Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3201

# Extending an Open Source BPEL Engine for Multi-Tenancy Support

Michael Baldauf



| | |
|---|---|
| **Course of Study:** | Computer Science |
| **Examiner:** | Prof. Dr. Frank Leymann |
| **Supervisors:** | Dipl.-Inf. Tobias Binz |
| | Dipl.-Inf. Steve Strauch |
| **Commenced:** | June 20, 2011 |
| **Completed:** | December 20, 2011 |
| **CR-Classification:** | D.2.11, H.3.5, H.4.1, H.5.3 |

# Abstract

WS-BPEL is the de-facto standard for orchestrating Web services into business processes. Workflow engines can execute WS-BPEL processes. Furthermore, workflow engines handle the communication with external service partners providing the Web services. One future goal is to achieve tenant-aware Web services and thereby tenant-aware workflow engines to handle Web services. These tenant-aware Web services are specifically configured for the tenants, and the tenant-aware engines are able to configure such a process instance and offer them on a per-tenant basis. This reduces provider costs and efforts.

The goal of this diploma thesis is to describe how workflow engines can support multi-tenancy, especially in the area of communication, and how this is of advantage for providers who offer their services over the Web using such a workflow engine. The providers should be able to offer tenant specific instances of applications on one workflow engine. In this diploma thesis, a concept to extend a workflow engine to handle a tenant context is developed. It is therefore an extension of the workflow engine in the area of communication. The concept is implemented by extending the open source WS-BPEL engine OW2 Orchestra.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# List of Abbreviations

ASF ............ Apache Software Foundation
BPEL ........... Business Process Execution Language
DB ............. Database
ESB ............ Enterprise Service Bus
GUI ............ Graphical User Interface
IaaS ............ Infrastructure as a Service
ID ............. Identification
IDE ............ Integrated Development Environment
ITHA .......... Isolated Tenancy Hosted Applications
JAX-RS ......... Java API for RESTful Web Services
JAX-WS ........ Java API for XML Web Services
JSP ............ JavaServer Pages
MBPC .......... Multi-Tenant Oriented Business Process Customization System
MSLD .......... Multi-Tenant, Secure, Load Disseminated SaaS Architecture
MTEA .......... Multi-Tenant Enabled Applications
MTS ........... Multi-Tenant System
NIST ........... National Institute of Standards and Technology
OASIS .......... Organization for the Advancement of Structured Information Standards
ORM ........... Object-Relational Mapping
OS ............. Operating System
PaaS .......... Platform as a Service
QoS ............ Quality of Service
RP ............. Request Parser
SaaS ............ Software as a Service
SM ............. Service Manager
SMBs ........... Small and Medium Businesses
SOA ............ Service-Oriented Architecture
STS ............. Single-Tenant System
SVBs .......... Very Small Businesses
TID ............ Tenant Identifier
UID ............ User Identifier
UUID .......... Universally Unique Identifier
VM ............. Virtual Machine
VRC ............ Validation Rule Component
XML ........... Extensible Markup Language
XSD ............ XML Schema Definition

# 1 Introduction

The concept of Web services and the associated architectural paradigm of Service-Oriented Architecture (SOA) are nowadays very important for the business IT [WCL$^+$05]. A definition of SOA is shown in [TOG]. One of the main characteristics of SOA is the reuseability of the software components like Web services. There are some proven technologies in SOA, e.g., workflow engines or an Enterprise Service Bus (ESB). With a workflow engine, modeled processes like BPEL processes can be executed. The ESB takes care of the service controlling, providing, and the communication between the individual services. To realize the SOA paradigm with the workflow engines and ESB, Cloud computing represents another significant and important paradigm. A definition of Cloud computing is shown in Section 2.3. A realation between SOA and Cloud computing is shown in [BGK$^+$11]. Cloud computing enables the SOA paradigm. It supports the service orientation of SOA with the help of the three service models IaaS, PaaS, and SaaS. Cloud computing adds the cloud characteristics to the delivered and consumed services. Furthermore, Cloud computing realize the SOA characteristics with the help of five essential characteristics. One of the main challenges is to provide the proven technologies in the cloud. One main feature of SOA and Cloud computing is to support tenant-aware applications, engines, and ESBs. Especially for the providers of applications tenant-aware workflow engines are a significant property, because one shared engine for a number of individual tenants is more resource efficient. The provider can offer tenant specific instances of the applications on one running workflow engine without setting up one workflow engine for each tenant.

## 1.1 Problem Statement

The goal of this thesis is to analyze existing concepts of multi-tenant composition engines and to extend the open source BPEL engine Orchestra to support multi-tenancy on a per-tenant basis. This means, that each tenant gets his own, specific, and configured process instance. Therefore, a Web service is called by processing a SOAP message including a *tenant context* explicit in its header. For the first time this tenant context includes a tenant identifier (TID) and a user identifier (UID) but should be expandable for further work. The specific instance is implemented based on tenant specific metadata.

The main focus of this diploma thesis is to add the multi-tenancy support in the area of the communication. This means the handling of the tenant context in the communication framework. More precisely to get the tenant context into a workflow engine, realize the internal passing, and include it into outgoing messages, e.g., to communicate with an ESB.

The management and configuration of the tenant specific process instances is not discussed or implemented in this diploma thesis.

Another important task is the investigation of an extensible and reusable concept of the tenant context and the integration of it into the SOAP message protocol as seen in Section 4.3. Based on this, the BPEL engine Orchestra is extended by extracting the tenant context from the incoming SOAP message into a variable as shown in Section 4.4. To realize the support of a tenant specific instance it is necessary to extend the process instance to manage the tenant context as shown in Section 4.5. The last step is to include the tenant context in the outgoing SOAP message as shown in Section 4.6.

## 1.2 Research Design

This Section demonstrates the individual steps that were necessary in preparing this diploma thesis to reach the goal of extending a workflow engine with multi-tenancy support in the area of communication. The first step was to achieve specialized knowledge through research on multi-tenancy approaches for compositions and applications as described in the Related Work Chapter 3. This knowledge was acquired by reading research papers, literature, thematically related diploma thesis, and definitions. The knowledge is analyzed to define the requirements for tenant-aware workflow engines. The concept and specification of this diploma thesis is created as shown in the Chapter 4. Therein, the requirements for a tenant-aware workflow engine and the use cases for the extension of the engine are presented. The concept is desgined and implemented in a open source workflow engine. Both, concept and implementation are evaluated on the basis of a example scenario.

## 1.3 Motivating Example

For a better understanding of tenant-aware workflow engines and the associated tenant specific process instances, sample use case scenarios are presented in this Section. Two simple example scenarios are described: The examples are the synchronous `echo` and the asynchronous `taxi` scenario. The two scenarios will be described in more detail in Section 5.4. For the sake of simplicity, simple Web services are used in this motivating example.

The synchronous *echo scenario* works as follows: A user sends a request SOAP message including a sample string in its body to the workflow engine. The executed BPEL process sends a request message including a sample string in its body to the `echo` Web service. The process directly receives the response message from the `echo` Web service and sends a reply message including the same string back to the user. Figure 5.3 in the implementation Chapter 5 shows the synchronous scenario.

In the asynchronous *taxi scenario*, a user calls a taxi company and wants to arrange for a taxi to a desired address. The taxi company sends a request SOAP message including the company's tenant context and the desired user address to the workflow engine. The process

is executed and in one step a Web service is invoked with the tenant context information. For example, this Web service could be a customer SMS notification service. The called Web service is choosen by the tenant's (taxi company's) specific preferences, e.g., to use a preferred low priced SMS service provider. This is realized by the communication with an multi-tenant ESB. The workflow engine calls the ESB and asks for the tenant specific Web service with the help of the tenant context. The ESB handles the received tenant context and invokes the tenant specific Web service[1]. The user is notified by the chosen Web service. After a delay, e.g., the user has to confirm the notification response, the process receives a response message including the tenant context from the Web service. This is the information that the customer is already successfull notified. Figure 5.4 in the implementation Chapter 5 shows the asynchronous scenario.

## 1.4 Definitions and Conventions

To better understand this document, some conventions are used. For reasons of comfort, we explain these here to avoid misunderstandings.

- Companies and departments representing tenants.
- Clients and subscribers representing users.
- Single instance multi-tenancy and native multi-tenancy are equal models.
- A tenant is not the same as a user. A tenant includes a lot of different users, who are his stakeholders.
- Cloud or Clouds refer to the the term Cloud computing which is described further in the Fundamentals chapter.
- An instance of an service is an instance of the application implementing the service [MUTL09].

## 1.5 Outline

This diploma thesis is divided into six chapters - introduction, fundamentals, related work, concept and specification, design and implementation, and a summary and outlook chapter at the end which summarizes the work done in this diploma thesis and gives an outlook on the future work.

In detail, each chapter covers the following topics:

**Chapter 1 - Introduction:** The introduction begins with the problem statement covered in this diploma thesis. After this, the research design of this diploma thesis and two motivating examples are described. Then the reader gets a short overview over the individual chapters.

---

[1]The communication with the multi-tenant ESB is not realized in this work. The ESB's invocation of the specific Web service is done Stefan Essl [Ess11]. This work realizes only the Web service invocation including the tenant context information

**Chapter 2 - Fundamentals:** Chapter 2 gives a brief introduction of the basic technologies and concepts used in this thesis. It covers the topics Service Oriented Architecture, the workflow engine Orchestra, Cloud computing, and Multi-Tenancy and its related subtopics.

**Chapter 3 - Related Work:** Chapter 3 covers the current research and industrial status of tenant-aware workflow engines and Web applications. It describes and explains the two multi-tenant PaaS compositions ActiveVOS and WSO2 Stratos, and two tenant-aware applications. Furthermore, multi-tenancy patterns to make applications tenant-aware are presented. In addition, research approaches of multi-tenant architectures and reengineering patterns are described. Then a comparison of tenant-aware workflow engines is presented.

**Chapter 4 - Concept and Specification:** The concepts and specifications of the work covered in this diploma thesis are discussed in Chapter 4. The Chapter starts with the specification of the tenant context, its structure, its life cycle, and its reuseability and extensibility. Then it describes the default Orchestra procedures to receive messages, invoke a process instance, send messages, and an approach to modify them. This Chapter is a starting point for the validation of the described approach.

**Chapter 5 - Design and Implementation:** Based on the Orchestra engine, this Chapter covers the implementation details of passing the tenant context through the system and all of the components involved in this process.

**Chapter 6 - Summary and Future Work:** At the end of this diploma thesis, a summary and outlook is given to summarize the realized work and discuss the future work in the context of this topic.

# 2 Fundamentals

This chapter will cover the main technologies and concepts used in this diploma thesis, which are fundamental to understand the concepts and principles introduced in the following chapters. The focus here is not to give a complete introduction of all topics, but rather give the reader the hands-on knowledge which is necessary for a better understanding. It covers topics such as Service-Oriented Architecture (SOA), Cloud computing, Multi-Tenancy, and related subtopics. In this chapter, the reader will also receive a short introduction to the open source BPEL engine Orchestra, since it is used to validate the concepts and specification.

## 2.1 Service-Oriented Architecture

Service-Oriented Architecture is a specific architecture style which supports service orientation [TOG]. A service constitute the substantial part of the SOA concept and gives an abstract view of a business function [WCL+05]. One of the proven technologies in SOA is the concept of an Enterprise Service Bus (ESB). An ESB implements the interaction between applications. Furthermore, it takes care of the controlling and providing of the application. The ESB receives a Web service invocation message, chooses the specific Web service, and invokes it. In the case of a tenant-aware system, the ESB receives the service invocation message including the tenant information (tenant context), chooses the tenant specific Web service, and invokes it. There are some definitions of SOA in the dedicated literature [WCL+05], from the OASIS Group [OAS06], and from the Open Group [TOG]. The definition of the OASIS Group [OAS06] is as follows:

"A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."

**2.1.1 SOAP**

SOAP is a flexible message protocol used to exchange information in a decentralized and distributed environment [W3C]. SOAP once stood for Simple Object Access Protocol. The structure of a SOAP message is shown in Figure 2.1. SOAP messages could be bind to different network transport protocols (for example http) and are based on the XML infoset [WCL$^+$05]. The SOAP envelope contains 1..N SOAP headers and 1..M SOAP bodys. The provided definition of SOAP in [WCL$^+$05] is as follows:

"SOAP is the fundamental messaging framework for Web services. With SOAP, you can access Web services through loosely coupled infrastructure that provides significant resilience, scalability, and flexibility in deployment using different implementation technologies and network transports."
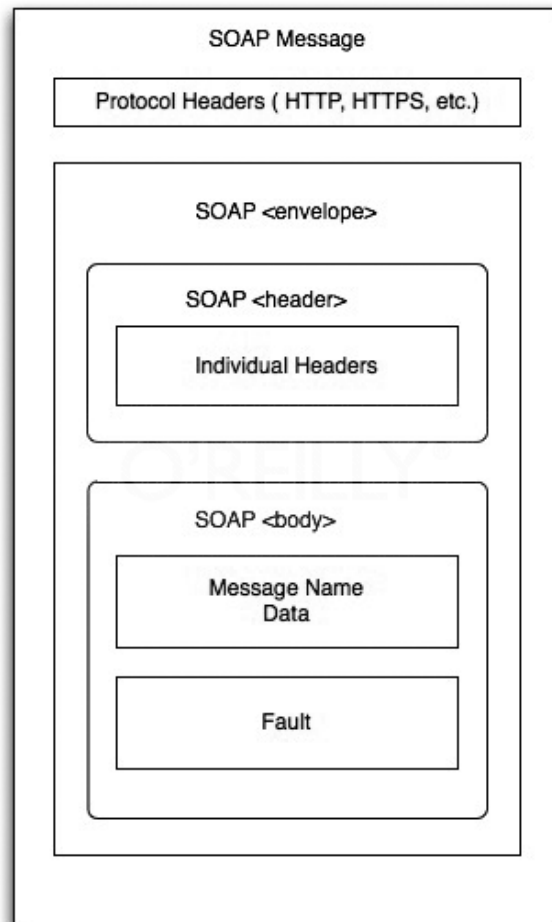


**Figure 2.1:** The Structure of a SOAP Message (Source: [MSDa])

### 2.1.2 Apache CXF

Apache CXF [APAa] is a open source Web service framework and a combination of the two open source projects Celtix[1] and XFire[2]. It is used by the open source BPEL engine Orchestra. It supports JAX-WS[3] and JAX-RS as API for Web service and RESTful Web service development. In addition maven[4] tooling is supported. Furthermore, according to [APAa], Apache CXF supports the following Web service standards:

- SOAP
- WS-Addressing
- WS-Policy
- WS-ReliableMessaging
- WS-SecureConversation
- WS-Security
- WS-SecurityPolicy

One important thing inside the CXF architecture [APAb] are `Interceptors` [APAd]. Interceptors can handle the messages. For example, reading the content, processing headers, or logging. Interceptors can be used for clients and servers and are stored in interceptor chains. The participant (client or server) who sends the message to the other participant has an outgoing interceptor chain. And the receiving participant has accordingly an incoming interceptor chain. Furthermore, there are outbound and inbound error handling chains to handle SOAP faults. The interceptor chains are divided up into phases. The phases of the incoming chain are shown in Table 2.1 and the phases of the outgoing chain are shown in Table 2.2. The implementation of each interceptor has a `handleMessage` method. It is also possible to log incoming or outgoing SOAP messages by adding an in/out interceptor to the associated interceptor chain [APAc].

---

[1]http://celtix.ow2.org/
[2]http://www.xfire.com/
[3]http://jax-ws.java.net/
[4]http://maven.apache.org/

| Phase | Functions |
|---|---|
| RECEIVE | Transport level processing |
| *PRE/USER/POST*_STREAM | Stream level processing/transformations |
| READ | This is where header reading typically occur |
| *PRE/USER/POST*_PROTOCOL | Protocol processing, such as JAX-WS SOAP handlers |
| UNMARSHAL | Unmarshalling of the request |
| *PRE/USER/POST*_LOGICAL | Processing of the umarshalled request |
| PRE_INVOKE | Pre invocation actions |
| INVOKE | Invocation of the service |
| POST_INVOKE | Invocation of the outgoing chain if there is one |

**Table 2.1:** The Incoming Interceptor Chain Phases (Source: [APAd])

| Phase | Functions |
|---|---|
| SETUP | Any set up for the following phases |
| *PRE/USER/POST*_LOGICAL | Processing of objects about to marshalled |
| PREPARE_SEND | Opening of the connection |
| PRE_STREAM | |
| PRE_PROTOCOL | Misc protocol actions |
| WRITE | Writing of the protocol message, such as the SOAP Envelope |
| MARSHAL | Marshalling of the objects |
| *USER/POST*_PROTOCOL | Processing of the protocol message |
| *USER/POST*_STREAM | Processing of the byte level message |
| SEND | |

**Table 2.2:** The Outgoing Interceptor Chain Phases (Source: [APAd])

## 2.2 The Workflow Engine Orchestra

In this Section the open source BPEL engine Orchestra[5] is presented. The engine is based on the OASIS[6] standard WS-BPEL 2.0 [OAS07]. The main focus is on the architecture of the engine. Orchestra's architecture is shown in Figure 2.2. The significant parts of the engine for this work are the open-source Web service framework Apache CXF in the Web service part on the right side in Figure 2.2 and the Invoker part of the service container in the core of the system.
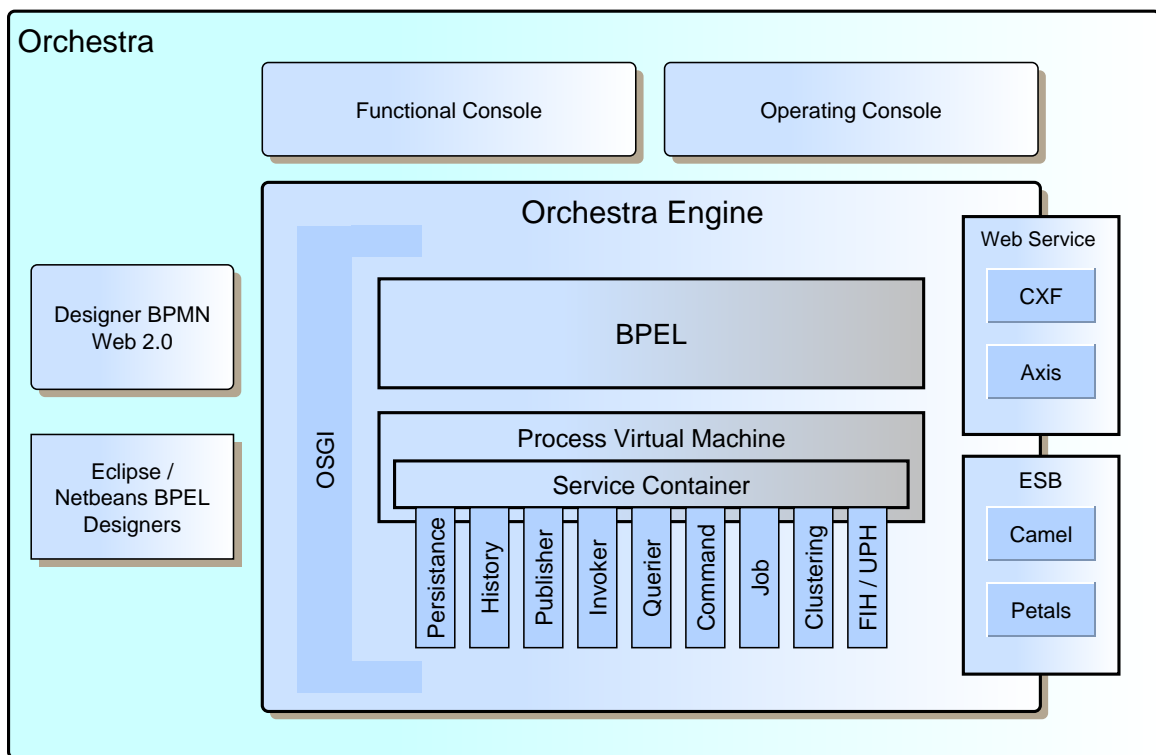


**Figure 2.2:** The Orchestra Architecture (adapted from [PLJ10])

---

## 2.3 Cloud Computing

Cloud computing is a model which enables offering and utilization of computer resources, software, and information. The resources have been provided and accessed over a network like the internet. The National Institute of Standards and Technology (NIST) provides a concise and specific definition:

"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models." [MG11].

Another definition of Cloud computing can be found in [Ley11]. In [VMCL09] the concept of Cloud computing, the definition of a Cloud, and the characteristics of cloud computing in the associated literature are discussed. In the following Subsections the five essential characteristics, three service models, and four deployment models of the NIST Cloud definition are described.

