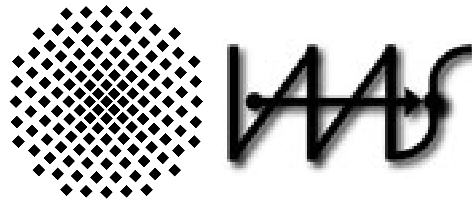
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3226

Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management

Dominik Muhler



Course of Study: Software Engineering

Examiner: Prof. Dr. Frank Leymann
Supervisor: Dipl.-Inf. Steve Strauch
Dipl.-Inf. Tobias Binz
Commenced: July 28, 2011
Completed: January 27, 2012

CR-Classification: C.2.4, D.2.11, H.2.1, H.3.4

Abstract

As part of cloud computing, the service model Platform-as-a-Service (PaaS) has emerged, where customers can develop and host internet-scale applications on cloud infrastructure. The Enterprise Service Bus (ESB) is one possible building block of a PaaS offering, providing integration capabilities for service-oriented architectures. Bringing the ESB to the cloud requires scalability and multi-tenancy support. When applied, these characteristics lead to economies of scale, reducing the costs per customer.

In this diploma thesis we specify, design, and implement a multi-tenant management application for an existing open source ESB. The management application grants tenant users limited configuration access to the ESB's connectivity and integration services. A tenant registry and a service registry serve as platform-wide databases. We ensure data isolation between tenants for the management application and ESB message flows. Furthermore, the management application can control clusters of ESB instances, retaining elasticity. These goals also involve extensions to the ESB itself, which implements the Java Business Integration (JBI) specification. As a result, an integration scenario emerged from the EU-funded project 4CaaSt was applied to the system.

Contents

1. Introduction 1
1.1. Motivating Scenario . . . 1
1.2. Scope of Work . . . 3
1.3. Outline . . . 3
1.4. Definitions and Conventions . . . 4

2. Fundamentals 9
2.1. Cloud Computing . . . 9
2.2. Service-oriented Architecture . . . 10
2.2.1. Web Services Platform . . . 11
2.3. Enterprise Service Bus . . . 12
2.4. Multi-tenancy . . . 14
2.5. Technologies . . . 15
2.5.1. Java Business Integration . . . 15
2.5.2. OSGi Framework . . . 17
2.5.3. Apache ServiceMix . . . 18

3. Related Works 19
3.1. WSO2 Platform-as-a-Service . . . 19
3.1.1. Multi-tenant SOA platform . . . 19
3.1.2. Multi-tenant BPEL engine . . . 20
3.2. Force.com Platform-as-a-Service . . . 21

4. Concept and Specification 23
4.1. System Overview . . . 23
4.1.1. Components . . . 23
4.1.2. Scenarios . . . 25
4.2. Service Registry . . . 26
4.3. Multi-tenancy . . . 27
4.3.1. Role-based Access Control . . . 27
4.3.2. Tenant Registry . . . 28
4.3.3. Configuration Registry . . . 29
4.3.4. Service Assembly Processing . . . 30
4.4. Use Cases . . . 31
4.5. Application Interfaces . . . 34
4.5.1. Web-based Graphical User Interface . . . 34
4.5.2. Web Service API . . . 36
4.6. Non-functional Requirements . . . 36

5. Design 39
5.1. Architectural Overview . . . 39
5.1.1. Components . . . 39
5.1.2. Integration . . . 42
5.1.3. Processing of JBI Artifacts . . . 43
5.2. Web Application . . . 43
5.2.1. Business Logic Access Layer . . . 43
5.2.2. Web Service . . . 45
5.2.3. Packaging and Deployment . . . 46
5.3. Database Schemes . . . 48
5.4. Extensions to ServiceMix . . . 50
5.4.1. Management Interface Over Messaging . . . 51
5.4.2. Multi-tenant JBI Components . . . 52

6. Implementation and Evaluation 55
6.1. Implementation . . . 55
6.1.1. Java Annotations for Role-based Access Control . . . 55
6.1.2. OSGi-based Management Service for Apache ServiceMix . . . 57
6.1.3. Multi-tenant Binding Component and Service Engine . . . 58
6.2. Evaluation . . . 59
6.2.1. Deployment and Initialization . . . 59
6.2.2. Tenant Context-based Routing . . . 63

7. Outcome and Future Work 67

A. Interface Definitions 69
A.1. Web Service Interface . . . 69
A.2. JBI Management Interface . . . 75

Bibliography 77

List of Figures

1.1. Taxi Scenario - Communication Diagram 2

2.1. Web Services Architecture 11
2.2. Taxi Scenario - Refactoring to ESB 13
2.3. JBI Architecture 16

4.1. Overview of the System JBIMulti2 24
4.2. Clustering Scenarios to Evaluate 25
4.3. Overview of Component Configuration Base Classes 30
4.4. Use Case Diagram 33
4.5. Web UI: Service Unit Quotas Content Panel 35
4.6. Web UI: Service Unit Contingents Content Panel 36

5.1. Overview of the JBIMulti2 Architecture 40
5.2. Business Logic Layer: AccessLayer 44
5.3. Enterprise Application Archive Packaging 47
5.4. Entity-relationship Diagram of Tenant Registry 48
5.5. Entity-relationship Diagram of Service Registry 49
5.6. Entity-relationship Diagram of Roles in Configuration Registry 49
5.7. Entity-relationship Diagram of JBI Components in Configuration Registry . . 50

6.1. Add Tenant Request with soapUI 61
6.2. Deploy Service Assembly Request with soapUI 62
6.3. Taxi Scenario - Tenant Context-based Routing 63

List of Tables

1.1. XML Namespaces 5

4.1. Permissions for Tenant Administrator Role and Tenant Operator Role 28
4.2. Properties in Tenant Registry Used by the System 28
4.3. Data Stored by Configuration Manager 29
4.4. XML Attributes Considered by Multi-tenant JBI Components 31
4.5. Description of Use Case: Delete Service Unit Contingent 32

List of Listings

5.1. Service Endpoint Replacing Patterns for HTTP BC 53

6.1. Permission Type Annotations in TenantAdminFacadeBean.java 56
6.2. OSGi Blueprint Descriptor of JMSManagementService 58
6.3. SOAP Binding Endpoint URLs Containing Tenant URI 64
6.4. Service Unit Configuration of Tenant Context Appender 65
6.5. Service Unit Configuration of Tenant Context-based Router 66

A.1. Tenant Context XSD of JBIMulti2 69
A.2. Policies XSD of JBIMulti2 70
A.3. Policies in WSDL Document of JBIMulti2 71
A.4. Example SOAP Message to JBIMulti2 73
A.5. Messages XSD of JBI Management Interface. 75

1. Introduction

Cloud computing is a recent paradigm for offering computing resources to customers on-demand via Web interfaces and standardized protocols. Offered computing resources are diverse, as they comprise business applications, services, storage, or virtual servers. Various cloud services were started in the past years. Platform-as-a-Service (PaaS) offerings, such as Force.com [WB09], provide application developers with facilities to build entire Web-based business applications. Apart from that, more flexible Infrastructure-as-a-Service (IaaS) offerings, such as Amazon EC2 [AMA], allow customers to provision virtual machines to run arbitrary software on them. As cloud service providers serve many customers, they can leverage economies of scale, reducing costs for the individual customer. Multi-tenancy describes how applications should be designed to maximize resource sharing among customers. This includes multi-tenant data architectures that isolate data between tenants and scalability to serve any number of tenants.

The EU-funded project 4CaaSt [4Ca] aims to create a PaaS cloud platform that provides application developers with base components for multi-tier Web-based applications. This includes an Enterprise Service Bus (ESB) middleware that service aggregators can use to integrate applications, benefiting from the principles of Service-oriented Architecture (SOA). To fulfill the characteristics of cloud computing, the ESB must be multi-tenant aware and scalable. This work examines for an open source ESB, how data isolation between tenants can be achieved, focusing on management and administration. Moreover, the concepts of this work aim to retain ESB clustering capabilities.

A multi-tenant ESB should provide a management interface for tenant users, allowing them to deploy configuration artifacts to the ESB. At the same time, configuration artifacts deployed by one tenant must result in messages being delivered to either services of the tenant or platform-wide services. This means, tenants must not get access to services of other tenants. Particularly, tenant users should not notice that there are other tenants using the same instances of the multi-tenant ESB. This work introduces a multi-tenant business application for managing and administrating the multi-tenant ESB. Furthermore, for a particular open source ESB, we introduce multi-tenant service components that host configuration data of tenant users and ensure data isolation for message exchanges.

1.1. Motivating Scenario

In this section we describe a concrete scenario that is used for evaluating the outcomes of this work. The scenario is composed of an application system that implements a taxi booking service. Taxi customers can request taxi companies to send a taxi driver for transportation.

For this purpose, the customer communicates the pick up location and the desired destination to the taxi company. Then, the taxi company consults an external taxi service provider about an available and nearby taxi cab. In the process, the taxi service provider requests taxi drivers that are near the pick up location to commit the transportation of the customer. Thus, each taxi driver has to carry a Taxi Transmitter that displays orders on a map and allows the taxi driver to confirm an individual order. Once a taxi driver has confirmed the commission, the taxi service provider sends a transportation information back to the taxi company that in turn informs the customer.

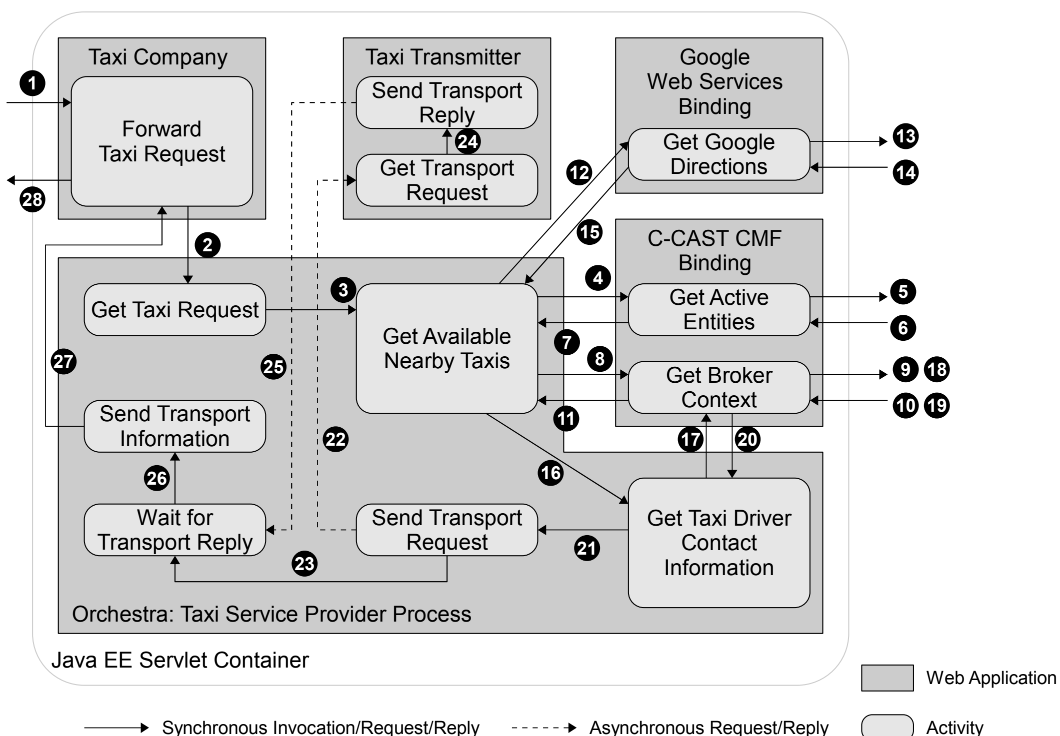


Figure 1.1.: Communication between Web applications that collaboratively accomplish prototype of Taxi Scenario. Condenses insights of Figure 1.1 and Figure 5.1 by Hagin [Hag11].

Hagin has developed a prototype of the Taxi Scenario by implementing a BPEL process that orchestrates the activities of the taxi service provider. The BPEL process leverages the Project Context Casting (C-CAST) Context-Management Framework (C-CAST CMF) [CCA], a context provisioning system that provides context information about taxi cab locations and taxi driver contact details. Moreover, Google Maps Web Services [GMA] provide distance calculations between the location of a taxi cab and the pick up location. All participants of the taxi booking service are Web applications deployed to a Java EE Servlet Container (see Fig. 1.1). The Taxi Company Web application acts as graphical user interface to the customer, while the Taxi Transmitter provides an interface for the taxi driver. Additionally, the BPEL engine Orchestra [OWO] is deployed as Web application, executing the taxi service provider process. As the service endpoints of the C-CAST CMF and the Google Maps Web

Services are incompatible to the BPEL process, two binding applications mediate between the BPEL process and this external services. All applications communicate via point to point messaging connections [Hag11].

In this work we develop concepts and implementation artifacts for a multi-tenant aware ESB. The Taxi Scenario prototype involves many point to point integrations between Web applications. Furthermore, each taxi company can publish an own Web application for communication with its customers and has own taxi drivers carrying Taxi Transmitters. Therefore, each taxi company and the taxi service provider are individual tenants and the outcomes of this work should simplify the integration of the various Web applications. For evaluation purposes, we integrate the Taxi Scenario using the multi-tenant aware ESB by utilizing tenant context-based routing (see Sect. 6.2).

1.2. Scope of Work

This diploma thesis has the goal to specify, design and implement a multi-tenant management and administration system for Apache ServiceMix 4.3.0 [ASM], an open source ESB. As Apache ServiceMix implements the Java Business Integration (JBI) specification [JBI05], tenant users must be provided with interfaces to deploy and undeploy service assemblies, which are ESB configuration artifacts. The system must simultaneously support multiple instances of Apache ServiceMix, retaining clustering support to ensure elasticity. This work emphasizes data isolation between tenants, regarding the management system as well as message flows inside the ESB that result from deployed service assemblies. While message flows are isolated between tenants, service assemblies must further be isolated on the level of tenant users.

As the management system will be embedded in a PaaS platform with other applications demanding similar information, this work must lay foundations for a shared registry of tenants and a shared registry of services. Both ensuring data isolation between tenants.

Out of scope is the design and implementation of different load balancing strategies. However, adapting the system for desired load balancing strategies must be facilitated. Possible scenarios include clusters of equal Apache ServiceMix instances or clusters of different interconnected Apache ServiceMix instances. Moreover, message flows resulting from policy-based routing [MvLW+09] are not considered in this work. It suffices if tenants can configure point to point messaging connections inside Apache ServiceMix. Furthermore, this work does not present concepts for ensuring performance isolation between service artifacts running on behave of different tenants.

1.3. Outline

The following six chapters cover different phases of work that have been pursued to accomplish the given goals and to conceive future tasks on the topic.

- **Fundamentals, Chapter 2**—In the beginning, relevant literature that covers the fundamentals of this diploma thesis was surveyed. The chapter gives an explanation of cloud computing, SOA, ESB, and multi-tenancy. Moreover, technologies widely used in this work are explained, such as JBI, OSGi, and Apache ServiceMix.
- **Related Works, Chapter 3**—As a second preliminary step, two existing PaaS platforms were examined for multi-tenancy concepts. Both the WSO2 organization and Salesforce.com offer multi-tenant PaaS platforms that provide developers with tools to create business applications.
- **Concept and Specification, Chapter 4**—Lessons learned in the first chapters were considered when formalizing the functional requirements and non-functional requirements of the multi-tenant ESB management system. This includes a conceptual overview, a Service Registry, a Tenant Registry, role-based access control, data isolation requirements, user interfaces, and a use-case analysis.
- **Design, Chapter 5**—An architectural overview devises components and their relations that together fulfill the described system requirements. A multi-tenant Java EE enterprise application is described that can control a cluster of Apache ServiceMix instances. Moreover, necessary extensions to Apache ServiceMix are illustrated that ensure data isolation in message flows.
- **Implementation and Evaluation, Chapter 6**—Chosen challenges that have occurred during the implementation of the system are explained. Finally, the implementation was evaluated by applying the Taxi Scenario to it.
- **Outcome and Future Work, Chapter 7**—Since a complete multi-tenant ESB can not be developed in the scope of a single diploma thesis, the last chapter summarizes the outcomes of this work and suggests future extensions to the system.

1.4. Definitions and Conventions

The following definitions and abbreviations should be inspected for understanding the descriptions in this work. They are used throughout the document.

Definitions

In this document the general term BPEL refers to the Web Services Business Process Execution Language (WS-BPEL) 2.0 specification [OAS07].

The following eXtensible Markup Language (XML) namespaces are used in this document and referenced by the listed prefix:

Prefix	Namespace	Specification
ctxjmu2	http://jbimulti2.iaas.uni-stuttgart.de/tenant-context	This document
wpjmu2	http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy	This document
camel	http://camel.apache.org/schema/spring	[APA11a]
cm	http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0	[ARS]
ds	http://www.w3.org/2000/09/xmldsig#	[XML02b]
ext	http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0	[ARS]
osgi	http://www.springframework.org/schema/osgi	[CHLP09]
soap	http://www.w3.org/2003/05/soap-envelope	[SOA07]
soap12	http://schemas.xmlsoap.org/wsdl/soap12	[WSD06]
sp	http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702	[OAS09]
wsdl	http://schemas.xmlsoap.org/wsdl	[WSD01]
wsp	http://www.w3.org/ns/ws-policy	[WSP07]
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd	[OAS06a]
xenc	http://www.w3.org/2001/04/xmlenc#	[XML02a]
xsd	http://www.w3.org/2001/XMLSchema	[XSD04]
xsi	http://www.w3.org/2001/XMLSchema-instance	[XSD04]

Table 1.1.: XML namespaces referenced in this document via listed prefix.

List of Abbreviations

The following list contains abbreviations used in this document. Full names by convention not valid or not used anymore are marked as deprecated.

ACID Atomicity, Consistency, Isolation, Durability

Apache ODE Apache Orchestration Director Engine

ASP Application Service Provider

Axis2 Apache eXtensible Interaction System v. 2

BC Binding Component

C-CAST CMF Project Context Casting (C-CAST) Context-Management Framework

BLOB Binary Large Object

BPEL Web Services Business Process Execution Language 2.0

CLOB Character Large Object

EAI Enterprise Application Integration

EAR Enterprise Archive

EJB Enteprise JavaBeans

ESB Enterprise Service Bus

IaaS Infrastructure-as-a-Service

IDE Integrated Development Environment

Java EE 5 Java Platform, Enterprise Edition v. 5

JAX-WS Java API for XML-Based Web Services

JAXB Java Architecture for XML Binding

JBI Java Business Integration

JBIMulti2 JBI Multi-tenancy Multi-container Support

JDBC Java Database Connectivity

JDK Java Development Kit

JMS Java Message Service

JMX Java Management Extensions

JOnAS Java Open Application Server

JPA Java Persistence API

JSF JavaServer Faces

JVM Java Virtual Machine

MBean Managed Bean

NIST National Institute of Standards and Technology

NMR Normalized Message Router

OSGi Open Services Gateway initiative *(deprecated)*

PaaS Platform-as-a-Service

POJO Plain Old Java Object

QoS Quality of Service

SaaS Software-as-a-Service

SE Service Engine

SOA Service-oriented Architecture

SOAP Simple Object Access Protocol *(deprecated)*

UDD Universal Data Dictionary

UDDI Universal Description, Discovery and Integration

UUID Universally Unique Identifier

W3C World Wide Web Consortium

WS* Web Services (Specifications)

WSDL Web Services Description Language

WSO2 Web Services Oxygen *(deprecated)*

WSO2 BPS WSO2 Business Process Server

WSS4J Apache Web Services Security for Java

XML eXtensible Markup Language

XSD XML Schema Definition

XSLT Extensible Stylesheet Language Transformation

2. Fundamentals

This diploma thesis relies on distinct conceptual and technological fundamentals that are clarified in this chapter. Together they form the context for the outcomes of this work. As the goal is to develop a system that will be part of a PaaS platform, the term *cloud computing* is defined. Moreover, explanations are given how the ESB facilitates implementing SOA. Different authors have addressed the various aspects of multi-tenancy. This chapter describes the aims of multi-tenant systems and how developers can realize them. Finally, technologies widely used in this work are explained.

2.1. Cloud Computing

Widely spread and recognized, the World Wide Web has opened a new distribution channel for application software. Over the last decade more and more software vendors have begun to rent their software online, thus acting as Application Service Provider (ASP). An ASP expects omission of distribution costs, a consistent user base, and a constant revenue stream. On the other side, customers benefit from omission of installation and maintenance efforts [Tao01].

A new paradigm is cloud computing, which goes further by not only providing application software via the Web, but all kinds of computing resources. This can be storage, servers, or services. These computing resources are offered on-demand by the service provider and provisioned or released by the customer. According to the National Institute of Standards and Technology (NIST), there are five *essential characteristics* for the cloud model [NIS11]. Firstly, the customer can provision computing resources without further intervention of the service provider. Moreover, the computing resources are accessible by various types of client platforms via standard protocols. Regarding the underlying technology, resources are pooled, automatically serving multiple tenants and scaling elastic on increasing demand. Finally, the service provider can measure resource usage to charge customers and to monitor computing capabilities. Three *service models* provide the customer with increasing control, while lowering the abstraction level of the computing resources.

- **Software-as-a-Service (SaaS)** is a service model where service providers offer applications running on a cloud infrastructure to their customers. The underlying cloud infrastructure enables cloud characteristics like elasticity and accessibility. In this model the customer can only control individual configurations that are applied while the customer uses the application.

- **Platform-as-a-Service (PaaS)** allows the customer to deploy and configure his own artifacts to the cloud infrastructure, such as applications, services, or libraries. These artifacts are defined using programming languages, description languages, or modeling languages predetermined by the service provider.
- **Infrastructure-as-a-Service (IaaS)** is the most flexible service model providing the customer with control on the level of operating systems and virtual machines. As a consequence, the customer can deploy any software that runs on the available operating systems.

Regarding the user base of a cloud service, four deployment models can be distinguished. A *private cloud* is used exclusively by one organization, possibly comprising multiple sub units. Whereas a *community cloud* is shared by multiple organizations. Both deployment models do not pretend the cloud infrastructure to be off premise. On the contrary, a *public cloud* is not restricted to a group of organizations, and therefore, must be provided by a separate service provider. A *hybrid cloud* combines at least two different of the previous deployment models. For instance, this allows an organization to fall back on a public cloud, if the own computing capabilities can not cope with the current load [NIS11].

Cloud services encourage using SOA as paradigm for implementation (see Sect. 2.2). According to IBM, all characteristics of SOA services can also be found in cloud services, such as flexible deployment models ranging from on premise deployment to solutions that spread over multiple enterprises [OPG11]. Therefore, cloud services can be realized with an extended SOA that additionally ensures all cloud characteristics.

2.2. Service-oriented Architecture

With the intensification of IT-based business process reengineering and optimization, new approaches towards system architectures and technologies have emerged. A widely used architectural paradigm is Service-oriented Architecture (SOA), where implementations of business activities are encapsulated as services. SOA provides the essential flexibility, by relying on loose coupling between services and interoperability [WCL+05]. The Open Group describes a service as a reusable implementation of a business activity that is self-contained, possibly aggregating other services, while hiding implementation details. SOA incorporates services as building blocks into enterprise business processes. Service orchestration requires services to provide business descriptions and access through open standard protocols [OPG06].

