inversion of Control through Dependency Injection

Figure 6.10 shows a simple example. The class `ClientConnector` has a reference to the `MessageRouter` interface. It does not initialize this reference with a concrete implementation. This is done from the outside by an Assembler.

Service Locators

Dependency management can also be delegated to a service locator. In this case the dependency is registered in a directory where it is identified by its interface. Through a service locator an application can then obtain a concrete implementation.

6.1.1.2 Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP) is an extension to existing programming paradigm like Object Oriented Programming[**?**]. It provides a unique way to handle crosscutting concerns.

Crosscutting Concerns

Software development treats with *core concerns* and *system concerns*. A core concern is related to a business problem which is tied to a use-case. System concerns are supportive

facilities that the core concerns indirectly depend on. An example could be the enforcement of user authorization at different points of a system, transaction management or monitoring.

System concerns are crosscutting core concerns and they have the peculiarity of change. For example security is often introduced at a later point of development. On the other hand monitoring and debug facilities need to be removed from production releases. This is why it is desirable to apply these crosscutting concerns in a flexible and transparent manner.

Object-oriented languages were not suitable because they do not provide possibilities to separate or hide system concerns from core concerns. This was one of the most important motivations for the development of J2EE and .NET which use heavy weight containers to approach this problem. The consequence is that software has to be tighly adapted to a concrete container implementation. Hence the development gets dependent on vendor specifics.

Aspect Oriented Programming uses a less invasive way by providing declarative language extensions to add crosscutting concerns. An AOP framework weaves the aspects and the core concerns together, the code remains unchanged.

Interceptor Strategy

In the terms of OO-programming the interceptor pattern enables to add code between a source and a target object to manipulate or monitor the data and control flow.

In AOP interceptors are referred to as *advice*. The main advantage is that the placement (*pointcut*) of such an advice is defined separately from the program code. An AOP weaver will then combine the program code and the interceptors. The weaving can take place before or during compilation as well as on run time. Aspects can be changed "on the fly".

Through the declarative definition the invocation of an advice is nearly possible at any level (e.g. methods, constructors, static fields etc.). Common types of advice are:

- before advice - invocation before calling the target
- after advice - invocation after returning from the target
- throws advice - when an error was thrown by the target
- around advice - before and/or after calling the target

Typical areas of usage are monitoring, transaction, encryption, error recovery or data type conversion.

6.1.2 The Spring Framework

Spring[1] is an open source enterprise application framework that combines Inversion of Control and Aspect oriented Programming. It uses a layered approach so that the developer can choose from a pallet of building blocks (e.g. persistence, transaction management etc.). In these layers existing technologies are integrated (e.g. hibernate persistence, acegi security and others). The most significant difference to J2EE is that the objects do not need to be written against tight and often overloaded specifications and that no heavy container is needed for execution. This is why in this context the terms lightweight container and Plain Old Java Object (POJO) are frequently used[2].

6.1.2.1 Overview

As mentioned above Spring is divided in different layers with the goal to clearly distinguish between certain concerns of enterprise application development. This way only the necessary components need to be integrated into a concrete development.

Figure 6.2 shows the six modules as well as some of the names of related projects which are integrated[JHA+07].

The whole framework is based on the *core* which provides the inversion of control facilities which have been discussed above. A so called bean-configuration manages the instantiation of classes and their parameterization through constructors and set methods which also allow referencing other beans.

The other packages are each dedicated to a certain aspect of enterprise application development:

- DAO aims to facilitate the usage of JDBC and provides transaction management.
- The ORM module integrates object binding frameworks.
- Aspect Oriented Programming is providing programming language extensions such as interceptors, pointcut and dynamic proxies.
- JEE contains support for enterprise application infrastructure services such as Java Messaging, Enterprise Java Beans integration or Java Management Extensions
- The Web package provides an integrated Model View Controller implementation as well as the possibility to link in many 3rd party technologies.

A more detailed introduction would exceed the scope of this document. The CSA will be using mainly the core functionality and the Spring Web Service module.

[1] The Spring Framework: `http://www.springframework.org/`
[2] Other lightweight application frameworks are e.g. the Pico Container or Have Mind

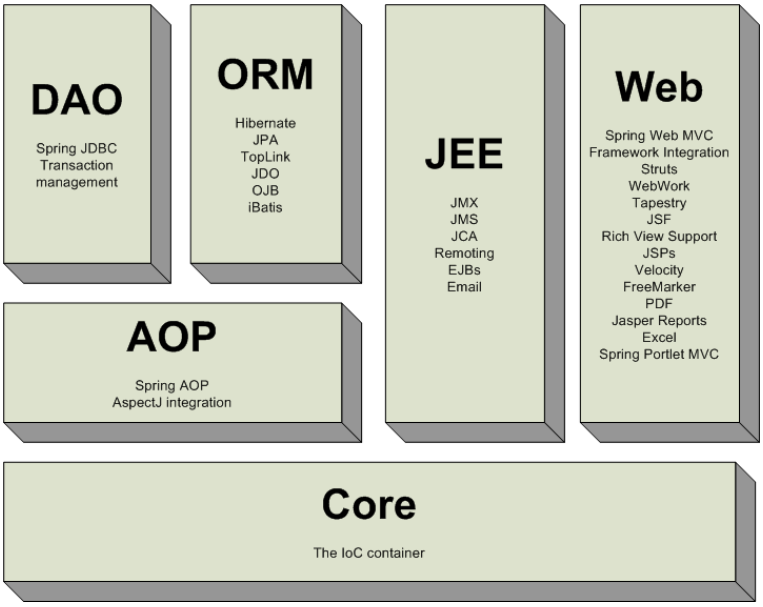


Figure 6.2: Spring Framework Architecture [JHA+07]

6.1.2.2 Spring Web-Services

Spring-WS is a framework that facilitates the development of Web Service based applications[AP]. It provides message dispatching, XML handling and a WS Security implementation.

Web Service Abstraction and XML Handling

Web Service technology is still evolving an a wide variety of implementations, so called *Web Service Stacks*, are available. The same can be stated for the related XML handling technologies which range from different Parsers, Document Object Models (DOM) to various Object Binding techniques. To make a choice is often hard because requirements of the future are difficult to estimate. Possible examples could be the discontinuation of a Web Service stack or that a concrete XML handling implementation does not scale as well as it was expected.

The Web Service module of Spring provides an abstraction layer on top of these technologies. The concrete implementation can be defined in the configuration (examples will be shown in the next section).

Message Dispatching and Interception

Apart from its abstraction capabilities, Spring-WS was chosen because it contains a flexible message dispatching framework which is is particularly suitable for the development of the CSA infrastructure. The central point is the `MessageDispatcher` that processes all incoming messages. It uses an `EndpointMapping` to identify the so called `Endpoint` which will handle the message. This mapping can be done in different ways:

- by name of the payload root tag (`PayloadRootQNameEndpointMapping`)
- by SOAP-Action attribute of the HTTP header (`SoapActionEndpointMapping`)
- by using the Web Service Addressing specification (as of version 2.5)
- by a default endpoint for 1:1 message dispatch

Figure 6.3 shows the request processing workflow. An endpoint can be practically any kind of Java class but there are also abstract endpoint implementations to ease the implementation (e.g. `AbstractMarshallingPayloadEndpoint`).

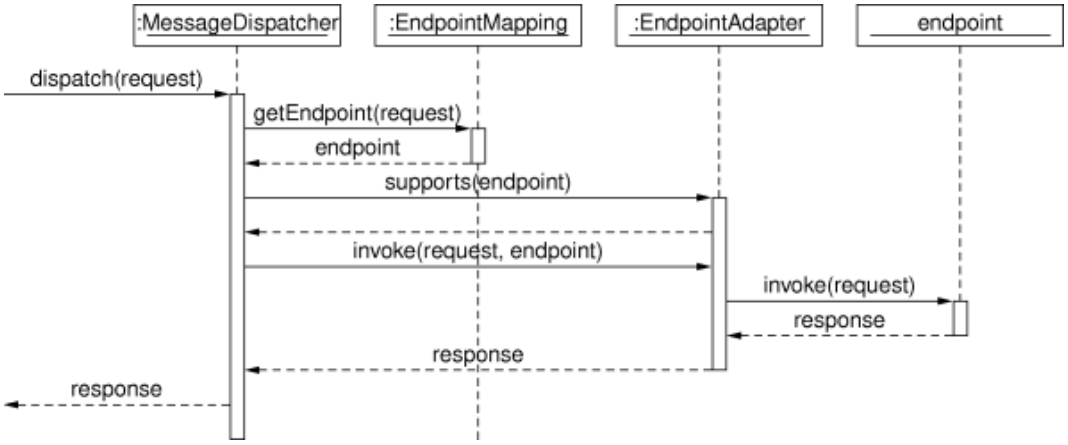


Figure 6.3: The request processing workflow in Spring Web Services[AP]

Crosscutting concerns can be delegated to classes that implement the `EndpointInterceptor` interface which can be plugged into the message flow to the endpoint if needed. Points of interception are on request, on response or in case of an soap fault from the endpoint.

Web Service Security

Web Service security is also part of the Spring WS framework. It covers three aspects:

Authentication to grand or deny access to a user, device or service

Digital-Signatures to ensure the integrity of a message

Encryption to provide a strong message layer security

These security mechanisms are based on the WS-Security specification that was explained on page 40. The integration is done by configuring the `XwsSecurityInterceptor` which is based on the Web Services Security package (XWSS)[3].

The aspect and level of security must be defined in an *xwss security policy* file which is used by the interceptor[**?**]. Figure 6.4 depicts an example which defines a public key encryption for the SOAP envelop body section. In a similar way the authentication or signature can be added.

```
2   <xwss:Encrypt>
3     <xwss:X509Token certificateAlias="s1as"/>
4     <xwss:KeyEncryptionMethod  algorithm="http://../xmlenc#rsa-oaepmgf1p"/>
5     <xwss:DataEncryptionMethod algorithm="http://../ xmlenc#aes128-cbc"/>
6     <xwss:EncryptionTarget type="xpath" value=".//SOAP- ENV:Body"/>
7   </xwss:Encrypt>
8 </xwss:SecurityConfiguration>
```

Figure 6.4: Example of a XWSS Security Policy File[**?**]

The `XwsSecurityInterceptor` acts as a mediator which does not directly handle the authentication or encryption. They are delegated to one or more *callback handler*. Spring offers a series of predefined handlers which allow e.g. authentication against a local key-store (`KeyStoreCallbackHandler`) the signing with X509 certificates (`SignatureKeyCallback`) or authentication against the Acegi Security Manager (`AcegiDigestPasswordValidationCallbackHandler`).

The assembly of interceptor, policy file and Callback Handlers is done in the central bean configuration file (example figure 6.5).

While Spring-WS provides flexible security integration, it does not directly act as an authentication provider. To fill this gap the Acegi Authentication Framework will be used.

6.1.3 Acegi Authentication Framework

An Authentication Manager is a central entity which addresses all authentication related concerns. The most important feature is the abstraction from the complexity of security mechanism and from concrete authentication sources. Such sources could be a simple files or database tables with user-name and password entries. But it could as well be a external providers that need to be integrated though directory access protocols or single sign-on services.