### 2.3.1 The Five Essential Characteristics

The five essential characteristics described in [MG11] are *on-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity*, and *measured service*. They are shown at the top of Figure 2.3. The first essential characteristic is called *On-Demand Self-Service*. This means the consumer can self obtain capabilities like network storage or server time. Another essential characteristic is *Broad Network Access*. Therein, the consumer can access network available capabilities through standard mechanisms. The characteristic *Resource Pooling* means that the consumer normally has no control or knowledge about the location of the used pooled resources like storage or virtual machines (VMs). *Rapid Elasticity* designate that the consumer can acquire the capabilities rapidely, elastically, in some cases automatically, at any time and unlimited. The last essential characteristic is *Measured Service*. It means that the automatically controlled resource usage of a cloud system leads to transparency for consumer and provider.

### 2.3.2 The Three Service Models

The three service models described in [MG11] are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) which are shown in the center of Figure 2.3. In all three models the provider offers a cloud infrastructure to a customer. *SaaS* means that the consumer use applications which are accessible over the Web, offered by a provider, and running on the providers cloud infrastructure. But the consumer cannot configure the

underlying hardware and software or the application itself. In contrast, in the *PaaS* model the consumer can deploy his own acquired applications and has to configure these. The *IaaS* model goes one step further and enables the customer to deploy and run arbitrary software like operating systems (OS) and applications on the providers cloud infrastructure. The user has to configure and control his own deployed and running software, but not the underlying cloud hardware.

### 2.3.3 The Four Deployment Models

The four deployment models described in [MG11] are private cloud, community cloud, public cloud, and hybrid cloud. They are shown at the bottom of Figure 2.3. In the first deployment model called *Private Cloud*, the cloud infrastructure is deployed only for one organization. The next deployment model is the *Community Cloud* model wherein several organizations share one cloud infrastructure. The model wherein the general public or large industry groups use the cloud infrastructure is called *Public Cloud*. The last deployment model is the *Hybrid Cloud*. Therein two or more of the three above described clouds *Private*, *Community*, or *Public* are connected together.
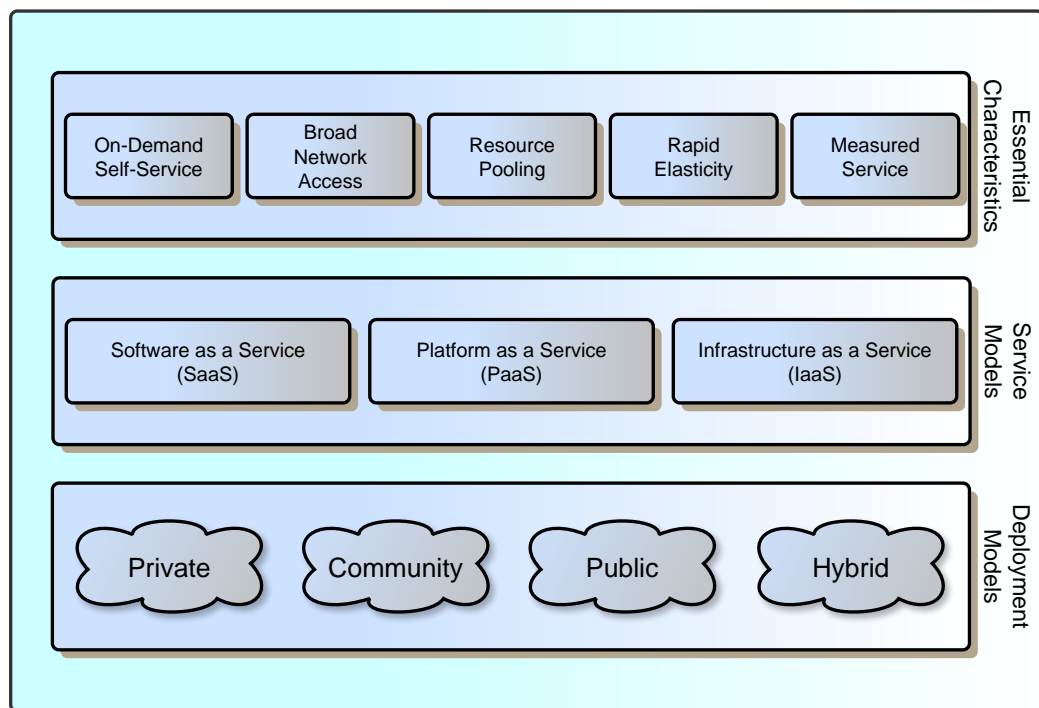


**Figure 2.3:** The Model of the NIST Cloud Computing Definition

## 2.4 Multi-Tenancy

Software as a Service (SaaS) is a software delivery model where different tenants can access, use, and rent applications from SaaS providers. Hosting one application for each tenant is not very efficient and economic. Due to this fact we need variable and customizeable instances of applications which support multi-tenancy.

The application instances of the tenants or companies have to be served concurrently. There are a lot of different approaches to realize the concept of multi-tenancy. One of the important techniques are isolation points and customization of applications [CWZ10]. Isolation points are some parts of an application which have a tenant specific value or behavior (e.g., a variable). This isolation points are stored in a metadata repository for each tenant. There are some approaches for better isolation techniques in security, performance, availability, and administration in [GSH⁺07]. With customization, tenant-specific behaviors can be enabled.

Multi-tenancy has a lot of benefits for both, providers and subscribers of Web applications. The providers get a higher profit margin because their delivery costs of applications are reduced. Thereby the subscribers or clients of Web applications have decreased costs. Reducing costs makes the application offering more attractive for tenants or clients like small and medium businesses (SMBs) and the very small businesses (SVBs). Multi-tenancy support makes expensive applications affordable for those SMBs and SVBs [KNL08]. But there are also disadvantages for providers and subscribers in relation to multi-tenancy like insufficient data or application isolation. Every tenant should use his own data set and shouldn't get access to the data of other tenants (data isolation). Furthermore, each tenant should not affect the other tenants by using the application (performance isolation).

Multi-tenancy support is not limited to SaaS applications. Another possibility is to make a server capable to support multi-tenancy. One way is that multiple virtual machines can be provided by one physical host. Each tenant accesses such a virtual machine as his own server.

### 2.4.1 Single-Tenancy

In the single-tenancy concept, only one single instance of an application for each tenant is provided. With single-tenancy each tenant gets his own (specific) software application by installing the software on his own IT infrastructure or rent it over the SaaS platform from a provider as a Web service. On the one hand single-tenant applications are cheaper to design and build, but on the other hand multi-tenancy applications are more flexible and customizable. This leads to Web applications for different tenants without rewriting the original source code. However the single-tenancy applications have to be redesigned and the code has to be rewritten for each tenant. This is one of many reasons why multi-tenancy applications are needed for the SaaS platform model.

### 2.4.2 Multi-Tenancy Models

Multi-tenancy can be classified into two types [GSH$^+$07], *native multi-tenancy* and *multiple instances multi-tenancy*. In the native multi-tenancy model, all tenants share one application instance. But every tenant has its own resources and configuration and does not realize something about the shared instance. With native multi-tenancy it is possible to support thousands of simultaneous tenants. However in the multiple instances multi-tenancy pattern, every tenant has its own application instance. Only the hardware or operating system is shared by all tenants. One problem of customizing a native multi-tenancy application for each tenant is that the services of other tenants could be affected, because they share the same application instance.

There are four levels of a multi-tenancy of an application instance [KNL08, CC06]. In level one every tenant has his own application instance with his own customized code. Level two devotes a configurable code of the application and every tenant gets his own instance. Level three utilize a configurable code and one individual customizeable instance of the application by the metadata of each tenant. Thereby each tenant has the impression that the application is his own seperate instance. Level four also utilize a configurable code and a set of individual customizeable instances of the application to support load-balancing.
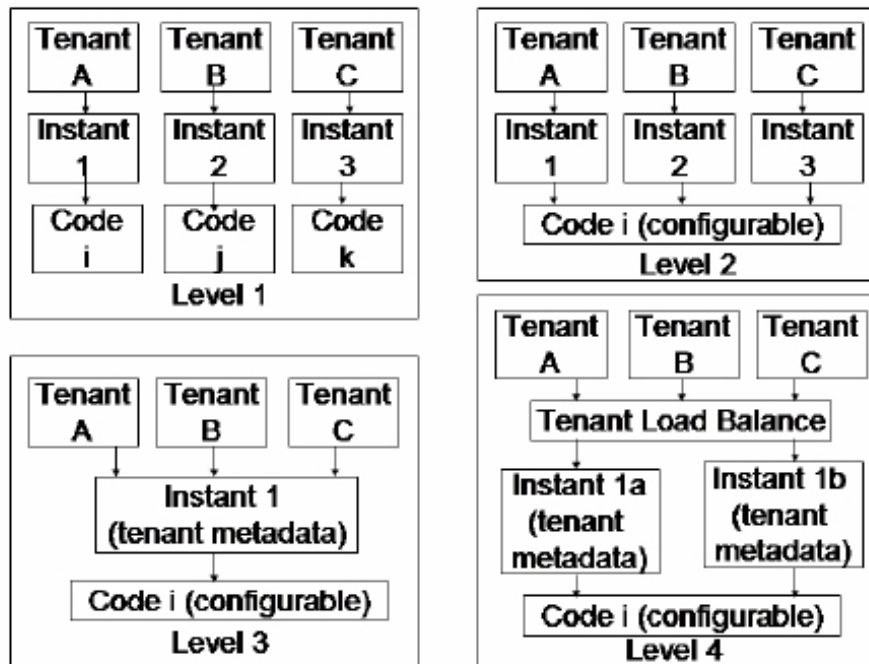


**Figure 2.4:** Four Levels of a Multi-Tenancy Model (Source: [KNL08])

# 3 Related Work

In this Chapter, the current research and industrial state of multi-tenant enabled workflow compositions and Web applications is investigated. First, two tenant-aware Web Applications are presented. After this, usage of multi-tenancy patterns to make applications tenant-aware is introduced. Then four architectures for multi-tenancy systems as well as methods to migrate and reengineer applications are described. Furthermore, the two multi-tenant PaaS products *ActiveVOS* and *WSO2 Stratos* and their functionality will be described. In addition, a diploma thesis about tenant-aware BPEL engines is reviewed. Last, a table of existing workflow engines is shown and they are compared in the area of the framework, compatibility, and tenant-awareness.

## 3.1 Tenant-Aware Web Applications

There are some approaches to support multi-tenancy in Web applications. One of these multi-tenant Web applications is the Electronic Contract Management Application described in [KNL08]. With the Electronic Contract Management Application, multiple tenants can manage their contracts very cheap and completely online. The application is used and hosted by IBM for a number of tenants. The conclusion of this industrial experience is that the application is stable and useful because it reduces cycletime and transaction costs. It also increases the productivity of sales and the efficience by saving a lot of time for the contracted users. The application realizes three important attributes of a multi-tenant application: (1) customizability, (2) efficiency, and (3) scalability.

The second multi-tenant application is called *Codename$^{MT}$*, a multi-tenant version of the single-tenant application *Codename*. *Codename$^{MT}$* is originated from *Codename* by the multi-tenant reengineering pattern proposed in [BZP$^+$10]. The migrating is done very cheap, quickly, and efficiently. *Codename$^{MT}$* supports the key benefits like increased usage of hardware resources, easier servicing, and non changed look-and-feel of the application. Tests of *Codename$^{MT}$* have shown that the major functionality of the original application is unaffected.

## 3.2 SaaS Applications and Multi-Tenancy Patterns

The recent work done by Ralph Mietzner et. al. in the area of multi-tenant SaaS applications [MMLP09, MUTL09, MLP08] proposes to use multi-tenancy patterns to make applications tenant-aware. Furthermore, there is several related work discussed, like a framework for

tenant-aware SaaS applications and whole tenant-aware applications. In contrast to this focus on the deployment of tenant-aware applications, the focus of [MUTL09] is more on the modeling of them. There are multi-tenancy patterns described to solve multi-tenancy problems. They also describe a real world running example application related to their multi-tenancy patterns. The application is called eCommerce Concept (eCCo) and used by the automotive industry to sell new and used vehicles. Thereby different dealers (tenants) can use the application and customize it to their needs like the specific tax rate of their market or the GUI. The application is implemented and operated only once, but the GUI is configurable by the tenant's specific metadata. The Web services are orchestrated using BPEL and run on an IBM WebSphere Process Server.

There are three basic types of instances in [MMLP09, MUTL09] to realize the tenant specific adaptation of SaaS applications, *single instance*, *single configurable instance*, and *multiple instances* of a service. The authors of [MUTL09] propose another taxonomy of the multi-tenancy patterns. Thereby Web services are either non-configurable or configurable. If its a non-configurable service and there is one instance used by all tenants, it is called *single instance service*. In contrast, the service is called *multiple instance service* when every tenant has his own separate instance of the service. If the service is configurable and one instance is used by all tenants, it is called *single configurable instance service*. When every tenant has his seperate instance of a configurable service, it is called *multiple configurable instances service*. The *multiple instances service* pattern is used when the service logic is very specific for each tenant, the data of a tenant is very private and critical, or the quality of services (QoS) is very specific for each tenant. However, this variant violates the notion of the single instance multi-tenancy architecture. Furthermore, it is harder to update the several multiple instances than one shared single instance. There are no process engines which support multi-tenancy on a per-tenant basis. Multi-tenancy on a per-tenant basis means that each tenant gets his own specific and different configured process instance in addiction to a tenant identifier (TID). Therefore, the idea is to call the Web service by processing a SOAP message which includes a *tenant context* explicit in its header[1] or implicit in its payload. There are two different notations of service invocations, a *basic invocation* and a *invocation under tenant context*. A so called *tenant handler* extracts the tenant context and retrieves the tenant's specific metadata from a database by the TID. The service is implemented based on this tenant specific metadata.

The eCommerce Concept (eCCo) SOA application described in [MUTL09] is an example for the described multi tenancy patterns. The described application is a automotive point-of-sales Web service application defined as BPEL process. The main functionality of the application is that car dealers can deal with the sale of new and used cars. A dealer corresponds to a user and a dealer organization to a tenant. The application is hosted centralized and can be used by several dealers. However, dealers or dealer organizations can be in different markets where different modificated instances of the application are used. The differences can be financial options, tax rates, exchange, or external services, and differences in the GUI. To support the offering of tenant-specific instances and not only the support of modifiable and market specific instances, the used workflow engine must be tenant-aware. This means it must be able to handle a tenant context to create dealer specific instances.

---

[1]This is similar to the approach in this thesis

18

## 3.3 Architectures

There are different architectures for multi-tenancy systems. This section prevents four kinds of multi-tenant architectures. The first architecture is the Multi-tenant Oriented Business Process Customization System (MBPC) in [SLLW09b] or also called Business Process Customization Framework in [SLLW09a]. This multi-tenancy system enables the customization of processes based on BPEL.

There are some problems in orchestrating BPEL business processes. The first problem is that a deployed BPEL process cannot be modified at runtime. Furthermore, it is difficult to realize business logic rules like QoS and handle exceptions caused by service invocations with BPEL processes. When tenants change services or the relationships between them, logical errors can occur. The architecture of the MBPC consists of a execution environment, a resource layer, and a kernel framework. Tenants can customize processes and services before they are deployed to the engine. The system can handle errors like unsuccessful process executions or prevent errors like wrong customization. The customization of the business processes is realized by adding points of variability in the BPEL process. The extension of BPEL is VxBPEL from Michiel Koning [KSS09].

The Multi-tenant Oriented Business Process Customization System (MBPC) consists of the three components: execution environment, resource layer, and kernel framework as shown in Figure 3.1:

- Execution environment: Includes the BPEL engine and external services.

- Resource layer: Includes the data storage needed by the system like tenant database or user database.

- Kernel framework: Is the core of the system. It includes five modules: business definition, service manager, service customizer, rule manager, and verification engine.

The second SaaS multi-tenant architecture is proposed in [PLL10]. The architecture is called Multi-Tenant, Secure, Load Disseminated SaaS Architecture (MSLD). The system as shown in Figure 3.2 is divided into five services:

- Responder Service: Handles the user requests.

- Routing Service: Includes three components, Service Manager (SM), Request Parser (RP) and the Validation Rule Component (VRC). The SM indexes every hosted service. The RP and VRC validate the incoming requests. VRC parse requests to check, e.g., the valid value range and valid data types. However the RP parses the requests to check if it concurs to the defined XML schema.

- Security Service: Controls authentication and authorization. Every user request passes the Security Service. The Responder Service and Routing Service depend on the response of the Security Service.

- Logging Service: There are two major tasks of the Logging Service, first logging the service failures and second logging the user service requests. This helps to realize the pay-as-you-go business model, where you only pay what you have used.
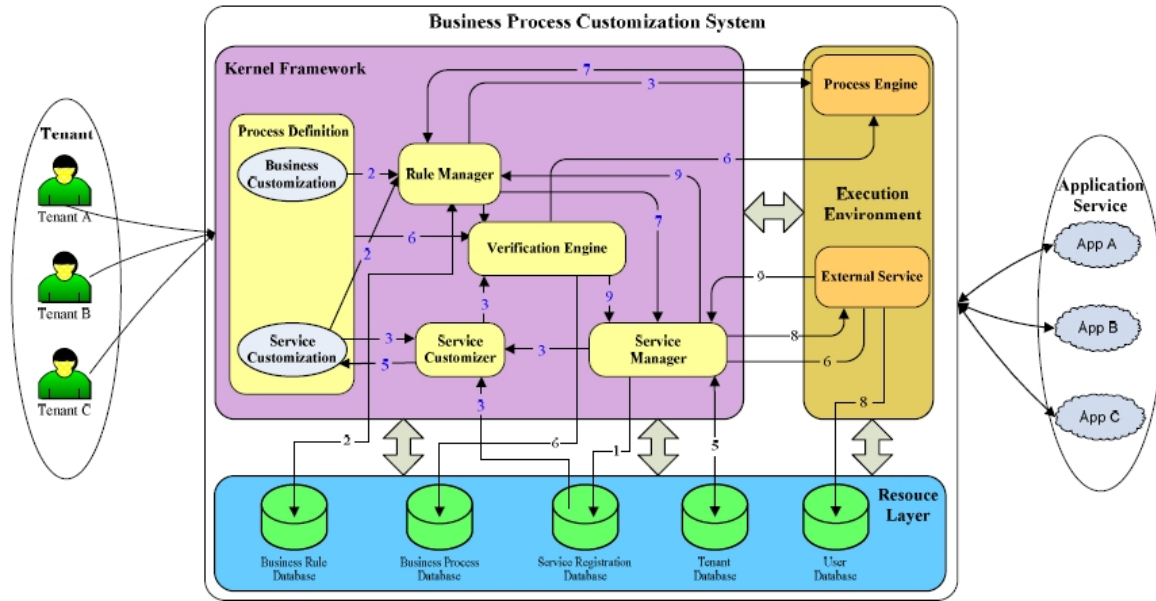
**Figure 3.1:** Multi-Tenant Oriented Business Process Customization System Architecture (Source: [SLLW09b])

- Realization Service: Hosts all offered services.

The Request propagation runs as follows: The Responder Service receives the user's service request and checks if it has a valid session and if its a valid customer. This validation of the request is realized by routing the request to the Security Service. When the Security Service validates the request its logged by the Logging Component and routed to the Routing Service. At the Routing Service the request is validated if its datatypes and value ranges are valid. Now, if the validation was successfull the Routing Service routes the request to the Realization Service. The Realization Service passes the request to the dedicated services. After the service is executed the response is routed to the Responder Service. The Responder Service checks the type of the service customer and transforms the response according to the skills of the customer. Thereby, the one instance of a service can be used by different customers.