Centerpiece of SOA is the *Publish, Find, Bind* pattern that describes the operations of participants. First, the service provider has to register a description of his service to a discovery facility. This allows a requestor to find a suitable service by consulting the discovery facility that replies by giving a concrete service endpoint. With this information, the requestor can than bind to the concrete service and finally execute a business activity [WCL+05]. An ESB simplifies this procedure for the requestor by finding a suitable service, binding to it, and executing the initial request in a single step (see Sect. 2.3).

2.2.1. Web Services Platform

The Web services technology introduces a set of specifications that cover a complete SOA stack (see Fig. 2.1). On top of common transport protocols, communication between services is realized by messaging. This reduces coupling between services and increases flexibility. The standard messaging protocol of the Web services stack is SOAP [SOA07]. Services are described using XML documents that comply to the Web Services Description Language (WSDL) specification [WSD01]. On the one hand, WSDL abstracts from the underlying service implementing technology. On the other hand, WSDL abstracts from the service provider, because it supports standardized Quality of Service (QoS) descriptions. Extensions for WSDL and SOAP that provide different QoS are defined in a set of separate specifications. For instance, WS-Security [OAS06a] specifies mechanisms to ensure integrity, confidentiality, and authentication of messages.

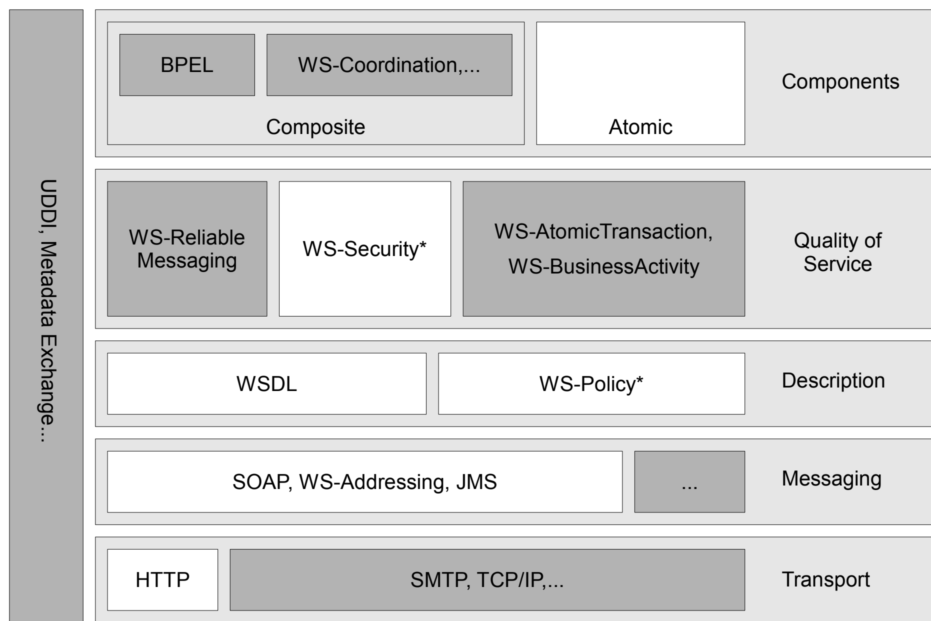


Figure 2.1.: Web services architecture according to Weerawarana et al. [WCL+05]. Specifications that are fundamental for this work are highlighted in white color.

The structure of SOAP (version 1.2) messages is defined as an XML Information Set, supporting arbitrary transport protocols. Each SOAP message contains a SOAP envelope that comprises a SOAP header and a SOAP body. A SOAP header can contain any meta-data about the message payload, such as routing information or authentication data. This supports processing of the message in a chain of receivers, with each chain node manipulating the message based on the information in the SOAP header.

A WSDL (version 1.1) document comprises an abstract definition and a concrete definition. The abstract definition contains one or more *port types* that are the abstract interface of the service. In contrast, the concrete definition of a WSDL document maps port types to concrete bindings and service endpoints. Different elements of a WSDL document can be annotated

with policies complying to the WS-Policy framework [WSP07]. Policies define claims about QoSs or information contained in SOAP messages. Therefore, policies define how the service operates, rather than what operations the service provides.

Completing the SOA stack, there are Web service specifications realizing discovery and composition of services. The former covered by Universal Description, Discovery and Integration (UDDI). The latter covered by WS-BPEL and other specifications [WCL+05].

2.3. Enterprise Service Bus

According to Ortiz Jr., enterprises have advanced their approach to application integration in multiple steps. At first, they have integrated enterprise applications manually by point-to-point connections. Since point-to-point connections lack flexibility and become increasingly unmanageable with a rising number of applications, Enterprise Application Integration (EAI) systems have emerged. Instead of directly communicating with each other, applications send messages via adapters to the EAI that is now responsible for correctly routing the messages. To omit the single point of failure of a centralized approach and to leverage SOA, software vendors have finally introduced the Enterprise Service Bus (ESB) [OJ07].

An ESB approaches integration problems by relying on standardized technologies, loose coupling, and distributed deployment. The core principle of an ESB is reliable messaging, which ensures loose coupling between applications. Using reliable messaging, application developers do not have to bother about resending data on failure. Additionally, if the target application is temporary unavailable, the messaging system can send the message as soon as the receiver is available again. An other foundation of loose coupling is a homogeneous representation format of data. As messages are represented as XML documents, mediation services can transform formats that are incompatible between applications. Implementing the principles of SOA, application logic is encapsulated behind a service interface. Thus, service endpoints are abstract and service implementations can simply be exchanged. An ESB encourages separating business process logic from the orchestrated underlying services. Therefore, applications do not need to know which other applications are dependent on the individual service interface. Adding new applications or changing their interactions can happen without changes to the individual applications.

Applications can be integrated beyond the borders of a single business location and even to other businesses. Each instance of the ESB middleware connects to other instances, building a distributed integration middleware. This allows a business unit to independently use the on-site ESB instance, while still being able to integrate services of other business units, on other locations. Nevertheless, administrators can configure and manage all ESB instances remotely, because management communication leverages the underlying distributed messaging middleware.

An integration architect configures connectivity services and integration services for each individual instance of the ESB. For this purpose, an ESB comprises service containers, which

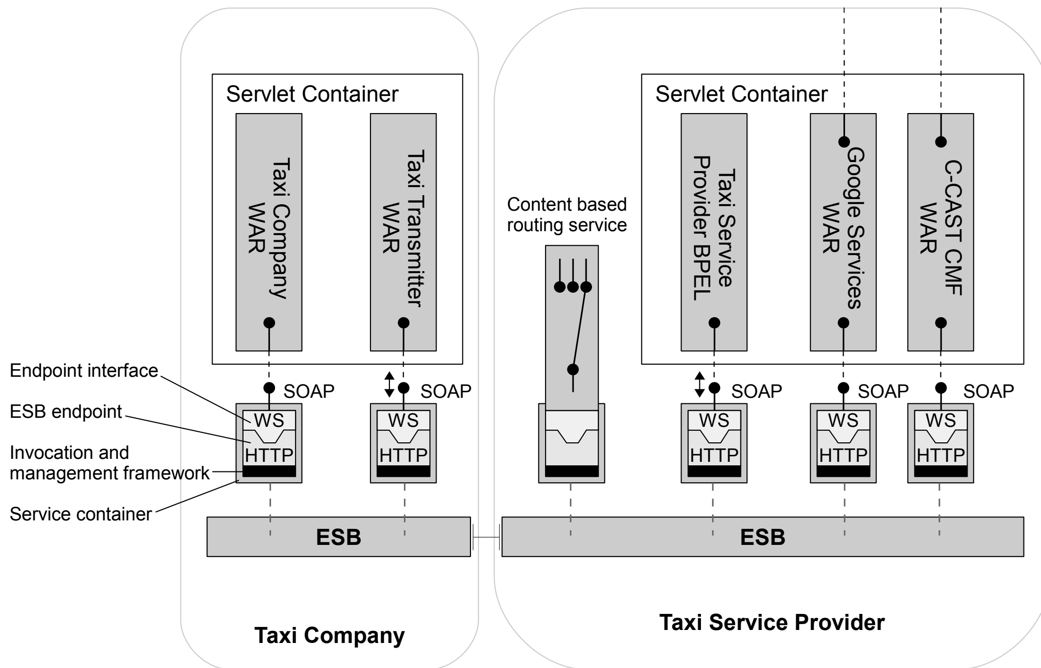


Figure 2.2.: Refactoring Taxi Scenario to ESB. Uses glyphs that have been introduced by Chappel [Cha04]. Double arrows indicate connectivity services that simultaneously publish a Web service interface for incoming messages and connect to an external Web service endpoint for outgoing messages.

host these services. The ESB service container provides an invocation and management framework that connects a deployed service to the ESB. Rather than programming connectivity services or integration services, the integration architect configures predefined services. The ESB must provide connectivity services for common protocols, such as SOAP or JMS, and integration services that transform messages or realize business process logic [Cha04].

The original implementation of the Taxi Scenario (see Sect. 1.1) integrates all participating applications via point-to-point SOAP/HTTP connections [Hag11]. Refactoring these connections to an ESB means that each application would instead communicate with an individual connectivity service running in an ESB service container. For instance, the taxi company Web application would send all customer requests to a connectivity service that publishes a Web service interface equivalent to the Web service interface of the taxi service provider process (see Fig. 2.2). Furthermore, the ESB could be distributed between the taxi company and the taxi service provider. In the original integration approach, the taxi service provider process manages direct endpoints of the orchestrated applications, tightening coupling. In an ESB approach the process could send all messages to a general ESB endpoint with a separate *content based router* [HW03] taking over responsibility for routing each message to the appropriate application.

2.4. Multi-tenancy

Using SaaS offerings instead of licensed, locally hosted applications, enterprises expect lower operational costs. Obviously, the software vendor benefits from synergies regarding administration and maintenance of similar applications running for many customers. But far more cost savings per customer can be achieved by scaling applications to serve many customers concurrently, making them multi-tenant aware. The reason for this is that the software vendor can better utilize computing capabilities by reducing no-load running and sharing common code base and data. Moreover, an automated sales and provisioning process can further reduce costs. With lower costs per customer the SaaS model can *catch the long tail* [CC06]. This describes the market segment of customers that previously could not afford licensing and hosting the individual software on their own. Depending on the market, the long tail can comprise many times more customers than have been reachable before.

Chong and Carraro introduce four maturity levels for SaaS architectures, assessing their capability to leverage economies of scale [CC06]. With increasing maturity, tenants share more code base and are less isolated. On the first level, each tenant is provided with a tailored application for exclusive use. The next level unifies the application code by giving tenants configuration tools. Nevertheless, each tenant still uses an own instance of the application. The third level adds multi-tenancy-efficiency by letting all tenants use the same instance of the application. However, the number of tenants one application instance can cope with is limited. Therefore, the fourth level of maturity requires scalability with tenants sharing a set of application instances.

Tenants need configuration flexibility regarding user interfaces, business logic, custom data models, and access control. Mature SaaS offerings should store tenant configurations in a metadata store, separate from the actual application data. For instance, administrators of a tenant could define roles and permissions in the metadata store that authorize access to application data and functions for other users of the tenant [CC06].

SOA integration architects meet additional challenges, when they deploy SaaS offerings that are compositions of multiple services. Taking up the previously described SaaS maturity levels, individual SOA services may as well differ in configurability and instance sharing. Accordingly, Mietzner et al. introduce *service tenancy patterns* including *single instance*, *single instance configurable*, and *multiple instance* [MUTL09]. The single instance pattern describes service instances that behave uniformly for all tenants. In contrast, the other two patterns describe services that expect a tenant context on each request. A configurable service instance uses the received tenant context to adjust its behavior, while a service exclusively provided to one tenant uses the tenant context for authentication. To choreograph these types of services, special enterprise integration patterns have been introduced. For instance, a *tenant context based router* can route a message by analyzing the tenant context contained in the message header. Furthermore, a SaaS vendor can apply tenant context based routing to distribute requests between heterogeneous computing resources. This allows the software vendor to offer its service with different quality levels and to optimize resource utilization [FLM10].

Regarding multi-tenant data architectures, there exist three approaches to isolate data between tenants in a database. When each tenant has an own database, recovery of tenant data on failure can be achieved with common tools and individual extensions to the data scheme are simple. Moreover, the database middleware ensures security. Certainly, this approach is not multi-tenant-efficient, because the number of databases per database server is limited. In a more shared architecture all data resides on the same database, but tenants have their data in separate tables. Still, custom columns can be used to adjust the data scheme for each tenant. Finally, the *shared database, shared schema* approach merges data of different tenants into the same tables. This third architecture allows more tenants per database server. However, if tenants have a high volume of data, there are many concurrent users, or tenants require flexible data schemes, the other architectures are worth considering. Application developers can apply security patterns to ensure data isolation between tenants, such as data encryption or SQL view filters [CCW06].

The previously described challenges may be unfamiliar to traditional application developers. A *multi-tenancy enablement layer* can encapsulate functions, such as data isolation, performance isolation, or configurability. Thus, allowing traditional application developers to concentrate on application logic and user interfaces. They invoke the underlying layer that then authenticates tenant users, controls access to resources, provides multi-tenant aware administration services and applies tenant configurations [GSH+07].

2.5. Technologies

The following sections describe technologies that realize concepts of SOA and the ESB. This work builds up on these technologies, thus an introduction in more detail is necessary.

2.5.1. Java Business Integration

As described previously, an ESB can act as integration middleware for SOA. But the common characteristics of an ESB are abstract. This means, a particular implementation of an ESB can indeed integrate various applications using different standardized protocols, but the building blocks of the ESB itself are still vendor-specific. Particularly, each software vendor may design an own service container and define interfaces to connectivity and integration services. The JBI specification, created by the Java Community Process, defines a Java framework that standardizes the interoperating between service containers, connectivity services, and integration services [JBI05]. Moreover, it specifies a management framework that integration architects can use for configuring those services to cope with individual integration tasks.

A JBI container provides facilities to plug in JBI-compliant components that interoperate through a central Normalized Message Router (NMR). A JBI component installed to a JBI container either acts as Binding Component (BC) or as Service Engine (SE). The former providing connectivity to external services, the latter implementing composition and transformation services (see Fig. 2.3). The NMR is a message-oriented mediator that ensures loose coupling

between JBI components. For this purpose, a JBI component consumes or provides services and describes them according to the WSDL 2.0 specification. Each JBI component can send *normalized messages* through the NMR by initiating a message exchange. A message exchange can target other services via abstract endpoints as well as concrete endpoints. Abstract endpoints are given as WSDL interface names or operation names and are routed by the NMR to a matching service endpoint. On the contrary, concrete endpoints are given as WSDL service names and service endpoint names. A normalized message comprises meta-data and a XML payload that corresponds to the XML Schema Definitions (XSDs) of the target service WSDL message elements. As a consequence, normalized messages are protocol neutral and BCs mediate between external protocols and the NMR.

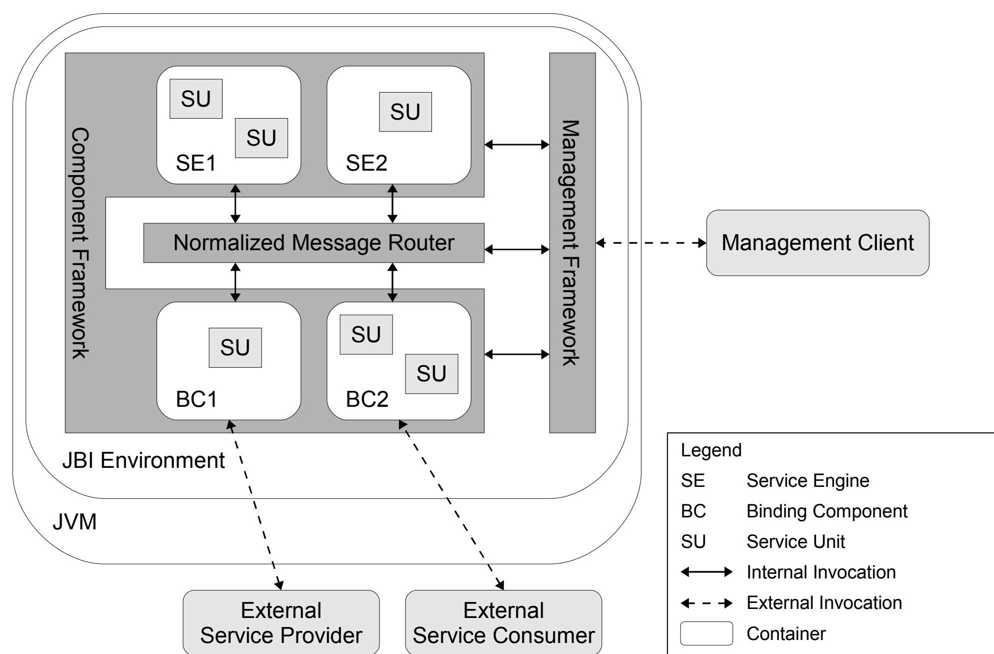


Figure 2.3.: Overview of JBI Architecture, based on Figure 4 in JBI specification document [JBI05]. JBI environment is illustrated as a container accommodating JBI components, which in turn accommodate service units.

The JBI specification defines four asynchronous message exchange patterns for communication between JBI components. They differ in being unidirectional or bidirectional and in being more or less reliable. Developers of JBI components use an API provided by the component framework to create and send message exchanges or to query service endpoints. Furthermore, the component framework demands of JBI components to implement several interfaces that allow the JBI container to manage their life-cycle and to deploy service artifacts to them.

A management framework allows administrators to install JBI components, to deploy service artifacts to them, and to control their state and the overall state of the JBI container. For this purpose, the JBI specification relies on the Java Management Extensions (JMX) specification. Consequently, each JBI component must implement predefined Managed Bean (MBean) interfaces. Via JMX, administrators can install new JBI components that are given as ZIP files

containing a JBI-compliant XML descriptor document. An integration architect can deploy service artifacts, called *service units* to JBI components. Often an integration problem can not be solved by only involving a single JBI component. Therefore, service units are packaged as *service assemblies* that likewise contain a XML descriptor document.

Contradicting an essential characteristic of an ESB, the JBI specification does explicitly not define how a distributed deployment of JBI containers must be accomplished [JBI05].

2.5.2. OSGi Framework

Designed for executability on devices with small memory, the OSGi Framework runs modular applications and encourages resource sharing between components within a single Java Virtual Machine (JVM). In OSGi, applications consist of executable and non-executable modules that are called *bundles*. A bundle is packaged as JAR file and contains, in addition to Java class files, meta-data describing capabilities it provides and requirements it demands. Capabilities include Java packages the bundle provides Java classes for, whereas requirements include Java packages the bundle relies on and does not deliver itself. After an administrator has installed a new bundle, the OSGi Framework tries to resolve its requirements by matching it with the capabilities of other installed bundles. A resolved bundle can provide classes and services for other bundles or execute functions by itself. For this purpose, the OSGi framework initializes a network of Java class loaders, allowing bundles to use system packages, framework packages, and packages exported by other bundles.

The OSGi Framework specifies a life cycle for bundles. Having been resolved, a bundle can be started by the framework. Therefore, executable bundles implement OSGi specific interfaces that allow the framework to start and stop the individual bundle. Furthermore, bundles can provide services to other bundles by dynamically registering *service objects* to the framework. Other bundles can then use an API to query the internal service registry for available service objects and finally bind to them. Additionally, bundles can listen to events that are published by registered services [OSG11].

To facilitate developing of modular applications in OSGi, the *Blueprint Container* specification [OSG09] has been introduced. A *blueprint bundle* contains a XML blueprint descriptor that defines the instantiation and wiring of objects. This provides *inversion of control*, meaning a blueprint bundle does not for itself create and connect objects, or bind to services. Instead these object dependencies are injected by a Blueprint Container according to the provided blueprint descriptor. As a result, the code base of blueprint bundles can be simpler and more reusable. Technically, a *Blueprint Extender* bundle listens for events that indicate the resolution of a new blueprint bundle. Then the Blueprint Extender creates a Blueprint Container that manages the instantiation and wiring of corresponding objects [Gaw09]. Another specification that covers inversion of control in OSGi environments introduces the concept of Spring Dynamic Modules [CHLP09]. As part of this work, we developed a blueprint bundle that creates multi-tenant aware JBI service assemblies (see Sect. 5.4.1).

2.5.3. Apache ServiceMix

The open source ESB we intend to extend for multi-tenancy support is Apache ServiceMix 4.3.0 [ASM] from the Apache Software Foundation, hereafter referred to as ServiceMix. It is based on the OSGi Framework implementation Apache Karaf [APA11b] that builds its kernel layer. ServiceMix also includes an OSGi Blueprint Container implementation that is provided by the Apache Aries project [ARS].

On top of the kernel layer, OSGi bundles realize the technology layer of ServiceMix. The technology layer brings in the ESB functionality complying to the JBI specification. Additionally, ServiceMix ships with various JBI components. BCs support diverse protocols, such as SOAP over HTTP, JMS, FTP, or SMTP. Whereas a SE that wraps Apache Camel [APA11a] provides enterprise integration patterns [HW03]. This comprises XML transformations, content-based routing, message splitting, or message aggregation. ServiceMix integrates the messaging broker Apache ActiveMQ [AMQ] as foundation of the NMR.

Administrators can manage an instance of ServiceMix via a command line console that is provided by Apache Karaf. However, the console not only allows managing OSGi bundles and services. Console extensions introduce commands for managing installed JBI components and deployed service assemblies. Artifacts, such as OSGi bundles, JBI components, or service assemblies, can be installed by putting them into a hot deployment directory. Although, the JBI specification does not define a distributed deployment of JBI containers, ServiceMix implements a clustering engine. As a result, within a cluster, each instance is aware of service endpoints created on other instances. Furthermore, developers are provided with plugins for the software build tool Apache Maven [AMV] that simplifies the process of developing JBI components and service assemblies [FUS11].

3. Related Works

This chapter presents the work of other authors who have designed multi-tenant systems for PaaS offerings. The WSO2 organization has developed a multi-tenant SOA platform and a multi-tenant BPEL engine, both based on existing open source middleware. Salesforce.com offers a PaaS platform that allows developers to create and run business applications. This analysis of existing approaches towards multi-tenant platforms shows, how existing open source middleware can be reused and how data isolation between tenants can be realized.

3.1. WSO2 Platform-as-a-Service

The organization WSO2 is a vendor of open source enterprise middleware. All products are component aggregations for WSO2 Carbon, an OSGi based platform. In order to implement a PaaS offering with their products, a multi-tenant SOA middleware was developed on top of WSO2 Carbon.

3.1.1. Multi-tenant SOA platform

A multi-tenant SOA platform based on the WSO2 Carbon platform allows tenant users to deploy components and services isolated from each other. Requests sent to the middleware may only be consumed by services of the corresponding tenant. The authors identify three architectural elements for a multi-tenant SOA middleware: Execution, Security and Data Isolation. Execution allows tenant users to deploy and run SOA components, such as services and processes, while preventing components of one tenant to interfere with components of other tenants. Security prevents deployed code of tenants to access resources of other tenants or resources of the platform. The proposed system also provides a framework that allows tenants to restrict their users to a limited set of system resources. Data isolation separates stored configurations of different tenants.

