Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis Nr. 3546

Approach and Realization of a Multi-tenant Service Composition Engine

Michael Hahn

Course of Study:	Software Engineering
Examiner:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Supervisor:	Dipl.-Inf., Dipl.-Wirt. Ing.(FH) Karolina Vukojevic-Haupt
Commenced:	June 17, 2013
Completed:	November 11, 2013
CR-Classification:	C.2.4, D.2.11, H.4.1, H.3.4

Abstract

The support of multi-tenancy is an essential requirement to leverage the full extent of Cloud computing. Multi-tenancy enables service providers to maximize the utilization of their infrastructure and to reduce the servicing costs per customer. With regard to the fact that nowadays new applications or services are often composed out of multiple existing services or applications, a middleware is required which enables these compositions. A *Service Composition Engine* (SCE) provides the required functionality to enable the definition and execution of service compositions.

In this diploma thesis we investigate the requirements and define an abstract architecture for the realization of a multi-tenant SCE. This architecture is prototypically realized with an open-source SCE and integrated into an existing multi-tenant aware ESB. The resulting middleware provides configurability for service compositions, tenant-aware messaging and tenant-based administration and management of the SCE and the ESB.

Contents

List of Abbreviations

API	Application Programming Interface
Axis2	Apache eXtensible Interaction System v. 2
BC	Binding Component
BLOB	Binary Large Object
BPEL	Business Process Execution Language
BPM	Business Process Management
DAO	Data Access Objects
DRL	Dynamic Recipient List
DSL	Domain Specific Language
EAI	Enterprise Application Integration
EIP	Enterprise Integration Pattern
ESB	Enterprise Service Bus
IaaS	Infrastructure as a Service
JaCOb	Java Concurrent Objects Framework
JAR	Java Archive
JBI	Java Business Integration
JMX	Java Management eXtension
JVM	Java Virtual Machine
MOM	Message Oriented Middleware
NIST	National Institute of Standards and Technology
NM	Normalized Message
NMF	Normalized Message Format
NMR	Normalized Message Router
ODE	Orchestration Director Engine
PaaS	Platform as a Service
PGF	Pluggable Framework
POJO	Plain Old Java Object
POM	Project Object Model
PSA	Process Service Assembly
SA	Service Assembly
SaaS	Software as a Service

SCE ..... Service Composition Engine
SCE-MT Manager Service Composition Engine Multi-tenancy Manager
SE ..... Service Engine
SLA ..... Service Level Agreement
SOA ..... Service Oriented Architecture
SU ..... Service Unit
SWfMS ..... Simulation Workflow Management System
UI ..... User Interface
UML ..... Unified Modeling Language
UUID ..... Universally Unique Identifier
WE ..... Workflow Engine
WfMC ..... Workflow Management Coalition
WfMS ..... Workflow Management System
WfRM ..... Workflow Reference Model
WS ..... Web Service
WSDL ..... Web Service Description Language
XML ..... eXtensible Markup Language

1 Introduction

Nowadays Cloud computing is one of the most important technologies in the IT landscape. It provides IT resources in an agile way over the Internet in a pay-per use model. The outsourcing of applications and services into the Cloud can provide many advantages like reduced operational costs and rapid elasticity for enterprises. To leverage the full extent Cloud computing provides, the outsourced applications and services have to be adapted to be multi-tenant aware. This means applications and services should be designed to maximize the resource sharing between multiple customers. Thus service providers are able to maximize the resource utilization and as a result reducing their servicing costs per customer [SALM12]. The main requirements to fulfill are isolation, configurability and scalability. Isolation between tenants is one of the most important requirements for outsourcing to the Cloud. All resources (e.g. data) which belong to a specific tenant should not be accessible by any other entity (e.g. tenant, service provider). If the same application or service is used to serve multiple tenants, it must be customizable in some degree to comply with the different requirements of its users. Configurability is the key concept to provide predefined configuration possibilities which allow the tenants to customize some parts of an application to their needs. Scalability is important to serve any number of tenants with a single instance of an application. In the scope of this diploma thesis a multi-tenant aware *Service Composition Engine* (SCE) is conceptualized and prototypically implemented. Furthermore the realized multi-tenant SCE prototype is integrated into the multi-tenant aware *Enterprise Service Bus* (ESB) realized by [Muh12], [Ess11] and [Sáe13]. The following sections provide the background of this thesis and a short motivation why a multi-tenant SCE integrated with a multi-tenant ESB should be realized.