The third SaaS multi-tenant architecture in [KNL08] is divided into three services, Tenant Specific Metadata Customization, multi-tenancy SaaS Security Service and multi-tenancy SaaS Data Model. In this System every tenant gets an unique identification (ID) and an own administrator. This administrator creates the user accounts for each tenant. There are three kinds of databases in the data model, one for the tenant information, one for the end user information, and one for the electronic contracts between customer and service provider. The tenant's administrator has to customize the application presentations, branding, user-interfaces, and Web pages by configuring the applications metadata. To realize this, they use the Customization Module. The users from different tenants have to be authenticated and authorized based on their unique ID by the Security Service. In the realized electronic contract management application in [KNL08] are eight modules: admin module, access control module, workflow
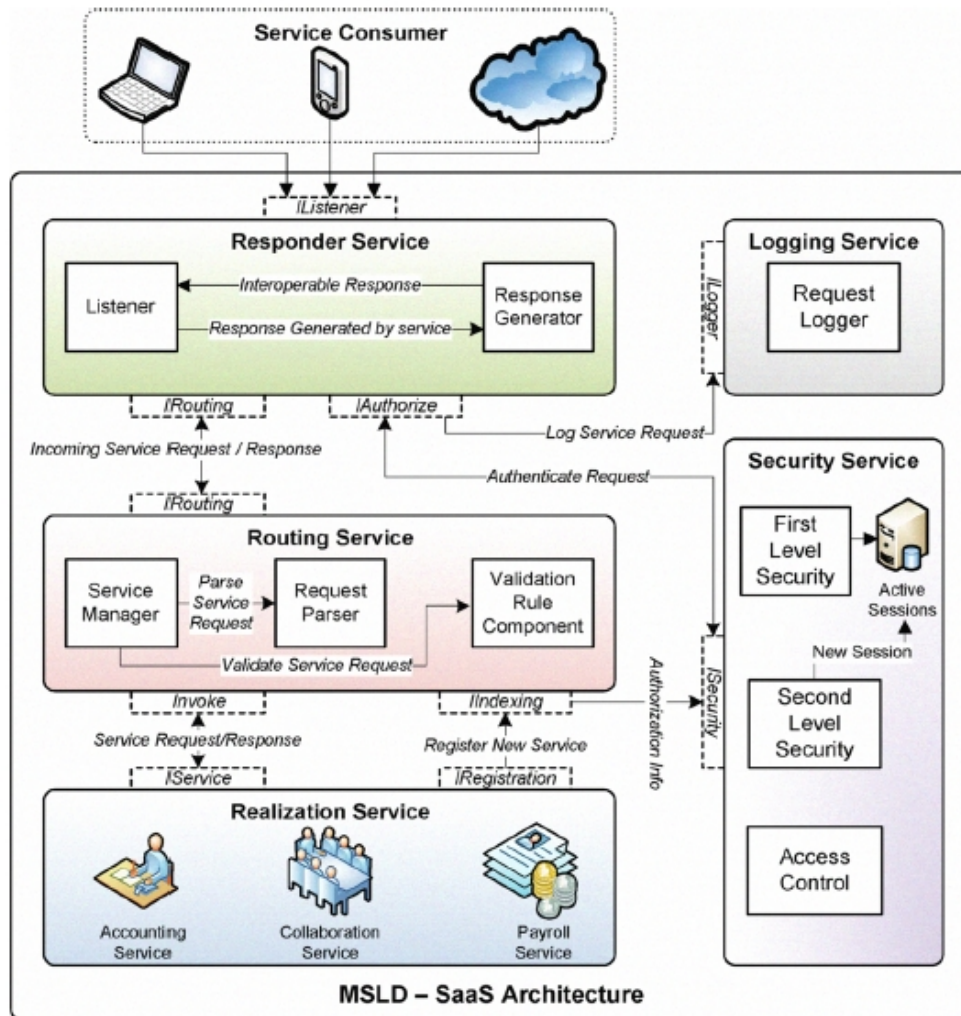
**Figure 3.2:** Multi-Tenant, Secure, Load Disseminated SaaS Architecture (Source: [PLL10])

module, email notification module, electronic signature module, document management module, data extraction and search module, and a life cycle management module. These eight modules are tenant specific by the unique tenant ID numbers and retrieve the tenant specific metadata by interacting with the Metadata Service. The hosted application by the application provider is shared by different tenants. Due to this fact the application has to be parametric to realize the customization. The Metadata Customization Model enables this customization of applications by each tenant with sets of customization templates. The application can, e.g., customized by setting the tenants company logo. The System uses a centralized authentication system. The users have roles according to their tenant IDs. The tenants and their users will be authenticated by the unique tenant ID and a password. The administrator can assign the user to an access level, add the general information of the user, and set the role of a user like submitter, signer, counter-signer, approver, or reviewer. The multi-tenancy Data Model has to be robust and secure. The tenant's specific data should

be isolated from the other tenants because they use the same application instance and each tenant has its own specific data. The multi-tenancy SaaS application uses a shared database and schema. The resources, software codes, and the data of the tenants are hosted in the same database on the same set of servers.

The fourth and last multi-tenant SaaS system presented in this section is shown in [SR11]. The goal of the system is to support the custom requirements of individual tenants and not only the common needs of several tenants. As the other presented systems and architectures in this section, this system has a shared an centrally-hosted application used by a large number of different tenants. Because of this its important to realize the application variations of the different tenants. We have to distinguish between common services used by all tenants and specific services created for and used by some tenants. However, the system is not only a collection of service variants. It should be able to analyze the service degree of variability. With thresholds, the evolution of the multi-tenant SaaS system can be controlled. The provider can offer a mix of re-used services and new developed services to the tenants. The higher the part of the re-used services the lower are the costs of extra developed services. And this increase the profit of the provider.

## 3.4 Migrating and Reengineering

A method to migrate isolated tenancy hosted applications (ITHA) into multi-tenant enabled applications (MTEA) is proposed in [ZSTC10]. The main focus is on the three aspects: data model, access control, and tenant management. To support multi-tenancy it is important to isolate the data of each tenant. To migrate from ITHA to MTEA it is important to be aware of the architectual similarities and differences of the two models. The data model in ITHA only describes the business requirement. In contrast, in MTEA the data model has to support the data isolation and take-up a large number of tenants. In ITHA there is a user authentication because there is only a single-tenant. In contrast, in MTEA there is a tenant authentication. The login is done by the three inputs: tenant name, user name, and password. MTEA has a tenant management console which does not exist in ITHA.

As shown in Figure 3.3, we need new or modified elements to migrate the ITHA to MTEA. The relational database which stores the business data of one tenant should be extended to hold the data of multiple tenants. In addition, a tenant database has to be added to hold the tenant and user information data. To support an access control in the MTEA, we need a security access control module. The tenant management module is added to manage multiple tenants and their data in the database. There are three kinds of data models (1) shared DB and shared schema, (2) separated DB, and (3) shared DB and separated schema. The target data model chosen for MTEA is shared DB and separated schema. In MTEA the tenant authentication has to be performed before the user authentication. Every tenant has his own schema address in the database. When a user logs in, the schema address of the tenant he belongs to is added to the user's session. Whenever a user asks for the specific business data after log in, he will get access to the data through the schema address. The resources

of one tenant should only be shared by users belonging to him and not by users from other tenants.
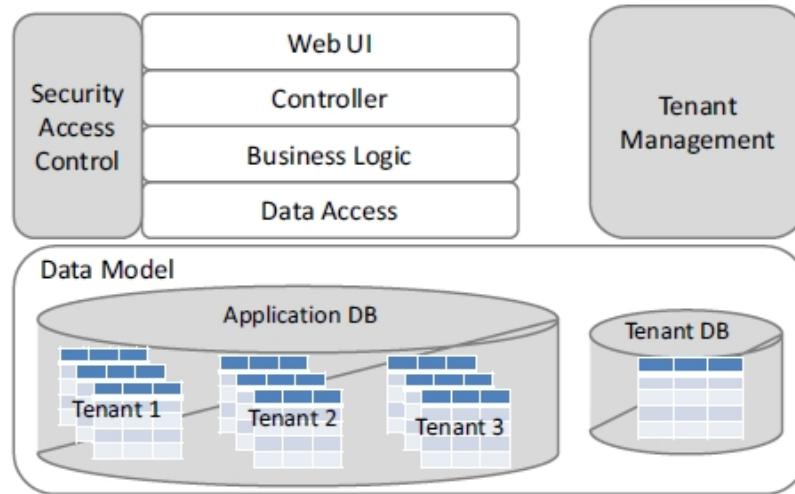


**Figure 3.3:** Architecture of Migrated Application (Source: [ZSTC10])

[BZP$^+$10] presents an approach to reengineer an existing industrial single-tenant software system[2] from Exact[3] into a multi-tenant one. The multi-tenancy reengineering pattern is shown in Figure 3.4. As in the other architectural approaches, a single instance of the software is shared by multiple tenants and is customized according to the requirements of each tenant. Because of this, the application must be customizeable or configurable by each tenant's requirements. There are two definitions in [BZP$^+$10]:

- **Multi tenant application:** "A multi-tenant application lets customers (tenants) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment."

- **Tenant:** "A tenant is the organizational entity which rents a multi-tenant SaaS solution. Typically, a tenant groups a number of users, which are the stakeholders in the organization."

The migration from a Single-Tenant System (STS) to a Multi-Tenant System (MTS) can be done easily, cost effective, transparently, and with little effects for the application developer. In a MTS tenants must be authenticated because they use the same environment but should only access their own data. Therefore a database layer ensures that the tenant specific data is stored or retrieved by the associated tenant. Furthermore, it is easier to add an additional tenant authentication mechanism than to change the current login mechanism. When a tenant logs in, a session ticket is generated. The application is loaded and configured by the information in this session ticket.

---

[2]The application Exact Codename, a research prototype
[3]A Dutch-based software company. http://www.exact.com
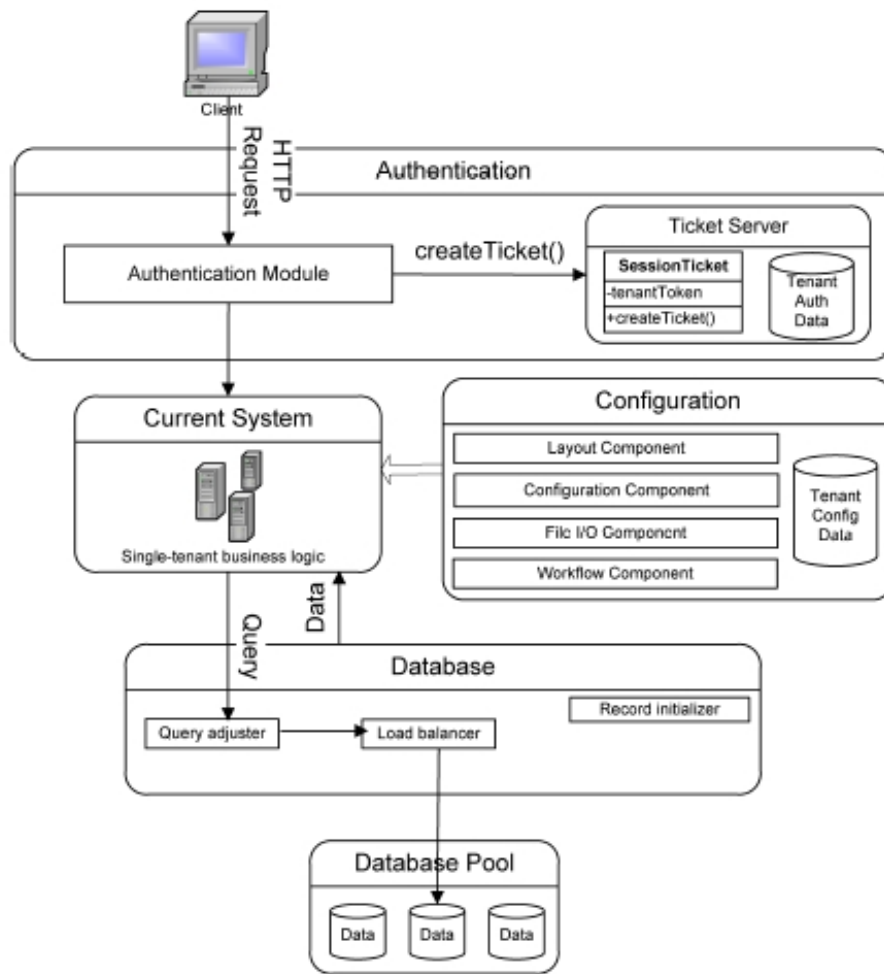
**Figure 3.4:** Multi-Tenancy Reengineering Pattern (Source: [BZP$^+$10])

## 3.5 ActiveVOS

The authors of [SLLW09b] have developed a prototype system based on the Multi-Tenant Oriented Business Process Customization System (MBPC) shown in Chapter 3.3. The main idea of the MBPC system is, that each tenant can customize the provided Web service based on his needs. In contrast, the approach of this diploma thesis will support multi-tenancy by providing tenant or user specific customized instances of the applications. The used process engine of the prototype system is ActiveVOS from Active Endpoints[4]. The experimental evaluation of the system simulates four tenants and each tenant has ten users. The author's conclusion after the evaluation of some statistics was that the MBPC system is suitable for multi-tenant customization of business processes based on BPEL.

With ActiveVOS, business processes can be executed. One of the new features of ActiveVOS

---

[4]http://www.activevos.com/

9.0 is the multi-tenancy support[5] [6] at runtime in the Data Center Edition. The $1..n$ tenants share a partitioned single instance of the ActiveVOS software as shown in Figure 3.5(a). Each tenant works on his own customized application instance. Thereby a tenant can create and manage his own processes and these processes are completely isolated. The multi-tenant architecture divides ActiveVOS into tenant groups which are controlled by the multi-tenant administration. Tenant groups reduce the management complexity. According to the multi-tenant architecture, each tenant has his own isolated shared hardware and software resources. Each of the $1..n$ tenants groups $1..m$ users, who are his stakeholders.



(a) ActiveVOS Model                    (b) WSO2 Stratos Model
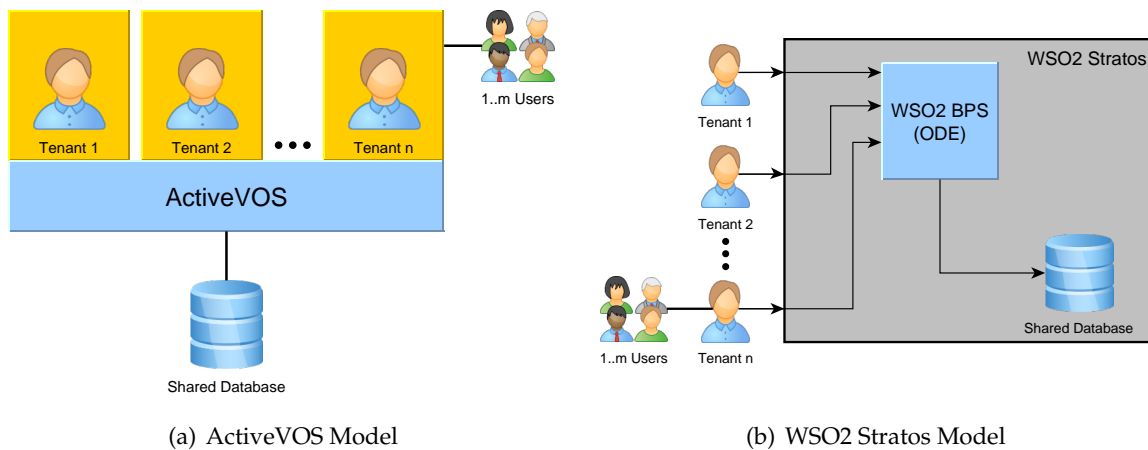
**Figure 3.5:** The Multi-Tenant PaaS Models

## 3.6 Apache ODE, WSO2 BPS, and WSO2 Stratos

The *WSO2 Business Process Server* (BPS)[7] is an open-source Business Process Server. It can execute WS-BPEL business processes and is powered by *Apache ODE*[8]. The business process and process instances can be deployed, managed, and monitored by a Web based graphical console. One of the features of the newest version WSO2 BPS 2.1.0 is the Carbon integration layer for Apache ODE with multi-tenancy support. It supports native multi-tenancy in the BPEL engine. Apache ODE is not multi-tenant capable by itself. The multi-tenancy support is enabled by including Apache ODE into in the *WSO2 Stratos* environment.

WSO2 Stratos[9] is described in [APW+11]. As shown in Figure 3.5(b), WSO2 Stratos handles the different tenants that they share one WSO2 BPS engine. WSO2 Stratos provides a series of functionalities like the tenant self-provisioning and tenant usage metering and billing. Server hardware, JVM, database, and instances of an ESB are shared by the WSO2 Stratos architecture. Furthermore WSO2 Stratos supports tenants, SMBs, and business units to develop their own

---

[5]http://www.activevos.com/cp/787/whats-new-in-activevos-9-0
[6]http://www.activevos.com/products/activevos-data-center/features
[7]http://wso2.org/library/bps
[8]http://ode.apache.org/
[9]http://wso2.com/cloud/stratos/

multi-tenant and single-tenant SaaS applications and APIs. From the supplier point of view it is possible to develop higher level APIs and SaaS applications and provide them to the customer. Like in ActiveVOS, in WSO2 Stratos each of the $1..n$ tenants groups $1..m$ users, who are his stakeholders.

## 3.7 Tenant-Aware BPEL Engines

A research of tenant-aware BPEL engines is done in the diploma thesis of Andreas Buchholz [Buc10]. Therein the multi-tenancy types, like *native multi-tenancy*, *multi-instance multi-tenancy*, as well as *single-tenancy* are presented and discussed. They are rated regarding to their advantages and disadvantages in the area of individuality, scalability, isolation, and resource usage. Furthermore, the Apache ODE engine is extended to support multi-tenancy by using multi tenant prototypes for the three modes IaaS, PaaS, and SaaS. The resulting prototype is called $ODE^{MT}$.

## 3.8 Existing Workflow Engines

This Section shows a current list of workflow engines and compares the engines in the area of the supported framework, the compatibility of standards, and the multi-tenancy awareness. Only ActiveVOS and WSO2 Stratos in combination with Apache ODE and WSO2 BPS are tenant-aware engines as described in Section 3.5 and 3.6.

| Name | Vendor | Framework | Compatibility | Multi-Tenancy |
|---|---|---|---|---|
| ActiveVOS [ACT] | Active Endpoints | Servlet or Java EE | BPMN 2.0, WS-BPEL, BPEL4People / WS-HumanTask | Yes |
| Apache ODE [APAe] | ASF (donated by Intalio) | Apache Axis, JBI, Java EE | BPEL4WS 1.1, WS-BPEL 2.0, (WS-HumanTask with Apache HISE), Apache | Yes |
| BizTalk Server [MBI] | Microsoft | .NET | BPEL, BPMN, RFID, WSDL, UDDI, WS-* | No |
| iBolt Server [IBO] | Magic Software Enterprises | Java EE | BPEL4WS | No |
| jBPM [JBP] | jBoss | Java EE | WS-BPEL | No |
| Oracle BPEL Process Manager [ORA] | Oracle Corporation | Java EE | WS-BPEL 2.0, BPMN | No |
| OW2 Orchestra [ORC] | OW2 | Apache Axis, Apache CXF, OSGi, Java EE | WS-BPEL 2.0 | No |
| Parasoft BPEL Maestro [MAE] | Parasoft | Servlet | WS-BPEL, BPEL4People / WS-HumanTask | No |
| Petals BPEL Engine [PET] | Petals Link | Java EE | WS-BPEL 2.0, WSDL 1.1 and 2.0 | No |
| SAP Exchange Infrastructure [SAP] | SAP AG | | BPEL | No |
| Virtuoso Universal Server [VUS] | OpenLink Software | | UDDI, WS-BPEL, WS-* | No |
| WebSphere Process Server [IBM] | IBM | Java EE | WS-BPEL | No |

**Table 3.1:** Comparison of Existing Workflow Engines (Source: Web research)

# 4 Concept and Specification

In this chapter, an approach for enabling multi-tenancy support in the Orchestra BPEL engine is presented to solve the challenges discovered in the previous chapters. A graphic of the multi-tenant system including the tenant-aware composition engine Orchestra, the tenant-aware enterprise service bus (ESB), and the multi-tenant service registry is shown in Figure 4.1. Therein the SOAP message with the tenant context in its header is sent to the multi-tenant Orchestra engine. This Engine handles the tenant context and creates a tenant specific process instance of the Web service. The engine can call the multi-tenant ESB with a SOAP message including the tenant context.
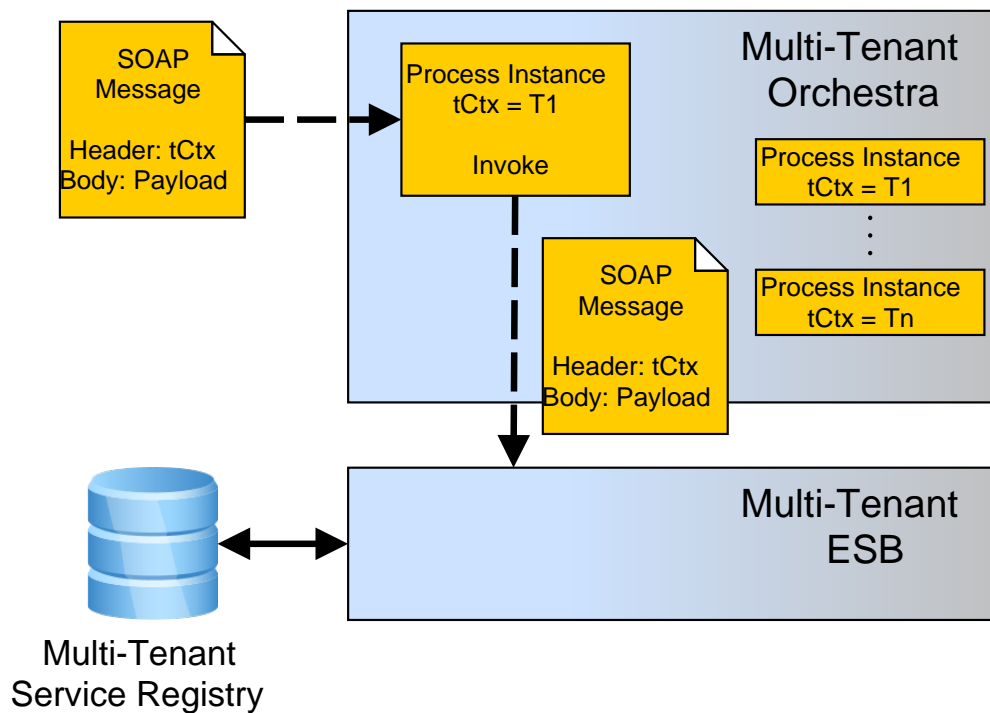


**Figure 4.1:** The Tenant-Aware Composition Engine's Architecture

The concept to realize the multi-tenancy support in the Orchestra BPEL engine is divided into several parts. At first, the requirements for a tenant-aware workflow engine are presented. Then the concept of the tenant-aware Orchestra engine is described. After this, the concept of a tenant context including its structure and lifecycle as well as the used message protocol is described (Section 4.3). The reuseability and extensibility of the tenant context is a important aspect of the concept and due to this fact specified in this Chapter. After that, a detailed

description of each step in the tenant context life cycle associated to the Orchestra BPEL engine is given. The first desribed step in Section 4.4 is how the tenant context is extracted from the message header and stored in a system variable. Then the extension of the Orchestra engine to handle the tenant context is described in Section 4.5. After that, the addition of the tenant context into the outgoing message header is explained in Section 4.6. One another thing, described in Section 4.7, is how to save and load the tenant context to and from a database to avoid loss of the tenant context information. The last subject of the concept is the external usage and internal handling of the above described tenant context in the Orchestra engine (Section 4.8).