[3]Java Web Services Security package (XWSS): `http://java.sun.com/webservices/xwss/`

```
1  <beans>
2    <bean id="wsSecurityInterceptor" class="org....XwsSecurityInterceptor">
3       <property name="policyConfiguration"
4                 value="classpath:securityPolicy.xml"/>
5       <property name="callbackHandlers">
6         <list>
7            <ref bean="keyStoreHandler"/>
8         </list>
9       </property>
10   </bean>
11   <bean id="keyStoreHandler" class="org.spring...KeyStoreCallbackHandler">
12      <property name="keyStore" ref="keyStore"/>
13      <property name="privateKeyPassword" value="changeit"/>
14   </bean>
15   <bean id="keyStore" class="org….KeyStoreFactoryBean">
16      <property name="location" value="classpath:keystore.jks"/>
17      <property name="password" value="changeit"/>
18   </bean>
19 </beans>
```

Figure 6.5: Example of a Spring-WS Security Configuration[AP].

The CSA will use Acegi Security as authentication manager. It is particularly suitable because it can be seamlessly integrated through the Spring IoC container. Among many features the following are of special importance:

- plug-able Security without change of code
- fine grain Access Control List definitions
- granularity ranges from basic authentication up to X509 certificate exchange
- support for all major authentication concepts (LDAP, SSO, etc.)
- policy definition and enforcement (e.g. channel security level)
- class and method layer security by using AOP

Further details of the features provided by Acegi can be found on the project web site[4].

Integration with Spring WS-Security

The following example demonstrates the integration of Acegi with Spring WS by using a special call-back handler inside the WS-Security Interceptor configuration (figure 6.6 lines 1-12) . The handler references a service which uses a data access object to retrieve the user credentials from a local data source(fig. 6.6 lines 8-9, 13-16). In addition a XWSS Security Configuration defines access constraints (6.6 bottom lines 1-5).

[4]Acegi Security Framework: `http://www.acegisecurity.org/`

```
1  <bean id="wsSecurityInterceptor"
2  class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
3     <property name="policyConfiguration"
4               value="classpath:com/amadeus/ori/csa/security/policy.xml"/>
5     <property name="callbackHandler">
6        <bean class="org.springframework.ws.soap.security.xwss.callback.
7                     acegi.AcegiDigestPasswordValidationCallbackHandler">
8           <property name="userDetailsService"
9                     ref="userCredentialProvider"/>
10       </bean>
11    </property>
12 </bean>
13 <bean id="userCredentialProvider"
14        class="com.amadeus.ori.csa.security.UserCredentialProvider">
15            <constructor-arg ref="userDAO"/>
16 </bean>
```

```
1  <xwss:SecurityConfiguration dumpMessages="false"
2            xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
3    <xwss:RequireUsernameToken
4            passwordDigestRequired="true" nonceRequired="true"/>
5  </xwss:SecurityConfiguration>
```

Figure 6.6: Spring WS Call-Back Handler using the Acegi Security Manger

Integration using Spring AOP

It was mentioned that Aspect Oriented Programming is especially suitable to introduce security concerns. The following example shows how it can be used protect the access to a method using the Acegi framework.

The position where the security aspect is introduced is marked by a point-cut definition (fig.6.7 line 2 and 3). For the point-cut an adviser must be defined. It has to decide whether to allow or reject the method invocation (this is why it is referred to as a *Voter*). To make the decision the authentication manager is called.

```
1  <aop:config>
2    <aop:pointcut id="getUserContext"
3         expression="execution(*com.amadeus.ori.csa.context.ContextManager+.*(..))"/>
4    <aop:advisor advice-ref="methodSecurityInterceptor"
5                 pointcut-ref="getUserContext"/>
6  </aop:config>
7  <bean id="methodSecurityInterceptor"
8        class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
9    <property name="authenticationManager">
10      <ref local="authenticationManager"/>
11   </property>
12   <property name="accessDecisionManager">
13      <bean class="org.acegisecurity.vote.AffirmativeBased">
14          <property name="decisionVoters">
15              <bean class="org.acegisecurity.vote.RoleVoter"/>
16           </property>
17      </bean>
18   </property>
19         …
20 </bean>
```

Figure 6.7: Method Access Protection with Spring AOP and Acegi

6.2 CSA Architecture

6.2.1 Overview

The architecture which will be described is a possible implementation of the concept that was developed in the previous chapter. There will be minor limitations of some complex features which are not required to evaluate the overall concept. A key requirement is to have possibilities for later extensions.

The components are build up on the Spring framework which was described in the previous section. All dependencies are controlled from the outside through configurations. To ensure the flexibility of the components itself, aspects are delegated to interceptors which can easily be added or removed.

6.2.2 Client Connector

The Client Connector component consists of a chain of Interceptors, and a final Endpoint which hands the message to the Message Router. This plug-able architecture allows a fine grain and flexible configuration of the components overall behavior. For instance it would be possible to plug in a logging interceptor for investigation or another interceptor that does a semantic validation.

The Interceptors and the Endpoint implement the corresponding Spring Web Service interfaces and infrastructure. All configurations are done through a separate bean-configuration file.

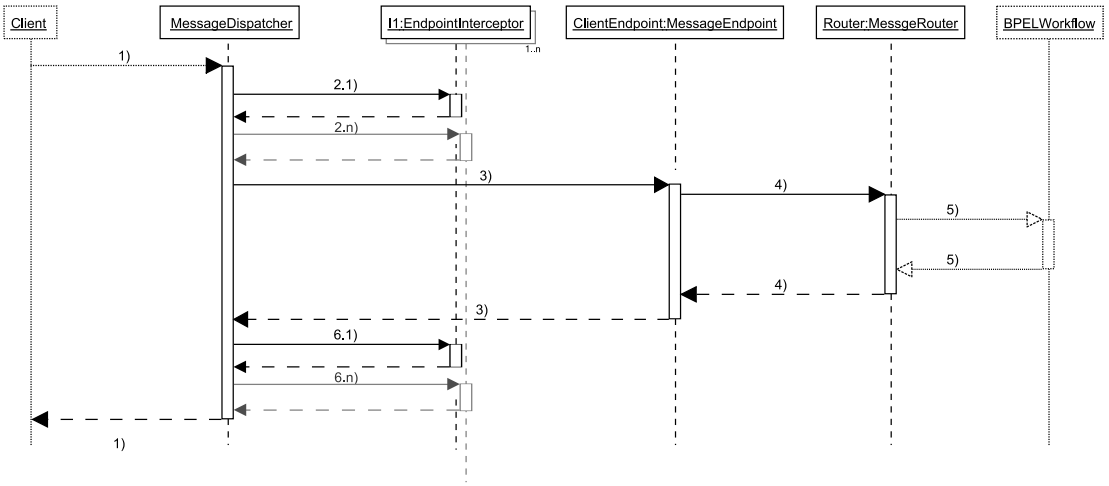


Figure 6.8: CSA Client Connector Workflow

A visualization of a client connector workflow is shown in figure 6.8[5]:

After a client request is received, the `MessageDispatcher` looks for the appropriate endpoint and retrieves a list of interceptors. These interceptors are sequentially invoked before the request is dispatched to the `ClientEndpoint`.

The Client Endpoint hands the request to the `MessageRouter` which analyzes the WS-Addressing attributes to look up the according route in the routing table. Subsequently the request is forwarded to the BPEL Workflow or Service Connector (see 6.2.3.3). The interceptors are invoked another time before the message is returned to the Client.

In the following a more detailed description of important will be given.

6.2.2.1 Client Request Message

The client request message is a standard soap envelope. The payload content will be processed by the WS-BPEL engine. All Information needed by the Client Connector and the Message Router is convoyed in WS-Addressing attributes in the SOAP header (see figure 6.9). It can is divided in three categories:

Originator.. the `<From>` element enables the identification of the individual client device and the application which is sending the request. It will be analyzed by interceptors (see below).

[5]The life-lines *Client* and *BPEL Workflow* are not part of the Client Connector

Destination.. The `<Action>` element uses a URI to define the logical identifier of the requested route within the CSA.

Reference-Parameters.. The complex element `<ReferenceParameters>` contains an element with a unique URI that allows to identify and load the session context which this message is belongs to.

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" . . . >
2  <soapenv:Header>
3    <w:From>
4     <w:Address>urn:labs.amadeus.com:mtw:0.0.1:6e8bc43…a66</w:Address>
5    </w:From>
6    <w:Action>urn:labs.amadeus.com:csa:TripInformation:0.0.1</w:Action>
7    <wsa:ReferenceParameters>
8       <csa:SessionID>870e8400-f25b-41d4-a716-446655184000</csa:SessionID>
9    </wsa:ReferenceParameters>
10 </soapenv:Header>
11 <soapenv:Body>
12       . . .
13 </soapenv:Body>
14 </soapenv:Envelope>
```

Figure 6.9: CSA Client Request Envelop with WS-Addressing Header and Sample Values

6.2.2.2 Logging Interceptor

Spring WS provides a `PayloadLoggingInterceptor` which was extended for the use with the CSA. The goal was not only to log the payload but also the contents of the Web Service Addressing header which contains important information about the client and the message flow.

The `CSAClientLoggingInterceptor` makes use of the *Apache Commons Logging API* and the *log4j* implementation[6] which is used throughout the whole project. Client access is configured to be logged to a separate file. The configuration would also allow a database as log target (*appender*). This can be useful at a later stage when a monitoring layer is added.

6.2.2.3 Validation Interceptor

The predefined `PayloadValidatingInterceptor` allows to validate the incoming and/or outgoing payload against a given XML Schema definition. For the CSA not only the format of the payload but also of the WS-Addressing header is important. This is why the `CSAClientValidationInterceptor` was derived which fulfills this task.

[6] Apache Logging: `http://commons.apache.org/logging/`, `http://logging.apache.org/log4j/`

Since XML Schema validation is presenting a certain overhead, this interceptor is mainly intended to be used during pre-production tests or in case of problem investigation.

6.2.2.4 Endpoint

The endpoint is invoked after the message has passed all interceptors (see also figure 6.8). It hands the entire SOAP message context to the router. The concrete message router implementation is also configured through dependency injection. This way both components are only tightly coupled in the sense that they must both live in a Spring application container. Alternatively the message could be converted to its XML representation to use a platform neutral transportation. In the current implementation this would mean an unnecessary overhead.

6.2.3 Message Routing

Web Services are usually a collection of methods that can be invoked by a client. For example a weather information service could offer one method to get the current temperature for a given location and another one for the average rainfall and so on.

All request messages are sent to a common endpoint which will dispatch them to the responsible processing entity which is identified by a so called action. It can be specified in different ways:

Most common is the usage of an attribute in the HTTP header (called `SOAP-Action`). This has the disadvantage of requiring HTTP as transport protocol.

Another possibility is for the endpoint to inspect the message payload to derive the right action to be performed. The XML root element often allows to make such assumption. This process is quite flexible but has certain limitations especially if different methods share identical requests.

The third option is to specify the action in the header of the SOAP envelop. This way was chosen by the WS-Addressing specification which reserves a dedicated element for this purpose (`<Action>` element).

In the CSA this transport layer agnostic way is used to realize the message routing. The client uses a logical identifier which the Message Router can map to a physical route to dispatch the message to the right composition for further processing. An example is shown in figure 6.9 line 9.

6.2.3.1 The Logical Identifier

An identifier is used by the client to specify the action to be performed upon the request. It is logical because it does not directly point to a physical resource but needs to be resolved by the message router. There are two reasons why the physical route[7] is not directly specified in the client request. On the one hand it ensures a certain measure of security against manipulation and on the other it enables modifications even at run time. In future versions of the CSA this will enable the dynamic change of routes in case of failure or to provide load balancing.

Prefix		Identifier		Version
`urn:labs.amadeus.com:csa`	:	`TripInformationProcess`	:	`0.0.1`

Table 6.1: Structure of the Logical Identifier

The logical identifier is expressed by using the URN/URI syntax and is placed in the WS-Addressing `<Action>`-element of the SOAP header (fig. 6.9 line 9). It consists of three - column separated - segments (see table 6.1):

- A static prefix: `urn:labs.amadeus.com:csa`
- An application identifier written in one capitalized word (e.g. `TripInformationProcess`)
- A version information (e.g. `0.0.1`)

The structure of the identifier enables the hosting and provisioning of several processes in parallel. The version information ensures backward compatibility in a production use.

6.2.3.2 The Routing Table

To resolve logical identifiers a routing table is used. It contains the physical address of the service composition that will process the request. The routing table gives the additional possibility of bypassing the BPEL engine.

In the prototype implementation the routing table is realized through a key - value map. All entries can be configured through the central Spring configuration. An example is shown in figure 6.10.

[7] In this context a *physical route* is a pointer to a concrete service by using the fully qualified address.