1.1 Background

This diploma thesis is based on two different research projects: SimTech and 4CaaSt. The result of this thesis will be used in both projects to solve different requirements. For a better understanding, these two projects are described in the following.

1.1.1 SimTech

The Simulation Technology (SimTech)[1] excellence cluster is a research cluster of the German Research Foundation (DFG) and is embedded in the Stuttgart Research Center for Simulation

[1] SimTech - Cluster of Excellence: http://www.simtech.uni-stuttgart.de/index.en.html

Technology. It represents a massive platform for developing scientific methods for modeling and simulation techniques. The Institute of Architecture of Application Systems (IAAS)[2] of the University of Stuttgart contributes four projects to the SimTech cluster which work on the provisioning of a *Workflow Management System* (WfMS) for simulation workflows, referred to as *Scientific Workflow Management System* (SWfMS). The SWfMS and a modeling tool (Eclipse BPEL Designer[3]) are especially adapted to conform to the scientists way of working (trial and error) and to hide the technical complexity of the underlying technology. Each of the four projects: Modeling, Runtime, Flexibility and Humans in Simulation Workflows therefore concentrates on the improvement of the SWfMS and the definition of new concepts and techniques to model and execute simulation workflows in a flexible manner. The realized SWfMS is based on the open source workflow engine Apache ODE[4] and is described in detail in Chapter 2.5. The goal of the project is to provide scientists an easy to use environment which enables them to dynamically model, execute and monitor their simulations based on workflow technology. The use of workflow technology provides therefore some advantages, like

- a new abstract layer to combine the isolated simulations of different scientists to much more powerful multi-domain, multi-physics or multi-scale simulations [SCLGK10] (Modeling),
- the possibility to dynamically influence the execution of simulations during their runtime [Nin11, Sch11, Tol11] (Flexibility, Humans in Simulation Workflows),
- the possibility to directly interact with a running simulation, e.g. to enable a scientist to specify how a simulation workflow should proceed in an exceptional situation [KDS+12] (Humans in Simulation Workflows),
- and the possibility to integrate existing simulation applications over Web Service Interfaces [Rut09, Hot10] or for example define quality of data (e.g. accuracy) which steers the execution of a simulation workflow [RBKK12] (Runtime).

The SWfMS and the modeling tool together with some additional components form the SimTech prototype which is shown in Figure 1.1. The SWfMS runs in a Java Servlet Container (Apache Tomcat[5]). Furthermore this container hosts the SimTech Auditing Service. The SimTech Modeling and Monitoring tool is based on Eclipse and therefore consists of a set of Eclipse Plugins. For example, the extended SimTech BPEL Designer is one of those plugins which enable the modeling and monitoring of simulation workflows. The communication between the SimTech Modeling and Monitoring Tool and the services hosted in the Servlet Container is enabled by a Messaging System and the Web Service APIs of the services. For example, the event data which is propagated by SWfMS is published to multiple consumers by the Messaging System. The SimTech Modeling and Monitoring Tool consumes those event messages to realize

[2]Institute of Architecture of Application Systems (IAAS), University of Stuttgart. SimTech Project Contribution: http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/
[3]The Eclipse Foundation, BPEL Designer Project: http://www.eclipse.org/bpel/
[4]The Apache Software Foundation, Apache ODE (Orchestration Director Engine): http://ode.apache.org
[5]The Apache Software Foundation, Apache Tomcat: http://tomcat.apache.org/