## 4.1 Requirements for a Tenant-Aware Workflow Engine

In this Section the requirements for an multi tenant aware workflow engine are described. There are general requirements, functional requirements, and non-functional requirements. Some of the requirements are derived from the needed properties of tenant-aware composition engines and applications described in the related work Chapter 3.

### 4.1.1 General Requirements

The general requirements for a workflow engine are the three functions described in [MSDb]. These three functions are described as follows:

- **Action Validation:** The workflow engine checks if the current action is valid for the current workflow state.

- **Check Permission:** The workflow engine checks if the current user has the permission to execute the action.

- **Evaluate Condition and Execute Procedure:** If *Action Validation* and *Check Permission* are fulfilled and the evaluation of the condition is true, the workflow engine executes the action and returns the result.

### 4.1.2 Funtional Requirements

The functional requirements for a tenant-aware workflow engine are:

- Extract the tenant context from the incoming message header to a system variable. This is explained in more detail in the use case table 4.1.

- Extend the process instance to handle the tenant context. This is explained in more detail in the use case table 4.2.

- Add the tenant context to the outgoing message header. This is explained in more detail in the use case table 4.3.

- Data, process, and performance isolation.

| Name | *Send Service Request including a tenant context to the workflow engine* |
|---|---|
| Goal | The tenant context of an incoming message has to be extracted and stored into the defined tenant context system variable. |
| Actor | A user sends a request including a tenant context to the workflow engine using a predefined endpoint of the Web service. |
| Pre-Condition | The workflow engine is up and running and the Web service is deployed. |
| Post-Condition | The workflow engine is waiting for service requests. |
| Post-Condition in Special Case | A notification is sent to the tenant. The workflow engine is waiting for service requests. |
| Normal Case | The engine receives the request message and extracts the tenant context from the message header into the defined tenant context system variable. |
| Special Case | The workflow engine is unable to extract the tenant context from the incoming message header and notifies the tenant and/or administrative staff thereof. Nevertheless the engine has to be able to accept further service requests. |

**Table 4.1:** Use Case: Send Service Request inclusive Tenant Context to the Workflow Engine

### 4.1.3 Non-Functional Requirements

There are three non-functional requirements for a tenant-aware workflow engine: extensibility, reusebility, and backwardcompatibility. They are described as follows:

- **Extensibility:** The workflow engine is extendable when the implementation takes into consideration future extensions of the engine. There are two kinds of extensions, adding new functionality to the engine or modificate existing functionality of the engine.

- **Reusability:** The implementation of the workflow engine has to be reusable to reduce implementation time for extensions or modifications.

- **Backward compatibility:** If the workflow engine is tenant-aware and it can receive and handle older non multi-tenant Web service calls, then it is backward compatible.

| Name | Passing the tenant context through the workflow engine to the process instance |
|---|---|
| Goal | The tenant context has to be passed through the system to the starting point of the process instance. |
| Actor | The engine passes the tenant context through the classes which are involved to start the process instance. |
| Pre-Condition | The workflow engine is up and running and the Web service is deployed. The tenant context has to be extracted successfully before. |
| Post-Condition | The workflow engine is waiting for service requests. |
| Post-Condition in Special Case | A notification is sent to the tenant. The workflow engine is waiting for service requests. |
| Normal Case | The tenant context is passed successfully through the system and the process instance is extended to handle this tenant context before the process is started. |
| Special Case | The workflow engine is unable to pass the tenant context through the system and so the process instance is not extended to handle the tenant context before the process is started. It notifies the tenant and/or administrative staff thereof. Noneetheless the engine has to be able to accept further service requests. |

**Table 4.2:** Use Case: Passing Tenant Context Through the Workflow Engine to the Process Instance

## 4.2 Orchestra^MT Model

There are two fundamental multi-tenant models. The one is the Multi-tenant PaaS Model as shown in Figure 3.5. Examples of this model are ActiveVOS and WSO2 Stratos and have been already described in Section 3. Therein multi-tenancy is realized from the customer's point of view. In both systems, multi-tenancy support is provided by sharing applications between different tenants. The tenants are partitioned in discrete groups with access to hardware and software. In Contrast, in the Orchestra multi-tenant model, each tenant gets his own specific process instance as shown in Figure 4.2. Therein each tenant calls the Web service offering the business process to the outside on the same endpoint of the engine. The open source BPEL engine Orchestra should be modified to create these tenant specific business process instances. In this model, multi-tenancy is realized from the point of view of the supplier. After the modification, the multi-tenancy supporting Orchestra engine is called Orchestra^MT.

| Name | Receive Service Response inclusive tenant context from the workflow engine |
|---|---|
| Goal | The tenant context has to be added to the outgoing message header. |
| Actor | The workflow engine sends a response message including the tenant context to the tenant. |
| Pre-Condition | The workflow engine is up and running and the Web service is deployed. The tenant context has to be available in the defined system variable after the process is executed. |
| Post-Condition | The workflow engine is waiting for service requests. |
| Post-Condition in Special Case | A notification is sent to the tenant. The workflow engine is waiting for service requests. |
| Normal Case | The tenant context is added to the outgoing response message header and the response message is sent to the tenant. |
| Special Case | The workflow engine is unable to add the tenant context to the outgoing message header and notifies the tenant and/or administrative staff thereof. Nevertheless the engine has to be able to accept further service requests. |

**Table 4.3:** Use Case: Send Service Response Including Tenant Context from the Workflow Engine

## 4.3 The Tenant Context

To achieve multi-tenancy in a composition engine, tenant specific process instances are needed. Due to this fact, with the help of a tenant context the needed information about a tenant can be kept. The tenant specific process instances in a tenant-aware composition engine can be realized with the tenant specific metadata and functionality. To receive this tenant specific metadata and functionality, each tenant has his own unique tenant identifier (TID). This TID is part of the tenant context. The tenant context is added to the header of a message protocol to transfer it from the tenant to the composition engine. The engine can extract the tenant context and use the information about the tenant to create the tenant specific instance.

### 4.3.1 Tenant Context Structure

The structure of the tenant context is compact. It consists of the two elements tenant identifier (TID) and user identifier (UID) and a arbitrary number of optional entries. This structure is represented graphically in Figure 4.3. The TID is a Universally Unique Identifier (UUID) which represents a company or a department. On the basis of the TID and its associated metadata, the engine can invoke the tenant specific process instance. The UID is a Universally Unique Identifier which represents the stakeholders of the associated company or department. The tenant context is implemented as XML Schema Definition (XSD) as shown in Listing 4.2. An example tenant context in a SOAP header is shown in Listing 4.1. To support consistency,
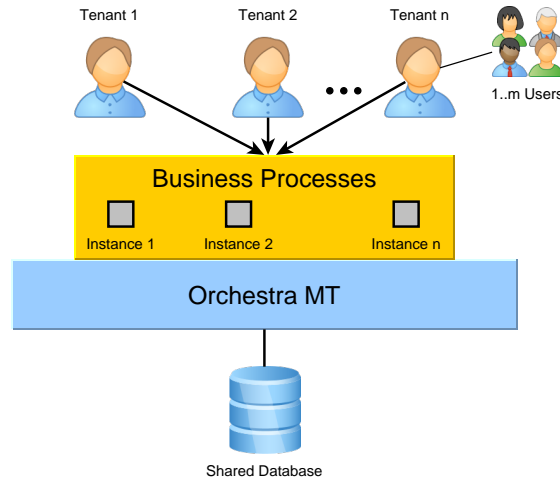
**Figure 4.2:** The Orchestra Multi-Tenant Model

the tenant context structure of this work is similar to the work done by Stefan Essl [Ess11]. The important thing for future work is the extensibility and reuseability of the tenant context. As a result of this, the tenant context definition in Listing 4.2 has an optional entry including a key and a value. The *optionalEntry* can be used as often as desired. The used communication message protocol in this diploma thesis is the SOAP message protocol. It is based on XML and so the tenant context defined in Listing 4.2 can be easily integrated. As shown in Section 2.1.1 a SOAP message consists of a header and body element. The SOAP message body contains the payload and does not care in our approach. It does not make sense to add the tenant context to the body of the SOAP message, because this part of the message includes the payload for the Web service. The called Web service should not be affected by this additional information. Due to this fact the tenant context is added to the SOAP message header.

**Listing 4.1:** A SOAP Header with Tenant Context

```
<soapenv:Header>
  <tenantContext>
      <tenantId>16c2025386054b679001935c50c8b707</tenantId>
      <userId>dc0b71dd4c994efb964bbc30efd552cc</userId>
      <optionalEntry>
         <key>tenantName</key>
         <value>Example Inc.</value>
      </optionalEntry>
      <optionalEntry>
         <key>userEmailAddress</key>
         <value>user@example.org</value>
      </optionalEntry>
  </tenantContext>
  <anotherHeader>
      <element1>value1</element1>
      <element2>value2</element2>
      <element3>value3</element3>
  </anotherHeader>
</soapenv:Header>
```

**Listing 4.2:** The XML Schema Definition of Tenant Context

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="UUIDType">
        <xsd:simpleType>
            <xsd:restriction base="ID">
                <xsd:pattern
                    value="[af09]{8}[af09]{4}[af09]{4}[af09]{4}[af09]{12}" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>

    <xsd:group name="tenantUserId">
        <xsd:sequence>
            <xsd:element name="tenantId" ref="UUIDType" />
            <xsd:element name="userId" ref="UUIDType" />
        </xsd:sequence>
    </xsd:group>

    <xsd:element name="tenantContext">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:choice>
                    <xsd:element name="tenantContextKey" ref="UUIDType" />
                    <xsd:group ref="tenantUserID" />
                </xsd:choice>
                <xsd:element name="optionalEntry" minOccurs="0" maxOccurs="unbounded"
                    >
                    <xsd:complexType>
                        <xsd:element name="key" type="xsd:string" />
                        <xsd:element name="value" type="xsd:anyType" />
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Another concept of the tenant context structure is only to add a reference of the tenant context information to the header of the SOAP message. The referenced tenant context information can be stored in an external database. This can be useful to handle voluminous tenant context information in future work. Because larger tenant context information can unnecessarily expand the message and, for example affect, the message transmission time. The procedure is as follows: The composition engine only extracts the tenant context reference and retrieve the whole tenant context information from the external database. But the tenant context structure in this work is very compact and because of this no reference to the tenant context information in an external database is used. Instead, for the sake of simplicity, the first concept to add the tenant context directly to the header of the SOAP message is used and implemented.
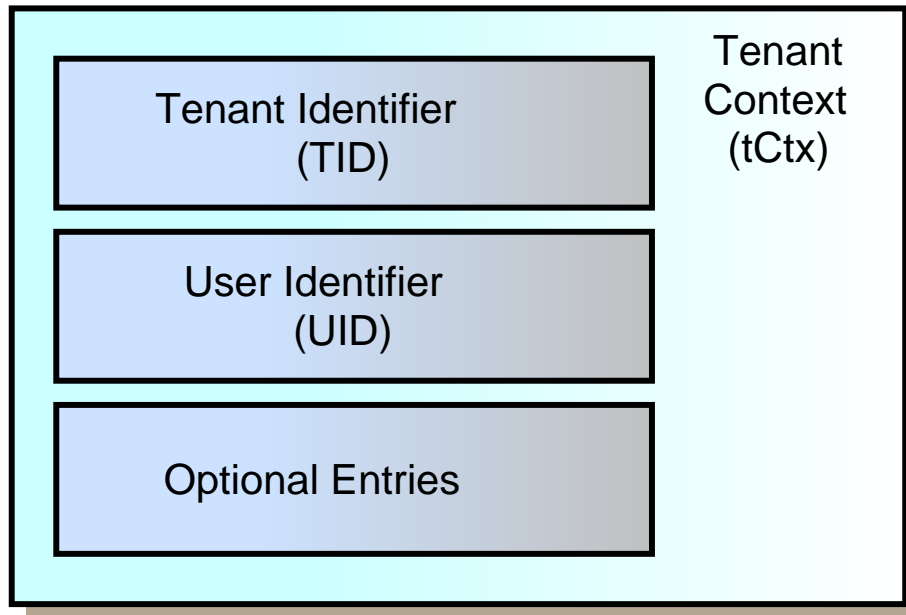
**Figure 4.3:** The Tenant Context Structure

### 4.3.2 Tenant Context Life Cycle

The tenant context described in the last Subsection goes through different steps as shown in Figure 4.4. At the beginning of the life cycle, the header of the incoming SOAP message includes the tenant context. In the first step the tenant context is extracted from the SOAP header by Orchestra$^{MT}$ and stored in a system variable. Secondly, the engine invokes the process instance and the instance handles the tenant context information. Finally, the tenant context is added to the header of the outgoing SOAP message. This could be a direct reply message to the tenant or an invocation message of an Web service.

### 4.3.3 Requirements for the Tenant Context Concept

The tenant context concept in this diploma thesis has to fulfill the two requirements reuseability and extensibility. This means that the concept is reusable and extensible for another or future work in this area. To realize this and keep the reuseability and extensibility, the structure of the tenant context cannot be static. At the moment there are only two elements in the tenant context, `tenantId` (TID) and `userId` (UID). But it is easy to add elements to the tenant context object by adding further `optionalEntry` elements to the header of the incoming SOAP message and parse those new elements, too.
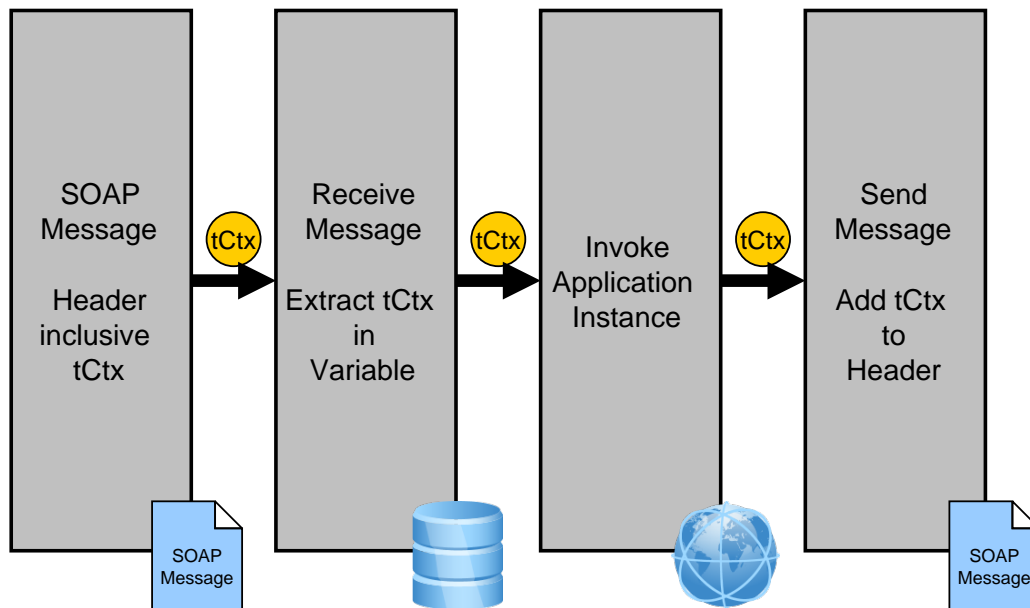
**Figure 4.4:** The Tenant Context Life Cycle

## 4.4  Extract Tenant Context from Incoming SOAP Message

The first step is to get the input message, extract the tenant context from the SOAP header, and store it in a system variable. The concept is shown in Algorithm 4.1. Therein the tenant context header in the SOAP message is searched and every element of this tenant context header is extracted to a set $M$ of tenant context header elements if the local name is *tenantId*, *userId*, or *optionalEntry*. The *valid* function in Algorithm 4.1 validates the extracted *tenantContextElement*. When the local name of the element is *optionalEntry*, the therein contained *key* and *value* information is extracted. Thereby, the extensibility of the tenant context for future work is guaranteed.

The processing in the original Orchestra source code is shown in Figure 4.5. Therein (1) first an CXF input message object is created. (2) Then the message content is extracted from the CXF input message to a *DOMSource list*. The Orchestra engine has it's own input message structure. (3) Due to this fact, an Orchestra input message has to be created from the CXF message. The default Orchestra processing has to be extended as follows: When the message content is extracted, the tenant context has to be extracted from the SOAP message header into a variable. The reuseability and extensibility described in Subsection 4.3.3 is achieved by the *optionalEntry* of the tenant context.