The underlying execution engine of WSO2 Carbon is Apache eXtensible Interaction System v. 2 (Axis2), a SOAP engine supporting different WS*-specifications. Users can deploy additional modules to Axis2, adding support for WS-Adressing [WSA04], WS-ReliableMessaging [BIMT05], or WS-Security [OAS06a]. Modules contain handlers for different phases of a message flow in both directions, into the engine or out of the engine [PHE+06]. Furthermore, users can deploy services as Axis2 archive files that contain a deployment descriptor, the service implementing classes, and additional libraries [Chi06].

The developed system takes advantage of the fact that Axis2 holds its static state under a single structure called AxisConfiguration and runtime state under a related structure called ConfigurationContext. Each tenant has its own AxisConfiguration, while there is still a master AxisConfiguration for other configurations. Resources assigned to one AxisConfiguration can not access resources of an other AxisConfiguration, which provides execution isolation and data isolation among tenants. When the engine receives a SOAP message, handlers in the master AxisConfiguration dispatch the message to the corresponding tenant AxisConfiguration. The system provides a unique URL for incoming messages that target services of different tenants. Such a URL comprises the identifier of the SOA platform, the tenant, and the target service. Thus allowing different tenants to have equal names for their services.

As tenants can run Java code on the multi-tenant platform, care is taken that the code can not access arbitrary resources. On the first level, the OSGi framework provides isolation by class loading mechanisms and configurable security mechanisms. On the second level, the JVM can be configured to run tenant code in a sandbox mode, preventing access to system resources.

The system uses shared database tables for tenants to secure isolation for relational data. Whereas file repositories are separated between tenants. But components and services do not access databases or file repositories directly, instead a multi-tenant aware API layer is positioned in between. It adds the tenant context to data access operations. Therefore, multi-tenancy is hidden from component and service developers [APG+10].

3.1.2. Multi-tenant BPEL engine

Part of the WSO2 PaaS is a multi-tenant BPEL engine called WSO2 Business Process Server (WSO2 BPS). Multiple tenants share the same instance of the BPEL engine, while business processes running inside are isolated from each other. As the system claims a tenant context on requests, the tenant context can be relayed to involved services. Therefore, multi-tenant shared business processes are deployable by orchestrating multi-tenant aware services.

The system is built on top of the BPEL engine Apache Orchestration Director Engine (Apache ODE) [AOD] and only modifies its integration layer, a component encapsulating the communication with other systems. Thus, the core components, such as the BPEL runtime are not changed. There exist two integration layers in the original version of Apache ODE, one using Axis2 and one using JBI. Incorporating a modified version of the Axis2 integration layer, WSO2 BPS adds multi-tenancy support to Apache ODE. The modified Axis2 integration layer leverages the functionality of the multi-tenant WSO2 Carbon platform (see Sect. 3.1.1).

When a new business process is deployed to WSO2 BPS, it is first stored in a multi-tenant aware repository provided by WSO2 Carbon. Then it is deployed to a multi-tenant aware ProcessStore in Apache ODE. The ProcessStore stores compiled business processes, which can directly be used by the BPEL runtime. Additionally, a service endpoint for the business process is added to the AxisConfiguration of the tenant. Once a request is sent to the Axis2 execution engine, it is relayed by the corresponding tenant service to WSO2 BPS.

The Apache ODE runtime inherently provides data and execution isolation, because the specifications of BPEL allow no access to resources belonging to an other BPEL process.

Clustering the WSO2 BPS is achieved by mapping a set of tenants to each instance. An upstream load balancer knows, which tenants and which business processes each instance serves and sends the request to a matching instance. This provides scalability regarding the number of tenants and the number of deployed business processes [MPW11].

3.2. Force.com Platform-as-a-Service

The application development platform Force.com is a PaaS offering operated by Salesforce.com. Tenant users can develop business applications by configuring a data model, an user interface, business logic, workflows, or other artifacts. The platform provides an interactive graphical user interface and a Web services API.

All artifacts that tenant users create are stored as metadata to a Universal Data Dictionary (UDD), founding the concept of a metadata-driven architecture. This architecture defines four components: The underlying runtime, application data, metadata for base functionality, and metadata for single tenants. As these components are decoupled from each other, it is possible for a tenant user to change the behavior of the tenant's application, without affecting other tenants or the whole platform.

Developers describe the data model of the application by specifying object types and associate fields with them. Each field has a standard data type assigned to it. This metadata is stored internally to database tables called Objects Metadata Table and Fields Meta Data Table. Concrete data derived from the metadata is stored in a large table that acts as a heap storage. This scheme allows tenant developers to create their own data model, while the application data is still stored in a central location. All schemes contain a column that identifies the tenant to which a record belongs.

When a user requests a tenant business application all components of the application are created dynamically from metadata. A query optimizer uses the tenant metadata, the application data, and pivot tables to efficiently retrieve information that the runtime uses to generate application components. Furthermore, metadata, application data and generated components are cached to reduce read operations to the central shared databases.

The platform provides the Apex programming language to application developers. Ensuring performance isolation among tenants the Force.com runtime limits CPU time, memory usage, and database queries of executed Apex code [WB09].

4. Concept and Specification

This chapter composes requirements for a multi-tenant ESB based on Apache ServiceMix [ASM] that can be part of a PaaS platform, like *4CaaSt* [4Ca]. The described system ensures data isolation by introducing multi-tenant aware management of ESB configuration artifacts. At first, an overview of the key functional requirements and components is given in Section 4.1. The following sections illustrate functional requirements in more detail and include a use case analysis. Moreover, non-functional requirements are listed in Section 4.6 giving guidance values for several software qualities.

4.1. System Overview

A system built on top of Apache ServiceMix [ASM] has to be developed that allows system administrators to configure connections to ServiceMix clusters and to a tenant database. System administrators can grant cluster access to tenants and monitor resource usage. In turn, tenant users can have different roles, allowing them to either manage access rights of other tenant users or to deploy services (see Sect. 4.3.1). In its final version, the system will provide a graphical Web-based user interface and a Web service interface, both running in parallel. Furthermore, the system must be independent of a specific version of the ServiceMix core implementation.

Configurations or services deployed by tenants to a ServiceMix cluster must not interfere with configurations of other tenants or system resources. A multi-tenant system must ensure data isolation and performance isolation. This specification focuses on requirements that provide data isolation for both static data and runtime data.

4.1.1. Components

As described in Section 2.5.3, ServiceMix complies with the JBI specification [JBI05]. Therefore, ServiceMix is designed as a JBI container that allows the installation of two types of JBI components. BCs connect the JBI container to external endpoints via different protocols, whereas SEs orchestrate the message flow inside the JBI container. Each JBI component uses service assemblies as configuration data that defines additional behavior, such as message flows, message transformations, or bindings to external endpoints. A multi-tenant ESB based on ServiceMix must manage service assemblies separately for each tenant.

Based on ServiceMix, the system conceived in this diploma thesis supports elasticity and additionally enables multi-tenant aware management of service assemblies and services. The

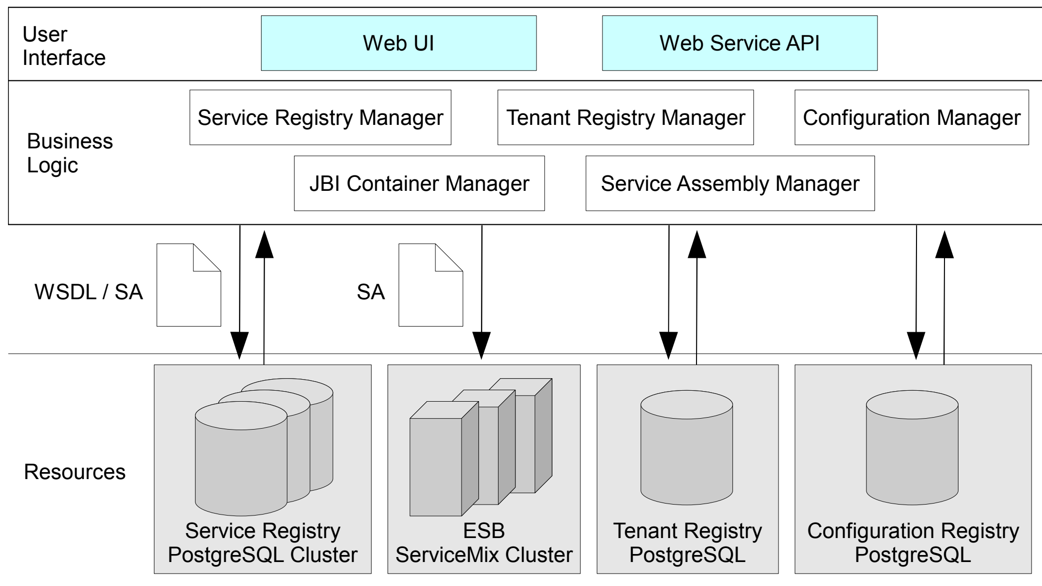


Figure 4.1.: Overview of the system JBIMulti2. Arrows illustrate data flow between core components and resources.

system is called *JBI Multi-tenancy Multi-container Support (JBIMulti2)*, because it supports many tenants and a cluster of ServiceMix instances.

JBIMulti2 relies on three data resources: the *Tenant Registry*, the *Service Registry*, and the *Configuration Registry*. Tenant contexts are obtained from the Tenant Registry. It delivers tenant users and their role inside the corresponding tenant (see Sect. 4.3.2). The Service Registry stores service assemblies and service descriptions for each tenant, while isolating the data of different tenants (see Sect. 4.2). All other non-tenant related and tenant related data is stored by the Configuration Registry. This includes configurations created by system administrators or tenant administrators. As service assemblies and services are stored to a separate Service Registry, they must not be stored to the Configuration Registry (see Sect. 4.3.3).

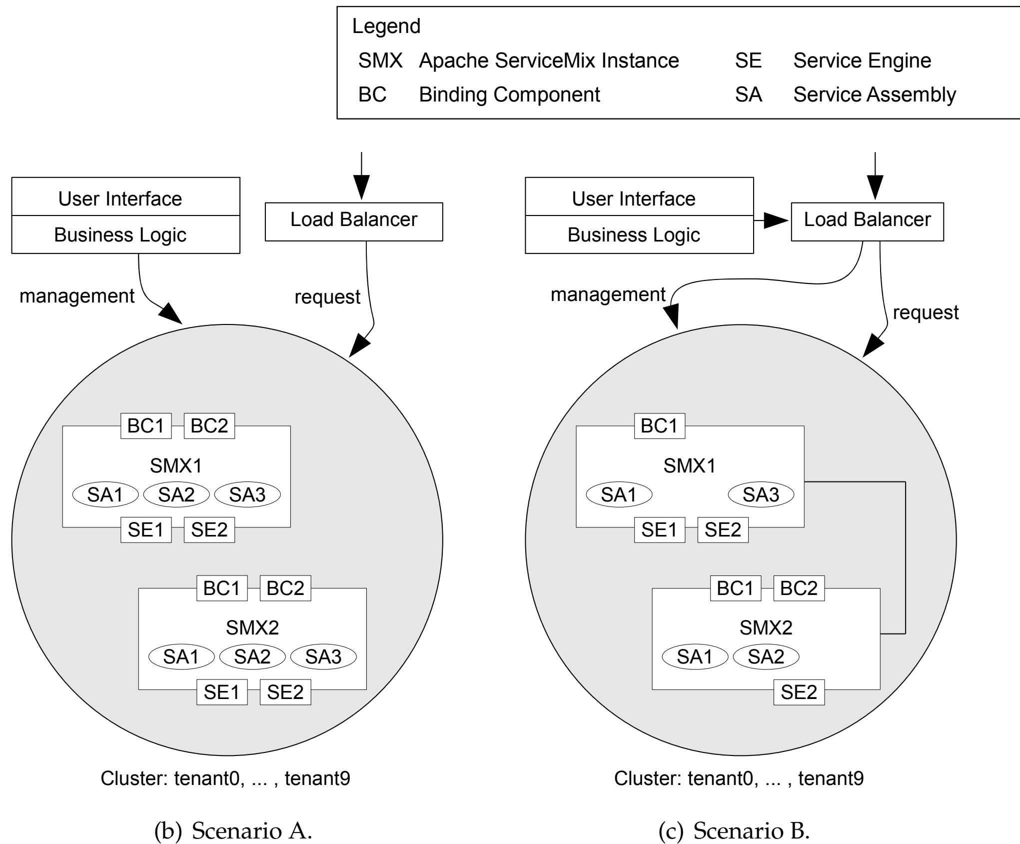
We propose a graphical Web-based user interface named *Web UI* that exposes the functionality of JBIMulti2. It can be used by system administrators and tenant users and supports all use cases listed in Section 4.4. Likewise, all functionality supported by the Web UI must be supported by a *Web Service API* as well (see Sect. 4.5.2). Currently, the prototype developed within the scope of this diploma thesis only implements the Web Service API. However, a concept for the layout of the Web UI can be found in Section 4.5.1.

The preceding components are built on top of an *Integration Layer* that encapsulates the access to databases and the ServiceMix clusters. Moreover, it can initiate atomic transactions among the resources. An extension component for ServiceMix provides integration with JBIMulti2. It installs JBI components, processes service assemblies by adding a tenant context, and deploys service assemblies. Once deployed, a service assembly must not interfere with the service assemblies of other tenants. To ensure data isolation, the tenant context inside the service units of a service assembly is processed by a set of new BCs and SEs. JBI components

are installed by a system administrator via the Web UI or Web Service API. Thus, they are predetermined for tenants using the system.

When designing the interfaces between the core components and the resources, maintainability and interchangeability have to be focused. Changing middleware vendors or endpoint configurations must not result in extensive changes to the core components.

4.1.2. Scenarios



(b) Scenario A.

(c) Scenario B.

Figure 4.2.: Clustering scenarios to evaluate. In scenario A all ServiceMix instances of a cluster have the same JBI components installed and the same service assemblies deployed. Whereas in scenario B all ServiceMix instances of a cluster are connected, allowing each one to have another set of JBI components and service assemblies.

Two scenarios regarding the topology of the ServiceMix clusters have to be evaluated (see Fig. 4.2). They arise from two different options for installing JBI components and deploying service assemblies to ServiceMix instances.

In the first scenario, the ServiceMix instances are not connected and therefore all instances are equal. JBI components have to be installed on all instances and service assemblies have to be deployed on all instances. Load balancing is easy as it does not matter, which ServiceMix

instance handles a request. The number of JBI components and service assemblies on one instance is limited. It has to be evaluated how many service assemblies a JBI component can handle and how the system performs with an increasing number of requests.

In the second scenario, the ServiceMix instances are connected and differ in the installed JBI components and deployed service assemblies. An intelligent deploying mechanism can distribute the service assemblies of tenants over the cluster to better utilize machine resources. An upstream load balancer then has to know, which ServiceMix instances can handle an ongoing request of a tenant. It has to be evaluated for two ServiceMix instances, how service units on one instance can communicate with service units on the other instance. Also it has to be evaluated how ServiceMix behaves, if the same service assembly is deployed to different instances.

The development of a load balancing mechanism or an intelligent deployment mechanism is not in the scope of the work. The current implementation of this specification only supports the first scenario.

4.2. Service Registry

The Service Registry is a platform-wide *PostgreSQL* [PSQ] database to store service assemblies and services. It is planned to reuse it as a shared database for other applications than the multi-tenant ESB.

JBIMulti2 writes service assemblies and services to the Service Registry in a tenant-isolated manner. Two kinds of resources are stored: service assemblies and WSDL descriptions, the former bundled as binary ZIP files and the latter represented as XML documents. The stored data is used to inform tenant users of their deployed resources. Apart from that, the WSDL data could be used by ServiceMix instances to dynamically establish JBI endpoints to external Web services. However, service assemblies are not dynamically loaded by the ServiceMix instances, instead, a subordinate load balancer will deploy service assemblies to the individual ServiceMix instances.

Neither dynamic loading of WSDL data nor load balancing is in the scope of the current implementation. But it must be taken care that it will be possible in a future extension of the system.

4.3. Multi-tenancy

Comparing isolated applications with multi-tenant applications, the latter demand an extended functionality. A multi-tenant system must ensure isolation for stored data not only between users, but also between different tenants. Moreover, authentication and access control to resources is executed on the two hierarchical levels of tenants and tenant users. Finally, there has to be a Tenant Registry and a corresponding management system that allows adding and removing tenants [ZSTC10]. The following sections describe how JBIMulti2 incorporates these functions.

4.3.1. Role-based Access Control

There are two distinct layers of roles participating in the system.

- The *system role* differentiates between *system administrators* and *tenant users*, with each tenant user belonging to one tenant. Both system roles are mutually exclusive and users of one system role have a completely separate user interface from users of the other system role.
- The *tenant role* classifies tenant users into *tenant administrators* and *tenant operators*.

A system administrator does not belong to a tenant. He configures the system and assigns quotas of resource usage to the tenants. The system administrator has unlimited permissions and consequently is allowed to interfere in the actions of the tenant users. On the other side, the tenant users consume the quotas of resource usage given by the system administrator to deploy service assemblies or to register services. Furthermore, they are classified into different tenant roles. Tenant administrators define roles and assign permissions to them. In turn, the roles are assigned to tenant operators who then access the resources using a resource contingent given by the tenant administrators. This scheme implements the *Role-Based Access Control (RBAC)* model [SCFY96]. Table 4.1 shows all permissions that are assignable to *tenant administrator roles* and *tenant operator roles*.

Each tenant user can have multiple tenant administrator roles and tenant operator roles. The union of the permissions of all tenant roles determines, which actions a tenant user can perform and which Web UI elements are visible to him.

A tenant administrator can manage contingents of service units and contingents of service registrations. The contingent defines a group of resources and the maximum number of resources the group can contain. All resources of a contingent are of the same type, such as BC for SOAP or WSDL service description. Therefore, a tenant administrator can partition the quota of resource usage he has obtained from the system administrator. For a tenant to be able to perform actions, the super administrator has to assign a default tenant administrator role to at least one tenant user. The initial tenant administrator can then appoint other tenant administrators or assign tenant operator roles to tenant users.

Tenant Administrator Role	Tenant Operator Role
Manage tenant administrator roles and their assignment to tenant administrators.	
Manage tenant operator roles and their assignment to tenant operators.	
Manage service unit contingents.	Deploy/undeploy service assembly using contingent x and view contingent x.
Manage service registration contingents.	Register/unregister service description using contingent x and view contingent x.
	View all service unit contingents.
	View all service registration contingents.

Table 4.1.: Permissions assignable to tenant administrator role and tenant operator role.

4.3.2. Tenant Registry

There is no existing Tenant Registry implementation given to use as data source for applications of the PaaS platform. Thus, a rudimentary tenant database has to be set up and tied to JBIMulti2 via the *Tenant Registry Manager* (see Fig. 4.1). The default middleware to use is a PostegreSQL 9.1.1 database. Though, the Tenant Registry Manager must facilitate changing the middleware that is delivering tenant information.

For each tenant the Tenant Registry stores a set of tenant users. Every tenant and every tenant user is identified by a Universally Unique Identifier (UUID) and can have associated properties represented as key-value pairs. The properties in Table 4.2 are used by JBIMulti2.

Subject	Property Name	Description
tenant	TName	Name of the tenant.
tenant	uri	URI that identifies the tenant domain. Used to name Web contexts in ServiceMix BCs for securing tenant isolation.
tenant user	UName	Name of the tenant user.
tenant user	tenant administrator role	Comma separated strings referencing tenant administrator roles in the Configuration Registry.
tenant user	tenant operator role	Comma separated strings referencing tenant operator roles in the Configuration Registry.
tenant user	password	Queried, when user tries to login. Represented as hash value generated with MD5.

Table 4.2.: Properties in Tenant Registry used by JBIMulti2.

4.3.3. Configuration Registry

All data created by system administrators or tenants, except for service assemblies and service registrations, is stored by JBIMulti2 via the *Configuration Manager* (see Fig. 4.1). As middleware PostgreSQL 9.1.1 is used and all tenants are assigned to the same database scheme, ensuring maximum resource sharing (see Sect. 2.4).

Table 4.3 lists the data stored by the Configuration Manager. Each data type is mapped to user roles that can have read or write access. A tenant user is additionally restricted to the actions defined in his roles. That means, for example, even if a tenant user has a tenant administrator role, he might not have the permission to change role assignments to other tenant administrators.

Data	Read Access	Write Access
Installed JBI components	system administrator	system administrator
Configuration of ServiceMix instances	system administrator	system administrator
Mapping of ServiceMix instances to tenants	system administrator	system administrator
Assignment of service unit quota for each installed JBI component	system administrator	system administrator
Assignment of service registration quota	system administrator	system administrator
Mapping of permissions to tenant administrator role	system administrator, tenant administrator	system administrator, tenant administrator
Assignment of tenant administrator role to tenant user	system administrator, tenant administrator	system administrator, tenant administrator
Mapping of permissions to tenant operator role	system administrator, tenant administrator	system administrator, tenant administrator, tenant operator
Assignment of tenant operator role to tenant user	system administrator, tenant administrator	system administrator, tenant administrator, tenant operator
Partitioning of service unit quota to service unit contingents	system administrator, tenant administrator	system administrator, tenant administrator
Partitioning of service registration quota to service registration contingents	system administrator, tenant administrator	system administrator, tenant administrator
Service unit contingent usage	system administrator, tenant operator	system administrator, tenant administrator, tenant operator
Service registration contingent usage	system administrator, tenant operator	system administrator, tenant administrator, tenant operator

Table 4.3.: Data stored by Configuration Manager.

4.3.4. Service Assembly Processing

Each JBI component defines what kind of configuration it relies on. The configuration has to be deployed as a service unit to ServiceMix. Multiple service units targeting different JBI components can be deployed together as a service assembly. JBIMulti2 provides an extension component that has to be installed on each ServiceMix instance. It adds the tenant context to each service unit contained in a service assembly, so that once deployed in ServiceMix, they do not interfere with service units of other tenants.