```
1 <map>
2  <entry key="urn:labs.amadeus.com:csa:TripInformationProcess:0.0.1">
3    <value>http://localhost/bpel/services/TripInformationService</value>
4  </entry>
5  <entry key="urn:labs.amadeus.com:csa:TripWeatherService:0.0.1">
6    <value>http://localhost/CSA/connector/services/WeatherService</value>
7  </entry>
8 </map>
```

Figure 6.10: Example of a CSA Rooting Configuration

6.2.3.3 BPEL Engine Bypass

Throughout this document, service aggregation was assumed as an omnipresent condition. In practice it can be very well possible that a concrete action can be satisfied by an existing service without any need of modification or conversion. In this case the BPEL workflow would only act as a repeater and cause an unnecessary overhead.

To avoid this it is possible to bypass the composition engine and route the message directly to a Service Connector. For this reason the complete physical route of each action can be individually defined. Figure 6.10 (line 5,6) shows how a weather service request configured to be forwarded to a service connector.

6.2.4 BPEL Engine

The design of CSA had the goal to keep the business process simple and clean from proprietary extensions. All communication with the BPEL engine is done through partner links and standard SOAP messages. As consequence there is no direct dependency on a concrete implementation.

For the prototype implementation the open source BPEL engine from Active Endpoints[8] will be used. It supports BPEL4WS 1.1 as well as WS-BPEL 2.0. It is a Java web application that can be run in the same servlet container as the CSA. Besides the usual setup there is no special customization to be done. The configuration of the business process will be explained in 6.3.

6.2.5 Service Connectors

The basic structure of the Service Connectors is quite similar to the client part. The requests that come from the composition engine (or through the bypass) are dispatched

[8] Active Endpoints: `http://www.active-endpoints.com`

to the `ServiceEndpoint` which can forward them to a intermediate `ServiceAdapter` or directly invoke a target Web Service (figure 6.11).

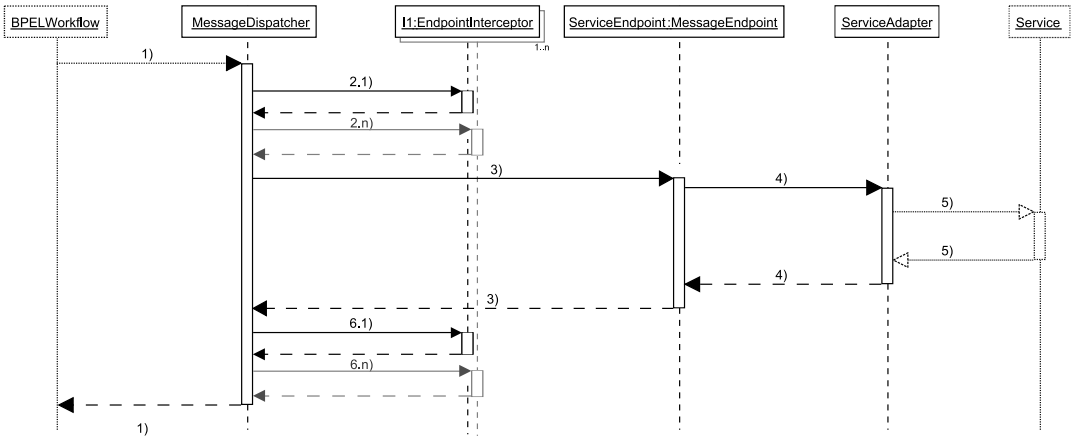


Figure 6.11: CSA Service Connector Workflow

The structure of a individual service connectors can be quite different depending of the nature of the service. Generally three categories can be distinguished:

Type1: Simple Service Connector

The request from the BPEL engine can be directly forwarded to the external Web Service and the reply does not need any special processing. The task of the connector is reduced to handle cross cutting concerns like security or accounting.

Type2: Service Transformation Connector

The Simple Connector is extended by introducing a XSL transformation interceptor on the query and/or reply. This enables to interconnect to Web Services which do not match the strict requirements of the BPEL engine. A trivial example could be missing name-spaces that need to be added in order to not break the business process.

In case of using the BPEL engine bypass a service transformation can be helpful to adapt the contents of a business service to the client application.

Type3: Custom Service Connector

If a transformation can not handle the kind of adaptation needed further customization of the connector has to be done. In this case the `ServiceAdaptor` class shown in figure 6.11 gains importance. Custom Java code can be placed to achieve the special needs.

To illustrate the need of a Custom Connector a simple example can be taken from the Client Application that will be described in the next section: In the workflow static information needed to be retrieved that did only exist in the form of a comma separated text file. A custom connector was written that exposes this data as a web service to the BPEL engine.

Possible other examples could be legacy protocol integration or transformations that can not be handled by XSLT.

6.2.6 Authentication and Security

Client Authentication and Security is based on the Spring Web Service Security integration using a `XwsSecurityInterceptor` (described on page 65). It provides different options for authentication, integrity and message level encryption. This flexibility is not limited by the CSA and it is depends on the concrete use-case which level of security or which authentication mechanism is best.

In the prototype implementation a layered authentication and authorization process was implemented, it is closely related to the session context. The following sequence illustrates how it works (numbers correspond to figure 6.12).

1. If the client does not possess a session identifier, it has to send an authentication request. This request must contain a WS-Security header with a valid Username Token (see figure 4.10).
2. In the Client Connector the `CSASessionValidationInterceptor` will forward the request to the `XwsSecurityInterceptor` which uses the Acegi security framework to validate the user credentials.
3. If the credentials are invalid or missing a SOAP fault is send back to the client.
4. A successful authenticated user will receive a reply containing a session identification parameter in the WS-Addressing header.
5. Subsequently the client can send a functional request. The Session Context Manager will be invoked by the `CSASessionValidationInterceptor` to validate the session key.
6. An invalid or missing identifier will lead to a SOAP fault.
7. If the request is accepted it will be forwarded to the Message Router (by passing the other interceptors)
8. The Message Router dispatches the request if the logical route exists.
9. If the logical route is missing or wrong (e.g. version no longer supported) the request is rejected.

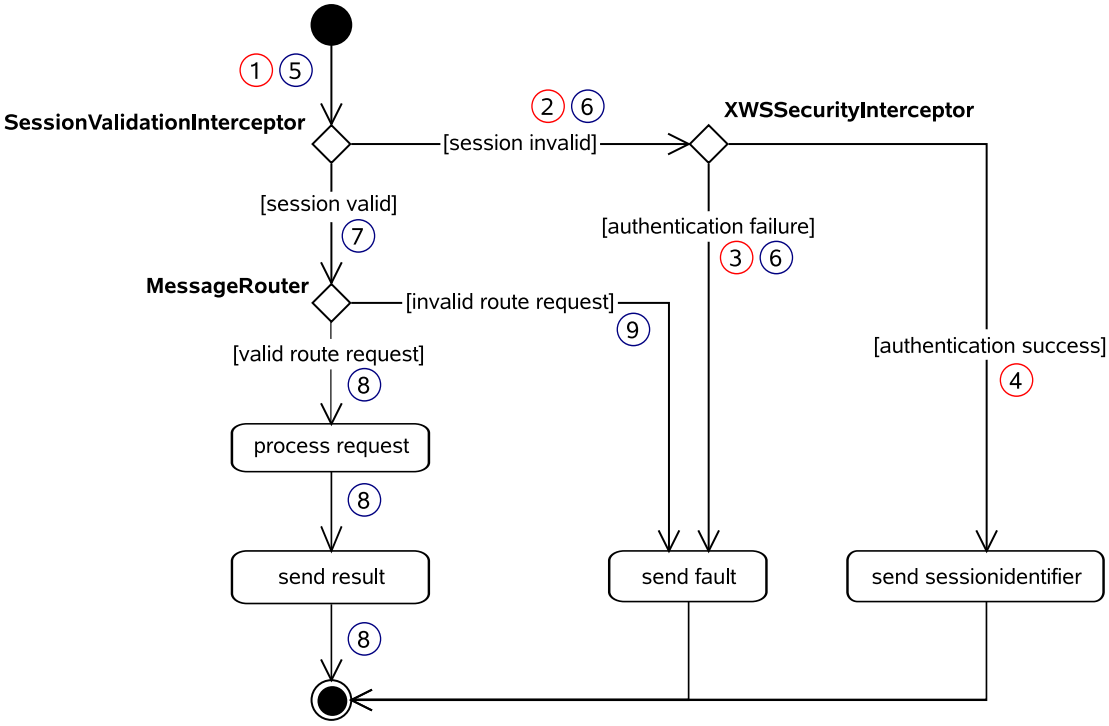


Figure 6.12: CSA Layered Authentication and Authorization Process

Confidentiality is currently ensured by transport layer security because the use-case implementation targets mobile devices with limited resources. Still the security configuration allows enforcing message encryption.

6.2.7 Session Context Manager

The Session Context Manager is a container that holds and manages the context information of a CSA user session. It has three tasks to handle:

- Context initialization (see 5.4.1.1)
- Uphold context information for the lifetime of a user session
- Allow store, retrieve and modification of parameter value pairs in a context

The context data structure is depicted in figure 6.13 it is rather simple. A session object with a unique identifier is referencing parameter objects which encapsulate name value pairs.

For current use-cases it is not necessary to persist session data over a long period of time. The data store is memory resident, every session has a timestamp that is updated

if an access occurs. If a timeout limit is exceeded the session gets removed from the store all data is lost.

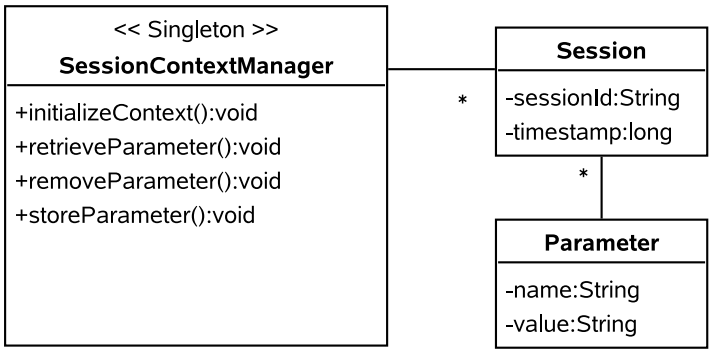


Figure 6.13: Context Manager Interface and Datastructure

6.2.7.1 Context Manager Interface

The Context Manager can be accessed through a native Java interface which is mainly intended to be used by the Java components of the CSA. In parallel there is also a Web Service which can be used by to integrate context interaction in the WS-BPEL composition. In both cases the same methods are offered:

Context Initialization

An invocation of the `InitializeContext` method will create a new session object in the context store. It is assigned a Universally Unique Identifier (UUID) which is returned by the method.

The Client Connector will use this method to start a new session and to obtain the identifier which will be added to the WS-Addressing header of the sign-in reply. This procedure is also done for anonymous users to provide the same functionality.

Parameter Retrieval and Modification

A context parameter is a name value pair (both of type String). It can be written to the session by the `StoreParameter` method which takes the name and value as arguments. If a parameter with the same name exists, it will be overwritten.

The `RetrieveParameter` method can be used to access a parameter value for a given name. `RemoveParameter` will delete an entry.

6.2.7.2 Context Manager Service

The concept section described two ways of accessing context parameters from within a composition (see 5.4.3.2). It was decided to implement the context pull scenario because currently consistency is less crucial and the overhead of copying the context to any composition was seen as drawback.

The Context Manager Service provides a Web Service interface to the operations which were described above (see figure 6.2.7.2). The implementation is similar a custom service connector which does not invoke an external provider but acts as mediator with the Context Manager.

To allocate the right context, the service must know which session the request is related to. This is achieved by copying the WS-Addressing header from the client request message to the request message of the Context Manager Service (see 4.2.1).

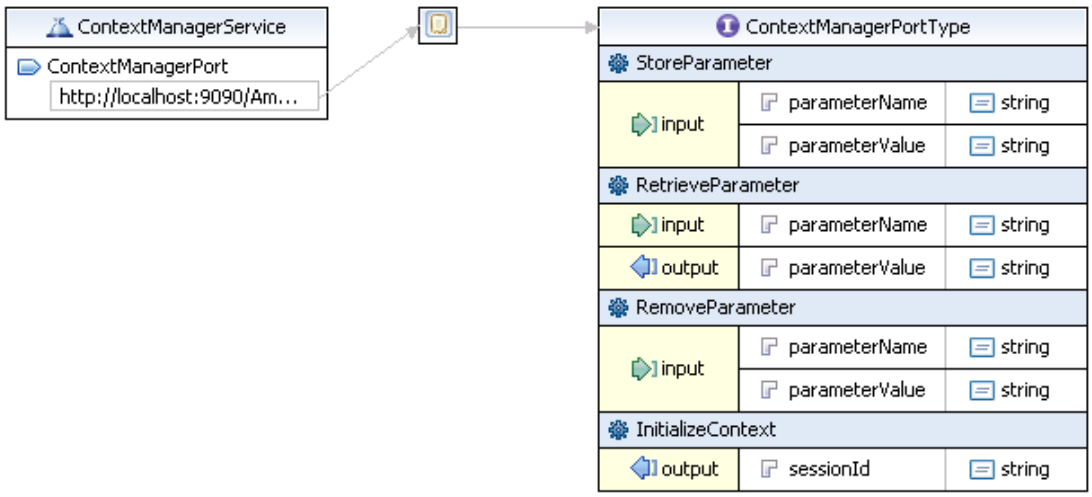


Figure 6.14: Overview of the Context Manager Service

6.3 Use-Case Implementation

This section will give a description of the way the use-case described in chapter 3 was implemented. The focus is clearly on the server side development and on the overall methodology.

6.3.1 Overview

The use-case consists of a client application which can run on mobile devices such as cellular phones or digital personal assistants. It is called the Mobile Travel Wallet because it aims to provide the traveller with all the necessary information of the trip. A functional overview was given in 3.1.

To obtain the information the client needs to connect to the CSA. Based on the booking reference number it can download the complete trip information. They are then stored on the mobile device to avoid the usage of expensive air-time or when no network is available (especially abroad or on a plane). Some information such as flight times or booking status can change on the way. This is why a second functionality was added to update only such crucial data. Finally the traveller can request the weather forecast for a given location.

Figure 6.15 gives an overview of the CSA configuration for the Mobile Travel Wallet (MTW). There are three operations which are exposed to the client which are used to:

- download trip information
- update previously downloaded trip information
- get weather forecast

Two compositions are loaded to extract and process the necessary information from service providers. The weather information is bypassing the composition engine to directly target a service connector.

Four service connectors are used:

- Amadeus Web Service connector to get the passenger record, flight information, hotel and car rental details
- IATA Resolver is used to translate codes in clear text
- The destination information service provides descriptive information about points of interest such as city or country guides
- The Information Content Exchange Server (ICE) provides detailed weather forecast data

6.3.2 Server Side Implementation

The Credential and Service Aggregator configuration consists of:

- service interface specification (WSDL and XSD-Schema)

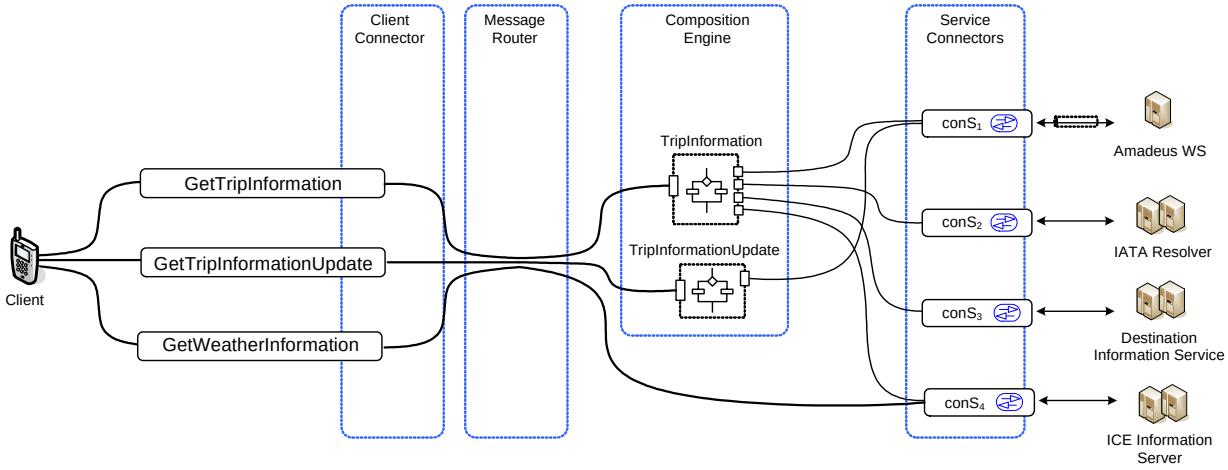


Figure 6.15: CSA Configuration Overview for Mobile Travel Wallet

- business process(es) (WS-BPEL definition, XSLT-style sheets, engine configuration)
- service connectors
- routing table
- security and credential definition
- session management configuration

These steps well be explained in detail below.

6.3.2.1 The Service Contract

The server side implementation was done by applying the *contract first* approach. It means that the service interface was specified first and then the configurations and workflows have been designed to fulfill the contract. This ensures a client centric design but also facilitates the server side development.

Operations

The service contract for the Mobile Travel Wallet consists of a WSDL definition which contains three operations (table 6.2):

`GetTripInformation` is the main operation which returns a collection of various information related to a given trip (itinerary details, weather, destination information etc.). In the request message the passengers booking identifier has to be provided.

Operation	Messages
`GetTripInformation`	`TripInformationRequest`
	`TripInformationReply`
`GetTripInformationUpdate`	`TripInformationUpdateRequest`
	`TripInformationUpdateReply`
`GetWeatherInformation`	`WeatherInformationRequest`
	`WeatherInformationReply`

Table 6.2: CSA Operations for the Mobile Travel Wallet

`GetTripInformationUpdate` is an operation that allows to update crucial parts of the trip information. For instance the times or status of segments but not static data like the address of a hotel. The update is a functionality required by the mobile client to quickly refresh existing data without requiring the whole data.

`GetWeatherInformation` enables updating the weather for a given location. It can be related to a trip or used to quickly get a weather forecast for an arbitrary place. This operation will be used to validate the composition engine bypass (see 6.2.3.3).

Message Schema

Each operation consists of a request and response message. The grammar of these messages has to be specified by an XML schema. The structure and content is designed to fulfill the requirements of the client. There is no unnecessary information and on client side no preprocessing or data-type conversion has to be done. This is an important advantage especially when mobile devices are targeted.

Figure 6.16 depicts an overview of the grammar of a `TripInformationReply`.

6.3.2.2 Message Routes

For each operation of the service contract a message route has to be defined. The logical identifiers are derived from the operation name and a version number is assigned (see 6.2.3.1). The following values have been added to the service contract and the routing table (WSDL WS-Addressing section):

- `urn:labs.amadeus.com:csa:TripInformationProcess:0.0.1`
- `urn:labs.amadeus.com:csa:TripInformationUpdateProcess:0.0.1`
- `urn:labs.amadeus.com:csa:TripWeatherService:0.0.1`

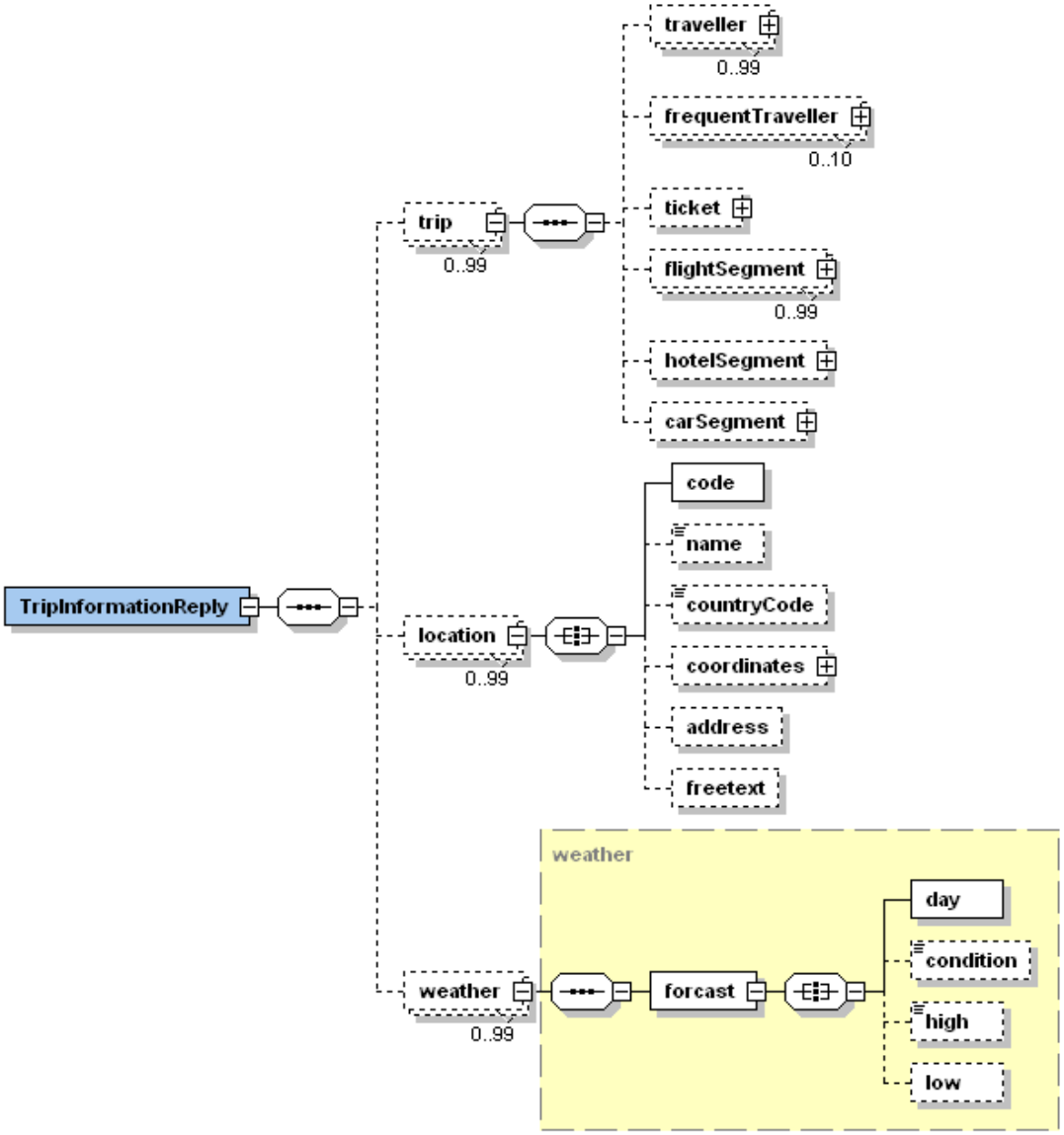


Figure 6.16: TripInformationReply Message for the Mobile Travel Wallet

For each logical identifier a physical route has to be added to the routing table (see 6.2.3.2). It can either point to the partner link of a composition or to a Service Connector (BPEL Engine Bypass).

The trip information and update request can only be fulfilled by a BPEL composition and the according endpoint address has to be specified. An example could look like this:

`http://localhost/active-bpel/services/TripInformationServicePartnerLinkService`

The weather information can be obtained directly from a third party service (see 6.3.2.4). This means that no service composition has to be defined nor executed and the message can be directly routed to the responsible service connector. So the URL of the Service Connector is saved in the routing table:

`http://localhost/AmadeusCSA/connector/services/ice`

An extract of the routing table was shown in figure 6.2.3.2.

6.3.2.3 BPEL Workflows

The message route analysis has shown that two service compositions are needed:

- `TripInformationProcess`
- `TripInformationUpdateProcess`

The implementation of this workflow was done in 4 steps which will be explained below. The `TripInformationProcess` is taken as example:

Refined Specification

During the specification phase of the Mobile Travel Wallet application, UML diagrams had been used to analyze the workflows (see 3.1). It was important to define at an early stage which service providers and which type of information is needed.

For the BPEL workflow implementation these diagrams are a valuable ground to start from. A detailed activity diagram can for instance be used as template for the implementation in WS-BPEL. Figure 6.17 depicts such a refined activity diagram of the `TripInformationProcess` (see also figure 3.3).

Partner Link and Variable Definitions

Before being able to implement the actual workflow all necessary assets need to be defined in the WS-BPEL file.

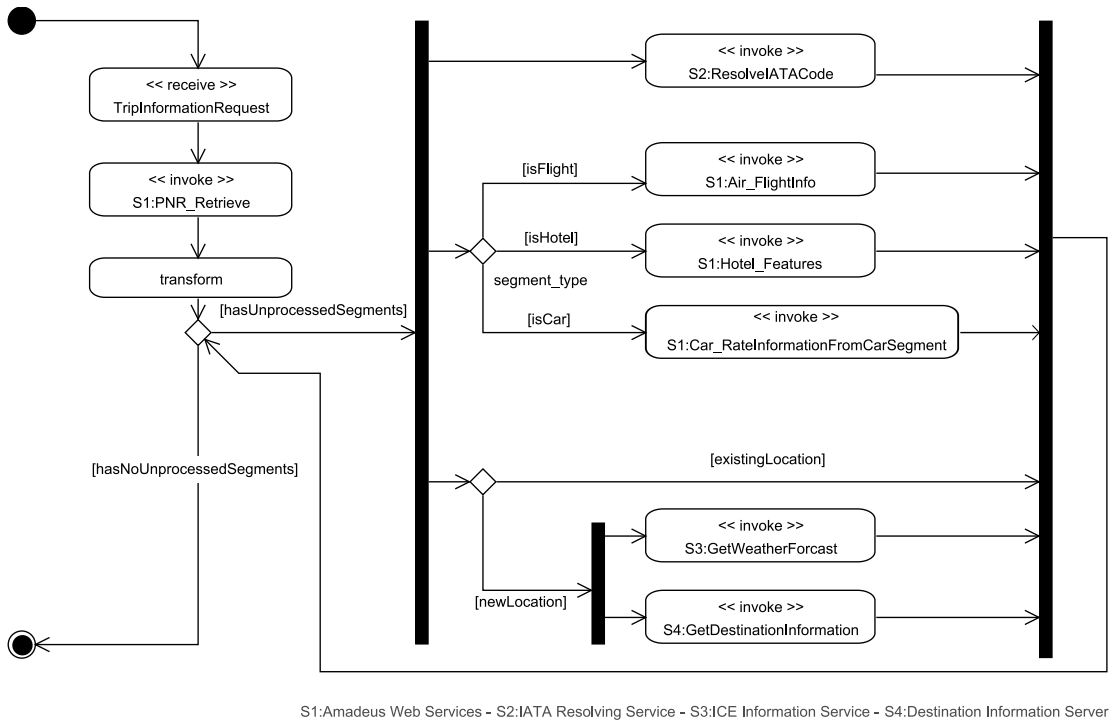


Figure 6.17: Refined Activity Diagram of a Trip Information Process

There are two mandatory global variables to be added. They correspond to the request and the response of the process.

In addition all partner-links have to be defined. There are two types, one for the workflow itself and one for each service that is invoked to complete the request. The partner link can be compared to a import statement in a traditional programming language. Three things have to be done[9]:

- a `partnerLinkType` element within the WSDL or in a wrapper if it is not locally available (see figure 6.18)
- an `import` element that references the wrapped WSDL specification
- a `partnerLink` element which makes the partner link type accessible from the composition