---

**Algorithm 4.1** Extract the Tenant Context Elements from the Message Header

$M := \varnothing$
**for all** $(headerElement \in soapHeader)$ **do**
  **if** $(headerElement = tenantContext)$ **then**
    **for all** $(tenantContextElement \in headerElement)$ **do**
      **if** $(valid(tenantContextElement) = true)$ **then**
        $M \leftarrow tenantContextElement$
      **end if**
    **end for**
  **end if**
**end for**

---
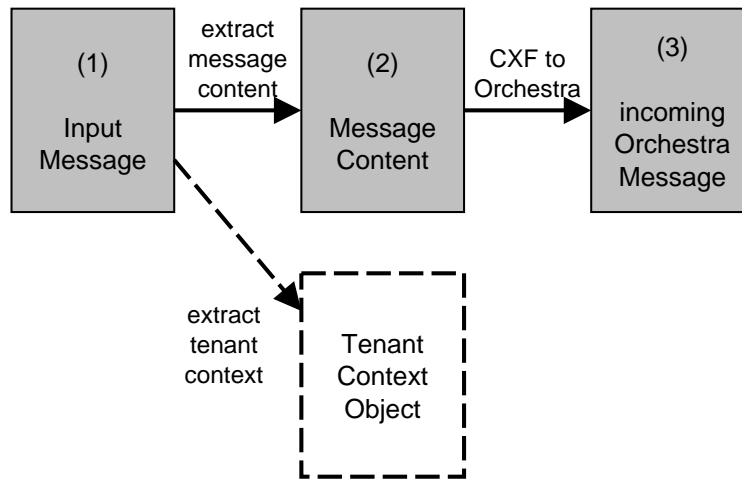


**Figure 4.5:** The Incoming Message Processing

## 4.5 Extend Process Instance to Manage Tenant Context

When the Web service offering the business process to the outside is called, the created BPEL process instance has to be extended to handle the tenant context. The tenant context is needed to realize the concept of a tenant specific instance of a BPEL process. The processing in the original Orchestra engine source code to create the process instance is shown in Figure 4.6. Therein (2) an instance of the BPEL execution is created with the input data of (1a) the incoming Orchestra message. When it is a two way operation, a message carrier object is created including a random Universally Unique Identifier (UUID). (1b) This message carrier is additional input data of the created BPEL execution instance. If there is no two way operation, no message carrier is created and added. The original Orchestra source code must be modified as follows: Whether it is a two way operation or not, a message carrier object is created. The engine must be able to handle the tenant context data within the message carrier object. Therefore, a get and set method in the implementation class of the message carrier interface is needed to add the tenant context to the message carrier.
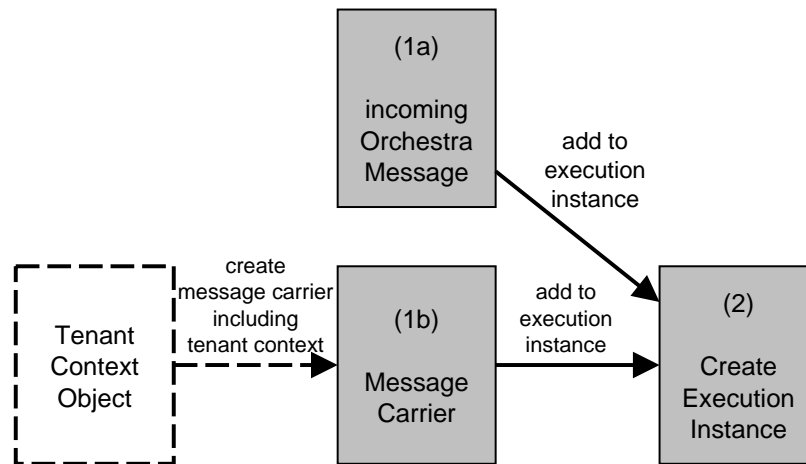
**Figure 4.6:** Create Process Instance

Figure 4.7 shows the internal flow of the execution after the process instance is created. The individual processing steps are described as follows: (1) After the process instance is created, the process is executed. The first step is to pass the message carrier object (MC) including the tenant context (tCtx) (2) with the help of a handle method to the starting point of the execution. Along this path the message carrier is added to a pending message object (PM). (3) At the starting point of the execution, the tenant context is extracted from the pending message object and added to the current execution. Therefore, methods to get and set the tenant context in the BpelExecution class have to be created. When the process gets started, the execution object (EX) including the tenant context is passed to the execution of the current specific BPEL process activity. (4) The executeActivity method is implemented by every activity and has to be extended to handle the tenant context. This means that the tenant context must be maintained in new created executions, like child executions. Therefore, the get and set method in the BPEL execution is used. The current parent execution provides the tenant context over its get tenant context method and the child execution gets the tenant context over its set tenant context method. When the executed BPEL activity is the invoke activity, the current execution object including the tenant context is stored in an InvokeJob object by a set method. This execution is fetched with the help of the corresponding getExecution method. (5) The invocation is executed by the executeInvoke method. If it is a two way operation, the tenant context inside the execution object is stored in a new message carrier object. The handle method in step (2), used on the way from the creation of the instance to the starting point of the execution, is called with this new message carrier object including the tenant context. This is the starting point of a cycle as long as synchronous invocations are processed. (6) When it is a one way operation, no message carrier object is created. The tenant context is extracted from the current execution and passed to the CxfInvoker class. Therein the CXF client is created and invokes the asynchronous Web service.
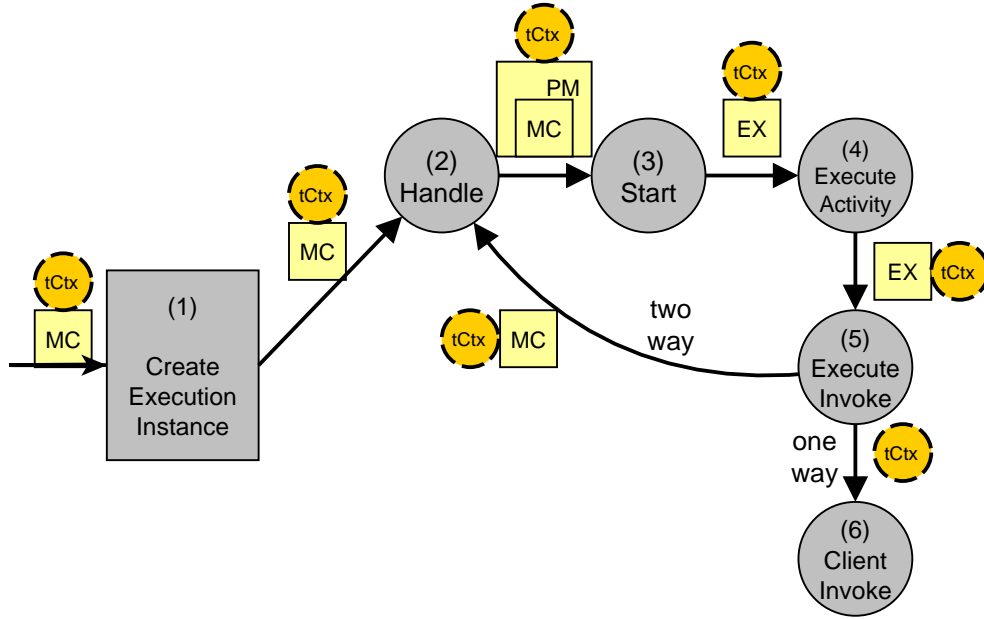
**Figure 4.7:** The Execution Flow

## 4.6 Include the Tenant Context in Outgoing SOAP Messages

The next step is to create the outgoing message and add the tenant context to its header. This concept is shown in Algorithm 4.2, where every tenant context element *tenantContextElement* of the tenant context map consists of a key-value mapping. For each element *tenantContextElement* a node *tenantContextElementNode* is created and added to the tenant context root node *tenantContextRootNode*. Finally, this tenant context root node is added as one header node to the header *MessageHeader* of the outgoing SOAP message.

---

**Algorithm 4.2** Include the Tenant Context Elements to the Message Header

---

$M := \{tenantContextElement \mid tenantContextElement \in tenantContext\}$
**for all** (*tenantContextElement* $\in M$) **do**
   *tenantContextElementNode* $\leftarrow$ *tenantContextElement*
   *tenantContextRootNode* $\leftarrow$ *tenantContextElementNode*
**end for**
*MessageHeader* $\leftarrow$ *tenantContextRootNode*

---

There are two variants of adding the outgoing message header. One for the synchronous communication and the other for the asynchronous communication. Both are realized with the help of an own implemented Apache CXF interceptor. The interceptor adds the tenant context information to the header of the outgoing message in the write phase of the CXF message processing. The two variants are described as follows:

- **Synchronous Variant**
  In the original Orchestra source code, (2) the CXF outgoing message is created from the (1) Orchestra outgoing message. The default Orchestra procedure has to be modified by adding the tenant context information to this CXF outgoing message. This is shown in Figure 4.8.
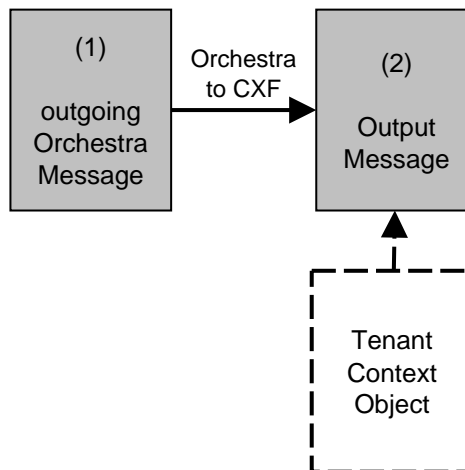


**Figure 4.8:** The Synchronous Outgoing Message Processing

- **Asynchronous Variant**
  In the original Orchestra source code, (2) first the CXF outgoing message is created from the (1) Orchestra outgoing message. (3) Then the Apache CXF client is invoked with the help of the created CXF outgoing message. The default Orchestra procedure has to be modified as follows: When the CXF outgoing message is created, the tenant context has to be added to it. Then the Apache CXF client is invoked with the modified CXF outgoing message. The processing and the modified step is shown in Figure 4.9.
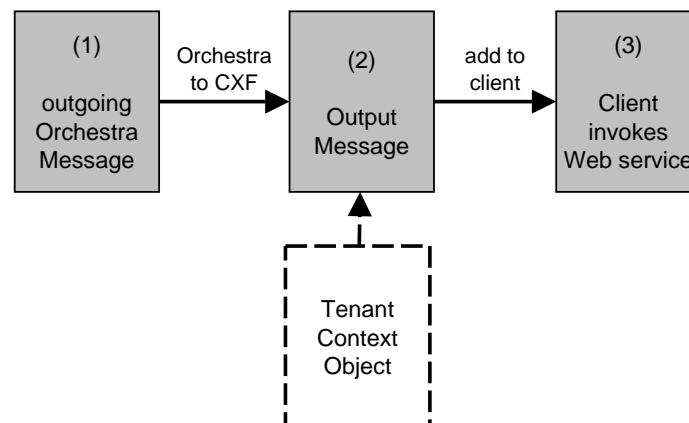


**Figure 4.9:** The Asynchronous Outgoing Message Processing

## 4.7 Save and Load the Tenant Context Data

The tenant context has to be stored in a database because the process instance including the tenant context is only in the system cache and can be lost when other process instances are being used. In this case, the process instance including the tenant context is fetched from the database. The backup of the process instance is done by the object-relational mapping (ORM) library for java called hibernate[1]. To realize this, the tenant context is converted to a string and this string is defined in the hibernate XML schema. The schema is mapped to the relational database as the other data of the Orchestra engine.

## 4.8 External Usage and Internal Handling

There are several input and output representations of the tenant context in the Orchestra$^{MT}$ engine. They are categorized in two main layers as shown in Figure 4.10. The two layer are described as follows:

- **External tenant context usage layer**
  Therein are the various types of the external tenant context usage like testing and logging outputs of the tenant context, input and output messages containing the tenant context, insertion and selection of the tenant context to/from database, and the optional output of the tenant context as file (for example as an xml file).

- **Internal tenant context handling layer**
  Includes the three main internal representations of the tenant context in the system: The tenant context object as main handled object in the system and the tenant context as String representation and as *DOMSource* as alternative types.

In the outer layer of Figure 4.10 there are the various components which use the tenant context in a different way. On the left side the tenant context handling of the in- and outgoing messages to and from the system is shown. Both messages have the tenant context in their header. The tenant context of the incoming message is extracted and stored in a created tenant context object variable as described in Section 4.4. The values, like TID and UID, of the tenant context object variable are added to the outgoing message as described in Section 4.6. For tests and logging of the system as shown in the upper part in Figure 4.10, the tenant context object has to be converted into a string. To save and load the tenant context via hibernate to and from a relational database as described in Section 4.7, the tenant context object has to be converted into a string. These two database external tenant context usages are shown on the right side in the outer layer of Figure 4.10. The last and optional part in the external layer is the output of a tenant context file as shown in the lower part of Figure 4.10. For this, the main tenant context object has to be converted to a string like for the database or testing/logging components. The internal layer of Figure 4.10 shows the internal tenant context as concept. Furthermore, the relationship between the internal tenant context and the components of the

---

[1]http://www.hibernate.org/

external tenant context usage layer is shown. The conversion between the different tenant context representations should be done by a tenant context handler.
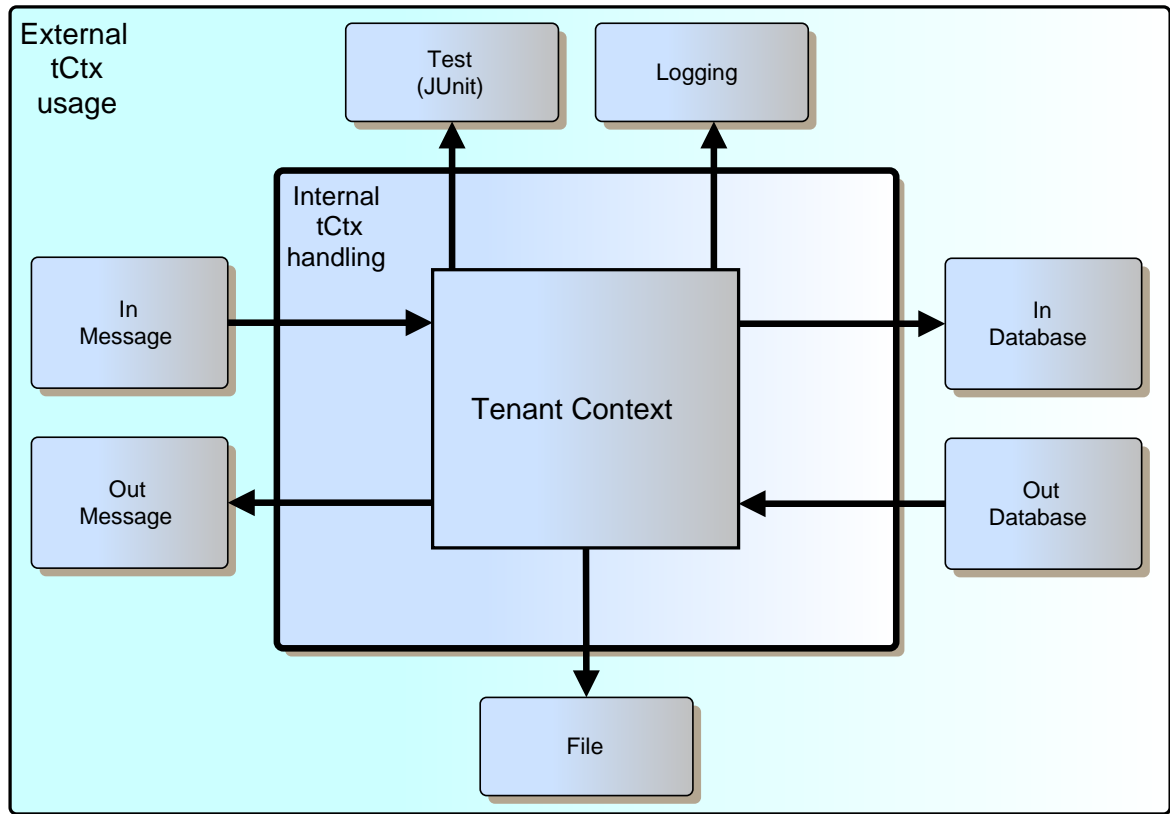


**Figure 4.10:** The Internal Tenant Context Handling and External Usage

# 5 Design and Implementation

In this Chapter, the concepts presented in the previous Chapter 4 are transferred into practice. First, the modified parts of the Orchestra architecture are shown and described. The second part of this Chapter covers the extension of the open source BPEL engine Orchestra with the tenant context concept presented in the Chapter before. This covers the implementation of the tenant context and the needed and used tenant context handlers. Furthermore, the implementation of the tenant context lifecycle phases, extracting the tenant context from the message header to a system variable, extending the process instance to handle the tenant context, and adding the tenant context information to the header of the outgoing message is presented.

## 5.1 Development Environment

The first step of the implementation was to set up a development server which includes the open source Orchestra BPEL engine version 4 (revision 6547). The used Integrated Development Environment (IDE) was Eclipse version 3.7.0 with Java 6. The graphical open source functional testing solution soapUI version 4.0.0 was used to address the deployed Web services. Apache Tomcat is an open source servlet container developed by the Apache Software Foundation (ASF) and a open source software implementation of the Java Servlet and JavaServer Pages (JSP) technologies. Apache Tomcat version 6.0.33 is used to run the open source Orchestra BPEL engine. Apache CXF is a open source Web service framework and used by the Orchestra BPEL engine. It supports SOAP and a lot of the other WS-* specification standards and is described in the previous Section 2.1.2. Apache Maven is used as software building and comprehension tool to build the Orchestra BPEL engine. A Guide how to build and run the Orchestra Engine and how to deploy and address a Web service with soapUI is included in the Appendix Chapter A.

## 5.2 Orchestra Architecture

The whole Orchestra architecture was already shown in Figure 2.2. The modified parts of the architecture are marked by the thick boundry in Figure 5.1. The first modified part is on the right side and called Web service. It includes the Web service framework Apache CXF already explained in Section 2.1.2. In this part the incoming and outgoing messages are modified to handle the tenant context inside their message header. This is realized with the help of the Apache CXF Interceptors (Section 2.1.2). Another modificated part is the Orchestra Invoker. The BPEL process instance is extended to handle the tenant context on invoke a Web service. A lot of Orchestra core classes are involved in the tenant context passing process and therefore listed in Section 5.3.4. Furthermore, the whole passing process is described there.
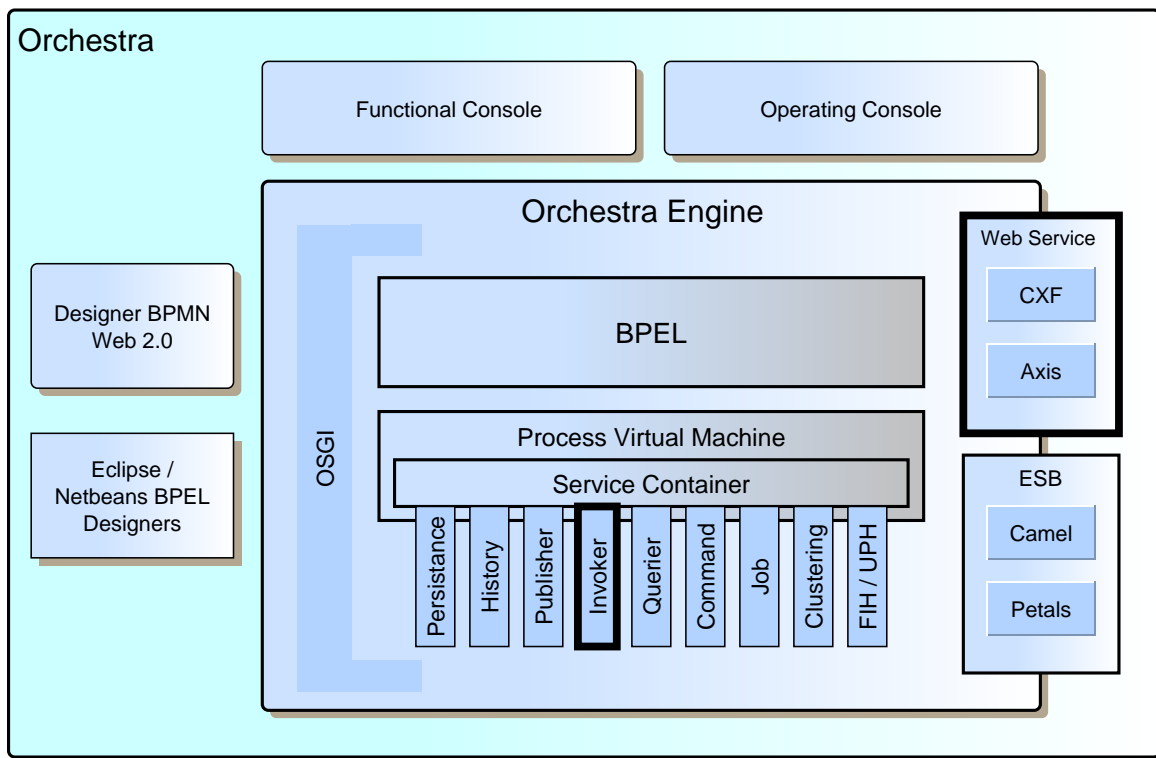


**Figure 5.1:** Modified Parts of the Orchestra Architecture

## 5.3 Extending Orchestra

One of the main tasks of this diploma thesis is to extend the Orchestra BPEL engine to enable multi-tenancy in communication. This means to extract the tenant context from the incoming message, extend the process instance to handle the tenant context, and include the tenant context in the outgoing message. The source code of the engine is written in Java. The extension of the engine is done by the modification of existing classes and the creation of new classes.

### 5.3.1 Tenant Context Class

The `TenantContext` class implements the defined tenant context. It contains a hash map for the tenant context elements like the TID and UID. The hash map is a mapping between the key of a tenant context element and its value. Each of the tenant context elements extracted from the incoming message header can be added to the tenant context hash map by the `addTenantContextElement()` method. Because of this hash map the tenant context is extensible in further work. The most important elements of the tenant context described in this diploma thesis are the TID and UID. Due to this fact, there are two get methods `getTenantID()` and `getUserID()` to support direct access to this elements. But there is also the possibility to get the whole tenant context map from the tenant context object by the get method `getTenantContextMap()`.

**Listing 5.1:** TenantContext.java Class

```java
package org.ow2.orchestra.services;
...
public class TenantContext {
  final String tenantIdString = "tenantId"; final String userIdString = "userId";
  private Map<String, String> tenantContextMap = new HashMap<String, String>();

  public void addTenantContextElement(String inElementNameString, String
      inElementValueString) {
    if ( (inElementNameString != null) && (inElementValueString != null) )
      this.tenantContextMap.put(inElementNameString, inElementValueString);
  }

  public Map<String, String> getTenantContextMap() {
    return this.tenantContextMap;
  }

  public String getUserID() {
    return this.getTenantContextElement(userIdString);
  }

  public String getTenantID() {
    return this.getTenantContextElement(tenantIdString);
  }
}
```

**5.3.2 Tenant Context Handler Class**

In the `TenantContextHandler` class, the methods to handle the different types of the tenant context in the several lifecycle phases are implemented. The main methods are:

- `stringToTenantContext()`: Converts an XML string into the tenant context object. This method is used when the tenant context information is loaded from the database.

- `tenantContextToString()`: Converts the tenant context object into an XML string. This method is used when the tenant context information is saved in the relational database by using hibernate.

- `tenantContextToXml()`: Converts the tenant context object into an XML DOM-Source object. This method is very useful for the conversion from the tenant context to the XML string. The first step is to convert the tenant context to a DOMSource. This DOMSource can be easily converted to a XML String or used to add the tenant context to the header of the outgoing SOAP message.

- `xmlToTenantContext()`: Converts an DOMSource object into the tenant context object. This method is very useful to create a tenant context object from the DOMSource structure and its elements.