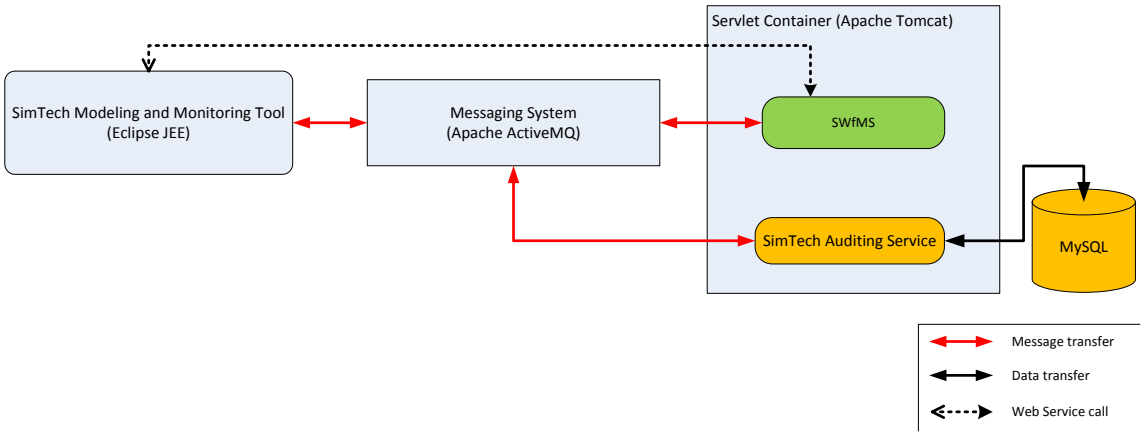

Figure 1.1: Architecture of the SimTech prototype

the monitoring of process executions and the SimTech Auditing Service persists the event messages in a database. Furthermore, the messaging system is used by the SimTech BPEL Designer to influence the execution of a process instance by sending corresponding command messages to SWfMS. For example, the registration of activity breakpoints or changing variable values are possible ways to influence a process instance during runtime. Chapter 2.5 provides some detailed descriptions of the SWfMS functionality.

1.1.2 4CaaSt

4CaaSt[6] is a research project funded by the European Union and aims to create a Cloud platform which "supports the optimized and elastic hosting of Internet-scale multi-tier applications". The main goal is to offer an advanced environment which provides all necessary features to design services and compositions based on Cloud-aware building blocks of different vendors [SALM12]. This will lower the entry barrier for small and medium enterprises to "create innovative applications leveraging the benefits of Cloud computing" [SALM12]. Figure 1.2 shows the architecture of the Taxi Scenario use case which is defined in the scope of 4CaaSt. In this scenario a service provider offers a taxi management application as a service to different taxi companies (tenants) [SALM12]. This management software can be used by the taxi companies to enable the communication between their customers (users of a tenant) and drivers. If a customer submits a transportation request over the *Customer GUI* to a taxi company, the management software contacts nearby taxi drivers. As soon as one of the contacted drivers confirms the request over the *Taxi Drivers' GUI*, the management software sends a transport notification containing the estimated arrival time to the customer [SALM12]. The taxi management software is realized as a BPEL process (*Taxi Service Provider Process*).