```
1  <plnk:partnerLinkType name="TripInformationService">
2     <plnk:role name="TripInformationServicePortTypeRole"
3                portType="tns:TripInformationServicePortType"/>
4  </plnk:partnerLinkType>
```

Figure 6.18: WSDL Partner Link Type Definition

[9] see also on page 27

Integrated Developer Environments for WS-BPEL usually generate partner-link and variable definitions automatically.

BPEL Workflow Implementation

The final implementation is basically a translation of the Activity Diagram like the one shown in figure 6.17:

Through the TripInformationProcessPartnerLink the requested booking reference is assigned to a service invocation which returns the complete booking record. This record is transformed in the reply format (figure 6.16). The design of the XML style sheet is part of the implementation work.

In the following a control element is introduced which iterates over all segments of the reply. Then in parallel different services are invoked to complement the information, to transform codes in human language and to add weather forecast and destination guides.

Workflow Refinement

The process of workflow refinement is ongoing beyond the boundaries of the actual implementation. Especially during load tests performance leaks can be attacked by parallelizing steps or improving transformations (see also 3.1).

Error handling and recovery is also from high importance. For example the missing or invalid reply from the weather service should not lead to a termination of the overall process.

6.3.2.4 Services and Connectors

There are 4 service providers that will be invoked by the Web Service composition. The integration of each of them contains specific challenges. This demonstrates the richness of possibilities provided by the service connector approach.

Amadeus Web Services

All booking information related to a traveller is stored in a Passenger Name Record (PNR) which is a standardized format in the global travel industry. The Amadeus Web Service provides the possibility to retrieve this information (`PNR_Retrieve` operation).

A PNR contains not only the name of the traveller(s) and itinerary information but also background data related to the handling (office identification, internal codes and other confidential data). An average PNR with two flight segments a hotel booking and a car rental has about 500 lines. A transformation in the BPEL process will filter the PNR and extract only the information that are needed for the Mobile Travel Wallet which is about 5% of the original data.

Figure 6.19 shows a small extract as an example. It contains one traveller and a flight segment with Air France from Nice to Paris Charles de Gaule.