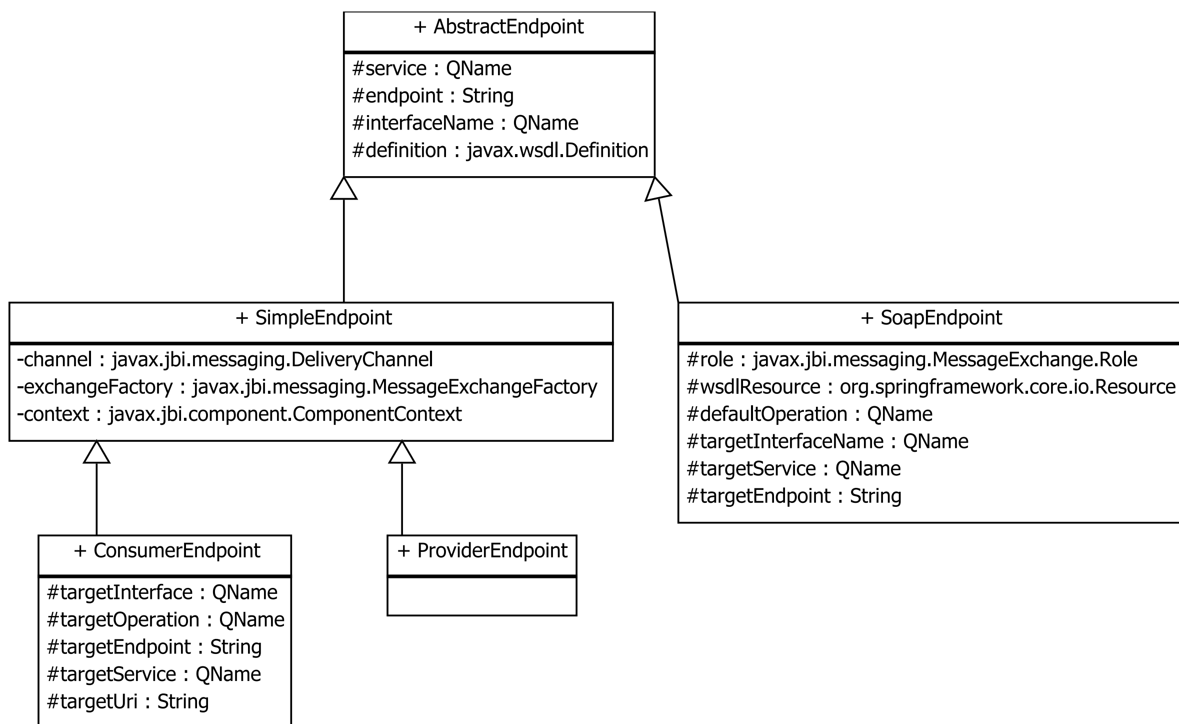


Figure 4.3.: Overview of component configuration base classes. XML elements are mapped to subtypes of this classes.

Each JBI component requires an other type of configuration data. Standard ServiceMix BCs, such as *servicemix-http*, *sevicemix-jms*, or *servicemix-mail* use XML configuration files. They each define an individual namespace with XML attributes mapped to internal Java class members of generated endpoints (see Fig. 4.3). Furthermore, the components servicemix-http and sevicemix-jms can also be configured by deploying a WSDL file. The SE *servicemix-camel* [APA11a] additionally supports custom Java class files as configuration data. Apache ODE [AOD] has a JBI integration layer that allows it to be installed as SE to ServiceMix. Service units for Apache ODE rely on BPEL definitions and a deployment descriptor that defines concrete endpoint definitions.

For each JBI component Table 4.4 shows, which XML elements and XML attributes are relevant for supporting multi-tenancy. Those have to be considered by the target multi-tenant JBI component in order to secure data isolation between tenants. A service assembly may not access endpoints that are configured by other tenants. Therefore, qualified names that

reference JBI service endpoints must be renamed during the internal transformation process (see Sect. 5.4.2).

JBI Component	XML elements	XML Attributes
servicemix-http	consumer, provider, soap-consumer, soap-provider	endpoint, interfaceName, service, targetEndpoint, targetService, targetInterface, targetOperation, targetUri, locationUri
servicemix-jms	consumer, provider, soap-consumer, soap-provider	endpoint, interfaceName, service, targetEndpoint, targetService, targetInterface, targetOperation, targetUri
servicemix-mail	poller, sender	endpoint, interfaceName, service, targetEndpoint, targetService, targetInterface, targetOperation, targetUri
servicemix-xmpp	receiver, sender	endpoint, interfaceName, service, targetEndpoint, targetService, targetInterface, targetOperation, targetUri
servicemix-camel	endpoint, from, to, wireTap, ...	uri, ref
Apache ODE	service	name, port

Table 4.4.: XML attributes considered by multi-tenant aware JBI components to secure data isolation between tenants.

4.4. Use Cases

Three actors take part in the use case analysis. They are defined in section 4.3.1: system administrator, tenant administrator, and tenant operator.

A system administrator has access to all resources of tenant administrators and tenant operators. Therefore, the system administrator inherits all use cases from the tenant administrator and the tenant operator (see Fig. 4.4). The system acts by modifying resources (see Table 4.4). As multiple resources can be changed during execution of one use case, resource changes must be included in a distributed transaction (see Sect. 5.1.2). All use case descriptions can be found in a separate requirements specification document [JBI12] belonging to this work.

Name	**Delete Service Unit Contingent**
Goal	The tenant administrator wants to delete a service unit contingent.
Actor	Tenant Administrator
Pre-Condition	The service unit contingent exists and the tenant administrator has the permission to manage service unit contingents.
Post-Condition	The service unit contingent is deleted.
Post-Condition in Special Case	The service unit contingent still exists and the service unit quota remains unchanged.
Normal Case	1. The tenant administrator commands the system to delete the service unit contingent. 2. The system asks the tenant administrator to confirm that all service assemblies using the service unit contingent will be undeployed. 3. The tenant administrator confirms. 4. The system starts a distributed atomic transaction. 5. The system initiates the undeployment of service assemblies at the JBI containers, deletes the records from the Service Registry and updates the Configuration Registry. 6. The system deletes the service unit contingent record from the Configuration Registry. 7. The system finishes the distributed atomic transaction.
Special Cases	3a. The tenant administrator aborts. a) The system does not delete the service unit contingent. 4a. Concurrently the tenant administrator has lost the permission to manage service unit contingents. a) The system shows an error message and aborts. 5a. The system can not finish the transaction with the JBI containers, the Service Registry and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message. 6a. The system can not finish the transaction with the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.5.: Description of Use Case *Delete Service Unit Contingent*.

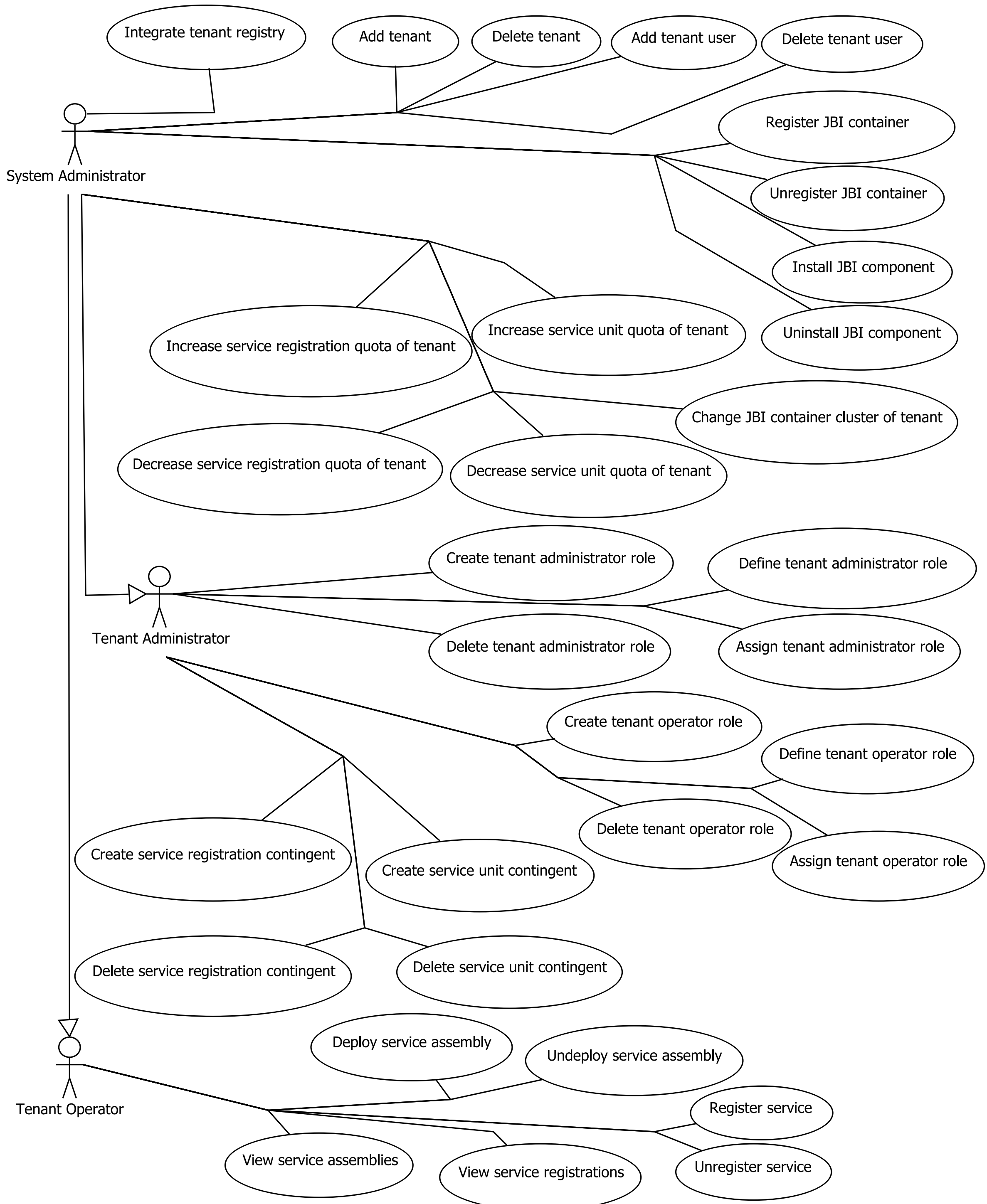


Figure 4.4.: Use case diagram.

4.5. Application Interfaces

This section describes a graphical user interface that makes the previously described functions available to humans. Moreover, a Web service API serves as interface to other applications and can be used to integrate JBIMulti2 into the PaaS platform.

4.5.1. Web-based Graphical User Interface

There are two separate Web-based user interfaces for system administrators and tenant users. This means on logging in, a system administrator is routed to an other interface than a tenant user.

Consisting of three parts, the user interface comprises a header, a menu and a content panel. The header shows the identity of the user. For tenant users additionally the name and URI of his tenant is shown. Via the header a user can log out. System administrators additionally can change to the perspective of a tenant and perform tenant action. Via the menu on the left side users can change the current content panel. The current content panel is shown on the right side. Every content panel subdivides its functionality into separate segments (see Fig. 4.5). For tenant users the roles they act as determine the available content panels. Whereas to system administrators all content panels are available.

Sketches of all content panels can be found in the separate requirements specification document [JBI12].

System Administrator Content Panels

For system administrators there exist five content panels, providing different functions.

The *Tenant Registry Content Panel* shows an overview of all tenants in the Tenant Registry. Furthermore, the system administrator can add and remove tenants and single tenant users. Using the *JBI Containers Content Panel*, on the one hand, the system administrator can assign a new JBI container to a cluster and install JBI components to known clusters. On the other hand, he can download the binaries of an installed JBI component or uninstall it again. To confirm his input the system administrator can view an overview of all JBI containers and installed JBI components.

With the *JBI Container Assignment Content Panel* the system administrator can assign an existing cluster name to a tenant. Then all JBI containers with this cluster name are available to the tenant. The *Service Unit Quotas Content Panel* allows system administrators to assign a quota of service units to a tenant for each JBI component (see Fig. 4.5). Only those JBI components are allowed that belong to a JBI container available to the tenant. To confirm his input, the system administrator can for each tenant inspect an overview of all allowed JBI components, the service unit quotas, and the number of currently deployed service units. Finally, the *Service Registration Quotas Content Panel* provides the system administrator with functions to assign a quota of service registrations to a tenant.

Figure 4.5.: Web UI: Sketch of Service Unit Quotas Content Panel.

Tenant Administrator Content Panels

Tenant administrators use the *Tenant Administrators Content Panel* to assign permissions to tenant administrator roles and tenant administrator roles to tenant users. Likewise, they use the *Tenant Operators Content Panel* to manage tenant operator roles. Each resource contingent has its own entry in the list of tenant operator permissions, which allows tenant administrators to separately grant permissions for using individual contingents.

With the *Service Unit Contingents Content Panel* the tenant administrator can partition the service unit quotas given by the system administrator into service unit contingents (see Fig. 4.6). Supporting the tenant administrator, an overview shows the quotas for different JBI components. Analogically, the *Service Registration Contingents Content Panel* allows a tenant administrator to partition the service registration quotas given by the system administrator into service registration contingents.

Tenant Operator Content Panels

Tenant operators are provided with two content panels. The *Service Assemblies Content Panel* allows tenant operators to deploy a service assembly by uploading it to the system as a ZIP file. He also chooses which service unit contingents may be charged for the service units contained in the service assembly. Apart from that, the tenant operator can download the ZIP file of an deployed service assembly or undeploy it again. Likewise, the tenant operator uses the *Service Registrations Content Panel* to upload a service registration as a WSDL file, download a WSDL file, or delete a service registration. WSDL files can also be registered by referencing them via URL.

Figure 4.6.: Web UI: Sketch of Service Unit Contingents Content Panel.

4.5.2. Web Service API

The Web Service API provides the same functionality as described in the use case analysis (see Sect. 4.4). It is stateless and integrates WS-Security by claiming a valid tenant context on each request.

When a ServiceMix instance wants to access information of the Service Registry or the Tenant Registry, it must call the Web service interface with system operator role. In usual cases a ServiceMix instance does not need to request information from JBIMulti2, because service assemblies are deployed via a push mechanism. A technical function that is not available in the Web UI, but needed by ServiceMix instances, is retrieving a tenant context by a tenant key. This function should additionally be supported by the Web Service API.

4.6. Non-functional Requirements

This section describes non-functional requirements that a productive version of JBIMulti2 should satisfy. The quantity structure describes, with how many computing resources of different types the system must cope. Moreover, considering the fact that JBIMulti2 acts as building block of a PaaS platform, we focused on different software qualities.

Quantity Structure

One JBIMulti2 instance connected to a cluster of two ServiceMix instances should handle the following quantities without impact on other non-functional requirements:

- The system can store 1000 registered service assemblies or services, 10 tenants with each having 1000 tenant users and corresponding records in the Configuration Registry.
- The system can run 100 service assemblies and 10 JBI components in total on one ServiceMix node.
- The system allows 50 tenant users or system administrators to execute use cases (see Sect. 4.4) concurrently over the Web UI and 100 management requests per second over the Web service interface.
- A management request to the Web UI or the Web service interface is responded to in 4 seconds on normal networking conditions.

Above values are based on experienced data. In the evaluation of this work, test criteria are limited to three service assemblies, two tenants with each having two tenant users, and not having concurrent requests (see Sect. 6.2).

Software Qualities

- **Data Consistency**—Transactions implementing Atomicity, Consistency, Isolation, Durability (ACID) have to secure data integrity between the Service Registry, the Tenant Registry and the Configuration Registry. State of JBI containers does not need to be consistent with the Service Registry, the Tenant Registry and the Configuration Registry at any time. But care has to be taken that once a change request to a JBI container has been initiated, it is successfully executed within 5 minutes. If a JBI container is unavailable, it has to execute missed requests once it has restarted. The execution order of requests must be adhered to.
- **Security**—Requests of tenant users and system administrators to the Web UI and the Web service interface have to be authorized, checked for integrity and treated as confidential. That means that requests to the Web UI have to use HTTPS and requests to the Web service interface have to use WS-Security. Internal communications between JBIMulti2, the databases, and ServiceMix instances are assumed to be exchanged in a demilitarized zone.
- **Maintainability**—Source code has to be well documented and the decoupling of system components has to facilitate changes in functionality. Interfaces to databases, other middleware and file systems have to be encapsulated. The system must be expandable with dynamic policy-based service discovery and load balancing in later stages of expansion.
- **Installation Ease**—The installation procedure has to be well documented. Executing all steps should lead to a running system.

5. Design

This chapter describes an architectural and technological solution for the concepts and specified system requirements of Chapter 4. First, an overview of the system decomposition and used technologies is given in Section 5.1. Then, the design of main components is illustrated in separate sections.

5.1. Architectural Overview

As described in Section 4.1, a configuration of JBIMulti2 consists of a management application that accesses a set of databases and clusters of JBI containers. The databases store service data, tenant data, or configuration data, which reference resources of the connected JBI containers (see Fig. 5.1).

5.1.1. Components

The management application, further called Web application, has a three-tier-architecture, with a presentation layer, a business logic layer, and underlying resources (see Sect. 5.2). The presentation layer is separated into a `WebGUI` component and a `WebService` component. Whereas the business logic is accessed via a superordinate `AccessLayer` that orchestrates use cases by calling the subjacent business logic components: `ServiceRegistry`, `TenantRegistry`, `ConfigurationRegistry`, and `JBIContainerManager`.

The Java Platform, Enterprise Edition v. 5 (Java EE 5) specification describes how different roles are involved in developing enterprise applications and acts as technological foundation. The Web application can be deployed on any application server that is compliant to Java EE 5 or a later version of the specification. Providing common services, the application server has various responsibilities, such as security, thread-pooling, transaction management, or resource management [JAV06].

As a consequence, the JBIMulti2 Web application only defines abstract endpoints to databases or the JBI containers. Concrete endpoints are not defined until an application deployer maps the abstract endpoints to concrete endpoints. Additionally, the Web application uses services of the application server that provide access to underlying resources. Therefore, the Web application does not need a specific integration layer to facilitate replacing the used database or messaging middleware. The Web application is tested with PostgreSQL 9.1.1 [PSQ] as database middleware and Apache ActiveMQ 5.3.1 [AMQ] as messaging middleware.

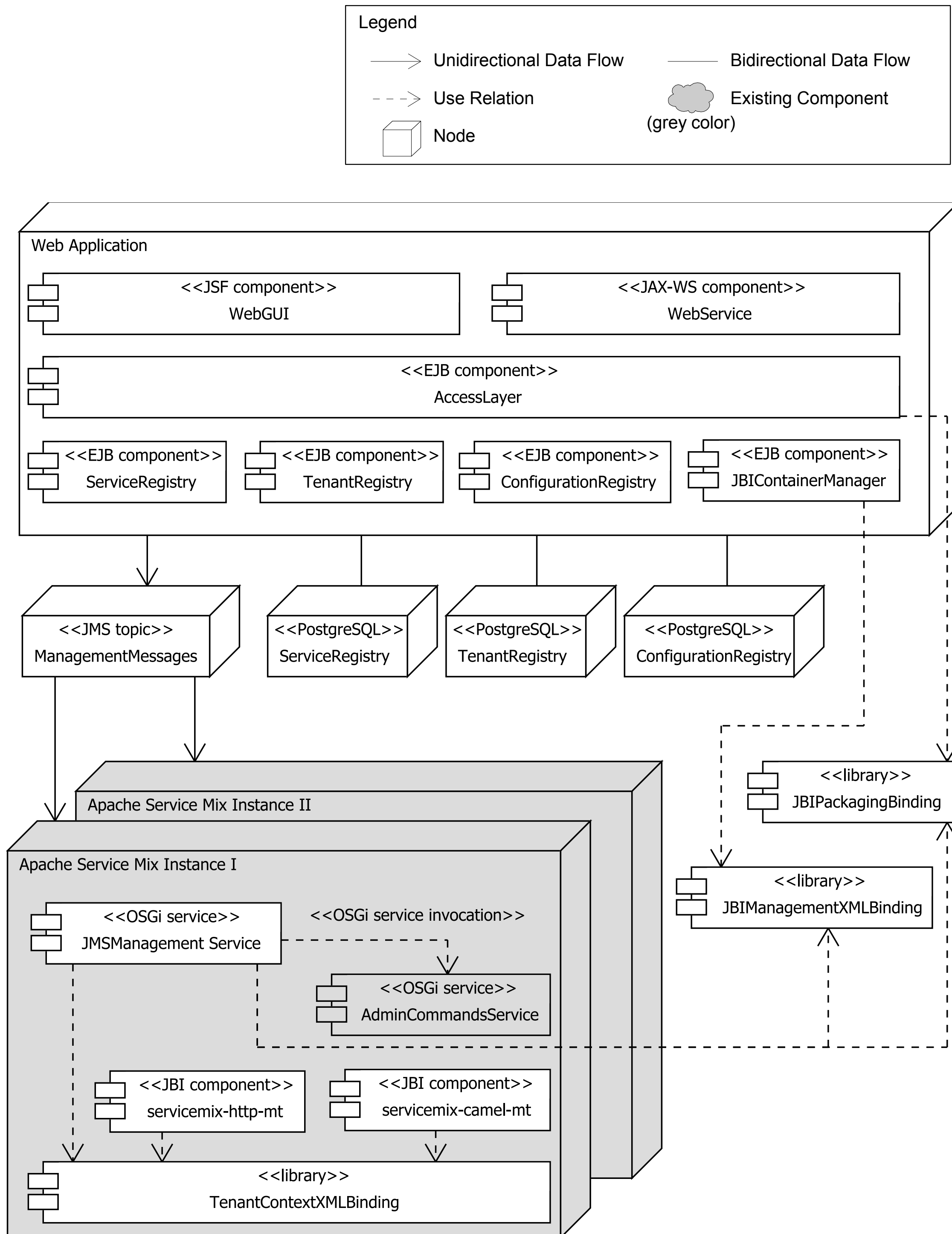


Figure 5.1.: Overview of the JBIMulti2 architecture.

Corresponding resource adapters are bundled together with the Web application in an Enterprise Archive (EAR) file (see Sect. 5.2.3).

On top of Java EE 5, a stack of technologies is built that provides APIs to implement business logic, Web services, Web frontends, and more. The Web application takes advantage of the following technologies provided by the application server:

- **Java API for XML-Based Web Services (JAX-WS) 2.0** to develop the Web service interface of the Web application. On the one hand, JAX-WS describes how Java interfaces are mapped to WSDL 1.1 and vice versa. On the other hand, JAX-WS sepecifies a Java API for developing Web service clients and Web service providers [JAX06b].
- **Java Message Service (JMS) 1.1** to interact with the JBI containers using messaging. The JMS specification defines a common messaging API for Java, intending to integrate messaging products that deliver a *JMS provider* interface [JMS02].
- **Enteprise JavaBeans (EJB) 3.0** to implement the business logic of the Web application. The EJB specification defines a component architecture that distinguishes between *stateless session beans*, *stateful session beans*, *entity beans*, and *message-driven beans*. A EJB developer defines enterprise beans by annotating his business logic classes, whereas the EJB container then takes care of instancing the enterprise beans and persisting related data. Enterprise beans can be annotated as JAX-WS clients or providers. Furthermore, enterprise beans can use container-managed connections to JMS queues and topics [EJB06].
- **JavaServer Faces (JSF) 1.2** to develop the graphical user interface of the Web application. The JSF specification defines a framework for creating Web-based graphical user interfaces. User interface components on the client side interact with *managed beans* running on the server side. Each request to the server runs through a processing lifecycle that involves validating, updating the server side model state, triggering events, and rendering a response. Its possible to instantiate managed beans in *request scope*, *session scope* or *application scope*. References to EJB components can be injected to managed beans [JSF06].
- **Java Architecture for XML Binding (JAXB) 2.0** to handle XML files. It allows a developer to generate Java classes from XSDs and an application to unmarshal XML documents to instances of these Java classes and vice versa [JAX06a].

The Java EE 5 certified Java Open Application Server (JOnAS) 5.2.2 [OWJ] is used for running the Web application. In addition to standard Java EE 5 deployment descriptors, the EAR file contains JOnAS specific deployment descriptors (see Sect. 5.2.3). Vendor specific deployment descriptors have to be added, if the Web application has to run on other application servers.