[6]The 4CaaSt project: http://www.4caast.eu/

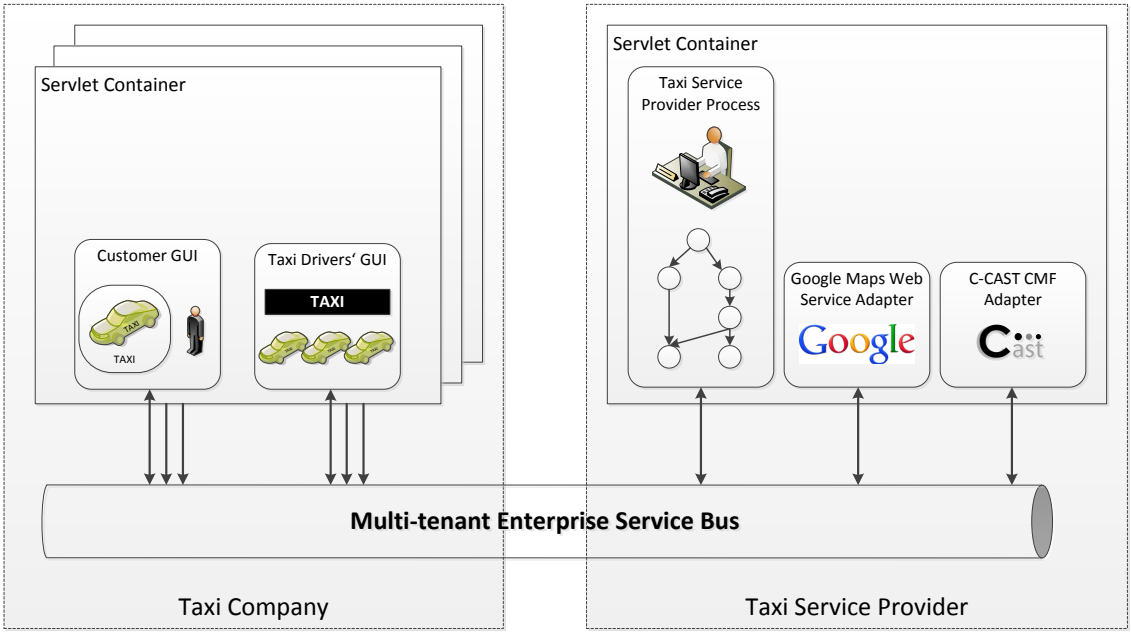

Figure 1.2: Architecture of the 4CaaSt Taxi Scenario [SALM12]

This process uses the Context Casting Context-Management Framework (C-CAST CMF)[7] and the Google Maps Web Services[8]. The C-CAST CMF provides information about the locations of the taxi cabs and how to contact the taxi drivers. The Google Maps Web Services are used by the process for distance calculations between the current location of a taxi cab and the pick up location of a customer. In case of incompatibilities between the service endpoints of the C-CAST CMF, Google Maps Web Services and the BPEL process, two adapters mediate between the process and these external services [SALM12]. The taxi service provider process is executed by the non multi-tenant aware BPEL engine *Orchestra*[9]. The communication between all components is realized over messaging.

The Institute of Architecture of Application Systems (IAAS) of the University of Stuttgart introduced a *multi-tenant ESB* (ESB^{MT}) as the messaging middleware to enable loose coupling and a flexible integration solution which avoids the use of hard-coded point-to-point connections [SALM12]. Furthermore, the multi-tenant ESB makes it possible that the same taxi management application can be offered as a service to multiple customers by a single provider. This maximizes the benefits for the taxi companies which outsource their software and also for the service providers which can serve their tenants more cost effective. In case

[7]The C-CAST project: http://www.ict-ccast.eu
[8]Google Inc., Google Maps API Web Services: http://code.google.com/intl/en/apis/maps/documentation/webservices/
[9]OW2 Consortium, Orchestra: http://orchestra.ow2.org

of that, the multi-tenant ESB is an essential building block of the 4CaaSt project [SALM12]. The multi-tenant aware SCE implementation which is provided by this diploma thesis should replace the currently used BPEL engine Orchestra.

1.2 Motivation

Adding multi-tenancy support to a SCE enables service providers to offer the same SCE as a service to multiple customers. As a result customers do not have to care about the underlying infrastructure, required middleware (e.g. an ESB, Servlet Container) or the management of the SCE. They can just use the SCE as an execution environment for their process models. Furthermore, service providers are able to maximize the utilization of their infrastructure, thus reduce the operational costs and as a result lower the entry barrier for customers which are not able to host an SCE on their own. Additionally the support of multi-tenancy on the process model level enables the definition of multi-tenant aware service compositions which again can be offered as a multi-tenant service. Another advantage multi-tenancy introduces is the configurability of the SCE and the process models which define service compositions. This enables customers to tailor the single SCE instance and a process model to their needs based on a set of predefined customization options. A multi-tenant SCE in combination with the multi-tenant ESB – realized in the 4CaaSt project – provides a powerful multi-tenant aware integration and messaging middleware. The main outcomes are configurable service compositions, tenant-aware messaging and tenant-based administration and management of all middleware components.