```
1   <travellerInformation>
2         <traveller><surname>FOSTER</surname></traveller>
3         <passenger><firstName>BOB</firstName></passenger>
4   </travellerInformation>
5   <originDestinationDetails>
6       <itineraryInfo>
7           <segmentName>AIR</segmentName>
8           <product>
9               <depDate>280108</depDate><depTime>0635</depTime>
10              <arrDate>280108</arrDate><arrTime>0810</arrTime>
11          </product>
12          <boardpointDetail>
13              <cityCode>NCE</cityCode>
14          </boardpointDetail>
15          <offpointDetail>
16              <cityCode>CDG</cityCode>
17          </offpointDetail>
18          <companyDetail>
19              <identification>AF</identification>
20          </companyDetail>
```

Figure 6.19: Extract of a structured Amadeus Passenger Name Record (PNR)

For each area of travel industry (air, car, hotel, cruise etc.) there are dedicated operations. Some of them are used to complement the passenger record with additional information:

`Air_FlightInfo` contains up-to-date data related to the operation of a flight before, during and after departure.

`Car_RateInformationFromCarSegment` allows to access information linked to a Car Booking stored in a PNR (e.g. phone number and address of pickup and drop-off location).

`Hotel_Features` provides details about a hotel such as address, telephone number, policies and services.

A custom connector (type 3) was needed to implement the complex and non standard session handling procedure. It does the authentication and stores the service session token in the client profile (see 5.4.1.3).

ICE Web Service

The Information Content Exchange server ICE[3](ice cube) is a Web Service API to expose general travel information such as the weather, currency conversion or airport information.

The service was build using a generated Web Service stack which wraps the XML payload in a CDATA element (see figure 6.20 line 2-6). This has the consequence that it is practically impossible to access the data through standard XML mechanism such as XQuery or XPath. Such a message can not correctly be consumed by a WS-BPEL workflow.

```
1 <m:WeatherConditionsResponse xmlns:m="Weather">
2    <result>&lt;?xml version="1.0" encoding="ISO-8859-1"
3 ?&gt;&lt;WeatherRequestCode="CDG"&gt;&lt;CurrentConditions LastUpdate="2008-01-
4 031:30"&gt;&lt;Temperature&gt;3&lt;/Temperature&gt;&lt;Humidity&gt;93
5 &lt;/Humidity&gt;&lt;HeatIndex&gt;3&lt;/HeatIndex&gt;&lt;WindChill&gt;1&lt;/WindChi
6 ll&gt;&lt;DewPoint&gt;2&lt;/DewPoint&gt;&lt;WindSpeed&gt;7&lt;/WindSpeed&gt;
7      ...
8      </result>
9 </m:WeatherConditionsResponse>
```

Figure 6.20: Extract of Web Service Response with Escaped Payload

A connector with transformation capabilities (type 2) was written to resolve this issue. It replaces the escape characters and removes the invalid XML header declaration (fig. 6.20 line 2). The result can then be accessed by the BPEL engine.

The connector also handles the HTTP authentication required by the service. The credentials are taken from the credential manager.

IATA Code Resolver

The International Air Transport Association (IATA) has assigned unique codes to enable the exact identification of entities like air ports or aircraft types. The following three examples show that these codes are not always obvious for travellers:

- the equipment code `E70` stands for a *Embraer 170* aircraft
- the airport code `PSY` identifies the *Port Stanley Airport* on the Falkland Islands
- the company code `SU` is a synonym for the Russian Airline *Aeroflot*

As shown in figure 6.19 such codes are used throughout the passenger record. They need to be translated before being displayed by the user. The fact that there are about

11.000 airport codes[10] shows that this can not be done on a mobile client. Instead the Web Service composition will use a custom service connector to solve this task.

This connector actually does not invoke an outside Web Service but retrieves the data from a local data-store. It is a good example how connectors can be used to integrate other services and information into a BPEL process.

Destination Guide

The Destination Guide service will deliver point of interest (POI) descriptions. A given location from the booking record is taken as request. The reply will contain categories such as *Restaurant* or *Theater* with sub items containing short descriptions and address information.

For logistical reasons the destination guide could not be integrated before the finalization of this document. This does not limit the validation capabilities of the prototype implementation as the other service connectors demonstrate all important adaption capabilities. Still it is mentioned to complement the functional content description.

6.3.2.5 Security Aspects

Originally the Mobile Travel Wallet required users to sign up on a Web Site and then to authenticate with the credentials that had been provided. During usebility tests it was decided to postpone this feature for a later version and to allow anonymous access.

The scenario that was described in 6.2.6 was therefore slightly modified. The authentication call still has to be done but with an empty WS-Security header. The `XwsSecurityInterceptor` has been disabled and a session is immediately established.

6.3.2.6 Context and Adaptation

The Mobile Travel Wallet will support multiple languages and not only the user interface elements but also the contents should adapt accordingly. Translation tables would take to much space on the client side so the composition must include such personalization features. This is possible because the services which are invoked offer the possibility to set a language code.

Semantically the language if part of the individual user context. It will therefore be used to validate the concept of the Context Manager.

[10] the number of codes multiplies by the number of supported languages

The language is determined on client side. It can be detected automatically from the operation environment or specified by the user in the configuration menu. To do correct internationalization language and country code are needed, they must comply with the according ISO standards[11].

In priciple the context adaption works as in the following example:

> Before the client can send the first request it has to do the authentication. If it is an anonymous user the authentication message is empty but still contains a parameter section with key-value pairs:

```
1     <csa:ContextParameter>
2             <parameterName>LOCALE</parameterName>
3             <parameterValue>EN_US</parameterValue>
4     </csa: ContextParameter>
5     <csa:ContextParameter>
6           <parameterName>PLATFORM</parameterName>
7           <parameterValue>RIM</parameterValue>
8     </csa: ContextParameter>
```

> The client connector will call the initialize method of the Context Manager to start a new session.
>
> Subsequently it will store all client parameters in the session context and finally send a reply which contains the session identifier in the WS-Addressing header.
>
> Later in the session the client will send a functional reply which is forwarded to the composition engine. Within the composition a weather service will be invoked.
>
> This service requires to specify a location and the code of the language which the result will be translated in. In order to obtain the language code the Composition has to invoke the Context Manager Service first:

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" >
2  <soapenv:Header>
3     . . .
4    <wsa:ReferenceParameters>
5        <csa:SessionID>870e8400-f25b-41d4-a716-446655184000</csa:SessionID>
6    </wsa:ReferenceParameters>
7  </soapenv:Header>
8   <soapenv:Body>
9     <con:RetrieveParameter xmlns:con="http://labs.amadeus.com/csa/context/ContextManager">
10            <parameterName>LOCALE</parameterName>
11    </con:RetrieveParameter>
12   </soapenv:Body>
13 </soapenv:Envelope>
```

> The Manger will get the language definition from the context store and send it back to the composition engine:

[11] Language codes specified by ISO 639 and country code by ISO 3166.