- `headerToTenantContext()`: Converts the extracted header object into the tenant context. Therein the key and value of each tenant context element are extracted into the hash map of the tenant context object.

- `tenantContextToElement()`: Converts the tenant context object into a W3C Dom element. This element is added as child node to the header of the outgoing message.

**5.3.3 Extract Tenant Context**

The first step in the tenant context life cycle is to extract it from the incoming SOAP message header. This is shown in Listing 5.2. To realize this, first the incoming message in the `CxfWSImpl` class is casted into a SOAP message. Secondly a new tenant context object is created. Then the defined tenant context header is extracted and the child nodes are added to the tenant context object with the `addTenantContextElement()` method described in Section 5.3.1. The key in the tenant context hash map is the local name of the child node and the value is the text content of the respective child node.

**Listing 5.2:** Extract Tenant Context Part in the CxfWSImpl.java Class

```java
package org.ow2.orchestra.cxf;
...
import org.apache.cxf.binding.soap.SoapMessage;

public final class CxfWSImpl implements org.apache.cxf.service.invoker.Invoker {
   TenantContext tenantContext = new TenantContext();
   TenantContextHandler tCtxH = new TenantContextHandler();
   ...
   public DOMSource[] invoke(final Exchange exchange, final Object payload) {
      ...
      final Message inMessage = exchange.getInMessage();
      ...
      SoapMessage soapMessage = (SoapMessage) inMessage;
         List<Header> soapHeaderList = soapMessage.getHeaders();

         if (soapHeaderList != null) {
         for (Header h: soapHeaderList) {
            Object obj = h.getObject();
            tCtxH.headerToTenantContext(obj, tenantContext);
         }
      }
      ...
   }
   ...
}
```

### 5.3.4 Extend Process Instance

The next step in the tenant context life cycle is to extend the BPEL process instance to handle the tenant context. After the extraction of the tenant context from the incoming SOAP message header, the created tenant context object has to be passed through the execution flow of the BPEL execution instance. The following Java classes are involved in the process flow:

- `org.ow2.orchestra.cxf.CxfWSImpl.java`
- `org.ow2.orchestra.cxf.CxfInvoker.java`
- `org.ow2.orchestra.services.MessageCarrier.java`
- `org.ow2.orchestra.services.MessageCarrierImpl.java`
- `org.ow2.orchestra.services.ReceivingService.java`
- `org.ow2.orchestra.services.AsyncAssociateMessage.java`
- `org.ow2.orchestra.services.PendingMessage.java`
- `org.ow2.orchestra.services.Receiver.java`
- `org.ow2.orchestra.services.InvokeExecutor.java`
- `org.ow2.orchestra.services.job.ExecuteInvokeJob.java`
- `org.ow2.orchestra.runtime.BpelExecution.java`

First of all, a `MessageCarrier` object is created in the `CxfWSImpl` class and the `TenantContext` object is stored in it. This `MessageCarrier` object including the `TenantContext` object is added to the BPEL process instance. This is shown in Listing 5.3.

**Listing 5.3:** Create Process Instance

```
package org.ow2.orchestra.cxf;
...
public final class CxfWSImpl implements org.apache.cxf.service.invoker.Invoker {
   TenantContext tenantContext = new TenantContext();
   ...
   public DOMSource[] invoke(final Exchange exchange, final Object payload) {
      ...
      MessageCarrierImpl messageCarrier;

      if (operationInfo.hasOutput()) {
         messageCarrier = new MessageCarrierImpl(tenantContext, true);
      }
      else
      {
          messageCarrier = new MessageCarrierImpl(tenantContext, false);
      }
      ...
      final BpelExecution instance = ReceivingService.handle(..., messageCarrier,
          ...);
      ...
      }
}
```

The BPEL process instance is created by calling the `handle` method of the `ReceivingService` class. If there is no existing execution waiting, a new `AsyncAssociateMessage` object without a start date of a parent instance and including the tenant context object is created. Otherwise, the start date of the parent instance is used. The declaration of the `handle` method is shown in Listing 5.4.

**Listing 5.4:** Call Handle Method

```
package org.ow2.orchestra.services;
...
public final class ReceivingService {
   ...
   public static BpelExecution handle(..., final MessageCarrier messageCarrier,
      ...)
   {
      return commandService.execute(
         new AsyncAssociateMessage(operationKey, incomingMessage, messageCarrier,
             lock, parentInstanceStartDate)
      );
   }
   ...
}
```

In the execution of the `AsyncAssociateMessage` class, first the `storeIncomingMessage` method from the `Receiver` class is called to create a `PendingMessage` object. This object

includes the `MessageCarrier` object. If there is a waiting execution, it is received by calling the `getWaitingExecution` method of the `Receiver` class. Otherwise, a new instance is created with the `startNewInstance` method from the `Receiver` class. This method calls the `createNewInstance` and therein, the modified bpelExecution retrieve the tenant context from the transferred pending message. To realize this, a `getTenantContext()` method is added to the `PendingMessage` class and a `setTenantContext` method is added to the `BpelExecution` class. The tenant context is added to the created bpelExecution before the process is started as shown in Listing 5.5.

**Listing 5.5:** The Process Start

```
package org.ow2.orchestra.services;
...
public class Receiver {
   ...
   private static ExecElementToSignal createNewInstance(
      final PendingMessage pendingMessage,
      ...) {
      ...
      bpelExecution.setTenantContext(pendingMessage.getTenantContext());
      ...
      bpelExecution.begin();

      return Receiver.getStartElementExecution(bpelExecution, startElement,
         pendingMessage);
   }
   ...
}
```

After this, the `BpelExecution` instance is created in the `CxfWSImpl` class. This instance includes the tenant context. When the process execution begins, the current specific BPEL process activity has to be executed by the `executeActivity` method. The `executeActivity` method is implemented by every BPEL activity. A list of all activities which implement the `executeActivity` method is shown in Figure 5.2.

The `executeActivity` methods create new executions, like child or parent executions. To maintain the tenant context, the `getTenantContext` and `setTenantContext` methods of the BPEL execution are used. When the executed BPEL activity is the invoke activity, the current execution object including the tenant context is stored in an `InvokeJob` object by a `setExecution` method. This execution is fetched with the help of the corresponding `getExecution` method in the `execute` method of the `ExecuteInvokeJob` class. Then the `executeInvoke` method of the `InvokeExecution` class receives the execution object. If its a two way operation, a new `messageCarrier` object is created and the tenant context of the execution object is added to it. The previously described `handle` method is called with the message carrier as input data as shown in Listing 5.4. When its a one way operation, the tenant context is extracted from the current execution and passed to the `CxfInvoker` class. Therein, the CXF client is created and invokes the asynchronous Web service.
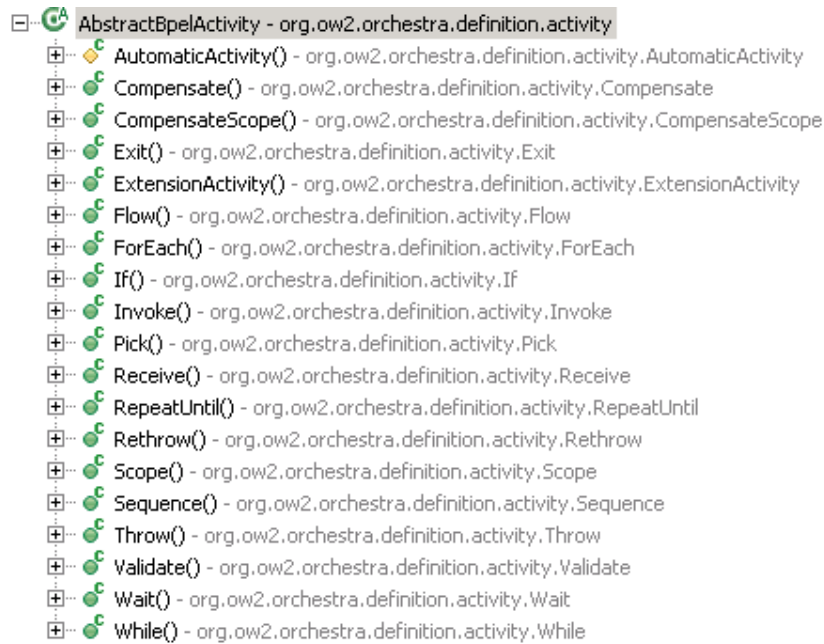
**Figure 5.2:** The BPEL Activities

### 5.3.5 Include Tenant Context

The last step in the tenant context life cycle is to add the tenant context to the header of the outgoing message. There is a need here to distinguish between calling a synchronous or asynchronous Web service:

- **Synchronous**
  If the process is called synchronously, a direct response message is replied. To add the tenant context information to the header of the outgoing message, a interceptor as shown in Listing 5.8 is used. This interceptor handles the outgoing message by adding the header to it. The `CxfWSImpl` class is modified by adding the interceptor to the InterceptorChain as shown in Listing 5.6.

**Listing 5.6:** Add Interceptor to InterceptorChain in the CxfWSImpl.java Class

```
package org.ow2.orchestra.cxf;
...
public final class CxfWSImpl implements org.apache.cxf.service.invoker.
    Invoker {
  ...
  public DOMSource[] invoke(final Exchange exchange, final Object payload) {
    ...
    if (operationInfo.hasOutput()) {
      ...
      exchange.getEndpoint().getOutInterceptors().add(
                new AddTctxHeaderInterceptor(tenantContext));
      exchange.getEndpoint().getOutInterceptors().add(
```

```
                       new LoggingOutInterceptor());
      exchange.getEndpoint().getOutFaultInterceptors().add(
                   new AddTctxHeaderToFaultMsgInterceptor(tenantContext));
      exchange.getEndpoint().getOutFaultInterceptors().add(
                   new LoggingOutInterceptor());
      ...
      return CxfMessageUtil.orchestraToCxfMessage(responseMessage,
          operationStyle, operation.getOutput().getMessage());
   }
   ...
   }
   ...
}
```

- **Asynchronous**

  The process can be called asynchronously. To add the tenant context information to the header of the outgoing message which invokes the Web service, the `CxfInvoker` class is modified as shown in Listing 5.7. The tenant context information is retrieved from the current BPEL instance and added to the header of the outgoing message by using an interceptor as shown in Listing 5.8. This interceptor handles the outgoing message by adding the header to it. The interceptor is added to the created cxf client as shown in Listing 5.7.

**Listing 5.7:** Add Interceptor to CXF Client in the CxfInvoker.java Class

```
package org.ow2.orchestra.cxf;
...
public class CxfInvoker implements Invoker {
   ...
   public MessageVariable invoke(final TenantContext tenantContext, ..)
   {
      final Client client = this.createCxfClient(operationKey,
          addressingInfo, definition, service, port);
      ...
      final DOMSource[] outMessage = CxfMessageUtil.orchestraToCxfMessage(
          requestMessage, operationStyle, operation.getInput().getMessage());
      ...
         client.getOutInterceptors().add(new AddTctxHeaderInterceptor(
            tenantContext));
         client.getOutInterceptors().add(new LoggingOutInterceptor());
            client.getOutFaultInterceptors().add(new
               AddTctxHeaderToFaultMsgInterceptor(tenantContext));
         client.getOutFaultInterceptors().add(new LoggingOutInterceptor());
      ...
      final Object[] res = client.invoke(operationKey.getOperationName(), (
          Object[]) outMessage);
      ...
   }
   ...
}
```

The used interceptor to add the tenant context information to the header of the `OutMessage` is shown in Listing 5.8. First, the interceptor receives the tenant context object. After this, it handles the message in the outgoing `WRITE` phase. The tenant context object is converted to a DOMSource Element by using the above described `tenantContextToElement()` method from the `TenantContextHandler` class. This DOMSource Element is added to the header list and this header list is added to the `OutMessage`. The interceptor to add the tenant context information to the header of the fault message is very similar. The only difference is that the `OutFaultMessage` is modified instead of the `OutMessage`.

**Listing 5.8:** The Interceptor to add the Tenant Context to the Message Header

```java
package org.ow2.orchestra.cxf;
...
import org.apache.cxf.message.Message;
import org.w3c.dom.Element;

public class AddTctxHeaderInterceptor extends AbstractPhaseInterceptor<Message> {

    private TenantContext tenantContext;

    public AddTctxHeaderInterceptor(TenantContext tenantContext) {
      super(Phase.WRITE);
      addAfter(SoapPreProtocolOutInterceptor.class.getName());
      this.tenantContext = tenantContext;
    }

    public void handleMessage(Message message) {
        List<Header> outHeaders = new ArrayList<Header>();
        TenantContextHandler tenantContextHandler = new TenantContextHandler();
        Element value = tenantContextHandler.tenantContextToElement(this.
            tenantContext);
        if (value != null) {
         Header outHeader = new Header(new QName(""), value);
         outHeaders.add(outHeader);
         Exchange exchange = message.getExchange();
         Message outMsg = exchange.getOutMessage();

         if(outMsg == null) {
           outMsg = new org.apache.cxf.message.MessageImpl();
           outMsg.setExchange(exchange);
           outMsg = exchange.getEndpoint().getBinding().createMessage(outMsg);
           exchange.setOutMessage(outMsg);
         }
         outMsg.put(Header.HEADER_LIST, outHeaders);
        }
    }
    ...
}
```

### 5.3.6 Export and Import Tenant Context Data

To save the tenant context information, first the data is converted to an (XML-)string with the `tenantContextToString()` method of the `TenantContextHandler` class. Then the string is mapped via hibernate to a relational database in an dedicated column. This new column is created in the database table of the `BpelExecution` class. The associated hibernate xml file is called `bpel.execution.hbm.xml` and modified as shown in Listing 5.9. It is stored in `/orchestra-core/src/main/resources/hibernate/core/`

The two files wherein the database table with the tenant context column will be created are:

- `org.hibernate.dialect.DerbyDialect-create.sql`
- `org.hibernate.dialect.H2Dialect-create.sql`

**Listing 5.9:** The Hibernate Mapping File

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC ...>

<hibernate-mapping package="org.ow2.orchestra.runtime"
   default-access="field" auto-import="false">

  <class name="BpelExecution" table="RUNT_EXECUTION" discriminator-value="E">
  ...
  <property name="xmlTenantContextString" column="TENANT_CONTEXT_"/>
  ...
  </class>
  ...
</hibernate-mapping>
```

### 5.3.7 Issues with Implementation

The implementation of the presented concept is a prototype. More extensive or additional parts of it are classified as future work. This affectes in particular the extension of the process instance to handle the tenant context. In this work, the first step up to a certain depth is done. Lost information of the tenant context during the life cycle was intercepted by stubs. Much effort has been put in maintaining the tenant context information in every step of the execution flow. Despite intensive endeavors, it was not possible to exactly locate the source of the lost tenant context information. To avoid this problem a dummy value was used to simulate and examine the concepts for its functionality. This has been tested by an extensive logging of the execution flow. The results of this logging tests show that the described concept of the execution flow works properly.

## 5.4 Example

To conclude this Chapter, the scenarios of the Motivating Examples Section 1.3 are presented. The two scenarios are the `Synchronous Echo Scenario` and the `Asynchronous Taxi Scenario`.

### 5.4.1 Synchronous Echo Scenario

First, the `Synchronous Echo Scenario` is presented. The associated example WS-BPEL process is shown in Figure 5.3. The process receives an input message of an user. The message including a string in the body and a tenant context information in the header. This string of the incoming message's body is added to the body of the outgoing reply message. The tenant context information is first extracted from the header of the incoming message. Then it is stored in a variable (the tenant context object) and is finally added to the header of the outgoing reply message after the process is executed. This reply message is sent to the process user. The source code of this example process is located in Appendix B.1.
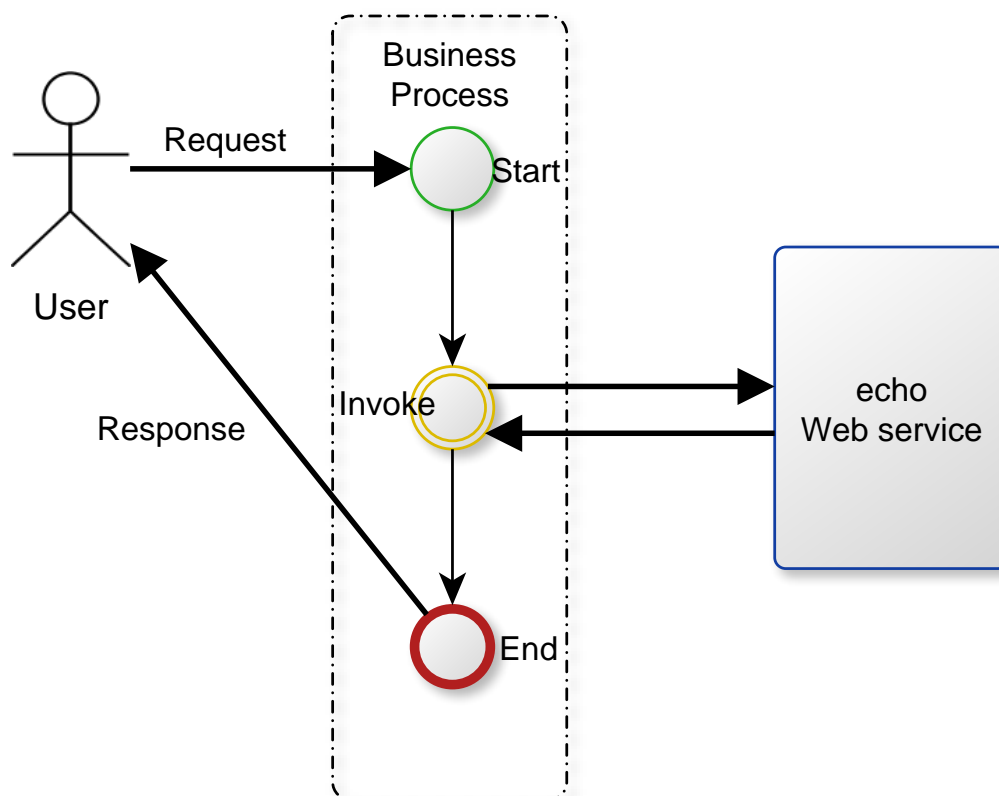


**Figure 5.3:** Synchronous Echo Scenario

### 5.4.2 Asynchronous Taxi Scenario

The other scenario is the `Asynchronous Taxi Scenario`. A user wants to order a taxi to a specific address at a certain point in time. (1) The called taxi company (tenant) receives the taxi order of the user. (2) The process as shown in Figure 5.4 receives an input message from the taxi company with the specific address and point in time information in the body and the tenant context information in the header. The tenant context information is extracted from the header of the incoming message and stored in a variable (the tenant context object). The user has to be notificated, e.g., by a SMS-notification service and he has to confirm this notification within a specified time. (3) The process realize this by calling the tenant preferred SMS-notification Web service on the basis of the tenant context information. (In this approach, the tenant context is added to the header of the Web service calling message). (4) The called Web service notifies the user. After a waiting period due to the (5) user confirmation, (6) the SMS-notification Web service sends a reply message back to the process. The reply message can be, e.g., that the user has confirmed the taxi order process. (7) Finally, the process sends a response message to the taxi company.
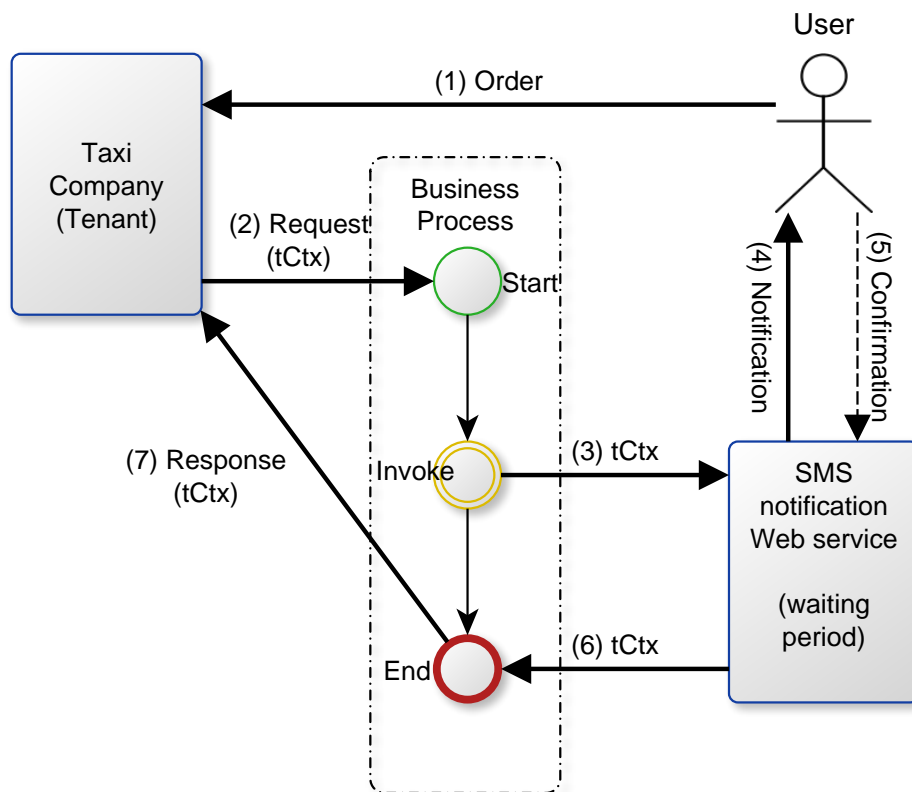


**Figure 5.4:** Asynchronous Taxi Scenario

## 5.5 Test and Test Cases

In this Section, the evaluation and test of the previously presented implementation is described.

### 5.5.1 Test

There are two test variants to realize the tests of the implementation: The first is the JUnit testing framework of Java. Using JUnit, the `TenantContext` and `TenantContextHandler` classes are tested. The second test variant is realized by using soapUI[1] and the previously described logging interceptors. To start, the Orchestra is running and workflows to test are deployed. Testing is done by sending a test request message including the tenant context and analyze the results. The result can be the response message in soapUI, the logging of the request or response message, or the logging of the

### 5.5.2 Test Cases

The described synchronous *echo* scenario is used due to its simplicity to test the addition of the tenant context elements to the outgoing message header. But it is not suitable for the fault cases. As a result of this, the `loan service` (located in Appendix B.2) process is used to test the four essentially test cases: (1) The tenant context element(s) are in the header and the payload is correct, (2) no tenant context element(s) are in the header and a faulty payload is sent, (3) the tenant context element(s) are in the header and a faulty payload is used, and (4) no tenant context element(s) are in the header and the correct payload is sent. This four test cases are shown in Table 5.1. Whereby `valid payload` means that it is a correct request and `invalid payload` means that it is a incorrect request. In the case of the loan service, a valid entry in the amount part of the request message only consists of numbers. The fault case is triggered by adding some characters to it. For simplicity, in Table 5.1, the whole tenant context or valid parts of it inside the request message header are called `valid tenant context`. Analogous to this, `invalid tenant context` means that there is no part of a valid tenant context header. An valid example of a loan Service request in soapUI is shown in Listing 5.10.

| Valid Payload | tCtx Header | Msg Body | Msg Header |
|---|---|---|---|
| false | false | Payload | false |
| false | true | Payload | tCtx Header |
| true | false | Fault Msg | false |
| true | true | Fault Msg | tCtx Header |

**Table 5.1:** The Test Cases

---

[1]http://www.soapui.org/

**Listing 5.10:** Example of a Valid Loan Service Request