On the side of the JBI containers, the OSGi service `JMSManagementService` is developed that receives JMS messages, analyzes their content, and installs contained JBI components or deploys contained service assemblies. Both the Web application and the `JMSManagementService` use the same separate libraries to marshal and unmarshal common XML documents

and JBI artifacts. Modified JBI components provide multi-tenant awareness by considering the tenant context contained in service units (see Sect. 5.4).

5.1.2. Integration

Accessing separate databases and JBI containers, the Web application must ensure consistency, when performing data operations. Thus, all operations to underlying resource managers are controlled by distributed transactions. The two-phase commit protocol defines the interaction between a transaction coordinator and resources that are modified during the distributed transaction.

A Java EE 5 application server can manage distributed transactions. The developer of EJB components can choose between *bean-managed transaction demarcation* and *container-managed transaction demarcation*. The former provides the developer with an API to start and commit distributed transactions. Whereas the latter allows the developer to define transaction attributes for whole business methods, which includes all resource changes inside the method to the distributed transaction. In both cases, the EJB developer does not need to take care about the internal two-phase commit protocol between the EJB container and the resource managers. Within the transaction context, all resource changes that the EJB developer initiates are controlled by the EJB container [EJB06].

The JBIMulti2 Web application uses container-managed transaction demarcation for distributed transactions between the Tenant Registry database, Service Registry database, Configuration Registry database, and JBI containers. As the system can contain many JBI containers, including every JBI container as resource manager to the distributed transaction may produce a performance bottleneck. Thus, the Web application subdivides the transaction to the JBI containers using messaging with *guaranteed delivery* [HW03]. In a first step, management operations to the JBI containers are sent to a messaging topic under the control of a distributed transaction. The distributed transaction commits successfully, as soon as the message is safely stored in the message topic. In the second step, a JBI container then acts as a selective, transactional, and durable subscriber. Therefore, the JBI container receives all messages in the topic, even if the topic or the JBI container is temporary unavailable. Moreover, a transaction between the topic and the JBI container ensures that the message is successfully processed before it gets deleted from the topic.

As API for messaging JMS is used. There is no Web service interface between the Web application and the JBI containers. By the time of this work, a specification for SOAP over JMS exists only as World Wide Web Consortium (W3C) working draft [SOA10]. Relying on other WS* specifications [WCL+05] would mean to integrate an additional component that stores and forwards messages to the JBI containers. Instead, we decided to use messaging over JMS and thus can leverage the message-oriented middleware of the ESB.

The stateless session bean `JBIContainerManager` sends JMS messages on the side of the Web application. Whereas the OSGi service `JMSManagementService` acts as message receiver on the side of the JBI container. Both components rely on the common, JAXB 2.0 based library

JBIManagementXMLBinding (see Fig. 5.1) to marshal the message content into XML before sending and unmarshal the message content again after receiving.

5.1.3. Processing of JBI Artifacts

Users upload service assemblies and JBI components to JBIMulti2 as single ZIP files. As a consequence, JBIMulti2 must process the contents of those artifacts and, in doing so, encapsulates convenient functionality into the separate library JBIPackagingBinding. This library is used by the Web application as well as the JMSManagementService installed on the JBI containers (see Fig. 5.1).

The Web application extracts data from uploaded service assemblies and JBI components, allowing it to implement access control mechanisms on the level of JBI service units. Furthermore, the JMSManagementService must process JBI artifacts it has received. This includes adding a tenant context XML document to all service units contained in a service assembly (see Sect. 5.4).

The JBI specification states that all JBI artifacts are packaged as ZIP archive and contain a *jbi.xml* deployment descriptor. Additionally, a service assembly contains a ZIP archive for each included service unit, which are themselves described by a *jbi.xml* deployment descriptor [JBI05]. Apache SerivceMix delivers a XSD file that provides all deployment descriptor definitions from the JBI specification [ASM]. The JBIPackagingBinding uses this XSD for a JAXB binding to Java classes. Moreover, it encapsulates the processing functionality that is necessary to read and modify ZIP files.

5.2. Web Application

As described in the preceding architectural overview, the management application controls clusters of ServiceMix instances via messaging. It is a three-tier enterprise application leveraging Java EE 5 technology. This section goes into more detail, regarding the decomposition of the business logic layer, the realization of the Web Service API, and the packaging of the enterprise application as an EAR.

5.2.1. Business Logic Access Layer

Marinescu [Mar02] describes design patterns for EJB projects, including common architectural approaches. The EJB design pattern *Session Façades* explains how an EJB developer can encapsulate the execution of use cases to minimize the number of calls to the EJB container. Rather then accessing business logic objects directly, clients initiate use cases on a superordinate layer, the Session Façade. The EJB developer can choose to provide multiple session beans, wrapping groups of use cases.

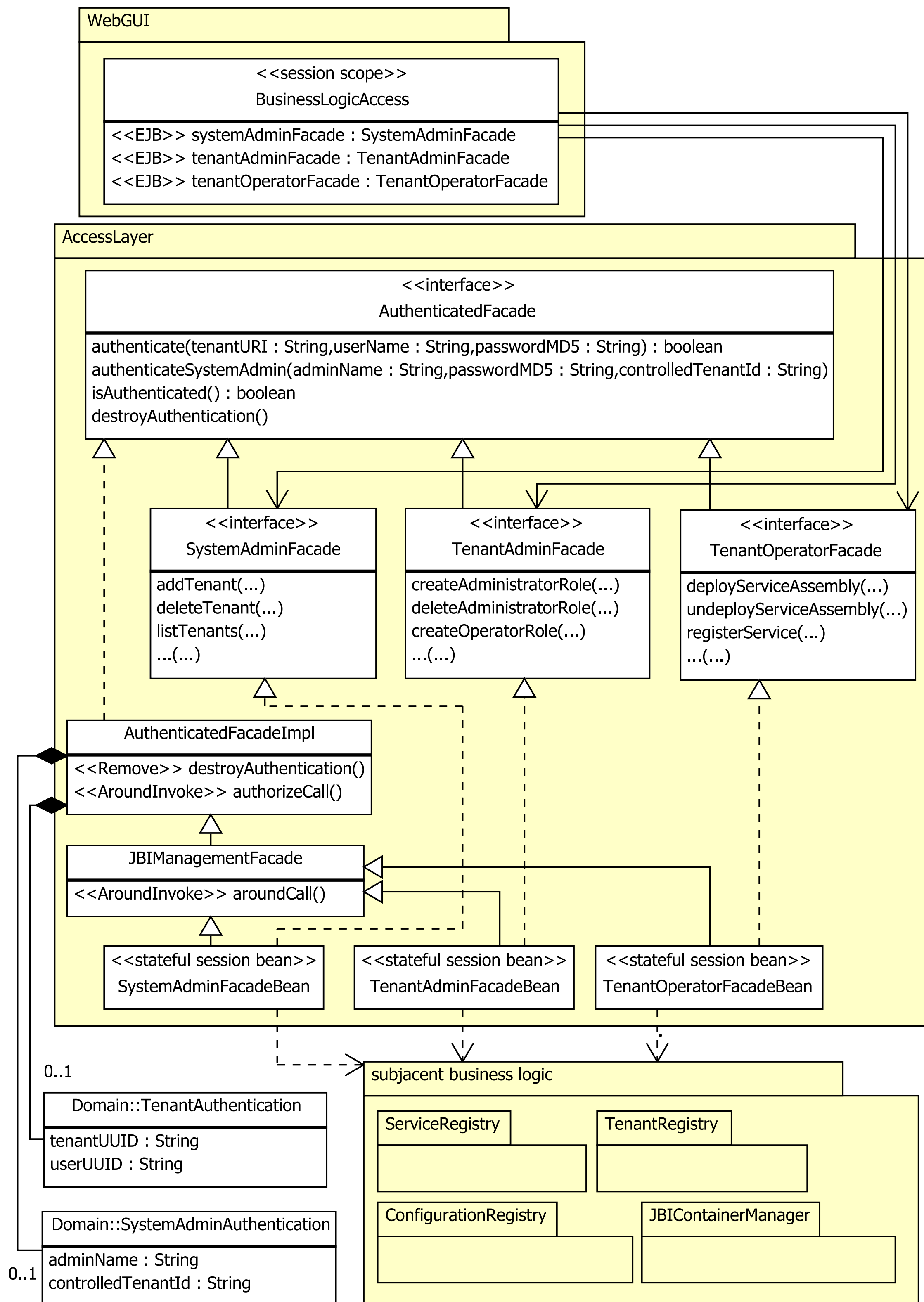


Figure 5.2.: Business logic layer `AccessLayer`, acting as Session Façade, multi-tenancy enablement layer and transaction demarcation.

The business logic layer of the JBIMulti2 Web application applies the Session Façade pattern with a superordinate AccessLayer (see Fig. 5.2). The three stateful session beans SystemAdminFacadeBean, TenantAdminFacadeBean, and TenantOperatorFacadeBean provide a method for each use case described in Section 4.4. A method call to one of these façades also demarcates a distributed transaction, including all resource changes in the underlying call tree.

Furthermore, AccessLayer acts as a *multi-tenancy enablement layer* [GSH+07]. Use case methods do not need a tenant context as parameter. When a user logs in to the graphical user interface, the WebGUI initializes the façade enterprise beans by authenticating the user with his tenant name, user name, and password. From this moment, all method calls to the façade enterprise bean are internally mapped to the proper tenant context. The superclass AuthenticatedFacadeImpl implements the authentication process and holds the tenant context as instance of TenantAuthentication, comprising the primary keys tenantUUID and userUUID. It is the responsibility of the JSF session scoped bean BusinessLogicAccess to correctly inject and destroy the façade enterprise beans of the user.

A third function that the AccessLayer provides is access control. Session bean methods that are annotated with the EJB annotation AroundInvoke are interceptors called by the application server before executing a business method. Such an interceptor is implemented to check, if the caller is authenticated and has a permission to execute the current business method. For this purpose, each business method is annotated with the required permission type. With this information the interceptor can check the authentication via a combined lookup in the Tenant Registry and Configuration Registry (see Sect. 5.3).

To minimize the number of JMS messages sent to the JBI containers, management commands emerging from a business method call are merged into a single JMS message. The class JBIManagementFacade contains an interceptor method that collects all management commands during method execution. The interceptor method hands over the result to the JBIContainerManager that finally sends a corresponding JMS message to the JBI containers.

5.2.2. Web Service

The JBIMulti2 Web service interface has to ensure integrity, confidentiality, and authentication of incoming messages. This means, third parties may not modify a message, inspect the contents of a message, or pretend to be the original initiator of the message. Mechanisms have to be implemented that abort the processing of a message, if one of the three security criteria can not be guaranteed.

Complying to the WS-Security specification, SOAP messages sent to the Web service can be signed and encrypted to ensure security criteria. Security tokens contain keys or other data that allow the receiver to validate signatures of message parts or decrypt message parts. Such information is included as SOAP header elements [OAS06a]. The Web service relies on the *WS-Security X.509 certificate token profile*, providing asymmetric encryption using key pairs. A X.509 certificate contains the public key and uniquely associates it with the owner of the public key, thus allowing validation from a certificate authority. When a sender wants to sign

a message, he generates a hash value of the message and encrypts it using his private key. The receiver can then use the public key of the sender to decrypt the signature and check, if the message was modified, thus ensuring integrity. Furthermore, a sender can encrypt the message using the public key of the receiver to ensure confidentiality [OAS06b].

We conceive confidentiality and integrity on a per tenant base, so that individual tenant users do not need to own a X.509 certificate. Each tenant using the Web service has to know the public key of JBIMulti2. In turn JBIMulti2 has to know the public keys of all tenants. Both parties can then encrypt a message targeted to the other party and validate a message signature. Public keys of tenants are stored as X.509 certificates in the Tenant Registry, represented as key value pair, but not validated at a certificate authority. Authentication at the Web service is implemented using a custom SOAP header element called `TenantContext`. It contains a UUID of the tenant, a UUID of the user, and the user password. This header element is encrypted and signed, preventing users of other tenants to act on behave of the sending tenant. This solution expects that tenant users only have credentials for exactly one tenant, otherwise tenant UUID and used X.509 certificate had also to be compared. Listing A.3 shows the WS-Policy definitions of the Web service interface, including WS-SecurityPolicy [OAS09] assertions and a custom assertion that claims a tenant context or a system administrator context. Furthermore, Appendix A.1 contains XSDs of the JBIMulti2 Web service policies and SOAP headers. Additionally, an example SOAP message is illustrated.

On the server side the policies can be ensured by two JAX-WS handlers. The first handler checks if incoming messages contain required WS-Security SOAP headers, decrypts message parts and validates signatures. For an outgoing message, it signs the message and encrypts message parts. Then the second handler is invoked, which checks if an incoming message contains a valid tenant context. The current prototype of JBIMulti2 (see Sect. 6.1) only provides the second handler. However, the open source library Apache Web Services Security for Java (WSS4J) could be used to implement WS-Security compliant encryption and signing. WSS4J can interpret the SOAP envelope and perform all required decryption and validating steps, relying on a static or in-memory Java keystore [AWS].

The Web service has three WSDL 1.1 `portType` elements: `SystemAdminServicePortType`, `TenantAdminServicePortType` and `TenantOperatorServicePortType`. Each `portType` contains only those operations, which match to a use case of the corresponding user role. A system administrator authenticates with a token containing his user name and password. Although not belonging to a tenant, a system administrator additionally provides a tenant UUID, when he executes functions associated with this tenant. Therefore, the `binding` element of the `SystemAdminPortType` does not reference the same policy as the tenant specific port types.

5.2.3. Packaging and Deployment

Java EE 5 defines the four module types *EJB Module*, *Web Module*, *Resource Adapter Module*, and *Application Client Module*. A complete deployment of the JBIMulti2 Web application comprises modules of the first three module types. Although each Java EE module can be

deployed separately, there are interface dependencies between different modules. Therefore, deploying them as a single archive is reasonable. Furthermore, different Java EE modules should share common libraries. For this reason, Java EE 5 provides the concept of an EAR file that packages different Java EE modules and required libraries into a single file as an enterprise application [JAV06].

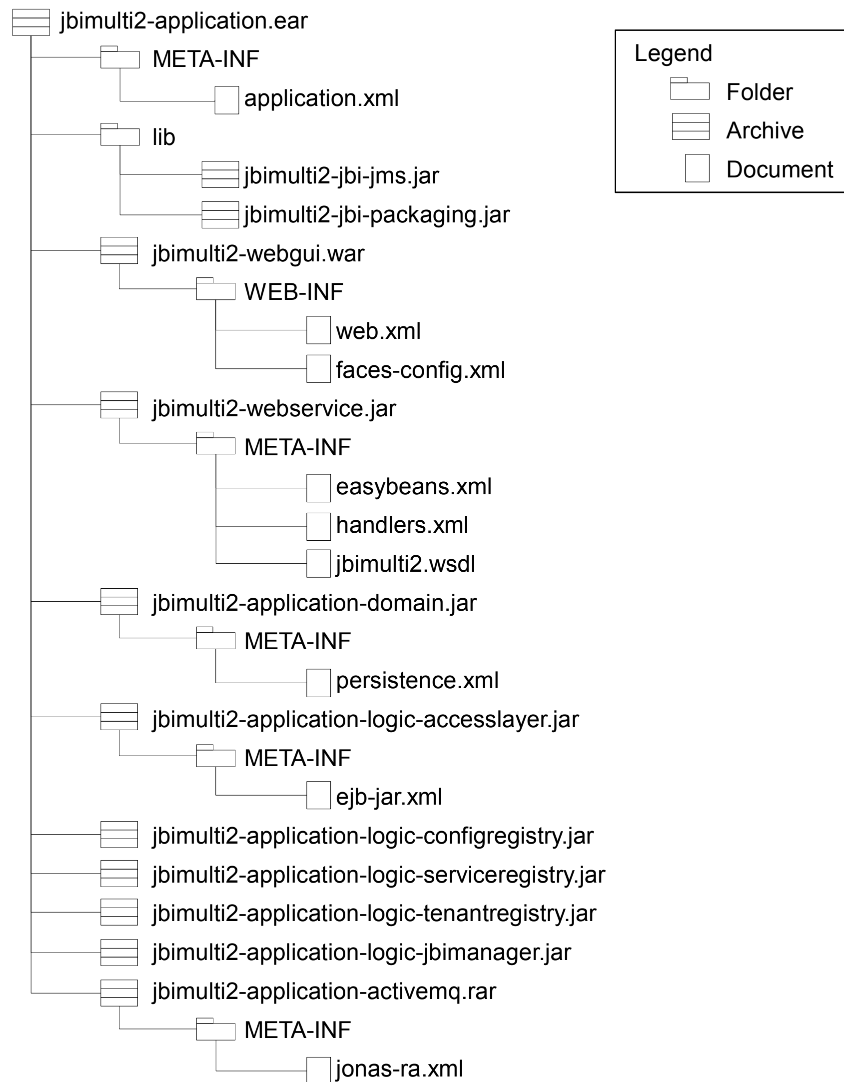


Figure 5.3.: EAR packaging of Web application. Illustrates all deployment descriptors that are JOnAS specific or that contain important configurations.

We automate the build process of all JBIMulti2 components with the tool Apache Maven [AMV]. The Maven plugin *maven-ear-plugin* facilitates the packaging of the EAR file and generates an *application.xml* deployment descriptor for the enterprise application. Other plugins package the different types of Java EE modules. The Java EE 5 specification states that a human deployer may modify deployment descriptors inside the EAR file to adjust application configurations according to the requirements of the given application server. The JBIMulti2 Web application already ships with a collection of JOnAS specific deployment

descriptors [OWJ] in different Java EE modules (see Fig. 5.3). All important configurations are located in XML documents that allow to modify system administrator credentials, JAX-WS handlers, Web context paths, JMS destinations, and persistence contexts.

5.3. Database Schemes

The database scheme of the Tenant Registry takes account of the fact that different applications must be able to use the same instance of the Tenant Registry as a shared database. Tenant users belong uniquely to one tenant, while both have a primary key represented as a UUID string. Tenant users and tenants can have arbitrary key-value pairs assigned to them, with each key-value pair only usable by a subset of applications.

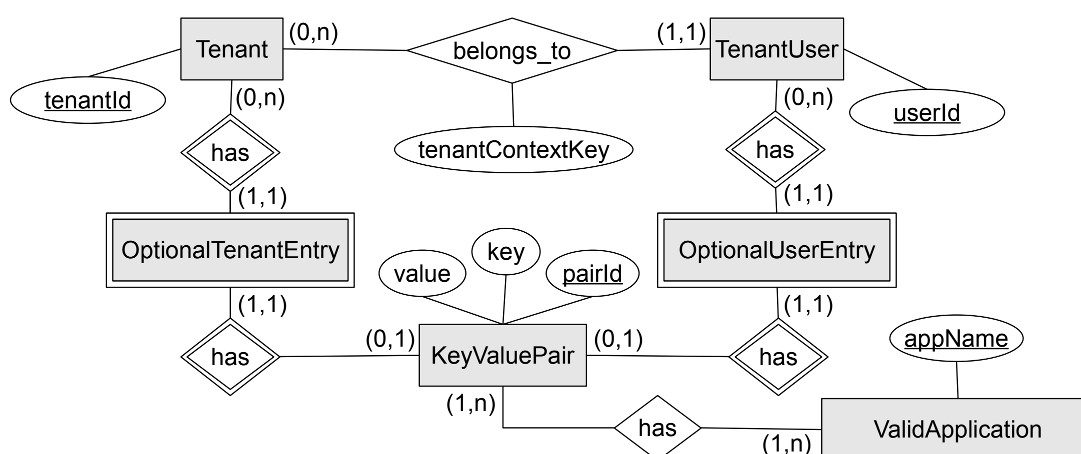


Figure 5.4.: Entity-relationship diagram of Tenant Registry using (Min,Max) Notation.

Figure 5.4 shows an entity-relationship diagram of the Tenant Registry. The attribute `tenant-ContextKey` is a UUID string that uniquely identifies a combination of tenant user and tenant. It can be contained in messages using protocols, which do not allow to add an structured tenant context with `tenantId`, `userId`, and optional key-value pairs. The `tenantContextKey` allows a message receiver to query the complete tenant context from the Tenant Registry. Rather than associating `KeyValuePair` directly with tenant user and tenant, the weak entity types `OptionalTenantEntry` and `OptionalUserEntry` represent intermediate relationship entities. This prevents `KeyValuePair` from having `null` entries, when transformed to a relational model. The downside of this solution is that the application has to prevent a tenant user or tenant from having two entries with the same key.

Likewise, the Service Registry has to be reusable by other applications to allow dynamic retrieval of services. As a consequence, services and service assemblies may have a valid combination of `tenantId` and `userId` as part of their primary key, but are not obligated to do so. The entity types `Service` and `ServiceAssembly` have an aggregated primary key that comprises an optional tenant context and a mandatory `serviceName` or respectively

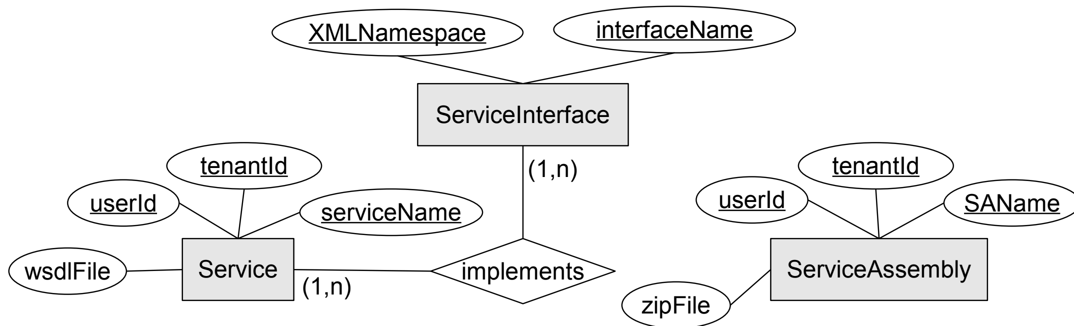


Figure 5.5.: Entity-relationship diagram of Service Registry using (Min,Max) Notation.

SAName. Moreover, interfaces a service implements are stored. This allows applications to query services for matching interface names (see Fig. 5.5). Service assembly ZIP files are stored as Binary Large Objects (BLOBs), whereas service WSDL files are stored as Character Large Objects (CLOBs).

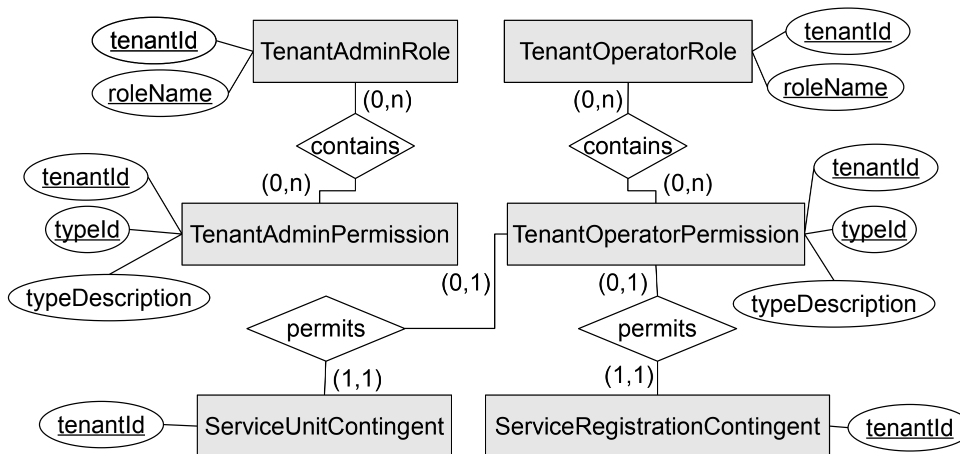


Figure 5.6.: Entity-relationship diagram of roles in Configuration Registry using (Min,Max) Notation.