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" >
2  <soapenv:Header>
3     . . .
4  </soapenv:Header>
5   <soapenv:Body>
6     <con:RetrieveParameter xmlns:con="http://labs.amadeus.com/csa/context/ContextManager">
7         <parameterValue>EN_US</parameterValue>
8     </con:RetrieveParameter>
9   </soapenv:Body>
10 </soapenv:Envelope>
```

Now the code can be copied to the Weather request to invoke the weather service. The result will be customized in the desired language code.

This example has shown how the client context can be successfully integrated into a composition. The Service Connectors can use the context in a similar way. In the use-case this was validated by the Amadeus Service Connector. It establishes a stateful conversation and needs to save the service session identifier in the user context for subsequent requests. This scenario was already explained in 5.4.1.3.

6.3.3 Client Application

The Mobile Travel Wallet is a client application that can be executed on mobile devices such as cellular phones or personal digital assistants which support the Java 2 Mobile Edition (J2ME).

Mobile application development is still a special challenge because of the limited capabilities of the devices and the heterogeneous infrastructure which they operate in. More than others they depend on a suitable server side interface. In the following will be shown how the CSA can facilitate the integration of such fragile platforms.

6.3.3.1 Functional Overview

The client application consists of:

- the *Graphical User Interface* and navigation structure
- a *Data Store* that is holding the domain objects and provides persistence
- the *CSA Client Proxy* with the capabilities to marshal and unmarshal the domain objects

The functional contents follow the specification of the use case given in chapter 3. The navigation structure is depicted in figure 6.21. After launching the user can authenticate or directly access to the main menu as anonymous guest. The central function is of course the display of trip information. The user navigates to a list of previously stored trips.

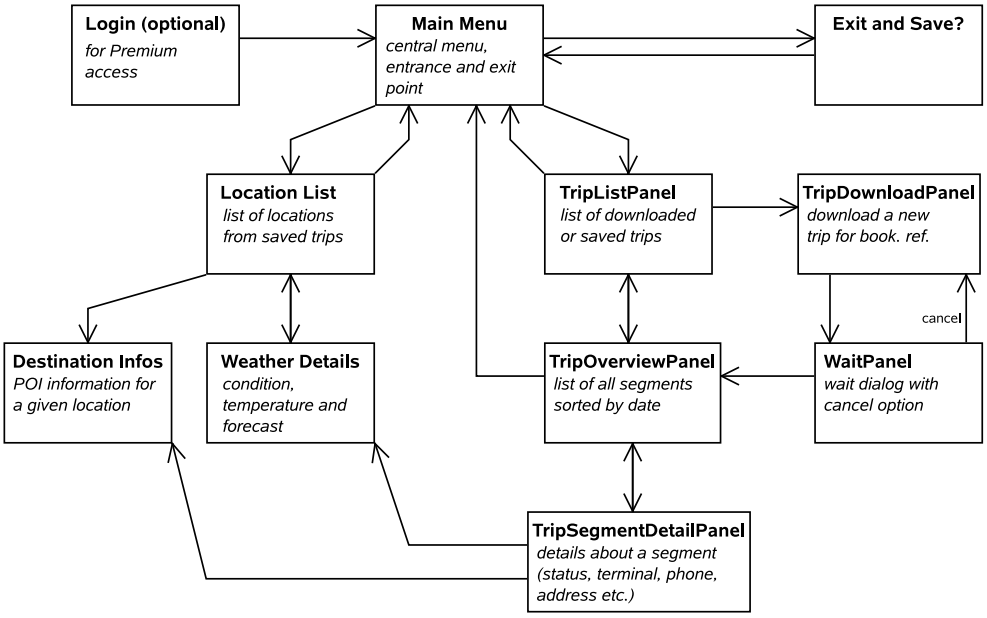


Figure 6.21: Mobile Travel Wallet - Functional Overview

To add a new trip a form needs to be filled. For security reasons not only the booking reference but also the name of the traveller is required. After sending the form a connection to the CSA is done to retrieve all information. The will be unmarshalled to the domain objects and can then be displayed.

Downloaded or saved trips can be displayed in two levels. A overview of all segments which is ordered by date and contains flight departure and arrivals, hotel check-in and check-outs as well as car rental pick-up and drop-off times. For each segment details can be displayed.

From the main menu there is also the option to directly access weather information or destination guides for downloaded trips. On exit the user can choose to persist the data to the J2ME record store.

6.3.3.2 Domain Model

The domain model of the client is rather simple, it contains the necessary data structure for the trip and the associated locations with weather and destination information. The central classes and their relations are depicted in figure 6.22.

The important fact about the domain model is, that it reflects the structure of the messages of the service contract (see 6.3.2.1). The reason is not that the client was adapted to the service interface but the other way around. This leads to a very slim client in-

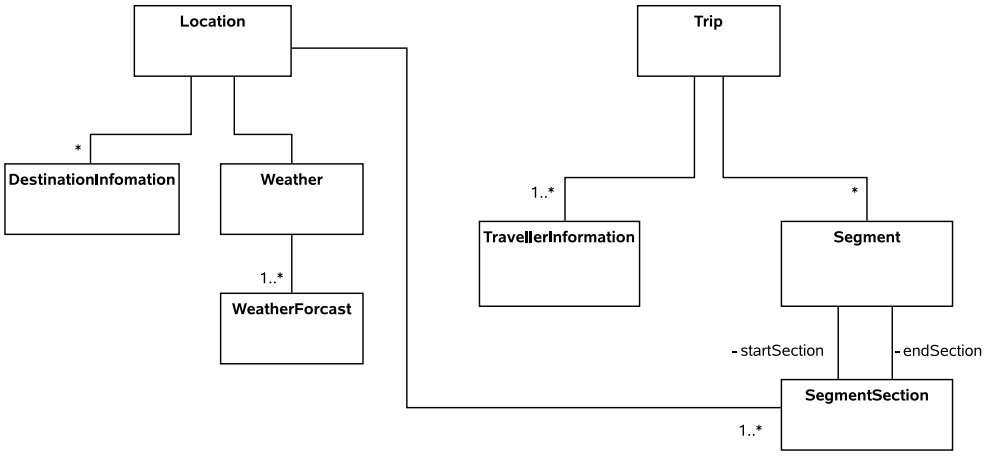


Figure 6.22: Travel Wallet Domain Model

tegration layer which is especially important on a limited platform such as a mobile device.

To have an effective domain model integration also reduces the work of migrating the client application to different run-time environments which is very likely in the area of mobile computing.

6.3.3.3 CSA Client Proxy

Turning a J2ME device into a Web Service client is still a challenge. In difference to a full fledged Java environment there is a much smaller selection of available frameworks to facilitate this task.

The JSR 172[12] specifies a Web Service API for J2ME as well as for handling XML marshalling and unmarshalling. Unfortunately this API is currently only available on a limited amount of devices. As a consequence the client proxy for the Mobile Travel Wallet was manually implemented to target a larger number of users. The proxy layer components are depicted in figure 6.23.

The CSA client proxy interface offers the corresponding methods of the server side contract. There are three implementations which can be initialized through the `CSAProxyFactory`. The HTTP proxy is the main implementation for production use. `DummyProxy` is for client side testing only, it emulates the server reply.

Finally a proxy was implemented which enables to communicate with the CSA through a bluetooth communication (`CSABtProxy`). The reason was to evaluate alternatives for places like airports or train stations to provide an inexpensive, kiosk-like connection.

[12]J2ME Web Service Specification (JSR-172): `http://jcp.org/en/jsr/detail?id=172`

Furthermore it was proved that the exclusive use of message layer specifications decisions was a right decision.

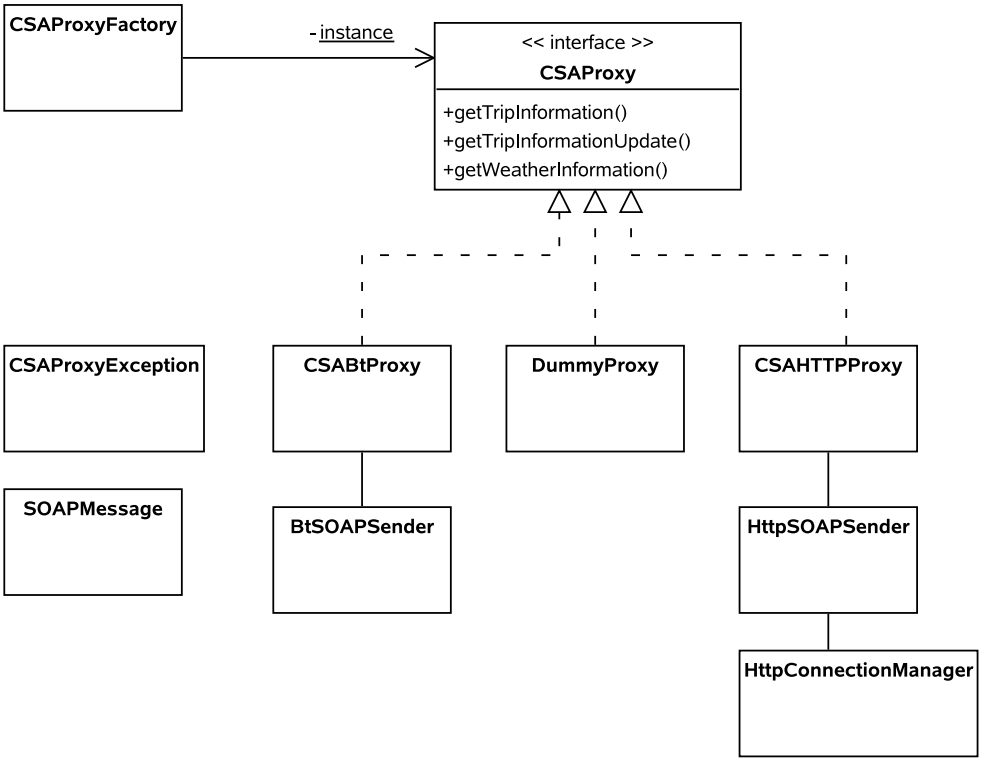


Figure 6.23: Client Proxy for Mobile Travel Wallet

The SOAPMessage class wraps the payload into a SOAP envelop and it provides methods to get and set the Web Service Addressing fields in the header. For performance reasons transport layer compression was implemented in the `HttpConnectionManager`.

6.3.3.4 Persistence

One of the key advantages of the Mobile Travel Wallet compared to a browser based solution is the fact that the extensive network traffic can be minimized and that the stored data can be accessed even if no connection is available (e.g. on an airplane or train). To achieve this, the data has to be persisted.

The J2ME record management system (RMS) provides a convenient way to store data to the device. It provides very basic functionality by mainly offering a persistent key value map. For this reason a persistence framework is used as abstraction layer to store the domain objects[13]. After annotating the persistable classes they can be saved or retrieved from the record store. The framework code is weaved into the client at compile time.

[13] *Floggy* persistance framework: `http://floggy.sourceforge.net/`

6.3.3.5 User Interface

The introduction of the client application will be finished with a quick look on the graphical user interface. A lightweight GUI framework for J2ME was used to facilitate the development[14].

A central interface description can be used to customize the look and feel of the display. This would enable the quick customization or branding of the Mobile Travel Wallet for a potential business partner.

Figure 6.24 depicts the main menu screen (top) a trip overview display (middle) and the details of a trip section (bottom).

6.3.4 Tests

6.3.4.1 Unit- and Stress Tests

A great advantage of the implemented component architecture is that the different units can be tested and debugged individually. This allows to locate bugs and performance issues throughout the whole development cycle.

A typical unit test would send a request to a component and then validate the result against given criteria. It can be easily replayed to ensure that code-changes did not affect the application behavior.

For the use-case implementation the following components have been tested with the JUnit framework:

- Service connector tests: `AmadeusConnectorTest`, `ICEConnectorTest`, `IATAConnectorTest`
- Test of the composition engine which is bypassing the client connector (`BPELEngineTest`)
- Test of the complete configuration including client connector, BPEL engine and service connectors (`MobileTravelWalletTest`). For each loaded composition a test case was written

Stress tests where done to validate the system behavior under real life condition. They have been based on the unit tests and where executed in sequence and in parallel.

6.3.4.2 Performance

A response time evaluation was done by injecting requests to the CSA and the individual components. The results are summarized in table 6.3.

[14]Fire themeable J2ME GUI framework: `http://sourceforge.net/projects/fire-j2me`

Main Menu

Overview of a Trip

Details of a Segment

Figure 6.24: Mobile Travel Wallet Display Samples

The focus of the test was mainly on the time relations between the service invocation and the overall composition processing. Roughly the setup is described by the following parameters:

- Client: Java HTML injector, average PC (Pentium 4, 2 GByte RAM, Windows XP)
- Server: CSA deployed to Tomcat 6 container running on average PC (Pentium 4, 2 GByte RAM, Linux Fedora Core 7)
- Network: Local Area Network 100MBit (~ ping 0.5ms)
- Cache: CSA caching and authentication abilities disabled

In the first step each connector was tested by injecting 1000 static queries. The IATA Code Resolver performed best (~15 ms) which is explained by the fact that it operates on a local data store. The Amadeus Connector which was charged with retrieving a regular PNR needed about 550ms. A weather forecast for a given location needed 859ms.

Figure 6.25 shows the response time for the three service invocations along with their distribution[15].

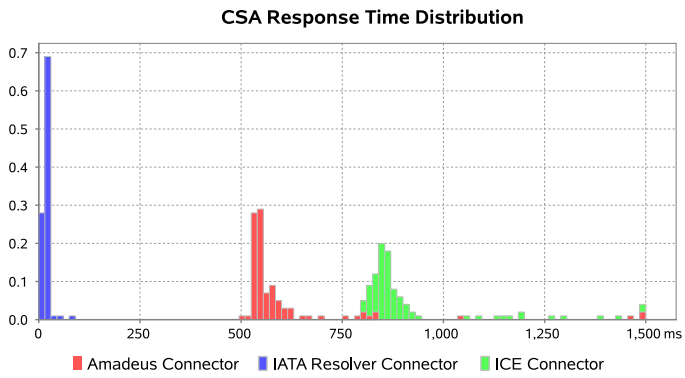


Figure 6.25: Service Connector Response Time Distribution

To measure the overall performance a trip information request was injected. A single passenger record was used which contained one traveller and four flight segments. The average processing time including the network traffic was 4.41 seconds (median).

Adding up the values for the single steps in the composition (1xAmadeus PNR, 4xWeather, 8xIATA) and subtracting their invocation time from the total leads to an estimate overhead of about 300 milliseconds.

Even if these estimations are not to be taken representative, they allow an important conclusions:

[15] the maximum outliers have been removed from the illustration

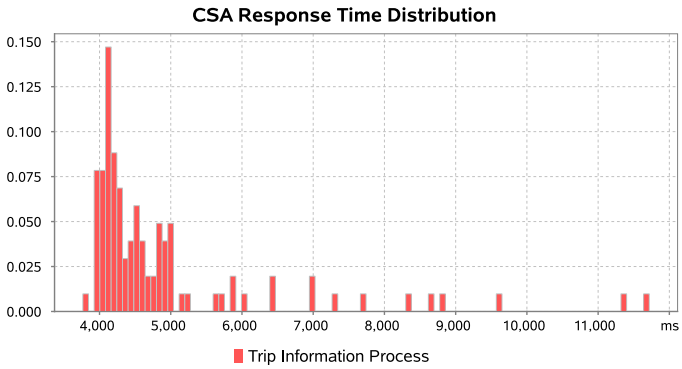


Figure 6.26: CSA Response Time Distribution

A service composition can not perform better then the sum of its components. Figure 6.26 depicts the distribution of the results and that in the worst case the processing time can be significantly larger. So optimization measures like the caching option in the CSA connectors have to be put in place.

Future work should also focus on quality of service enforcement. In the example of the trip information process a badly performing weather service can strongly impact the response time even if the importance of this data is less crucial then for instance the retrieval of the PNR. Currently such conditions can only indirectly be defined (timers).

	Median	Mean	Minimum	Maximum	Observations
Amadeus Connector	547.0	620.47	500	2672	1000
IATA Connector	15.0	12.34	0	78	1000
ICE Connector	859.0	944.70	797	3172	1000
Trip Information Process	4406.5	4916.60	3766	11718	1000

Table 6.3: Performance Test Results (time in ms)

6.4 Summary

In this chapter was shown how the concept of a Credential and Service Aggregator can be realized. The use of existing frameworks such as Spring Web Services has imposed a flexible and powerful architecture. All components are customized through configuration files and can therefore be adapted to extend this solution for different types of application.

The use-case has shown that the CSA is capable to successfully integrate client application into a business service infrastructure without the need of compromising service credentials. It also confirmed a significant reduction of the client side implementation

effort. First tests have proved that the solution is meeting the performance and quality criteria for production use.

In the next chapter a conclusion will be given along with a list of future extensions to the overall architecture.

7 Conclusions and Outlook

In this document a solution was proposed that supports client application integration into a business service infrastructure. This chapter explains how and to which extend the requirements have been reached and what improvements can be done in future work.

7.1 Overview

In general it can be said that the overall goal was achieved. The use-case confirmed that the client implementation was significantly facilitated. Adapting the server side components to the use-case was limited to the implementation of service connectors and configuration files which underlines the generic nature of this approach. The key requirements were satisfied.

In addition the CSA could be used as an integration layer on top of the services offered by a company which includes legacy systems. Customized services could be offered to business customers.

Currently the handling of the CSA requires in-depth knowledge also about the underlying frameworks and Web Services in general. This effort can be justified if it is expected that multiple application or a family of similar products will be developed.

Especially client application which are highly responsive are not always suitable for this solution. They would be negatively impacted by the processing times of a composition. In such a case it is better to handle the data processing directly and for instance invoke services in the background.

7.2 Requirement Assessment

7.2.1 Service Composition Requirements

It was shown that the WS-BPEL language and available run-time environments are able to fulfill all requirements to realize the composition of multiple Web Services.

The services can be invoked in parallel or in sequence. Constructs like conditions or loops allow controlling the execution. All elements of the process algebra that was defined in 3.2.2 can be realized.

Interface Adaptation and Data-type Conversion is ensured by the integration of the XPath- and the XML document transformation language XSLT. Even if general purpose programming languages are more powerful and allow more complex operations there is no limitations to the use cases which this document is targeting. In addition most BPEL run-time environments allow using Java Script for such purposes.

7.2.2 Security and Accounting

7.2.2.1 Authentication and Authorization

The application of the WS-Security specification in combination with supportive frameworks ensures a wide range of authentication mechanism and the integration of heterogeneous credential sources.

Authorization was implemented and validated at the level of the client connector. The routing mechanism has shown how second level authorization can be realized. In the concrete example the version information from the logical route request can be used to deny the access for a certain group of clients (e.g. if out of date).

7.2.2.2 Service Authentication

Services which are needed throughout a composition often require authentication. In the proposed solution the three most common authentication types were implemented (HTTP Basic, session with sign-in request, certificates). Furthermore the concept of service connectors provides room for the integration of other authentication mechanism.