1.3 Outline

This diploma thesis contains the following chapters.

Chapter 1 – Introduction introduces the background and motivates the topic of this diploma thesis.

Chapter 2 – Fundamentals provides descriptions of the necessary technologies and concepts used for the realization of this thesis.

Chapter 3 – Related Works contains a general overview of related works that deal with multi-tenancy and the requirements and challenges to realize multi-tenant aware applications or services.

Chapter 4 – Requirements and Concepts provides the identified challenges and requirements to enable multi-tenancy for Service Composition Engines and Process Models. Furthermore, two conceptual solution approaches are described and compared. One of these two concepts is then used as the basis for the implementation of a multi-tenant SCE based on the open-source BPEL engine Apache ODE.

Chapter 5 – Implementation specifies how the general concept is realized and implemented. Furthermore, the most important management and runtime scenarios of the resulting system and the integration with JBIMulti2 and ESB^{MT} are described.

Chapter 6 – Conclusion and Future Work: The last chapter summarizes the outcomes of this diploma thesis and suggests some future extensions for the defined concepts and the resulting system.

2 Fundamentals

This chapter provides short introductions to the main technologies and concepts which are used in this diploma thesis. The descriptions should provide some basic knowledge to the reader for a better understanding of the following chapters.

2.1 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is defined by The Open Group as "an architectural style that supports service-orientation – a way of thinking in terms of services, service-based development and the outcomes of services" [SOA13]. A service is therefore a logical representation of a business function that is self-contained, possibly composed of other services while hiding any implementation details [SOA13]. As a result SOA provides the required flexibility for building distributed systems by relying on loose coupling, interoperability, efficiency and standardization [WCL+05].

The main roles in which the different participants of a SOA are distinguished are service requester, service provider and service broker. A service provider creates a description of its service and publishes this to the service broker. A service requestor is therefore able to discover a required service at the service broker by searching through all the registered service descriptions. As soon as a service is found which fits the needs of the service requestor, the concrete endpoint of the service is replied by the discovering facility. With this information, the requestor can then bind to the concrete service and finally invoke a business activity [WCL+05].

The use of an *Enterprise Service Bus* (ESB) simplifies the described procedure for the service requestor by combining the three steps (find, bind and invoke) into a single one. The ESB takes care of finding an appropriate service, binding to it and invoking the service by forwarding the initial request of the service requestor to it [WCL+05]. The following section provides some more details of the functionality of an ESB.

2.2 Enterprise Service Bus

Business drivers like changing economic conditions or new regulatory compliance have led to the need for a new broadly applicable and standardized integration solution [Cha04]. The ESB concept consists of a variety of previously existing standards, concepts and technologies like *Enterprise Application Integration* (EAI), SOA, Web Services or *Message Oriented Middleware*

(MOM) and combines all their advantages in a new type of integration middleware. As a result, an ESB provides a loosely coupled, event-driven SOA with a highly distributed universe of named routing destinations across a multi-protocol message bus [Cha04]. All connected applications are abstractly decoupled from each other by using logical endpoints that are exposed as services. This enables that services, routings or data transformations can be configured rather than written into code. A programmer has only to implement the binding to a logical endpoint exposed as a service. This decouples the implementation of a service from its integration with other components on the bus. To enable loose coupling, one of the core principles of an ESB is reliable messaging. Therefore applications do not need to care about resending messages on failure or if the target application is currently unavailable.