The Configuration Registry ensures data isolation between tenants by having a tenantId primary key on entity types as described in Chong et al. [CCW06] (see Fig. 5.6). It references the tenant in the Tenant Registry. Each tenant can have entities of type TenantAdminRole and TenantOperatorRole, which are associated with pre-defined permissions and can be assigned to tenant users. For each ServiceUnitContingent and ServiceRegistrationContinent a corresponding permission entity is created automatically.

The amount value of a contingent entity is a share of a ServiceUnitQuota or ServiceRegistrationQuota entity (see Fig. 5.7). Each ServiceUnitQuota is uniquely related to a JBIComponent entity and a tenant. A JBIComponent entity stores its content as ZIP file and can be associated with many JBIContainer entities. Not belonging to a tenant, the entity types JBICluster, JBIContainer, and JBIComponent do not have a tenantId attribute.

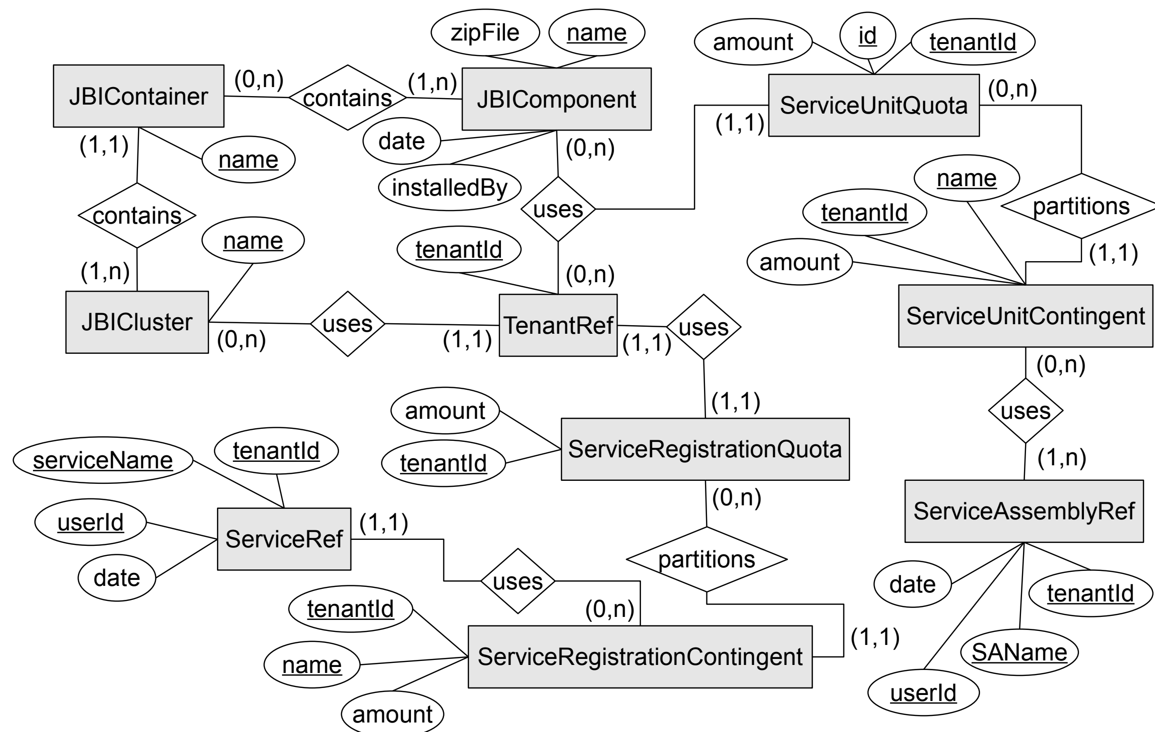


Figure 5.7.: Entity-relationship Diagram of JBI components in Configuration Registry using (Min,Max) Notation.

One of the three EJB specification documents [EJB06] describes Java Persistence API (JPA), providing *object/relational mapping*, which automatically transfers database records to entity bean instances. By annotating entity beans with meta data, a developer can influence this process. There are Java annotations for defining class attributes as primary keys, for defining relations between entity beans and their cardinality, or for defining table names and row names. The business logic layer of the JBIMulti2 Web application (see Fig. 5.2) contains a `Domain` component with entity beans implementing the entity-relationship model described in this section. Entity bean instances are persisted by using the JPA `EntityManager`. Once persisted, an entity bean instance is associated to a database row and modifications to the entity bean instance can be passed down.

5.4. Extensions to ServiceMix

For the JBIMulti2 system to work, extensions to the underlying Apache ServiceMix middleware are necessary. As ServiceMix does not ship with a Web service interface for managing JBI artifacts, we have designed an OSGi blueprint bundle that consumes management messages coming from the Web application. Moreover, message flows resulting from deployed service assemblies of tenant users must be isolated from message flows of other tenants. For this purpose, multi-tenant JBI components are introduced that read the tenant context from service units to ensure data isolation.

5.4.1. Management Interface Over Messaging

A selective message consumer, called `JMSManagementService`, is implemented for ServiceMix that receives management messages from the JBIMulti2 Web application. The management messages are sent to a JMS topic all ServiceMix instances subscribe to. As ServiceMix does not provide a Web service interface for installing JBI components or deploying service assemblies, the `JMSManagementService` calls methods on an existing internal JBI administration OSGi service (see Fig. 5.1). Therefore, we leverage the OSGi platform [OSG11] ServiceMix is built on. The `JMSManagementService` is also implemented as an OSGi bundle that has to be installed, configured, and started on an ServiceMix instance, prior to registering the instance to the JBIMulti2 Web application.

Messages sent to the `JMSManagementService` contain a combination of the following elements, each representing a management command:

- *JBI Component Install Command* instructs the ServiceMix instance to install all JBI components sent together with the command as binary data.
- *JBI Service Assembly Deploy Command* instructs the ServiceMix instance to deploy all service assemblies sent together with the command as binary data. For each service assembly a tenant context element is mandatory.
- *JBI Service Assembly Undeploy Command* instructs the ServiceMix instance to undeploy all service assemblies referenced by name. For each service assembly a tenant context element is mandatory.
- *JBI Component Uninstall Command* instructs the ServiceMix instance to uninstall all JBI components referenced by name.

One message can be addressed to many JBI containers. If a message contains different management commands, they are processed in the order above. As `JMSManagementService` is a *durable subscriber* and a *transactional client* [HW03], messages are guaranteed to be processed in the order they were sent by the JBIMulti2 Web application. If a message is malformed or invalid, it is sent to a *dead letter queue*.

Messages contain a XML document with the root element `JBIManagement` (see Appendix A.2). Binary data of JBI components and service assemblies is encoded as Base64 [FB96] character strings and added as sub elements to the corresponding command element of a message. To serialize XML documents in the JBIMulti2 Web application and to deserialize them again in the `JMSManagementService`, the JAXB 2.0 based library `JBIManagementXMLBinding` is used (see Fig. 5.1).

Before the `JMSManagamentService` deploys a service assembly, it reads the tenant context from the JMS message and then adds the `tenantId` and `userId` to the service assembly name, service unit names, and JBI deployment descriptors. This makes sure that different tenant users can deploy service assemblies and service units with the same name. The `JMSManagementService` must add the tenant context as a separate XML document to each service unit contained in the service assembly, allowing data isolation on the level of JBI

components. This solution aims to implement multi-tenancy, while still complying with the JBI specification. A XSD containing the tenant context can be found in Listing A.1. We have developed the tenant context XSD together with Essl [Ess11].

5.4.2. Multi-tenant JBI Components

A multi-tenant JBI component must ensure that a message exchange only involves service units of the same tenant. We achieve this by introducing multi-tenant aware internal JBI service endpoints. A service endpoint comprises a service name and an endpoint name. To ensure data isolation, we replace the namespace prefix of the service name by a namespace that contains the `tenantId` (see Listing 5.1). Interfaces are not multi-tenant aware in this approach. Each JBI component provides a service unit manager that manages the life cycle of service units, such as deploying, starting, or stopping [JBI05]. A multi-tenant aware service unit contains a tenant context, represented as XML document. The service unit manager of a multi-tenant JBI component must read the tenant context and inject it into those classes that implement service endpoint registrations and message exchanges.

All JBI components included in ServiceMix are inherited from the common class `DefaultComponent`. In turn, this class and its base classes implement the two JBI interfaces `Component` and `ComponentLifeCycle`, thus fulfilling the JBI contract. The former provides runtime information to the JBI container, such as the service unit manager, used for deploying service units. The latter provides life cycle control to the JBI container, such as starting and stopping the JBI component.

At runtime, the `DefaultComponent` manages a registry containing objects that implement the internal interfaces `Endpoint` and `ServiceUnit`. The class `BaseServiceUnitManager` is used by `DefaultComponent` as JBI service unit manager. It manages the life cycle of service units and adds deployed service units to the registry. But instead of deploying service units given by file path on its own, the `BaseServiceUnitManager` delegates this work to subordinate deployers. There are two main types of deployers that rely on different artifacts contained in a service unit: a *WSDL Deployer* generates endpoints from WSDL files, and a *XBean Deployer* generates endpoints from a XBean file. The `BaseServiceUnitManager` asks consecutively every deployer, if the given service unit contains the desired artifact. If the deployer confirms, it is then commanded to process the deployment process. The result of the deployment process is a `ServiceUnit` implementing instance that comprises all generated `Endpoint` implementing instances. Each subclass of `Endpoint` processes JBI message exchanges and implements a prescribed life cycle.

To ensure data isolation, multi-tenant aware endpoints and the new deployer `XBeanDeployerMT` are implemented. In addition to creating endpoint objects from a XBean file, `XBeanDeployerMT` reads a tenant context file. The tenant context is injected into all multi-tenant aware endpoints that have been created in the process. Each multi-tenant aware endpoint implements the interface `TenantEndpoint`, which is a subtype of `Endpoint` and expects a tenant context with the `tenantId`. For target service endpoints the tenant operator may add

a option to the XBean definition that inhibits replacing the service namespace. This allows targeting common platform-wide service endpoints.

Regarding the Apache Camel SE, the previous insights keep its validity, even though the `CamelJbiComponent` is only a wrapper for the external Apache Camel implementation. There is already a separate camel context created for each deployed service unit, thus data isolation is ensured inside Apache Camel. Nevertheless JBI service endpoints extracted from the camel context by the `CamelJbiComponent` have to be extended with multi-tenant awareness.

```
1  /*
2    input: tenantId, tenantUri, serviceLocalPart, endpointName, configuredLocationUriPrefix
3    example: http://localhost:8193/tenant-services/54ed4755-5965-4b47-a121-d25907e29c04/
         ExampleService/ep
4   */
5  locationUri       ::= locationUriPrefix ( tenantId | tenantUri ) serviceEndpoint
6  locationUriPrefix ::= "http://localhost:8193/tenant-services/" | configuredLocationUriPrefix
7  serviceEndpoint   ::= "/" serviceLocalPart "/" endpointName
8
9  /*
10   input: tenantId, serviceLocalPart, endpointName, configuredServiceNamespacePrefix
11   example: {jbimulti2:tenant-endpoints/54ed4755-5965-4b47-a121-d25907e29c04}ExampleService:
         ep
12  */
13 serviceEndpoint        ::= serviceName ":" endpointName
14 serviceName            ::= "{" serviceNamespacePrefix tenantId "}" serviceLocalPart
15 serviceNamespacePrefix ::= "jbimulti2:tenant-endpoints/" | configuredServiceNamespacePrefix
```

Listing 5.1: Location URI (top) and service endpoint (bottom) replacing patterns for the multi-tenant HTTP BC *servicemix-http-mt* in Extended Backus–Naur Form (EBNF).

The original ServiceMix BC for HTTP accepts a location URI defining, where to listen to requests. This parameter is no longer allowed in the multi-tenant aware version of the component. Instead, the location URI is dynamically built from the given tenant context. The generated location URI is a concatenation of a common prefix and either the `tenantUri` or the `tenantId`, if the context does not contain a tenant specific URI (see Listing 5.1).

Similar to the approach described in Section 3.1.2, a multi-tenant Apache ODE SE can be realized. For this purpose, only the JBI integration layer of Apache ODE must be extended. The internal process store of the Apache ODE JBI component uses the service unit file path as the location of deployment units. As we separate service units of different tenants by a name pattern inside ServiceMix, the Apache ODE process store is already multi-tenant aware. A modified service unit manager in the JBI integration layer must load the tenant context prior to delegating the deployment process to the process store. The tenant context is then used throughout the JBI integration layer to ensure multi-tenant aware service endpoints.

6. Implementation and Evaluation

Various challenges occurred during the implementation of a prototype that complies to the concepts in Chapter 4 and conforms to the design of Chapter 5. This chapter describes chosen implementation challenges in more detail. Furthermore, we evaluate the developed prototype by utilizing it as an integration system for the Taxi Scenario (see Sect. 1.1).

6.1. Implementation

The current prototype of JBIMulti2 has been implemented as a hierarchical Maven project [AMV]. Apache Maven executes various tasks during the build process of the system. Different Maven plugins have been used to adapt the build process to the needs of the JBIMulti2 components. This includes packaging Java EE modules and aggregating them to an EAR, including a JOnAS specific deployment descriptor to the ActiveMQ resource adapter, generating source code from XSDs for JAXB, packaging the `JMSManagementService` as an OSGi blueprint bundle, and setting up projects for the modification of existing JBI components. With the help of Apache Maven developers can modularize software systems, restricting the dependencies between modules. Altogether, the current prototype consists of two main Maven modules that implement the JBIMulti2 Web application and the Apache ServiceMix extensions. Additionally, a third Maven module contains components for the Taxi Scenario tailored to JBIMulti2. Project files for the Integrated Development Environment (IDE) Eclipse 3.7 have been generated with Apache Maven and used during development. All Java source code was compiled with the Java Development Kit (JDK) 6.

Out of scope is the implementation of a Web-based graphical user interface described in Section 4.5.1, the implementation of a JAX-WS handler for controlling the WS-Security policies designed in Section 5.2.2, the implementation of Web service operations that go beyond the use cases of Section 4.4, and the development of a multi-tenant Apache ODE [AOD] JBI component.

6.1.1. Java Annotations for Role-based Access Control

As described in Section 5.2, an EJB method interceptor is used to check authentication and authorization of each call to the business logic access layer of the JBIMulti2 Web application. In avoidance of duplicate source code, business methods do not check authorization of calling users themselves. Instead, authorization functionality is taken over by the EJB method interceptor `authorizeCall()` defined in the superclass `AuthenticatedFacadeImpl` of all façade enterprise beans (see Fig. 5.2). The EJB interceptor is a method

that is annotated with the EJB annotation `@AroundInvoke` [EJB06] and executed for every business method call. A method annotated with `@AroundInvoke` must claim an instance of `javax.interceptor.InvocationContext` as method parameter. It can execute functionality preceding and following the targeted business method call. For this purpose, the interceptor calls the business method by invoking `InvocationContext::proceed()` at any point of execution.

```
1      ...
2
3      @PermissionType(type = PermissionTypeEnum.MANAGE_TENANT_OPERATOR_ROLES)
4      public Collection<TenantOperatorPermissionEntry> listOperatorPermissions(String roleName
           ) throws AuthorizationException, ExecutionException {
5          ...
6      }
7
8      @PermissionTypes({ @PermissionType(type = PermissionTypeEnum.MANAGE_TENANT_ADMIN_ROLES),
            @PermissionType(type = PermissionTypeEnum.MANAGE_TENANT_OPERATOR_ROLES) })
9      public Collection<TenantUserEntry> listTenantUsers() throws AuthorizationException,
           ExecutionException {
10         ...
11     }
12
13     @PermissionType(type = PermissionTypeEnum.MANAGE_SERVICE_UNIT_CONTINGENTS)
14     public ServiceUnitContingentEntry createServiceUnitContingent(String jbiComponent,
           String contingentName, int amount) throws AuthorizationException, ExecutionException
            {
15         ...
16     }
17
18     ...
```

Listing 6.1: Excerpt from `TenantAdminFacadeBean.java` focusing on Java annotations that define required permissions. A tenant administrator must have the respective permissions to execute a business method. Only three chosen business methods are shown and implementation details are omitted.

The EJB method interceptor `authorizeCall()` queries the Tenant Registry and the Configuration Registry to check if the caller has a matching permission in one of his roles to execute the current business method. Whenever a system administrator or tenant user invokes one of the authentication methods in the superclass `AuthenticatedFacadeImpl`, an authentication context is assigned to a private instance variable. The method interceptor can then authorize a method invocation using the identification information in the previously stored authentication context. As the method interceptor must not check authorization of the authentication methods, the method interceptor immediately accepts all invocations of business methods declared in the superclass `AuthenticatedFacadeImpl`. Thus, the method interceptor only checks authorization for invocations of business methods defined in the `SystemAdminFacadeBean`, `TenantAdminFacadeBean`, and `TenantOperatorFacadeBean`. These façade enterprise beans annotate business methods with the Java annotation `@PermissionType` that inform the method interceptor of the permission a caller must have. Where a business method allows choosing from multiple permission types, the aggregation annotation `@PermissionTypes`

is used (see Listing 6.1). The method interceptor retrieves all annotations of the targeted business method via the `InvocationContext` object.

6.1.2. OSGi-based Management Service for Apache ServiceMix

We have implemented the OSGi blueprint bundle `JMSManagementService` that must be deployed on Apache ServiceMix and listens to a JMS topic for incoming management messages from the JBIMulti2 Web application (see Sect. 5.4.1). The Maven plugin *maven-bundle-plugin* by the Apache Software Foundation facilitates developing OSGi bundles. As the XML descriptor of a Maven project contains dependencies to other Maven projects, the maven-bundle-plugin can add these dependencies as requirements to the OSGi descriptor of the current Maven bundle project. In the case of the `JMSManagementService`, there are dependencies to the Apache ActiveMQ [AMQ] Maven artifact `activemq-core` and to the Apache ServiceMix Maven artifact `org.apache.servicemix.jbi.deployer`. The latter provides the interface `AdminCommandsService` that declares methods for installing JBI components and deploying service assemblies. Thus, during resolving of OSGi requirements in ServiceMix, the `JMSManagementService` is wired to an ActiveMQ JMS implementation and to a JBI deployment service. Both are already installed on ServiceMix.

As the `JMSManagementService` is an OSGi blueprint bundle, it contains an OSGi blueprint descriptor that controls the instantiation and wiring of Java objects (see Listing 6.2). The main class `JMSManagementService` is a Plain Old Java Object (POJO) that is initialized by the Blueprint Container. In this process, the Blueprint Container sets all properties of the instance to values defined in the blueprint descriptor. One property references an implementation of the `AdminCommandsService`, for which the Blueprint Container searches in the service registry of the OSGi framework. After all properties have been set, the Blueprint Container calls the *init-method* of the created `JMSManagementService` instance.

During initialization the `JMSManagementService` instance creates a JMS connection to the topic that provides JBI management messages. Additionally, a JMS connection to a dead letter queue for unprocessable messages is created. A JMS durable topic subscriber object is periodically queried for new messages in a separate thread. Once a new message has arrived, it is unmarshalled to a Java object with JAXB. For the processing of tasks contained in the incoming message, we have created a simple handler framework. The unmarshalled message is validated and processed by a chain of four handlers that consists of an `InstallJBIComponentHandler`, a `DeployServiceAssemblyHandler`, an `UndeployServiceAssemblyHandler`, and an `UninstallJBIComponentHandler`. All four handlers implement the same interface with a `validate()` method and a `process()` method. Therefore, before the handlers process the management tasks contained in the message, each must first report, if the corresponding task is valid. For instance, an invalid message contains a JBI component that is already installed. Only the handlers can directly call methods on the `AdminCommandsService`. Once all handlers have been called, the `JMSManagementService` commits the processing of the JMS message to the ActiveMQ broker. A commit is performed, even when the `JMSManagementService` has redirected the message to the dead letter queue.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="..." xmlns:cm="..." xmlns:ext="...">

    <bean id="jmsManagementService" class="de.unistuttgart.iaas.jbimulti2.jbi.servicemix.
        jmsmanagement.JMSManagementService"
        init-method="afterPropertiesSet" destroy-method="destroy">
        <property name="containerName" value="${containerName}" />
        <property name="adminCommandsService" ref="adminCommandsService" />
        <property name="connectionUrl" value="${connectionUrl}" />
        <property name="topicName" value="${topicName}" />
        <property name="deadLetterQueue" value ="${deadLetterQueue}" />
        <property name="jmsUserName" value="${jmsUserName}" />
        <property name="jmsPassword" value="${jmsPassword}" />
    </bean>

    <reference id="adminCommandsService" interface="org.apache.servicemix.jbi.deployer.
        AdminCommandsService" />

    <cm:property-placeholder persistent-id="de.unistuttgart.iaas.jbimulti2.jbi.servicemix.
        jmsmanagement">
        <cm:default-properties>
            <cm:property name="containerName" value="jbi1" />
            ...
        </cm:default-properties>
    </cm:property-placeholder>