7.2.2.3 Accounting

In the prototype every request by a client and every invocation of an external service is registered in a protocol file. It also includes the unique client identifier so that it is possible to backtrack and account the usage down to the level of individual customers.

The protocol also enables to validate the fees claimed by external service providers and to compute usage statistics. In a future extension tools to analyze and automate these accounting works will be developed.

7.2.3 Context Management

7.2.3.1 Stateful Service Access

A key requirement was the ability to integrate stateful service access into a service composition with the possibility to run transactions even over a sequence of executed compositions. To solve this task the Session Context Manager in combination with a stateful client connection was introduced.

In the use-case implementation this concept was used to uphold a stateful connection between a client session and the Amadeus Web Service by storing the service session identifier in the client context and reusing it for subsequent requests.

7.2.3.2 Adaptation of Composition Processes

To be able to adapt a composition to an individual user it has to be possible to access context information within a composition. The WS-BPEL language has no native construct to do this.

Two possible solutions have been discussed in the concept chapter (see 5.4.3). For the prototype validation the context-pull mechanism was selected to introduce internationalization into the use-case workflow. It allows providing the content in the language of the client which invokes the composition.

7.2.4 Non-Functional Requirements

7.2.4.1 Tool Support

Different composition technologies were evaluated in 4.1 and WS-BPEL was also selected because of the availability of a wide range of tools. During the use-case implementation these tools proofed as very valuable to quickly implement, test and deploy workflows. Due to the high popularity of the language it can be expected to have even better support in the future.

7.2.4.2 Availability of Composition Engines

The same that was said about the tools is also valid for the availability of WS-BPEL composition engines. During non representative tests it was found that the level of maturity is rather heterogeneous. Furthermore the future trend is hard estimate, some solutions might disappear and new ones will come up. For this reason a lot of attention was paid

to develop a concept that is agnostic to the concrete WS-BPEL run-time environment and the use of proprietary extensions was strictly avoided.

For the prototype and use-case validation the Active Endpoint implementation was used. The Apache Orchestration Director Engine (ODE) was tested as backup system.

7.2.4.3 Robustness and Quality of Service

The component based approach allows ensuring the robustness and service quality of the overall solution. Caching for example was used in the service connectors (IATA code resolver and the ICE Weather Service) to significantly improve the response time.

Stress tests with real-time and simulated service replies have allowed to validate the performance. Other tests have also confirmed that the failure of non crucial service invocations (e.g. weather) could be recovered by the service connectors to enable a partial completion of the composition.

7.3 Future Improvements

The proposed architecture was designed with a special emphasize on extensibility. In the following some possible advancements are listed:

7.3.1 Service Composition

WS-BPEL as well as available its run-time environments supports asynchronous composition workflows. Currently this has not been implemented in the overall architecture. It would be very interesting to combine this with different transport mechanism.

To explain this idea the use-case of the Mobile Travel Wallet could include a rebooking feature. Rebooking involves different business partners like a travel agent, air line system, hotel property management systems etc. Depending on the nature of the trip and the number of segments it can take from several minutes up to hours. In this case the client should just trigger the request and not wait for the response.

If the rebooking workflow is completed (or failed) the Credential and Service Aggregator could push the result to the client by using for instance SMS to notify the mobile device.

7.3.2 Security and Accounting

The implemented use-case was targeting mobile devices with limited resources. For this reason full fledged security mechanism especially message layer encryption was not used. If a client solution involves sensible operations and data like credit card numbers such an extension would make sense.

Accounting is currently only supported on a basic level by tracking client and service access. A valuable improvement would be a visual tool to analyze the access logs and to integrate automatic accounting solutions.

To avoid fraud attacks a monitoring system could be put in place that enables automatic notification or denial of service in case of abused user credentials.

7.3.3 Context Management

The abilities to define adaptations and to access the context have been provided by this work. Still the idea of combining context information with service composition offers far more possibilities than the internationalization that was realized in the use case.

An interesting improvement would also be the automatic gathering of context information (e.g. from the user agent) and a common way of integrating external context-sources (e.g. from a frequent flyer profile).

7.3.4 Other Improvements

7.3.4.1 Tool Support

The tool support for designing compositions or XML transformations is already very good. It could be complemented by an integrational development environment for the overall configuration which is currently done by hand. The visualization of message routes etc. would also minimize possible design errors.

7.3.4.2 Robustness and Quality of Service

The proposed architecture is not monolithic and the components as well as the composition engines can be distributed to multiple physical machines. The message router could also be extended to implement load balancing by targeting redundant instances of the composition engine and the service connectors. There are quite some difficulties like the consistency of the client context to be taken in account.

7.3.4.3 Monitoring and Management Console

What is missing today is a tool for surveillance of the server and all of its components with a graphical display of the status (heart beat mechanism). It should include features to directly interact to solve problems or update configuration (routes etc.).

7.4 Summary

The paradigm of service orientation is a unique and promising way to overcome the limitations of current software systems. Service composition is a key concept for application development in a service oriented architecture. The practical implementation remains a challenge especially when trying to bridge the gap between client application development and enterprise service infrastructures.

This document proposed a generic solution which is based on existing standards and technologies. It combines service composition and credential management to allow the secure and accountable aggregation of business services to facilitate client application integration.

It was also shown how the concept of a stateful context can be used to build adaptable compositions which off a wide range of new possibilities and help to reduce computation on client side.

The concept and its prototypical implementation have been validated by a practical use-case which confirmed that the client development was significantly facilitated and that the initial requirements for service composition, security and accounting could be fulfilled. First tests have also shown that the performance and quality meets the requirements for production use.

Finally possible improvements have been shown to underline the flexibility of the component based concept and to give an introduction of how this work can be continued.

Nomenclature

CSA Credential and Service Aggregator - middleware component that unites service composition with authentication, authorization and accounting capabilities

GDS Global Distribution System

IATA The International Air Transport Association

IDE Integrated Development Environment

J2ME Java 2 Mobile Edition

JSR Java Specification Requests of the Java Community Process

OASIS Organization for the Advancement of Structured Information Standards

PAX Abbreviation for Passenger(s)

PNR Passenger Name Record - IATA standard for saving reservation information for one or multiple travellers

POI Point Of Interest

RMS J2ME Record Management System (RMS)

URI Uniform Resource Identifier

W3C World Wide Web Consortium

List of Figures

2.1 Services can encapsulate varying amounts of logic.[Erl05] 5
2.2 first generation Web Service components[fujitsu] 8

3.1 Travel Wallet - Trip Overview and Itinerary Detail 13
3.2 Travel Wallet - Destination Information 13
3.3 Composite Trip Information Service Example 15
3.4 Use-Case: User Groups 16
3.5 Categories of Interface Adaptation (schematic samples) 18
3.6 Example: Schematic Flow of a Service Provider Context 22

4.1 Black-Box Composition [Ass03] 25
4.2 BPEL Partner Link with Web Services 26
4.3 Screenshot of a WS-BPEL development tool 30
4.4 OWL-S Service Class and Relations 32
4.5 OWL-S Atomic Process Example(schematic/notation) 33
4.6 Example: OWL-S Composite Process Example (schematic/notation) . . . 34
4.7 OWL-S Service Grounding [SD05] 35
4.8 OWL-Editor Screenshot 36
4.9 Example of WS-Addressing SOAP Header Elements[**?**] 39
4.10 WS-Security Example for a UsernameToken[AN04] 41

5.1 CSA Architecture Overview 44
5.2 CSA Components: Composition Engine 45
5.3 CSA Components: Client Connector 46
5.4 CSA Components: Message Router 47
5.5 CSA Components: Service Connectors 47
5.6 CSA Credential Manager 49
5.7 Data Model of the Local Credential Store 50
5.8 CSA Session Context Manager 51
5.9 Context Lifecycle - Initialization 52
5.10 Context Lifecycle - Session Load 53
5.11 Context Lifecycle - Service Provider Transaction 55
5.12 Composition Engine Integration - Control Flow Adaptation (Example) . . 56
5.13 Composition Engine Integration - Content Personalization (Example) . . . 57
5.14 Composition Engine Integration - Context Push 58

5.15 Composition Engine Integration - Context Pull 58

6.1 Inversion of Control through Dependency Injection 61
6.2 Spring Framework Architecture [JHA+07] 64
6.3 The request processing workflow in Spring Web Services[AP] 65
6.4 Example of a XWSS Security Policy File[**?**] 66
6.5 Example of a Spring-WS Security Configuration[AP]. 67
6.6 Spring WS Call-Back Handler using the Acegi Security Manger 68
6.7 Method Access Protection with Spring AOP and Acegi 69
6.8 CSA Client Connector Workflow 70
6.9 CSA Client Request Envelop with WS-Addressing Header and Sample Values 71
6.10 Example of a CSA Rooting Configuration 74
6.11 CSA Service Connector Workflow 75
6.12 CSA Layered Authentication and Authorization Process 77
6.13 Context Manager Interface and Datastructure 78
6.14 Overview of the Context Manager Service 79
6.15 CSA Configuration Overview for Mobile Travel Wallet 81
6.16 TripInformationReply Message for the Mobile Travel Wallet 83
6.17 Refined Activity Diagram of a Trip Information Process 85
6.18 WSDL Partner Link Type Definition 85
6.19 Extract of a structured Amadeus Passenger Name Record (PNR) 87
6.20 Extract of Web Service Response with Escaped Payload 88
6.21 Mobile Travel Wallet - Functional Overview 92
6.22 Travel Wallet Domain Model 93
6.23 Client Proxy for Mobile Travel Wallet 94
6.24 Mobile Travel Wallet Display Samples 96
6.25 Service Connector Response Time Distribution 97
6.26 CSA Response Time Distribution 98

List of Tables

4.1 Support for selected WS-* Specification in Popular Web Service Stacks [Source: `http://xfire.codehaus.org/Stack+Comparison`] 38

6.1 Structure of the Logical Identifier 73
6.2 CSA Operations for the Mobile Travel Wallet 82
6.3 Performance Test Results (time in ms) 98

Bibliography

[AAB+07] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu. *Web Services Business Process Execution Language Version 2.0*. OASIS `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, April 2007.

[AMA07] *Amadeus Corporate Profile*. Amadeus IT Group SA, 2007.

[AN04] C. Kaler P. Hallam-Baker R. Monzillo editors A. Nadalin, P. Griffin. *Web Services Security, UsernameToken Profile 1.0*. OASIS `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`, March 2004.

[AP] Rick Evans Arjen Poutsma.

[Ass03] Uwe Assman. *Invasive Software Composition*. Springer Verlag, 1 edition, 4 2003.

[BHL+04] M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. *OWL-S: Semantic Markup for Web Services*. W3C `http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/`, November 2004.

[BvHH+03] S. Bechhofer, F. v. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. Andrea Stein. *OWL Web Ontology Language Reference*. W3C `http://www.w3.org/TR/owl-ref/`, 2003.

[Car07] Sandy Carter. *The New Language of Business: SOA & Web 2.0*. IBM Press, 1 edition, 2 2007.

[CHvRR04] L. Clement, A. Hately, C. v. Riegen, and T. Rogers. *UDDI Version 3.0.2*. OASIS, `http://uddi.org/pubs/uddi-v3.0.2-20041019.htm`, October 2004.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C `http://www.w3.org/TR/2007/REC-wsdl20-20070626`, June 2007.

[Erl04] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 4 2004.

[Erl05] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, 8 2005.

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Frystyk Nielsen, A. Karmarkar, and Y. Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C `http://www.w3.org/TR/2007/REC-soap12-part1-20070427`, April 2007.

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the Fourteenth Australasian Database Conference (ADC 2003)*, Adelaide, Australia, 2003.

[Her07] Ivan Herman. *Semantic Web Activity Statement*. W3C `http://www.w3.org/2001/sw/Activity`, May 2007.

[JHA+07] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, and R. Laddad. *The Spring Framework - Reference Documentation Version 2.5*. Spring Project `http://static.springframework.org/spring/docs/2.5.x/spring-reference.pdf`, 2007.

[LKN+06] K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, and P. Hallam-Baker. *Web Services Security, SOAP Message Security 1.1*. OASIS `http://docs.oasis-open.org/wss/v1.1/`, February 2006.

[QP06] R. Quintero and V. Pelechano. Conceptual modeling of service composition using aggregation and specialization relationships. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 452–457, New York, NY, USA, 2006. ACM Press.

[SD05] Shahin Saadati and Grit Denker. *An OWL-S Editor Tutorial - Version 1.1*. SRI International `http://owlseditor.semwebcentral.org/documents/tutorial.pdf`, April 2005.