Chappel denotes the combination of loosely coupled interfaces and asynchronous interactions as a key concept of the bus terminology [Cha04]. Services or applications plugged into the bus have access to everything else on the bus without to be concerned about how the communication is realized. As a result, the ESB makes the three SOA steps find, bind and invoke transparent to the user. The only thing a user has to do, is to plug into the bus, post data to it and receive the response data from the bus [Cha04]. The service requestor only has to send a service description and a collection of data to the bus. With the service description the bus selects the service which best fits to the requirements of the requestor. After that the bus binds the requestor to the target service by creating a route between their logical endpoints. At the end the bus uses the created route to enable the communication between the two parties and executes any required data transformations for the exchange of data between the requestor and the target service.

2.3 Cloud Computing

Cloud computing is a new paradigm in the IT world to provide IT resources in an agile way over the Internet in a pay-per use model. The *National Institute of Standards and Technology* (NIST) defines Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [MG11]. The five essential characteristics of which the Cloud computing model is composed of are described in [MG11]:

On-demand self-service A Cloud consumer can provision computing resource automatically without the need of human interaction.

Broad network access All computing capabilities are available over the network and can be accessed through standard mechanisms.

Resource Pooling The computing capabilities offered by a Cloud provider are virtualized and pooled to serve multiple consumers using a multi-tenant model. Therefore the Cloud consumer generally has no sense of the location of the provided resources.

Rapid Elasticity Based on the Cloud consumers' demand the computing capabilities can be elastically – and in some cases automatically – provisioned and released.

Measured Service The resource usage can be monitored and measured providing transparency for both the Cloud consumer and the provider for any control and optimization purposes.

The Cloud computing model offers three service models which provide different levels of control the cloud consumer has over the provided computing resources. *Software-as-a-Service* (SaaS) provides to the cloud consumer the use of cloud provider's applications running on a cloud infrastructure [MG11]. The consumer has no control over the underlying infrastructure to which the application is deployed. Only provided user-specific configuration settings can be used by the consumer to control individual application capabilities. *Platform-as-a-Service* (PaaS) provides the consumer the capabilities to deploy applications created using programming languages, required libraries, services and tools supported by the provider [MG11]. The consumer has no control over the underlying infrastructure but has control over the deployed applications. *Infrastructure-as-a-Service* (IaaS) is the service model which offers the most control to the consumer. As a result, the consumer is able to deploy and run arbitrary software and has also the control over operating systems, storage and deployed applications. The management and control of the underlaying cloud infrastructure remains at the cloud provider.

The NIST defines four deployment models how a cloud infrastructure can be provisioned. A *private cloud* is provisioned to be exclusively used by a single organization and its members. A *community cloud* is a cloud infrastructure that is used by a specific collection of organizations which share the same requirements. A *public cloud* infrastructure is accessible and usable by the general public. The *hybrid cloud* deployment model combines two or more distinct cloud infrastructures which are bound together but each of them remain as a unique entity.

2.4 Workflow Technology

A business process consists of a set of activities that are carried out in a specific sequence and passing data from one activity to another [LR00]. A process model describes the structure of a business process and defines all possible paths and the actions which need to be performed [LR00]. The process model is used as a template to create instances of it – so called process instances – which actually execute a business process. Each process instance executes one of the possible paths specified in the process model. This path is determined by a set of instance-specific values (e.g. input data) and a set of rules which are defined in the process model [LR00]. Not necessarily all parts of a process model must be executed by a computer. The parts of a process model which can be run on a computer are called *workflow models* [LR00]. Similar as for process models an instance of a workflow model is defined as a workflow instance. The *Workflow Management Coalition* (WfMC) defines workflows as "computerised facilitation or automation of a business process, in whole or part" [Hol95]. A *Workflow Management System* (WfMS) provides the software environment for workflows. It is defined as "a system that completely defines, manages and executes 'workflows' through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [Hol95]. The "computer representation of the workflow logic" is realized by the use of

standardized modeling languages like the *Web Services Business Process Execution Language* (BPEL, [BPE07]) described in the next Section.