</blueprint>
```

Listing 6.2: OSGi blueprint bundle descriptor [OSG09] of JMSManagementService referencing an OSGi service interface provided by Apache ServiceMix that allows installing JBI components and deploying service assemblies.

6.1.3. Multi-tenant Binding Component and Service Engine

As part of this work, the original ServiceMix BC for HTTP (version 2011.01) and the original Apache Camel SE (version 2011.01) [ASM] have been extended to support multi-tenancy. A future version of JBIMulti2 should also ship with a multi-tenant Apache ODE SE.

For the multi-tenant HTTP BC *servicemix-http-mt*, the design of Section 5.4.2 anticipates all necessary extensions that have been made. The two endpoint types `HttpConsumerEndpoint` and `HttpProviderEndpoint` implement the new interface `TenantEndpoint` that declares a method for applying the tenant UUID to the endpoint configuration. The `HttpSoapConsumerEndpoint` inherits from the `HttpConsumerEndpoint`. A problem occurred, when the `HttpSoapConsumerEndpoint` tried to processes the mandatory WSDL file contained in a service unit. Before publishing the WSDL file, the `HttpSoapConsumerEndpoint` replaces the location URL of the SOAP binding with the proper Web address. As service endpoint names are replaced to support multi-tenancy, the `HttpSoapConsumerEndpoint` was no longer able to find the correct service definition in the WSDL file. Thus, the corrected version of the multi-tenant HTTP BC uses the original service endpoint names when processing WSDL files.

For the multi-tenant Camel SE *servicemix-camel-mt*, the design of Section 5.4.2 only holds true for the `CamelProviderEndpoint`. As other JBI endpoints are targeted dynamically, depending on the Camel routing definitions, there is no explicit consumer endpoint object. However, a class named `CamelConsumerEndpoint` exists that dynamically transforms Camel URIs to JBI service endpoint names for outgoing message exchanges. The multi-tenant version of the `CamelConsumerEndpoint` contains the method `configureTenancyAwareExchange()` that replaces the original transformation method to also include the tenant UUID.

It should be noted that from version 4.0.0 of Apache ServiceMix all JBI components are packaged as OSGi bundles rather than as JBI packages. We noticed some incompatibility issues, when deploying them as JBI packages via the `AdminCommandsService`. Particularly, the Apache ServiceMix JBI shared library must be deployed additionally, otherwise the Java class loader does not find necessary libraries. Furthermore, it was not possible to use a `HttpSoapProviderEndpoint` with a JBI packaged HTTP BC.

6.2. Evaluation

For evaluation of the developed prototype, the Taxi Scenario (see Sect. 1.1) has been modified to use Apache ServiceMix for communication between the involved Web applications. For this purpose, JBIMulti2 is used to manage tenants and to deploy the required integration services on one instance of Apache ServiceMix.

6.2.1. Deployment and Initialization

The evaluation takes place on a single virtual machine. The test system keeps the Web applications of the Taxi Scenario prototype [Hag11] deployed on a separate Apache Tomcat 7.0.23 [ATC] Web container. An instance of the Java EE 5 certified application server JOnAS 5.2.2 [OWJ] hosts the JBIMulti2 EAR package. Currently, the Taxi Scenario Web applications can not be deployed on JOnAS, because they ship with Java libraries that are also provided by JOnAS, leading to class loading failures.

Three database connections are configured on JOnAS that connect to the Tenant Registry, Service Registry and Configuration Registry created on PostgreSQL 9.1.1 middleware [PSQ]. In this process, the corresponding `postgresql-9.1-901.jdbc3.jar` Java Database Connectivity (JDBC) driver must be installed on JOnAS. Additionally, one instance of Apache ServiceMix 4.3.0 [ASM] has been installed. To establish the connection between ServiceMix and the JBIMulti2 business logic, the `JMSManagementService` OSGi bundle must be deployed on ServiceMix. According to the initial settings of the `JMSManagementService`, management messages are expected on the ActiveMQ 5.3.1 [AMQ] message broker that runs inside ServiceMix. The `jonas-ra.xml` deployment descriptor (see Sect. 5.2.3) initially targets the standard URL of this internal message broker. Therefore, no configuration is required on the side of JOnAS.

To interact with JBIMulti2 over the Web Service API, we use soapUI 4.0.1 [SOA], a graphical SOAP-based Web services testing tool. Once Tomcat, JOnAS, PostgreSQL, and ServiceMix are running, the system administrator registers the ServiceMix instance to JBIMulti2 via the Web Service API. The logical name of each individual ServiceMix instance is defined in the configuration file of the `JMSManagementService` OSGi bundle. Then, the system administrator installs the multi-tenant HTTP BC and the multi-tenant Camel SE. For this purpose, the JBI component ZIP files are encoded as Base64 [FB96] character strings and included in the respective SOAP request. From this moment on, JBIMulti2 is operational and the system administrator can begin to add tenants to the Tenant Registry.

The Taxi Scenario is adapted for the evaluation of the multi-tenancy capabilities of JBIMulti2, with tenants representing taxi companies. Thus, the system administrator adds two tenants (see Fig. 6.1), assigns them to the cluster of the ServiceMix instance, and for both tenants adds two tenant users. As a consequence, the system administrator can configure the quota of service units for each tenant and each JBI component installed on the ServiceMix instance. Finally, one tenant user per tenant obtains a tenant administrator role from the system administrator. In doing so, the system administrator configures entities belonging to a tenant. Therefore, the corresponding tenant UUID is additionally provided in the SOAP header.

At this point, the tenant administrator can appoint a tenant operator. The tenant administrator creates service unit contingents and adds the corresponding permissions to the tenant operator role, enabling the tenant operator to deploy service assemblies (see Fig. 6.2). In conclusion, each taxi company can now deploy service units to integrate the Taxi Company Web application, the Taxi Transmitter Web application, and the external Taxi Service Provider Process (see Sect. 1.1).

For the configuration steps described above, we have added an appropriate soapUI project to the deliverables of this work. It contains a test case that gradually sends the required SOAP requests.

Figure 6.1.: Add Tenant Request executed by system administrator with soapUI 4.0.1 [SOA]. On the left side the SOAP request message is displayed, while on the right side the corresponding SOAP response message is displayed. The response contains the UUID generated by JBIMulti2 for the tenant.

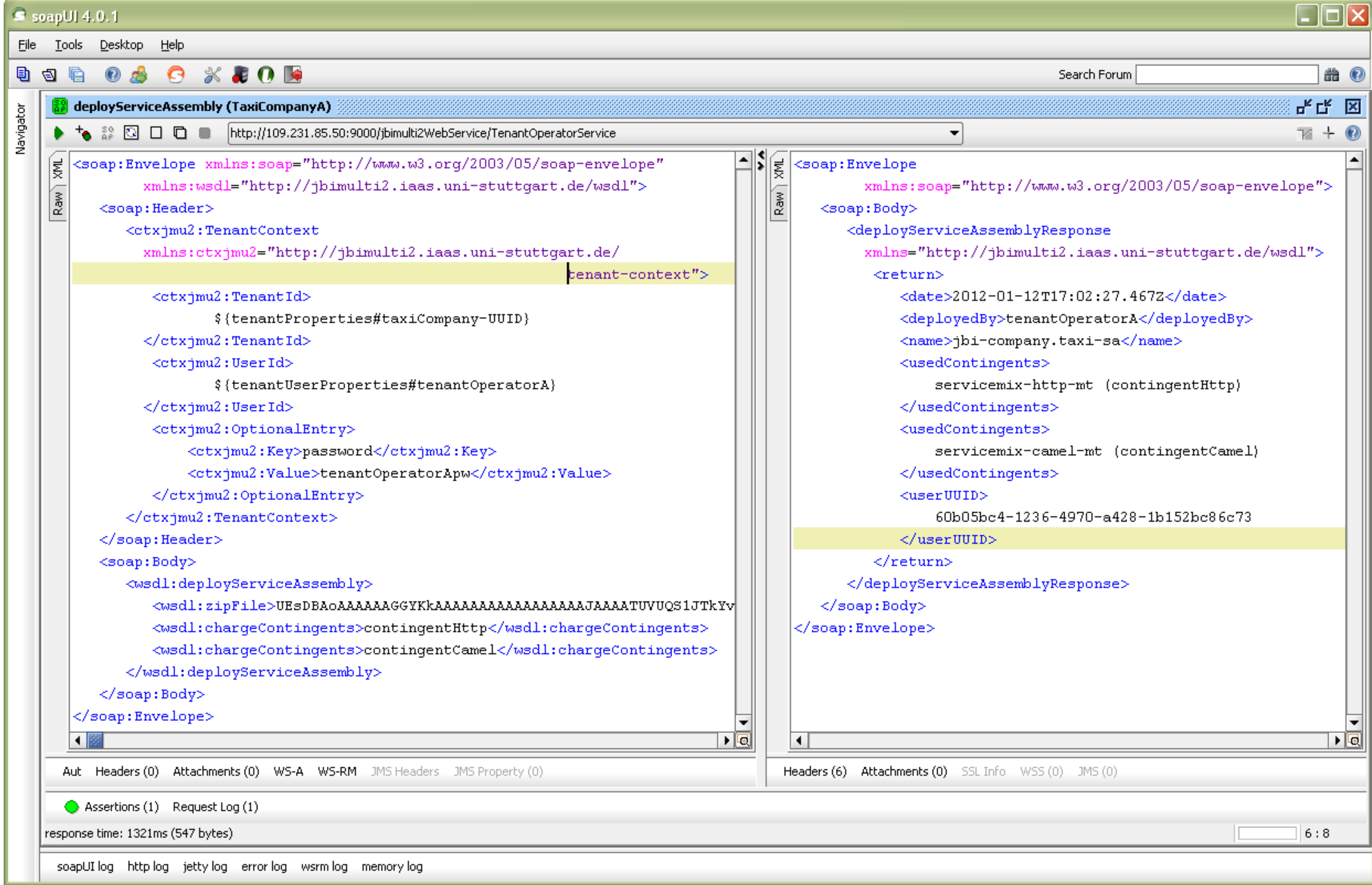


Figure 6.2.: Deploy Service Assembly Request executed by tenant operator with soapUI 4.0.1 [SOA]. On the left side the SOAP request message is displayed, while on the right side the corresponding SOAP response message is displayed. The tenant context contains soapUI variables referencing a tenant operator that belongs to the tenant named Taxi Company.

6.2.2. Tenant Context-based Routing

For this evaluation we assume that taxi companies host the Taxi Company Web application and the Taxi Transmitter Web application on an own servlet container. Both Web applications communicate with the Taxi Service Provider Process via a single ServiceMix instance and are not extended regarding multi-tenancy. By contrast, the Taxi Service Provider Process should be multi-tenant aware, which is not the case in the current Taxi Scenario prototype [Hag11]. Therefore, we have extended the interfaces of the Taxi Service Provider Process to accept a tenant UUID as an additional XML element. The multi-tenancy business integration patterns by Mietzner et al. [MUTL09] are used to integrate the non-multi-tenant aware Web applications of the taxi company with the multi-tenant aware BPEL process of the taxi service provider (see Fig. 6.3). As the taxi company Web applications are not multi-tenant aware, this evaluation is not based on the work of Essl [Ess11]. He describes how BCs add the tenant context to the JBI normalized message format as a set of message properties, separating it from the message metadata.

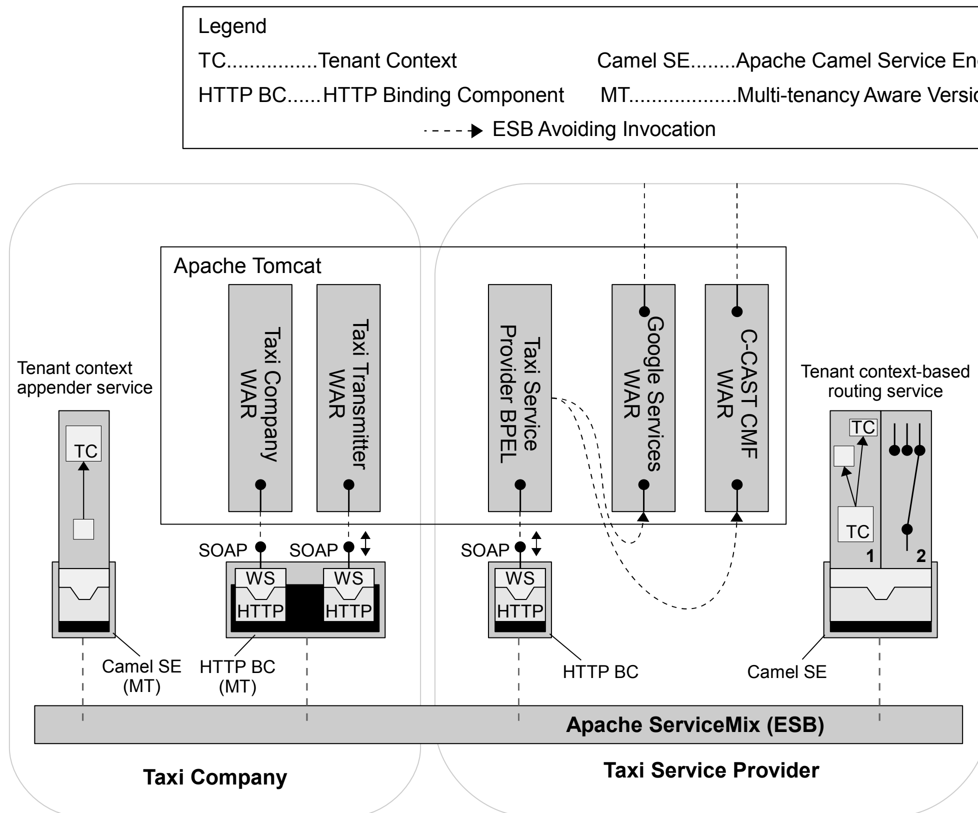


Figure 6.3.: Integrate Taxi Scenario with tenant context-based routing [MUTL09] in JBI-Multi2 environment. Uses glyphs that have been introduced by Chappel [Cha04]. For bidirectional connectivity services incoming and outgoing ESB endpoints are merged to simplify illustration.

The tenant operator of the taxi company must deploy service units that provide connectivity services to the Web service interfaces of the Taxi Company Web application and the Taxi

Transmitter Web application. Moreover, messages targeting the Taxi Service Provider Process must be enriched with a tenant context, according to the *tenant context appender* pattern. On the side of the taxi service provider, the tenant context contained in incoming booking requests is used to route messages only to those taxis that belong to the respective taxi company. This complies with the *tenant context-based router* pattern.

In this evaluation all Web applications are deployed on a single Tomcat Web container. However, tenant operators on the side of the taxi company are free to configure bindings to arbitrary Web service endpoints.

Taxi Company Integration

The current service assembly on the side of the taxi company contains three service units. Two service units are deployed on the multi-tenant HTTP BC and connect ServiceMix to the Web services of the Taxi Company Web application and the Taxi Transmitter Web application. The third service unit is deployed on the multi-tenant Apache Camel SE and appends a tenant context to messages before forwarding them to the Taxi Service Provider Process. As these service units are deployed on the multi-tenant aware version of the JBI components, created service endpoints are modified to ensure data isolation inside ServiceMix. For instance, the original service namespace of the Taxi Transmitter `{http://www.taxiserviceprovider.eu/transmitter/definitions}` is replaced with the namespace `{jbi:endpoint:jbimulti2:tenant-endpoints/`*tenantUUID*`}` for each service endpoint created by a service unit. Thus, except for the configuration of the external Web service endpoints, the service assembly can be reused by other taxi companies.

Both the Taxi Company Web application and the Taxi Transmitter Web application require a binding to the Web service port `TaxiServiceProviderPort`, provided by the Taxi Service Provider Process. Additionally, the Taxi Transmitter Web application calls the `GetTaxiDriverInformationPort` of the Taxi Service Provider Process. As the tenant operator of a taxi company deploys a service unit that substitutes the original SOAP bindings, the Web applications now bind to other endpoint URLs. Each endpoint URL generated from the multi-tenant HTTP BC consists of a common prefix, the tenant URI configured in the Tenant Registry, and the service endpoint name (see Listing 6.3).

```
1 http://hostname:8193/tenant-services/taxicompany.example.org/TaxiServiceProvider/
      TaxiServiceProviderPort/
2 http://hostname:8193/tenant-services/taxicompany.example.org/GetTaxiDriverInformation/
      GetTaxiDriverInformationPort/
```

Listing 6.3: Endpoint URLs generated for inbound SOAP bindings, resulting from service unit deployed on multi-tenant HTTP binding component. The constituent `taxicompany.example.org` matches the tenant URI defined in the Tenant Registry.

Listing 6.4 shows the configuration of the tenant context appender. The Camel route transforms all incoming messages by calling an Extensible Stylesheet Language Transformation (XSLT) document that can read the appropriate tenant UUID from the JBI message property `javax.jbi.ServiceName`. This property is provided by the JBI framework and delivers the inbound service endpoint name of the current service unit. As service endpoints must always contain the tenant UUID, this is a convenient method to transfer the tenant UUID from the service unit to a message.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="..." xmlns:xsi="..." xmlns:osgi="..."
3         xmlns:camel="..." xsi:schemaLocation="..." >
4
5      <camelContext xmlns="..." id="taxi-tenant-context-enricher-su">
6          <route streamCache="true" >
7              <from uri="jbi:endpoint:http://jbimulti2.iaas.uni-stuttgart.de/taxiscenario/
                   TaxiTenantContextEnricher/ep" />
8              <to uri="xslt:tenantContextEnrich.xsl" />
9              <inOut uri="jbi:endpoint:http://www.taxiserviceprovider.eu/definitions/
                   TaxiServiceProvider/TaxiServiceProviderPort?targetServiceGlobal=true" />
10         </route>
11     </camelContext>
12 </beans>
```

Listing 6.4: Service unit configuration `context.xml` for the multi-tenant Camel SE that appends a tenant UUID to messages targeting the Taxi Service Provider. The transformation rules are defined in a separate XSLT document.

The outbound service endpoint has the option `targetServiceGlobal` set to `true`, because the outgoing messages target service endpoints of the taxi service provider. Otherwise, the outgoing messages would be routed to an analogous service endpoint that contains the tenant UUID of the current taxi company, which does not exist.

Taxi Service Provider Integration

As opposed to the taxi company, the service assembly of the taxi service provider is global and must be directly deployed to ServiceMix without going through JBIMulti2. Therefore, the two service units are deployed on the original ServiceMix HTTP BC and Camel SE, which run in parallel to the multi-tenant aware versions. The first service unit provides a binding to the Web service interface of the BPEL process and a binding to the Web service interface of the Taxi Transmitter Web application. The former accepts outgoing booking requests of taxi customers, while the latter accepts incoming transport requests to taxi drivers. The second service unit is deployed on the Camel SE and routes transport requests to a service endpoint for the Taxi Transmitter Web application that refers to the appropriate tenant.

Listing 6.5 illustrates the configuration for the tenant context-based router. The tenant UUID is queried from incoming messages and added as the header element `tenantId` to the message. Then, the tenant UUID element is removed from the message metadata, because the Taxi

Transmitter Web application does not accept a tenant context. Finally, the message is routed to a service endpoint of the tenant that corresponds to the information contained in the `tenantId` header element.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="..." xmlns:xsi="..." xmlns:osgi="..."
3         xmlns:camel="..." xsi:schemaLocation="..." >
4
5      <camelContext xmlns="..." xmlns:soap="..."
6                    xmlns:tr="http://www.taxiserviceprovider.eu/transmitter/types"
7                    id="taxi-transmitter-router-su" >
8          <route streamCache="true" >
9              <from uri="jbi:endpoint:http://jbimulti2.iaas.uni-stuttgart.de/taxiscenario/
                   TaxiTransmitterRouter/ep" />
10             <setHeader headerName="tenantId" >
11                 <xpath resultType="java.lang.String">//tr:tenantId[1]/text()</xpath>
12             </setHeader>
13             <to uri="xslt:tenantContextFilter.xsl" />
14             <recipientList stopOnException="true" >
15                 <simple>jbi:endpoint:jbimulti2:tenant-endpoints/${header.tenantId}/
                       TaxiTransmitter/TaxiTransmitterSOAP</simple>
16             </recipientList>
17         </route>
18     </camelContext>
19 </beans>
```

Listing 6.5: Service unit configuration `context.xml` for the standard Apache ServiceMix Camel SE that routes a message to the appropriate Taxi Transmitter.

7. Outcome and Future Work

Contributing one building-block of a PaaS platform, this diploma thesis has originated concepts and implementation strategies for a multi-tenant ESB based on Apache ServiceMix [ASM]. In Chapter 2 we have presented relevant fundamentals, like cloud computing, SOA, the ESB, multi-tenancy, and JBI. After an investigation of previous work on multi-tenant PaaS platforms, we have introduced concepts for a multi-tenant ESB in Chapter 4. Attention was focused on the possibility to integrate the multi-tenant ESB with other PaaS applications by sharing a platform-wide registry of tenants and services. Regarding multi-tenancy, we have focused on data isolation between tenants [CC06] for message flows inside the ESB as well as for the management of ESB configuration artifacts. For this purpose, the JBI specification [JBI05] was analyzed and it was identified that integration services must be deployed on multi-tenant JBI components. This diploma thesis has conceived a multi-tenant enterprise application based on Java EE technology that allows tenant users to deploy JBI service assemblies to JBI containers like Apache ServiceMix. We have leveraged *role-based access control* [SCFY96] to distinguish between tenant administrators and tenant operators, the former giving limited access permissions to the latter. A use-case analysis has brought out a JBI specific management concept for the ESB. Furthermore, we have conceived two clustering scenarios for JBI containers to ensure scalability. Together, these concepts target the fourth level of the SaaS maturity model [CC06].

These concepts have led to a system design in Chapter 5 that describes a multi-tenant Java EE enterprise application for managing a cluster of JBI containers. Additionally, extensions to the JBI container Apache ServiceMix have been designed for communication with the enterprise application and to ensure data isolation for deployed JBI artifacts. We have leveraged distributed transactions to ensure consistency between separate databases and the messaging middleware used for management messages. Care has been taken that records stored in databases are related to individual tenants as stated by Chong et al. [CCW06]. Moreover, we have oriented towards the *multi-tenancy enablement layer* by Guo et al. [GSH+07] when designing the business logic of the management application. A prototype that complies to the design has been implemented and evaluated with the Taxi Scenario described in Chapter 1. The prototype provides a Web service interface that allows system administrators and tenant users to manage the mulit-tenant ESB.

Currently, the Taxi Scenario only uses one instance of Apache ServiceMix. However, the system was designed to handle clusters of ServiceMix instances. All data structures of the system allow for more than one ServiceMix instance. Furthermore, intelligent load balancing for installing JBI components and deploying service assemblies to individual ServiceMix instances were anticipated. As a consequence, the single component *JBI Container Manager* (see Sect. 4.1.1) in the management application was designated to communicate with the

underlying ESB implementation. In the future, the management application should get informed if management tasks have been successfully executed by Apache ServiceMix. There already is a *dead letter queue* for unprocessable management messages (see Sect. 5.4.1). The management enterprise application could reuse these messages to inform tenant users of the status of tasks. In the scope of this diploma thesis, a multi-tenant HTTP binding component and a multi-tenant Apache Camel service engine have been developed for Apache ServiceMix (see Sect. 5.4.2). The lessons learned in this process can assist developers that implement other multi-tenant JBI components. We have already conceived the possibility of a multi-tenant Apache ODE [AOD] service engine.

It is also not clarified, how external endpoint references in Apache ServiceMix should be handled, as they could lead to one tenant invoking external services of an other tenant via internal JBI service endpoints. Moreover, only service endpoint names are multi-tenant aware in the scope of this work. Accordingly, concepts for service interface names and operation names have to be considered (see Sect. 5.4.2). Otherwise, messages could flow through service endpoints of other tenants, too.

There is a Web service interface for the management application, but it is missing an implementation of the designed WS-Security mechanisms (see Sect. 5.2.2) as well as convenient operations for other applications of a PaaS platform. Future versions of the Web service interface should include operations for the retrieval of tenant context data and for the query of registered services. The latter can be used in policy-based routing scenarios [MvLW+09], where services are dynamically chosen dependent on the respective tenant and other parameters. The Service Registry conceptualized in this diploma thesis and the business logic of the management application (see Sect. 4.2) already support multi-tenant aware registering of services. There also should be an implementation for the Web-based graphical user interface described in Section 4.5.1. Finally, performance isolation, an important characteristic of a multi-tenant system, is out of scope in this work. But a multi-tenant ESB must ensure that services executed on behave of one tenant do not influence services of other tenants executed on the same instance.

Appendix A.

Interface Definitions

This chapter lists XSDs that illustrate the WS-ScurityPolicy [OAS09] based authentication, integrity, and confidentiality mechanisms of the JBIMulti2 Web service interface (see Sect. 5.2.2). Furthermore, a XSD for management messages targeting the `JMSManagementService` (see Sect. 5.4.1) is presented. For a detailed description of used XML namespaces look up the specifications referenced in Table 1.1.

A.1. Web Service Interface

Except for the definition of the tenant context, definitions in this section are not supported by the current implementation of JBIMulti2. They are solely intended to clarify the WS-Security [OAS06a] and WS-ScurityPolicy [OAS09] concepts introduced in Section 5.2.2. This includes a XSD of JBIMulti2-specific policies, a WSDL document extended with policy assertions, and an example SOAP message complying to the previous definitions.