```xml
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:loan="http://orchestra.ow2.org/loanApproval/loanService">
    <soapenv:Header>
    <tenantContext>
        <userId>dc0b71dd4c994efb964bbc30efd552cc</userId>
        <tenantId>16c2025386054b679001935c50c8b707</tenantId>
        <optionalEntry>
            <key>userEmailAddress</key>
            <value>user@example.org</value>
        </optionalEntry>
    </tenantContext>
    <anotherHeader>
        <element1>value1</element1>
        <element2>value2</element2>
        <element3>value3</element3>
    </anotherHeader>
    </soapenv:Header>
    <soapenv:Body>
        <loan:request>
            <firstName>Sample</firstName>
            <name>User</name>
            <amount>40000</amount>
        </loan:request>
    </soapenv:Body>
</soapenv:Envelope>
```

For the sake of simplicity, the echo Web service to test the invocation of an asynchronous Web service is used. It is invoked in the asynchronous BPEL process. Like the synchronous echo scenario, it is very simple to test the addition of the tenant context elements to the outgoing message header. In this example there is no reply message of the invoked echo service. Therefore, the outgoing and client invoke message is logged by an logging interceptor. The asynchronous test cases are the same four test cases as in the synchronous test and shown in Table 5.1.

# 6 Summary and Future Work

In this diploma thesis the extension of a BPEL engine for multi-tenancy support in the area of communication was researched. The examination of other work in the area of tenant-aware composition engines and applications presented a number of requirements for workflow engines to support Web services on a per tenant basis, like data, process, and performance isolation. For a better understanding of the work done in this diploma thesis, the needed fundamentals like basic knowledge are given in the Fundamentals Chapter 2. After that, an extensible and reusable concept of an tenant context and how to integrate it into a message protocol is presented in the Concept Chapter 4. Furthermore, the life cycle phases of the tenant context information inside the engine are described.

To validate the concepts presented in the Concept Chapter 4, the open source BPEL engine Orchestra [ORC] was extended to support multi-tenancy in the area of communication. This includes the tenant context handling, like extracting the tenant context from the incoming message, handling the tenant context information inside the system and the process instance, and adding the tenant context to the outgoing messages of the business process. The message protocol used to evaluate the concept of the Tenant Context is SOAP which was described in 2.1.1. The details of the concept implementation are described in the Design and Implementation Chapter 5.

Before the open source BPEL engine Orchestra could be extended, much effort has been put into understanding the original source code and architecture of the engine. The acquired knowledge about developing Orchestra are documented in Appendix A. This knowledge is useful for future developers who want to extend the Orchestra engine.

The work done in this diploma thesis does not cover all the aspects of tenant-awareness on workflow compositions, only those in the area of communication. The future work to realize the whole tenant-aware system is to realize the communication with the tenant-aware Enterprise Service Bus (ESB). For example, the asynchronous Web service described in 5.4.2 is invoked by a message including the Tenant Context. But this invocation should be realized by the ESB. Another future work is to solve the issues of the implementation. Furthermore, Web service providers can make their services tenant-aware using the multi-tenancy patterns presented by Mietzner et. al. [MMLP09, MUTL09, MLP08].
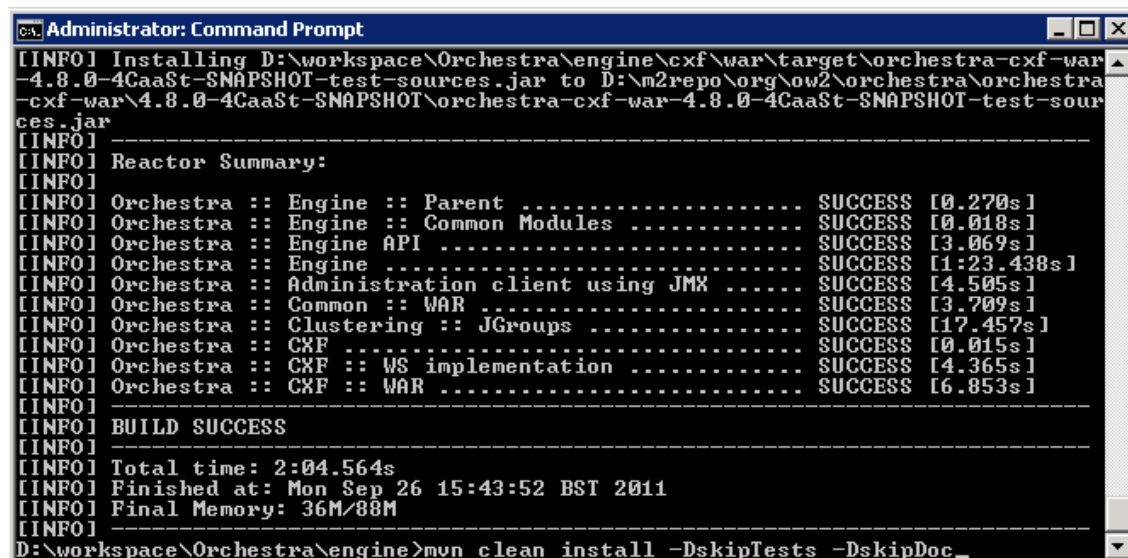
# Appendix A

# Developing Orchestra

In this Appendix Chapter important manuals how to setup the Orchestra engine is presented. This includes manuals how to build and run the engine as well as how to deploy and address a Web service.

## A.1 How to Build and Run the Engine

The first step after the code changes is to build the engine that the changes will take effect. To do this, open the command line and go to the */workspace/Orchestra/engine/* directory. After this enter *mvn clean install -DskipTests -DskipDoc* to repackage the jar files and install it in the maven local repository as shown in Figure A.1. The switches $-DskipTests$ and $-DskipDoc$ can be used to shorten the build. *-DskipTests* will skip test execution and *-DskipDoc* will skip the user guide, javadoc and source-jar creation.



```
[INFO] Installing D:\workspace\Orchestra\engine\cxf\war\target\orchestra-cxf-war
-4.8.0-4CaaSt-SNAPSHOT-test-sources.jar to D:\m2repo\org\ow2\orchestra\orchestra
-cxf-war\4.8.0-4CaaSt-SNAPSHOT\orchestra-cxf-war-4.8.0-4CaaSt-SNAPSHOT-test-sour
ces.jar
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Orchestra :: Engine :: Parent ..................... SUCCESS [0.270s]
[INFO] Orchestra :: Engine :: Common Modules ............. SUCCESS [0.018s]
[INFO] Orchestra :: Engine API ........................... SUCCESS [3.069s]
[INFO] Orchestra :: Engine ............................... SUCCESS [1:23.438s]
[INFO] Orchestra :: Administration client using JMX ...... SUCCESS [4.505s]
[INFO] Orchestra :: Common :: WAR ........................ SUCCESS [3.709s]
[INFO] Orchestra :: Clustering :: JGroups ................ SUCCESS [17.457s]
[INFO] Orchestra :: CXF .................................. SUCCESS [0.015s]
[INFO] Orchestra :: CXF :: WS implementation ............. SUCCESS [4.365s]
[INFO] Orchestra :: CXF :: WAR ........................... SUCCESS [6.853s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2:04.564s
[INFO] Finished at: Mon Sep 26 15:43:52 BST 2011
[INFO] Final Memory: 36M/88M
[INFO] ------------------------------------------------------------------------
D:\workspace\Orchestra\engine>mvn clean install -DskipTests -DskipDoc
```

**Figure A.1:** Build the Engine

After the rebuild of the engine, the next step is to run the engine. This is accomplished by executing *mvn jetty:run-war* in the */workspace/Orchestra/engine/cxf/war* directory as shown in

Figure A.2. Now the Orchestra engine is running on the localhost and port *8080*. To test it, open *http://localhost:8080/orchestra/* in a Web browser. If the engine is running and there are no Web services deployed yet, the text "No services are running" must appear. Otherwise the deployed Web services are listed as shown in Figure A.3.
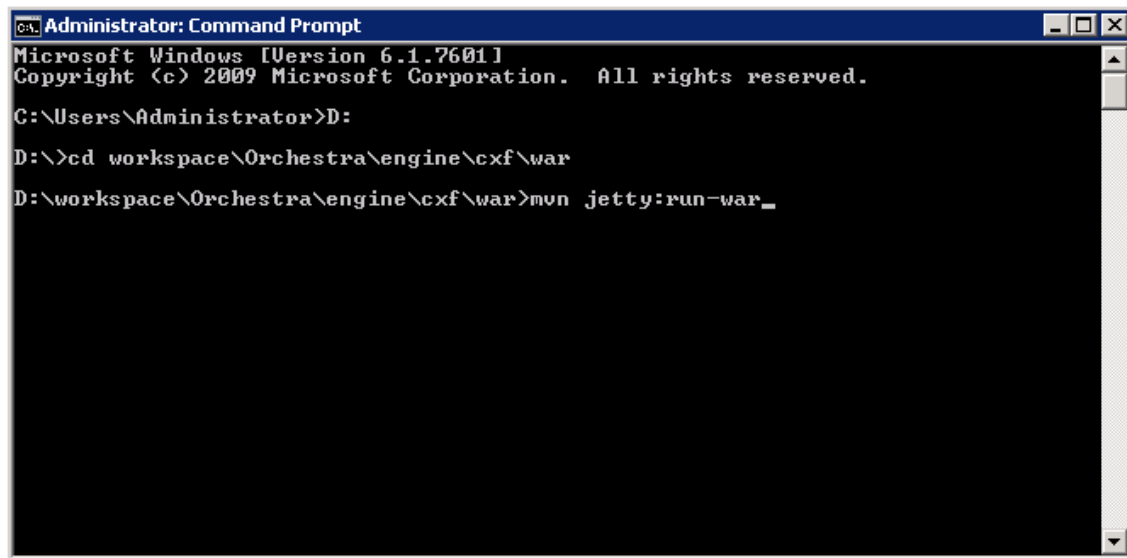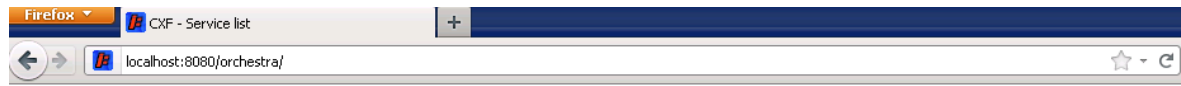


**Figure A.2:** Run the Engine

## A.2  How to Deploy and Address a Web Service

To deploy a Web service, go to the examples directory in */orchestra-cxf-tomcat-4-8-02/*, choose an example Web service and execute *ant deploy* inside the directory of the Web service. For example run *ant deploy* in */orchestra-cxf-tomcat-4-8-02/echo/* to deploy the *echo* Web service as shown in Figure A.4.

Adressing a deployed Web service is performed as follows: Install and open the functional testing solution soapUI[1]. Create a new soapUI project and choose the related wsdl file of the deployed Web service. The binding and a suitable default request message is created automatically by soapUI. Change the endpoint address of the request message to the endpoint address of the deployed Web service as shown at *http://localhost:8080/orchestra/*. Finally the default request message can be submitted to the specified endpoint adress and the return message of the adressed and deployed Web service appears on the right side as shown in Figure A.5.

---

[1]http://www.soapui.org/

**Figure A.3:** Listing of Deployed Web Services

## A.3 Orchestra Logging
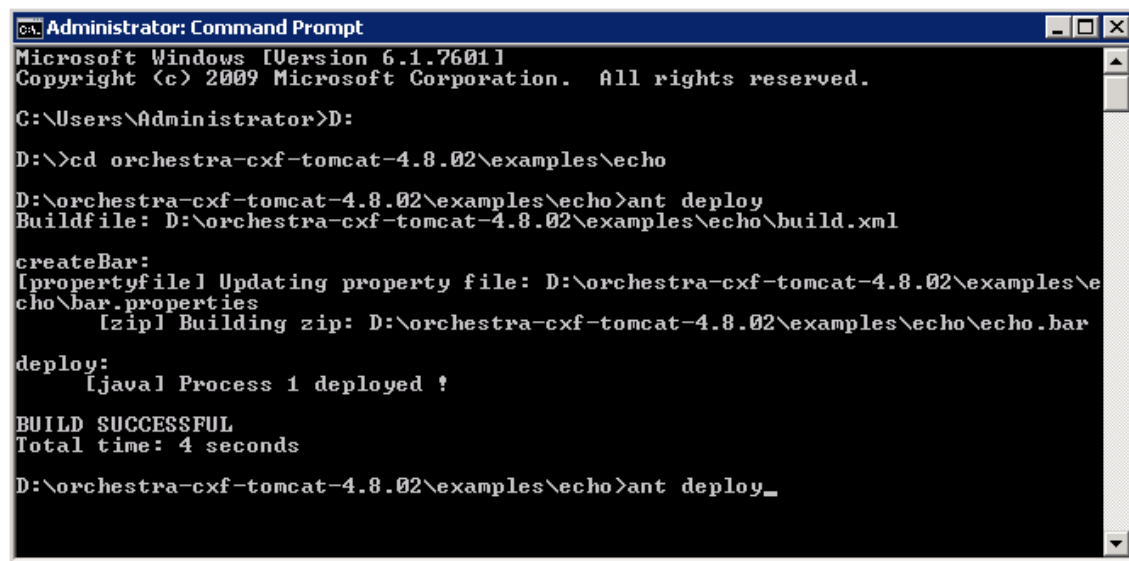
In this Appendix Chapter the logging process in the Orchestra engine is described. This is important to test parts of the engine during the developing process. The responsible logging class is the `java.util.logging` class. To change the logging properties, the `logging.properties` file as shown in Figure A.6 has to be configured. Therein the logging status of the respective classes can be set. The file is stored in `/workspace/Orchestra/packages/common`

But it is possible to define the location of the logging configuration file by setting the `java.util.logging.config.file` system property to e.g.:

```
mvn jetty:run-war –Djava.util.logging.config.file myLogging.properties
```

There are several logging levels:

- *OFF*
- *INFO*
- *FINE*
- *WARNING*
- *ALL*
- *SEVERE*

**Figure A.4:** Deploy a Web Service

The level is set in the properties file by adding the project class and set their logging level. E.g., for the CxfWSImpl.java class and the logging level `ALL`:

```
org.ow2.orchestra.cxf.CxfWSImpl.level=ALL
```

In the associated class, a logger has to be created, e.g., for the CxfWSImpl.java class. Therein the needed packages are imported and a logger function implemented. This funktion is used to choose the logging level and set the output logging string. This is shown in Listing A.1.

**Listing A.1:** The Logger Function

```
import java.util.logging.Level
import java.util.logging.Logger
...
private static Logger log = Logger.getLogger(CxfWSImpl.class.getName());
...
CxfWSImpl.log.<LOGGING LEVEL>(<LOGGING STRING>);
```
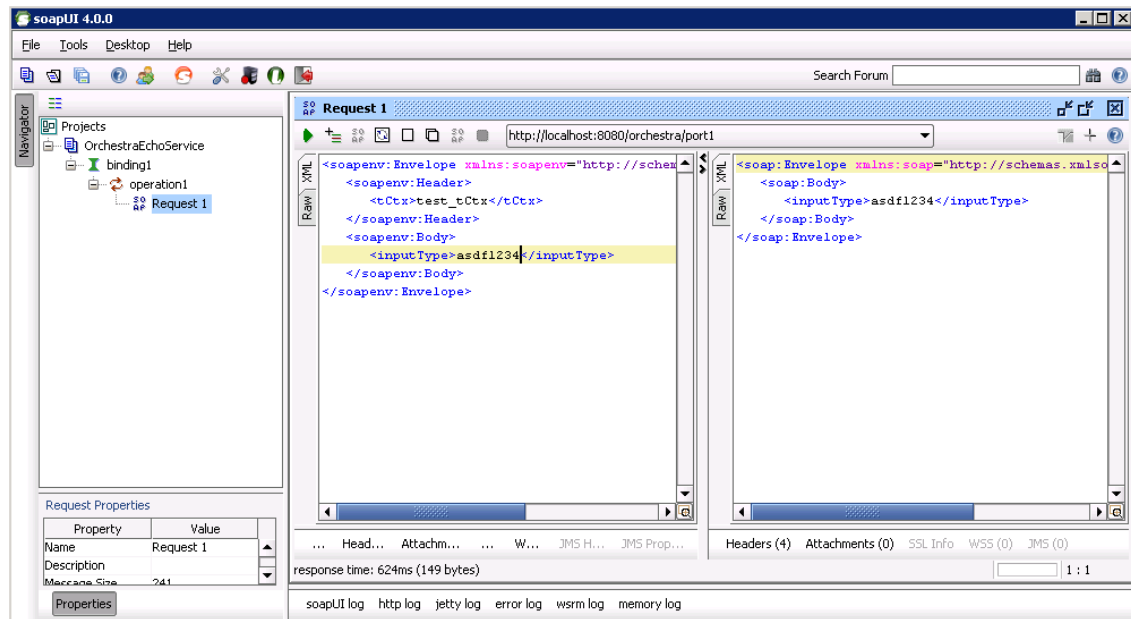
**Figure A.5:** Address a Web Service