```
1  <xsd:schema xmlns:xsd="..."
2    xmlns:tns="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context"
3    targetNamespace="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context">
4
5    <xsd:simpleType name="uuidType">
6      <xsd:restriction base="xsd:string">
7        <xsd:pattern value="[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}" />
8      </xsd:restriction>
9    </xsd:simpleType>
10
11   <xsd:group name="tenantUserId">
12     <xsd:sequence>
13       <xsd:element name="TenantId" type="tns:uuidType" />
14       <xsd:element name="UserId" type="tns:uuidType" />
15     </xsd:sequence>
16   </xsd:group>
17
18   <xsd:element name="TenantContext">
19     <xsd:complexType>
20       <xsd:sequence>
21         <xsd:choice>
22           <xsd:element name="TenantContextKey" type="tns:uuidType" />
```

```
23          <xsd:group ref="tns:tenantUserId" />
24        </xsd:choice>
25        <xsd:element name="OptionalEntry" minOccurs="0" maxOccurs="unbounded">
26          <xsd:complexType>
27            <xsd:sequence>
28              <xsd:element name="Key" type="xsd:string" />
29              <xsd:element name="Value" type="xsd:anyType" />
30            </xsd:sequence>
31          </xsd:complexType>
32        </xsd:element>
33      </xsd:sequence>
34    </xsd:complexType>
35  </xsd:element>
36
37  <xsd:element name="SystemAdminContext">
38    <xsd:complexType>
39      <xsd:sequence>
40        <xsd:element name="UserId" type="xsd:string" />
41        <xsd:element name="Password" type="xsd:string" />
42        <xsd:element name="OpTenantID" type="tns:uuidType" minOccurs="0" />
43      </xsd:sequence>
44    </xsd:complexType>
45  </xsd:element>
46
47 </xsd:schema>
```

Listing A.1: Tenant context XSD of JBIMulti2. Used in SOAP header and as element in JMS management messages. This XSD was developed together with Essl [Ess11].

```
1  <xsd:schema xmlns:xsd="..."
2    xmlns:tns="http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy"
3    targetNamespace="http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy">
4
5    <xsd:element name="TenantContext">
6      <xsd:complexType>
7        <xsd:sequence>
8          <xsd:element name="RequireValue" minOccurs="0">
9            <xsd:complexType>
10             <xs:attribute name="key" type="xsd:string" />
11           </xsd:complexType>
12         </xsd:element>
13       </xsd:sequence>
14   </xsd:element>
15
16   <xsd:element name="SystemAdminContext">
17     <xsd:complexType>
18       <xsd:sequence>
19         <xsd:element name="RequireOpTenantId" minOccurs="0">
20           <xsd:complexType />
21         </xsd:element>
22       </xsd:sequence>
```

```
23      </xsd:complexType>
24    </xsd:element>
25
26  </xsd:schema>
```

Listing A.2: Policies XSD of JBIMulti2.

```
1   <wsdl:definitions targetNamespace="http://jbimulti2.iaas.uni-stuttgart.de/wsdl"
2     xmlns:tns="http://jbimulti2.iaas.uni-stuttgart.de/wsdl" xmlns:wsdl="..."
3     xmlns:soap12="..." xmlns:wsp="..." xmlns:wsu="...">
4
5     <wsp:Policy wsu:Id="TenantAuthPolicy">
6       <wsp:ExactlyOne>
7         <wsp:All>
8           <wpjmu2:TenantContext
9             xmlns:wpjmu2="http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy">
10            <wpjmu2:RequireValue key="password" />
11          </wpjmu2:TenantContext>
12          <wsp:PolicyReference URI="#X509SignEncrypt" />
13        </wsp:All>
14        <wsp:All>
15          <wpjmu2:SystemAdminContext
16            xmlns:wpjmu2="http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy">
17            <wpjmu2:RequireOpTenantId />
18          </wpjmu2:SystemAdminContext>
19          <wsp:PolicyReference URI="#X509SignEncrypt" />
20        </wsp:All>
21      </wsp:ExactlyOne>
22    </wsp:Policy>
23
24    <wsp:Policy wsu:Id="SystemAdminAuthPolicy">
25      <wpjmu2:SystemAdminContext
26        xmlns:wpjmu2="http://jbimulti2.iaas.uni-stuttgart.de/wsdl/policy" />
27      <wsp:PolicyReference URI="#X509SignEncrypt" />
28    </wsp:Policy>
29
30    <wsp:Policy wsu:Id="X509SignEncrypt" xmlns:sp="...">
31      <wsp:ExactlyOne>
32        <wsp:All>
33          <sp:AsymmetricBinding>
34            <wsp:Policy>
35              <sp:InitiatorToken>
36                <sp:X509Token sp:IncludeToken="http://.../IncludeToken/Never">
37                  <wsp:Policy> <sp:RequireThumbprintReference /> </wsp:Policy>
38                </sp:X509Token>
39              </sp:InitiatorToken>
40              <sp:RecipientToken>
41                <wsp:Policy>
42                  <sp:X509Token sp:IncludeToken="http://.../IncludeToken/Never">
43                    <wsp:Policy> <sp:RequireThumbprintReference /> </wsp:Policy>
44                  </sp:X509Token>
```

```
45              </wsp:Policy>
46            </sp:RecipientToken>
47            <sp:AlgorithmSuite>
48              <wsp:Policy> <sp:TripleDesRsa15 /> </wsp:Policy>
49            </sp:AlgorithmSuite>
50            <sp:Layout>
51              <wsp:Policy> <sp:Strict /> </wsp:Policy>
52            </sp:Layout>
53          </wsp:Policy>
54        </sp:AsymmetricBinding>
55        <sp:SignedParts>
56          <sp:Body />
57          <sp:Header Namespace="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context" />
58        </sp:SignedParts>
59        <sp:EncryptedParts>
60          <sp:Body />
61          <sp:Header Namespace="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context" />
62        </sp:EncryptedParts>
63      </wsp:All>
64    </wsp:ExactlyOne>
65  </wsp:Policy>
66  ...
67  <wsdl:binding name="SystemAdminServiceSoap12Binding" type="tns:SystemAdminServicePortType"
        >
68    <wsp:PolicyReference URI="#SystemAdminAuthPolicy" />
69    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
70    ...
71  </wsdl:binding>
72
73  <wsdl:binding name="TenantAdminServiceSoap12Binding" type="tns:TenantAdminServicePortType"
        >
74    <wsp:PolicyReference URI="#TenantAuthPolicy" />
75    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
76    ...
77  </wsdl:binding>
78
79  <wsdl:binding name="TenantOperatorServiceSoap12Binding" type="
        tns:TenantOperatorServicePortType">
80    <wsp:PolicyReference URI="#TenantAuthPolicy" />
81    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
82    ...
83  </wsdl:binding>
84  ...
85 </wsdl:definitions>
```

Listing A.3: Policies in WSDL document of JBIMulti2. This is an extension of the original WSDL document of the JBIMulti2 implementation.

```
<soap:Envelope xmlns:soap="" xmlns:xenc="">
  <soap:Header>
    <ctxjmu2:TenantContext wsu:Id="Id-8e01e303"
      xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context">
      <xenc:EncryptedData Id="EncDataId-8e01e303" Type="http://...#Content"
        soap:mustUnderstand="1">
        ...
      </xenc:EncryptedData>
    </ctxjmu2:TenantContext>
    <wsse:Security xmlns:wsse="" soap:mustUnderstand="1">
      <xenc:EncryptedKey Id="EncKeyId-urn:uuid:74738ff5536759589aee98fffdcd1876">
        <xenc:EncryptionMethod Algorithm="http://...#rsa-1_5" />
        <ds:KeyInfo xmlns:ds="">
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier EncodingType="http://...#Base64Binary"
              ValueType="http://...#ThumbprintSHA1">1PUOqBg9B88D+d78PpDk2Zs9kpE=
            </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        <xenc:CipherData>
          <xenc:CipherValue>wOmcJy...2+KP48=</xenc:CipherValue>
        </xenc:CipherData>
        <xenc:ReferenceList>
          <xenc:DataReference URI="#EncDataId-8e01e303" />
          <xenc:DataReference URI="#EncDataId-6704f01a" />
        </xenc:ReferenceList>
      </xenc:EncryptedKey>
      <ds:Signature xmlns:ds="" Id="Signature-af014604">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="http://...xml-exc-c14n#" />
          <ds:SignatureMethod Algorithm="http://...#rsa-sha1" />
          <ds:Reference URI="#Id-8e01e303">
            <ds:Transforms>
              <ds:Transform Algorithm="http://...xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://...#sha1" />
            <ds:DigestValue>HFnzW9vpQov4goxllf51qhsUVdY=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Id-6704f01a">
            <ds:Transforms>
              <ds:Transform Algorithm="http://...xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://...#sha1" />
            <ds:DigestValue>EpdZEMPmNStbK97oH6L8RlOlvVk=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>Q7aBOB...wMkEeY=</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier EncodingType="http://...#Base64Binary"
              ValueType="http://...#ThumbprintSHA1">/YzLL5spPxILKhTmO5OjKDKnpO4=
            </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
```

```
55        </ds:Signature>
56      </wsse:Security>
57    </soap:Header>
58    <soap:Body xmlns:wsu="" wsu:Id="Id-6704f01a">
59      <xenc:EncryptedData Id="EncDataId-6704f01a" Type="http://...#Content">
60        <xenc:EncryptionMethod Algorithm="http://...#tripledes-cbc" />
61        <ds:KeyInfo xmlns:ds="">
62          <wsse:SecurityTokenReference xmlns:wsse="">
63            <wsse:Reference
64              URI="#EncKeyId-urn:uuid:74738ff5536759589aee98fffdcd1876" />
65          </wsse:SecurityTokenReference>
66        </ds:KeyInfo>
67        <xenc:CipherData>
68          <xenc:CipherValue>+STotL...kR9x1w=</xenc:CipherValue>
69        </xenc:CipherData>
70      </xenc:EncryptedData>
71    </soap:Body>
72  </soap:Envelope>
```

Listing A.4: Example SOAP message to JBIMulti2 including WS-Security [OAS06a] definitions. SOAP messages with security tokens are not supported by the current implementation of the JBIMulti2 Web service interface.

A.2. JBI Management Interface

Management messages sent by the JBIMulti2 Web application to the `JMSManagementService` conform to the following XSD. A description of this definition is given in Section 5.4.1.

```
<xsd:schema xmlns:xsd="..."
  xmlns:tns="http://jbimulti2.iaas.uni-stuttgart.de/jbi/jms"
  xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context"
  targetNamespace="http://jbimulti2.iaas.uni-stuttgart.de/jbi/jms">

  <xsd:import namespace="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context" />

  <xsd:element name="JBIManagement">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="TargetJBIContainers" type="tns:targetJBIContainers" />
        <xsd:element name="InstallJBIComponents" type="tns:installJBIComponents" minOccurs="
            0" />
        <xsd:element name="DeployServiceAssemblies" type="tns:deployServiceAssemblies"
            minOccurs="0" />
        <xsd:element name="UndeployServiceAssemblies" type="tns:undeployServiceAssemblies"
            minOccurs="0" />
        <xsd:element name="UninstallJBIComponents" type="tns:uninstallJBIComponents"
            minOccurs="0" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="targetJBIContainers">
    <xsd:sequence>
      <xsd:element name="JBIContainer" type="xsd:string" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="installJBIComponents">
    <xsd:sequence>
      <xsd:element name="JBIComponent" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Name" type="xsd:string" />
            <xsd:element name="ContentZIP" type="xsd:base64Binary" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="deployServiceAssemblies">
    <xsd:sequence>
      <xsd:element name="ServiceAssembly" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
```

```
44              <xsd:element name="Name" type="xsd:string" />
45              <xsd:element ref="ctxjmu2:TenantContext" />
46              <xsd:element name="ContentZIP" type="xsd:base64Binary" />
47            </xsd:sequence>
48          </xsd:complexType>
49        </xsd:element>
50      </xsd:sequence>
51    </xsd:complexType>
52
53    <xsd:complexType name="undeployServiceAssemblies">
54      <xsd:sequence>
55        <xsd:element name="ServiceAssembly" maxOccurs="unbounded">
56          <xsd:complexType>
57            <xsd:sequence>
58              <xsd:element name="Name" type="xsd:string" />
59              <xsd:element ref="ctxjmu2:TenantContext" />
60            </xsd:sequence>
61          </xsd:complexType>
62        </xsd:element>
63      </xsd:sequence>
64    </xsd:complexType>
65
66    <xsd:complexType name="uninstallJBIComponents">
67      <xsd:sequence>
68        <xsd:element name="JBIComponent" maxOccurs="unbounded">
69          <xsd:complexType>
70            <xsd:sequence>
71              <xsd:element name="Name" type="xsd:string" />
72            </xsd:sequence>
73          </xsd:complexType>
74        </xsd:element>
75      </xsd:sequence>
76    </xsd:complexType>
77
78  </xsd:schema>
```

Listing A.5: Messages XSD of JBI management interface.

Bibliography

[4Ca] 4CaaSt – EU Project. http://www.4caast.eu/.

[AMA] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/en/ec2/.

[AMQ] The Apache Software Foundation. Apache ActiveMQ. http://activemq.apache.org/.

[AMV] The Apache Software Foundation. Apache Maven. http://maven.apache.org/.

[AOD] The Apache Software Foundation. Apache ODE (Orchestration Director Engine). http://ode.apache.org/.

[APA11a] The Apache Software Foundation. *Apache Camel User Guide 2.7.0*, 2011. http://camel.apache.org/manual/camel-manual-2.7.0.pdf.

[APA11b] The Apache Software Foundation. *Apache Karaf Users' Guide 2.2.5*, 2011. http://repo1.maven.org/maven2/org/apache/karaf/manual/2.2.5/manual-2.2.5.pdf.

[APG+10] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA Middleware for Cloud Computing. In *Proc. IEEE 3rd Int Cloud Computing (CLOUD '10) Conf.*, pages 458–465, 2010.

[ARS] The Apache Software Foundation. Apache Aries. http://aries.apache.org/.

[ASM] The Apache Software Foundation. Apache ServiceMix. http://servicemix.apache.org/.

[ATC] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[AWS] The Apache Software Foundation. Apache WSS4J. http://ws.apache.org/wss4j/.

[BIMT05] BEA Systems, IBM, Microsoft, and TIBCO Software. Web Services Reliable Messaging Protocol (WS-ReliableMessaging), 2005. IBM developerworks, http://www.ibm.com/developerworks/library/specification/ws-rm/.

[CC06] F. Chong and G. Carraro. Architecture Strategies for Catching the Long Tail, 2006. MSDN, http://msdn.microsoft.com/en-us/library/aa479069.aspx.

[CCA] EU ICT Project Context Casting (C-CAST). http://www.ict-ccast.eu/.

[CCW06] F. Chong, G. Carraro, and R. Wolter. Multi-Tenant Data Architecture, 2006. MSDN, http://msdn.microsoft.com/en-us/library/aa479086.aspx.

[Cha04] D. A. Chappel. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[Chi06] E. Chinthaka. Web services and Axis2 architecture, 2006. IBM developerworks, http://www.ibm.com/developerworks/webservices/library/ws-apacheaxis2/.

[CHLP09] A. M. Colyer, H. Hildebrand, C. Leau, and A. Piper. Spring Dynamic Modules Reference Guide 1.2.1, 2009. http://static.springsource.org/osgi/docs/1.2.1/reference/pdf/spring-dm-reference.pdf.

[EJB06] Enterprise JavaBeans (EJB) 3.0, Final Release, 2006. JSR-220, http://jcp.org/aboutJava/communityprocess/final/jsr220/.

[Ess11] S. Essl. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2011.

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, 1996. RFC 2045, http://www.ietf.org/rfc/rfc2045.txt.

[FLM10] C. Fehling, F. Leymann, and R. Mietzner. A Framework for Optimized Distribution of Tenants in Cloud Applications. In *Proc. IEEE 3rd Int Cloud Computing (CLOUD '10) Conf.*, pages 252–259, 2010.

[FUS11] FuseSource. *Fuse ESB 4.4 – Using Java Business Integration*, 2011. http://fusesource.com/docs/esb/4.4/jbi/.

[Gaw09] J. Gawor. Building OSGi applications with the Blueprint Container specification, 2009. IBM developerworks, http://www.ibm.com/developerworks/opensource/library/os-osgiblueprint/.

[GMA] Google Maps API Web Services. http://code.google.com/intl/en/apis/maps/documentation/webservices/.

[GSH+07] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A Framework for Native Multi-Tenancy Application Development and Management. In *Proc. 9th IEEE Int. Conf. E-Commerce Technology and the 4th IEEE Int. Conf. Enterprise Computing, E-Commerce, and E-Services CEC/EEE '07*, pages 551–558, 2007.

[Hag11] R. Hagin. Enabling Integration and Aggregation of Context Information into WS-BPEL Processes. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2011.

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[JAV06] Java Platform, Enterprise Edition 5 (Java EE), Final Release, 2006. JSR-244, http://jcp.org/aboutJava/communityprocess/final/jsr244/.

[JAX06a] The Java Architecture for XML Binding (JAXB) 2.0, Final Release, 2006. JSR-222, http://jcp.org/aboutJava/communityprocess/final/jsr222/.

[JAX06b] The Java API for XML-Based Web Services (JAX-WS) 2.0, Final Release, 2006. JSR-224, http://jcp.org/aboutJava/communityprocess/final/jsr224/.

[JBI05] Java Business Integration (JBI) 1.0, Final Release, 2005. JSR-208, http://jcp.org/aboutJava/communityprocess/final/jsr208/.

[JBI12] Dominik Muhler. JBI Multi-tenancy Multi-container Support (JBIMulti2) – Requirements Specification, 2012. On CD-ROM belonging to this diploma thesis.

[JMS02] Java Message Service (JMS) 1.1, Final Release, 2002. JSR-914, http://jcp.org/aboutJava/communityprocess/final/jsr914/.

[JSF06] JavaServer Faces Specification (JSF) 1.2, Final Release, 2006. JSR-252, http://jcp.org/aboutJava/communityprocess/final/jsr252/.

[Mar02] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., 2002.

[MPW11] I. K. Milinda Pathirage, Srinath Perera and S. Weerawarana. A Multi-tenant Architecture for Business Process Executions. In *Proc. IEEE 9th Int Conf. on Web Services (ICWS '11)*, pages 121–128, 2011.

[MUTL09] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In *Proc. IEEE Int. Enterprise Distributed Object Computing Conf. EDOC '09*, pages 131–140, 2009.

[MvLW+09] R. Mietzner, T. van Lessen, A. Wiese, M. Wieland, D. Karastoyanova, and F. Leymann. Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus. In *Proc. IEEE Int. Conf. Web Services ICWS '09*, pages 617–624, 2009.

[NIS11] National Institute of Standards and Technology. The NIST Definition of Cloud Computing, 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[OAS06a] OASIS. Web Services Security (WS-Security) 1.1, 2006. http://www.oasis-open.org/standards/.

[OAS06b] OASIS. Web Services Security X.509 Certificate Token Profile 1.1, 2006. http://www.oasis-open.org/standards/.

[OAS07] OASIS. Web Services Business Process Execution Language (WS-BPEL) 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html.

[OAS09] OASIS. WS-SecurityPolicy 1.3, 2009. http://www.oasis-open.org/standards/.

[OJ07] S. Ortiz Jr. Getting on Board the Enterprise Service Bus. *Computer*, 40:15–17, April 2007.

[OPG06] The Open Group. The SOA Work Group: Definition of SOA, 2006. http://www.opengroup.org/soa/soa/def.htm.

[OPG11] The Open Group. IBM Cloud Computing Reference Architecture 2.0, 2011. https://www.opengroup.org/cloudcomputing/uploads/40/23840/CCRA.IBMSubmission.02282011.doc.

[OSG09] OSGi Alliance. OSGi Service Platform: Service Compendium Version 4.2, 2009. http://www.osgi.org/Download/Release4V42/.

[OSG11] OSGi Alliance. OSGi Service Platform: Core Specification Version 4.3, 2011. http://www.osgi.org/Download/Release4V43/.

[OWJ] OW2 Consortium. JOnAS: Java Open Application Server. http://wiki.jonas.ow2.org/.

[OWO] OW2 Consortium. Orchestra: Open Source BPEL / BPM Solution. http://orchestra.ow2.org/.

[PHE+06] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. Axis2, Middleware for Next Generation Web Services. In *Proc. IEEE Conf. Web Services (ICWS '06)*, pages 833–840, 2006.

[PSQ] PostgreSQL. http://www.postgresql.org/.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based Access Control Models. *Computer*, 29:38–47, February 1996.

[SOA] SmartBear Software. soapUI. http://www.soapui.org.

[SOA07] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. W3C Recommendation, http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[SOA10] SOAP over Java Message Service 1.0, 2010. W3C Working Draft, http://www.w3.org/TR/2010/WD-soapjms-20101026/.

[Tao01] L. Tao. Shifting Paradigms with the Application Service Provider Model. *Computer*, 34:32–39, October 2001.

[WB09] C. D. Weissman and S. Bobrowski. The Design of the Force.com Multitenant Internet Application Development Platform. In *Proc. SIGMOD International Conf. on Management of Data SIGMOD'09*, pages 889–896. ACM, 2009.

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

[WSA04] Web Services Addressing (WS-Addressing), 2004. W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/.

[WSD01] Web Services Description Language (WSDL) 1.1, 2001. W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[WSD06] WSDL 1.1 Binding Extension for SOAP 1.2, 2006. W3C Member Submission, http://www.w3.org/Submission/2006/SUBM-wsdl11soap12-20060405/.

[WSP07] Web Services Policy 1.5 - Framework, 2007. W3C Recommendation, http://www.w3.org/TR/2007/REC-ws-policy-20070904/.

[XML02a] XML Encryption Syntax and Processing, 2002. W3C Recommendation, http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[XML02b] XML-Signature Syntax and Processing, 2002. W3C Recommendation, http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[XSD04] XML Schema Part 1: Structures Second Edition, 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[ZSTC10] X. Zhang, B. Shen, X. Tang, and W. Chen. From Isolated Tenancy Hosted Application to Multi-tenancy: Toward a Systematic Migration Method for Web Application. In *Proc. IEEE Int Software Engineering and Service Sciences (ICSESS '10) Conf.*, pages 209–212, 2010.

All links were last followed on January 22, 2012.

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, January 27, 2012 ____________________

(Dominik Muhler)