```
 1  handlers= java.util.logging.ConsoleHandler
 2  .level= FINEST
 3  #java.util.logging.ConsoleHandler.level = FINEST
 4  java.util.logging.ConsoleHandler.formatter = org.ow2.orchestra.util.TraceFormatter
 5
 6  org.ow2.orchestra.util.TraceFormatter.alias=\
 7  org.ow2.orchestra.pvm.internal.wire.descriptor.HibernateConfigurationDescriptor~hibernateConfiguration,\
 8  org.ow2.orchestra.deployment.Deployer~deployer,\
 9  org.ow2.orchestra.facade.jmx.RemoteDeployerMBean~api,\
10  org.ow2.orchestra.persistence.log.LoggerRecorder~recorder,\
11  org.ow2.orchestra.persistence.log.LoggerArchiver~archiver
12  # For example, set the com.xyz.foo logger to only log SEVERE messages:
13  # com.xyz.foo.level = SEVERE
14
15  org.hibernate.level=SEVERE
16  org.ow2.orchestra.pvm.internal.wire.descriptor.HibernateConfigurationDescriptor.level=FINE
17  org.hibernate.event.def.AbstractFlushingEventListener.level=OFF
18  org.ow2.orchestra.jmx.level=INFO
19  org.ow2.orchestra.osgi.Engine.level=INFO
20  org.ow2.orchestra.pvm.internal.svc.DefaultCommandService.level=OFF
21  org.ow2.orchestra.pvm.internal.tx.StandardTransactionInterceptor.level=OFF
22  org.ow2.orchestra.deployment.Deployer.level=FINE
23  org.ow2.orchestra.test.EnvironmentTestCase.level=FINE
24  org.ow2.orchestra.test.remote.RemoteTestCase.level=FINE
25  org.ow2.orchestra.pvm.level=WARNING
26  org.ow2.orchestra.level=WARNING
27  org.ow2.orchestra.StartupListener.level=INFO
28  #org.ow2.orchestra.persistence.log.level=FINE
29  org.ow2.orchestra.cxf.CxfWSImpl.level=ALL
30  org.ow2.orchestra.cxf.CxfInvoker.level=ALL
31  org.ow2.orchestra.cxf.MessageCarrierImpl.level=ALL
32  org.ow2.orchestra.cxf.AsyncAssociateMessage.level=ALL
33
```

**Figure A.6:** The Logging Properties File

# Appendix B

# BPEL and WSDL Examples

This Chapter presents some of the BPEL and WSDL files of the used Web services in the motivating examples and tests. The examples are from the Orchestra CXF tomcat 4.8.0 package[1].

## B.1 Echo BPEL Process and WSDL Example

**Listing B.1:** Echo BPEL Process File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process name="echo"
        targetNamespace="http://enterprise.netbeans.org/bpel/echo/echo_1"
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension
            /Trace"
        xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/
            SUNExtension/Editor"
        xmlns:wsdlNS="http://enterprise.netbeans.org/bpel/echo/echo_1"
        xmlns:ns1="http://localhost/echo/echo"
        xmlns:ns2="http://xml.netbeans.org/schema/echo">

    <documentation>
        The synchronous BPEL process illustrates a simple synchronous
        flow. The process receives an input message and sends it back
        synchronously. A client starts the synchronous process by invoking
        a request-response operation. After invoking a synchronous process,
        the client is blocked until the process finishes and returns the result.
    </documentation>

    <import namespace="http://localhost/echo/echo"
            location="echo.wsdl"
            importType="http://schemas.xmlsoap.org/wsdl/"/>

    <partnerLinks>
        <partnerLink
            name="echo"
            partnerLinkType="ns1:partnerlinktype1"
```

---

[1]Available at http://repo1.maven.org/maven2/org/ow2/orchestra/orchestra-cxf-tomcat/

```
            myRole="partnerlinktyperole1">
            <documentation>
                This partner link represents the client who sends an
                input message to the process and receives the response.
            </documentation>
        </partnerLink>
    </partnerLinks>

    <variables>
        <variable name="outputVar" messageType="ns1:responseMessage">
            <documentation>Output variable.</documentation>
        </variable>
        <variable name="inputVar" messageType="ns1:requestMessage">
            <documentation>Input variable.</documentation>
        </variable>
    </variables>

    <sequence>
        <documentation>
            The sequence includes several activities
            which are executed in lexical order.
        </documentation>
        <receive
            name="start"
            partnerLink="echo"
            operation="operation1"
            portType="ns1:portType1"
            variable="inputVar"
            createInstance="yes">
            <documentation>
                The Receive activity makes the process
                to wait for the incoming message to arrive.
            </documentation>
        </receive>
        <assign name="Assign1">
            <copy>
                <from variable="inputVar" part="inputType"/>
                <to variable="outputVar" part="resultType"/>
            </copy>
        </assign>
        <reply
            name="end"
            partnerLink="echo"
            operation="operation1"
            portType="ns1:portType1"
            variable="outputVar">
            <documentation>
                The Reply activity returns a message from the process
                to the  partner which initiated the communication.
            </documentation>
        </reply>
    </sequence>
</process>
```

**Listing B.2:** Echo WSDL File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="echo" targetNamespace="
        http://localhost/echo/echo"
    xmlns:tns="http://localhost/echo/echo"
    xmlns:ns="http://xml.netbeans.org/schema/echo"
    xmlns:plink="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/varprop">


    <message name="requestMessage">
        <part name="inputType" type="xsd:string"/>
    </message>

    <message name="responseMessage">
        <part name="resultType" type="xsd:string"/>
    </message>

    <portType name="portType1">
        <operation name="operation1">
            <input name="input1" message="tns:requestMessage"/>
            <output name="output1" message="tns:responseMessage"/>
        </operation>
    </portType>

    <binding name="binding1" type="tns:portType1">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="
            document"/>
        <operation name="operation1">
            <soap:operation soapAction="http://orchestra.ow2.org/echo/operation1"
                />
            <input name="input1">
                <soap:body use="literal"/>
            </input>
            <output name="output1">
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="service1">
        <port name="port1" binding="tns:binding1">
            <soap:address location="http://localhost:${HttpDefaultPort}/orchestra
                /port1"/>
        </port>
    </service>

    <plink:partnerLinkType name="partnerlinktype1">
        <plink:role name="partnerlinktyperole1" portType="tns:portType1"/>
    </plink:partnerLinkType>
```

```
</definitions>
```

## B.2  LoanService BPEL Process and WSDL Example

**Listing B.3:** LoanService BPEL Process File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bpws:process exitOnStandardFault="no" name="loanService"
    suppressJoinFailure="yes"
    targetNamespace="http://orchestra.ow2.org/loanApproval/loanService"
    xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:ns="http://orchestra.ow2.org/loanApproval/common"
    xmlns:ns0="http://orchestra.ow2.org/loanApproval/riskAssessment"
    xmlns:ns1="http://orchestra.ow2.org/loanApproval/approval" xmlns:tns="http://
        orchestra.ow2.org/loanApproval/loanService">

    <bpws:import importType="http://schemas.xmlsoap.org/wsdl/"
        location="loanService.wsdl" namespace="http://orchestra.ow2.org/
            loanApproval/loanService"/>
    <bpws:import importType="http://schemas.xmlsoap.org/wsdl/"
        location="riskAssessment/riskAssessment.wsdl" namespace="http://orchestra
            .ow2.org/loanApproval/riskAssessment"/>
    <bpws:import importType="http://schemas.xmlsoap.org/wsdl/"
        location="approval/approval.wsdl" namespace="http://orchestra.ow2.org/
            loanApproval/approval"/>
    <bpws:import importType="http://schemas.xmlsoap.org/wsdl/"
        location="common.wsdl" namespace="http://orchestra.ow2.org/loanApproval/
            common"/>

    <bpws:partnerLinks>
        <bpws:partnerLink myRole="loanService" name="customer" partnerLinkType="
            tns:loanServiceLT"/>
        <bpws:partnerLink name="approver"
            partnerLinkType="ns1:approvalLT" partnerRole="approver"/>
        <bpws:partnerLink name="assessor"
            partnerLinkType="ns0:riskAssessmentLT" partnerRole="assessor"/>
    </bpws:partnerLinks>
    <bpws:variables>
        <bpws:variable messageType="ns:creditInformationMessage" name="request"/>
        <bpws:variable messageType="ns0:riskAssessmentMessage" name="risk"/>
        <bpws:variable messageType="ns:approvalMessage" name="approval"/>
    </bpws:variables>
    <bpws:flow name="Flow">
        <bpws:links>
            <bpws:link name="receive-to-assess"/>
            <bpws:link name="receive-to-approval"/>
            <bpws:link name="approval-to-reply"/>
            <bpws:link name="assess-to-setMessage"/>
            <bpws:link name="setMessage-to-reply"/>
            <bpws:link name="assess-to-approval"/>
        </bpws:links>
        <bpws:receive createInstance="yes" name="Receive"
```

```
            operation="request" partnerLink="customer"
            portType="tns:loanServicePT" variable="request">
            <bpws:sources>
                <bpws:source linkName="receive-to-assess">
                    <bpws:transitionCondition><![CDATA[$request.amount < 10000]]>
                        </bpws:transitionCondition>
                </bpws:source>
                <bpws:source linkName="receive-to-approval">
                    <bpws:transitionCondition><![CDATA[$request.amount >= 10000]]
                        ></bpws:transitionCondition>
                </bpws:source>
            </bpws:sources>
        </bpws:receive>
        <bpws:invoke inputVariable="request" name="Assess"
            operation="check" outputVariable="risk"
            partnerLink="assessor" portType="ns0:riskAssessmentPT">
            <bpws:targets>
                <bpws:target linkName="receive-to-assess"/>
            </bpws:targets>
            <bpws:sources>
                <bpws:source linkName="assess-to-setMessage">
                    <bpws:transitionCondition><![CDATA[$risk.level='low']]></
                        bpws:transitionCondition>
                </bpws:source>
                <bpws:source linkName="assess-to-approval">
                    <bpws:transitionCondition><![CDATA[$risk.level!='low']]></
                        bpws:transitionCondition>
                </bpws:source>
            </bpws:sources>
        </bpws:invoke>
        <bpws:assign name="SetMessage" validate="no">
            <bpws:targets>
                <bpws:target linkName="assess-to-setMessage"/>
            </bpws:targets>
            <bpws:sources>
                <bpws:source linkName="setMessage-to-reply"/>
            </bpws:sources>
            <bpws:copy>
                <bpws:from>
                    <bpws:literal>yes</bpws:literal>
                </bpws:from>
                <bpws:to part="accept" variable="approval"/>
            </bpws:copy>
        </bpws:assign>
        <bpws:invoke inputVariable="request" name="Approval"
            operation="approve" outputVariable="approval"
            partnerLink="approver" portType="ns1:approvalPT">
            <bpws:targets>
                <bpws:target linkName="receive-to-approval"/>
                <bpws:target linkName="assess-to-approval"/>
            </bpws:targets>
            <bpws:sources>
                <bpws:source linkName="approval-to-reply"/>
            </bpws:sources>
        </bpws:invoke>
```

73

```
        <bpws:reply name="Reply" operation="request"
            partnerLink="customer" portType="tns:loanServicePT" variable="
                approval">
            <bpws:targets>
                <bpws:target linkName="approval-to-reply"/>
                <bpws:target linkName="setMessage-to-reply"/>
            </bpws:targets>
        </bpws:reply>
    </bpws:flow>
</bpws:process>
```

**Listing B.4:** RiskAssessment WSDL File

```
<wsdl:definitions
  targetNamespace="http://orchestra.ow2.org/loanApproval/riskAssessment"
  xmlns:tns="http://orchestra.ow2.org/loanApproval/riskAssessment"
  xmlns:common="http://orchestra.ow2.org/loanApproval/common"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:import location="../common.wsdl" namespace="http://orchestra.ow2.org/
    loanApproval/common"/>

  <plnk:partnerLinkType name="riskAssessmentLT">
    <plnk:role name="assessor" portType="tns:riskAssessmentPT" />
  </plnk:partnerLinkType>

  <wsdl:message name="riskAssessmentMessage">
    <wsdl:part name="level" type="xsd:string"/>
  </wsdl:message>

  <wsdl:portType name="riskAssessmentPT">
    <wsdl:operation name="check">
      <wsdl:input message="common:creditInformationMessage" />
      <wsdl:output message="tns:riskAssessmentMessage" />
    </wsdl:operation>
  </wsdl:portType>

  <binding name="riskAssessmentBinding" type="tns:riskAssessmentPT">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="check">
      <soap:operation soapAction="http://orchestra.ow2.org/loanApproval/
        riskAssessment"/>
      <input>
        <soap:body use="literal" namespace="http://orchestra.ow2.org/loanApproval
          /riskAssessment" />
      </input>
      <output>
        <soap:body use="literal" namespace="http://orchestra.ow2.org/loanApproval
          /riskAssessment" />
      </output>
    </operation>
```

```
    </binding>

    <service name="riskAssessmentServiceBP">
      <port name="riskAssessmentPort" binding="tns:riskAssessmentBinding">
        <soap:address location="http://localhost:${HttpDefaultPort}/orchestra/
            riskAssessmentPort"/>
      </port>
    </service>

</wsdl:definitions>
```

**Listing B.5:** Approval WSDL File

```
<wsdl:definitions
  targetNamespace="http://orchestra.ow2.org/loanApproval/approval"
  xmlns:tns="http://orchestra.ow2.org/loanApproval/approval"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:common="http://orchestra.ow2.org/loanApproval/common"
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:import location="../common.wsdl" namespace="http://orchestra.ow2.org/
      loanApproval/common"/>

  <plnk:partnerLinkType name="approvalLT">
    <plnk:role name="approver" portType="tns:approvalPT" />
  </plnk:partnerLinkType>

  <wsdl:portType name="approvalPT">
    <wsdl:operation name="approve">
      <wsdl:input message="common:creditInformationMessage" />
      <wsdl:output message="common:approvalMessage" />
    </wsdl:operation>
  </wsdl:portType>

  <binding name="approvalBinding" type="tns:approvalPT">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="approve">
      <soap:operation soapAction="http://orchestra.ow2.org/loanApproval/approval"
          />
      <input>
        <soap:body use="literal" namespace="http://orchestra.ow2.org/loanApproval
            /approval" />
      </input>
      <output>
        <soap:body use="literal" namespace="http://orchestra.ow2.org/loanApproval
            /approval" />
      </output>
    </operation>
  </binding>

  <service name="approvalServiceBP">
    <port name="approvalPort" binding="tns:approvalBinding">
```

```
      <soap:address location="http://localhost:${HttpDefaultPort}/orchestra/
          approvalPort"/>
    </port>
  </service>

</wsdl:definitions>
```

# Bibliography

[ACT]      ActiveVOS BPMS from Active Endpoints. Website. Available online at `http://www.activevos.com/`.

[APAa]     Apache CXF. Website. Available online at `http://cxf.apache.org/`.

[APAb]     Apache CXF Architecture. Website. Available online at `http://cxf.apache.org/docs/cxf-architecture.html`.

[APAc]     Apache CXF Debugging and Logging. Website. Available online at `http://cxf.apache.org/docs/debugging-and-logging.html`.

[APAd]     Apache CXF Interceptors and Phases. Website. Available online at `http://cxf.apache.org/docs/interceptors.html`.

[APAe]     Apache ODE. Website. Available online at `http://ode.apache.org/`.

[APW$^+$11] A. Azeez, S. Perera, S. Weerawarana, P. Fremantle, S. Uthaiyashankar, S. Abesinghe. WSO2 Stratos: An Application Stack to Support Cloud Computing. 2011.

[BGK$^+$11] M. Behrendt, B. Glasner, P. Kopp, R. Dieckmann, G. Breiter, S. Pappe, H. Kreger, A. Arsanjani. Introduction and Architecture Overview IBM Cloud Computing Reference Architecture 2.0. 2011. Available online at `https://www.opengroup.org/cloudcomputing/uploads/40/23840/CCRA.IBMSubmission.02282011.doc`.

[Buc10]    A. Buchholz. Multi-tenant-fähige BPEL-Engines. *Diplomarbeit Nr. 2995, University of Stuttgart*, 2010.

[BZP$^+$10] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, A. Hart. Enabling Multi-Tenancy: An Industrial Experience Report. *26th IEEE International Conference on Software Maintenance in Timisoara, Romania*, 2010.

[CC06]     F. Chong, G. Carraro. Architecture Strategies for Catching the Long Tail. Website, 2006. Available online at `http://msdn.microsoft.com/en-us/library/aa479069.aspx` visited on July 16th 2011.

[CWZ10]    H. Cai, N. Wang, M. Zhou. A Transparent Approach of Enabling SaaS Multi-tenancy in the Cloud. *IEEE 6th World Congress on Services*, 2010.

[Ess11]    S. Essl. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support. *Masterarbeit Nr. 3166, University of Stuttgart*, 2011.

[GSH$^+$07] C. Guo, W. Sun, Y. Huang, Z. Wang, B. Gao. A Framework for Native Multi-Tenancy Application Development and Management. *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services*, 2007.

[IBM]      IBM Business Process Automation WebSphere Process Server. Website. Available online at `http://www-01.ibm.com/software/integration/wps/`.

[IBO]      Business and Process Integration Suite. Website. Available online at `http://www.magicsoftware.com/en/products/?catID=41`.

[JBP]      jBPM JBoss Community. Website. Available online at `http://www.jboss.org/jbpm`.

[KNL08]    T. Kwok, T. Nguyen, L. Lam. A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application. *IEEE International Conference on Services Computing*, 2008.

[KSS09]    M. Koning, C.-A. Sun, M. Sinnema. VxBPEL: Supporting variability for Web services in BPEL. *Information and Software Technology*, 51:258–269, 2009.

[Ley11]    F. Leymann. Cloud Computing: The Next Revolution in IT. 2011. Available online at `http://www.iaas.uni-stuttgart.de/institut/mitarbeiter/leymann/publications/INPROC-2009-65%20-%20Leymann%20-%20Cloud%20Computing%20-%20PhoWo.pdf`.

[MAE]      BPEL Engine & Toolkit: BPEL Maestro Parasoft. Website. Available online at `http://www.parasoft.com/jsp/products/bpel.jsp?itemId=114`.

[MBI]      Microsoft BizTalk Server. Website. Available online at `http://www.microsoft.com/germany/biztalk/default.mspx`.

[MG11]     P. Mell, T. Grance. The NIST Definition of Cloud Computing. 2011. Available online at `http://docs.ismgcorp.com/files/external/Draft-SP-800-145_cloud-definition.pdf`.

[MLP08]    R. Mietzner, F. Leymann, M. P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. pp. 156–161, 2008. doi:10.1109/ICIW.2008.68. URL `http://dl.acm.org/citation.cfm?id=1381304.1381988`.

[MMLP09]   R. Mietzner, A. Metzger, F. Leymann, K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. *Principles of Engineering Service Oriented Systems, ICSE Workshop on*, 0:18–25, 2009. doi:http://doi.ieeecomputersociety.org/10.1109/PESOS.2009.5068815.

[MSDa]     Introducing Windows Communication Foundation: Accessible Service-Oriented Architecture. Website. Available online at `http://msdn.microsoft.com/en-us/library/orm-9780596527563-01-10.aspx`.

[MSDb]     The Workflow Engine Model. Website. Available online at `http://msdn.microsoft.com/en-us/library/aa188337%28office.10%29.aspx`.

[MUTL09]   R. Mietzner, T. Unger, R. Titze, F. Leymann. Combining Different Multi-Tenancy Patterns in Service-Oriented Applications. *2009 IEEE International Enterprise Distributed Object Computing Conference*, 2009.

[OAS06]    Reference Model for Service Oriented Architecture 1.0. 2006. Available online at `docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`.

[OAS07]     Web Services Business Process Execution Language Version 2.0. 2007. Available online at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`.

[ORA]       Oracle BPEL Process Manager. Website. Available online at `http://www.oracle.com/technetwork/middleware/bpel/overview/index.html`.

[ORC]       Orchestra: Open Source BPEL / BPM Solution. Website. Available online at `http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome`.

[PET]       Petals Link Research Home. Website. Available online at `http://research.petalslink.org/display/research/Petals+Link+Research+Home`.

[PLJ10]     G. Porcher, G. Le Jeune. Orchestra. 2010. Available online at `http://orchestra.ow2.org/xwiki/bin/download/Main/Documentation/Orchestra_en.pdf`.

[PLL10]     Z. Pervez, S. Lee, Y.-K. Lee. Multi-Tenant, Secure, Load Disseminated SaaS Architecture. 2010.

[SAP]       SAP Exchange Infrastructure. Website. Available online at `http://help.sap.com/saphelp_nw04/helpdata/de/0f/80243b4a66ae0ce10000000a11402f/content.htm`.

[SLLW09a]   Y. Shi, S. Luan, Q. Li, H. Wang. A Flexible Business Process Customization Framework for SaaS. *2009 WASE International Conference on Information Engineering*, 2009.

[SLLW09b]   Y. Shi, S. Luan, Q. Li, H. Wang. A Multi-Tenant Oriented Business Process Customization System. *International Conference on New Trends in Information and Service Science*, 2009.

[SR11]      B. Sengupta, A. Roychoudhury. Engineering Multi-Tenant Software-as-a-Service Systems. 2011.

[TOG]       The SOA Work Group : Definition of SOA. Website. Available online at `http://www.opengroup.org/soa/soa/def.htm`.

[VMCL09]    L. Vaquero, L. Merino, J. Caceres, M. Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1), 2009.

[VUS]       Virtuoso Universal Server. Website. Available online at `http://virtuoso.openlinksw.com/`.

[W3C]       W3C Recommendation - SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Website. Available online at `http://www.w3.org/TR/soap12-part1/`.

[WCL+05]    S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[ZSTC10]    X. Zhang, B. Shen, X. Tang, W. Chen. From Isolated Tenancy Hosted Application
to Multi-tenancy: Toward a Systematic Migration Method for Web Application.
2010.

All links have been last checked on December 19, 2011

## Declaration

All the work contained within this thesis, except where otherwise
acknowledged, was solely the effort of the author. At no stage was
any collaboration entered into with any other party.


Stuttgart, December 20, 2011          ————————————

                                          (Michael Baldauf)