The WfMC has defined a *Workflow Reference Model* (WfRM, [Hol95]) which specifies the main components and interfaces that each WfMS should provide. The central component of the model is the *Workflow Enactment Service* which provides the runtime environment for one or more *Workflow Engines* (WE). These realize the runtime and execution environment for workflow instances [Hol95]. Defined workflow models are deployed to a WE which interprets the models and maybe transforms them to an engine-internal representation. The main functionality of a WE is the control of the lifecycle of workflow instances, navigation between modeled activities, passing of workflow instance data and the invocation of external applications or services [Hol95]. The Apache Orchestration Director Engine (ODE)[1] used in the context of this diploma thesis is such a WE.

The terms process and workflow are often used interchangeably. We follow this convention in this document and use the terms process model and process instance when we are talking about executable business processes. In the following we will have a look at the executable process definition language BPEL.

Business Process Execution Language

BPEL is an XML-based language for the definition of executable business processes also known as workflows. A BPEL process model describes the orchestration of Web Services and can be itself exposed as a Web Service. Thus BPEL provides a recursive aggregation model for Web Services. A BPEL process model consists of the following artifacts.

Process The process artifact is the root of a BPEL process model and contains all other artifacts. It provides properties to specify the name, a target namespace or the expression language (e.g. XPath or XQuery) amongst other things for the defined process model.

Partner Links are used to "directly model peer-to-peer conversational partner relationships" [BPE07]. A partner link encapsulates a service with which a business process interacts or itself provides. Partner links are characterized by a partner link type [BPE07]. This construct defines some kind of conversation channel between two partners by specifying the port type of each of the partners' WSDL service interface and the role they are playing in the conversation. For example, if they provide a service or require a service from an external service provider. During the runtime of a process model, the partner link holds the relevant binding and communication data (e.g. service endpoint address) to enable the conversation between the process model and any partners. The endpoint address of a referenced service can be dynamically assigned during runtime or statically defined during the deployment of the process model.

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

Variables are containers which persist the data of a process instance. They can be used to exchange data with partners or to hold any other process related data (e.g. constant values, intermediate results). Variables can be defined on the process level or in `<scope>` activities. Their values can be set or manipulated over `<assign>` activities.

Correlation Sets enable the automatic routing of messages to the correct process instances based on business data instead of using artificial identifiers. Therefore a correlation set consists of a list of properties. A property is an abstract container for correlation data which is located in the messages during runtime. Property aliases are used to define a mapping between a property and the part of the message which should be used for correlation. The defined correlation sets can then be referenced in any conversation activity (e.g. `<invoke>`, `<receive>`, `<reply>`) to enable message correlation.

Handlers enable the modeling of specific control flow which should be executed to "handle" different types of occurrences during the runtime of an instance of the process model. BPEL provides handlers to react on faults (`<faultHandlers>`), different events (`<eventHandlers>`), the termination of an instance (`<terminationHandler>`) or to enable the compensation of a collection of activities (`<compensationHandler>`).

Activities The BPEL specification provides a variety of different activity types. They can be divided into two categories: basic activities and structured activities. The former provide atomic artifacts with specific functionality (e.g. invocation of external services, data assignment) and the latter realize the composition of basic activities by prescribing the order in which they should be executed [BPE07]. BPEL supports a combination of graph-based (e.g. `<flow>`) and block-based (e.g. `<sequence>`) modeling for business processes.

Furthermore BPEL provides language extensibility by the import of other XML namespaces. This allows adding constructs defined in an imported XML namespace to any BPEL construct. The BPEL specification provides also two explicit extension constructs: *Extension Activities* and *Extension Assign Operations*. The former can be used to specify new activity types which can then be used in a process model and the latter to define specialized data manipulation constructs which can be used in `<assign>` activities.

2.5 Extentend Apache ODE

As introduced in Chapter 1.1.1, a modified version of the open source BPEL engine Apache Orchestration Director Engine (ODE)[2] is used. Since BPEL enables the orchestration of services and therefore the definition of service compositions, the BPEL engine Apache ODE provides also an implementation of a *Service Composition Engine* (SCE). The original version of Apache ODE is extended for the SimTech project. Figure 2.1 shows the architecture of the resulting SWfMS. The four layers and their components are described in the following.

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

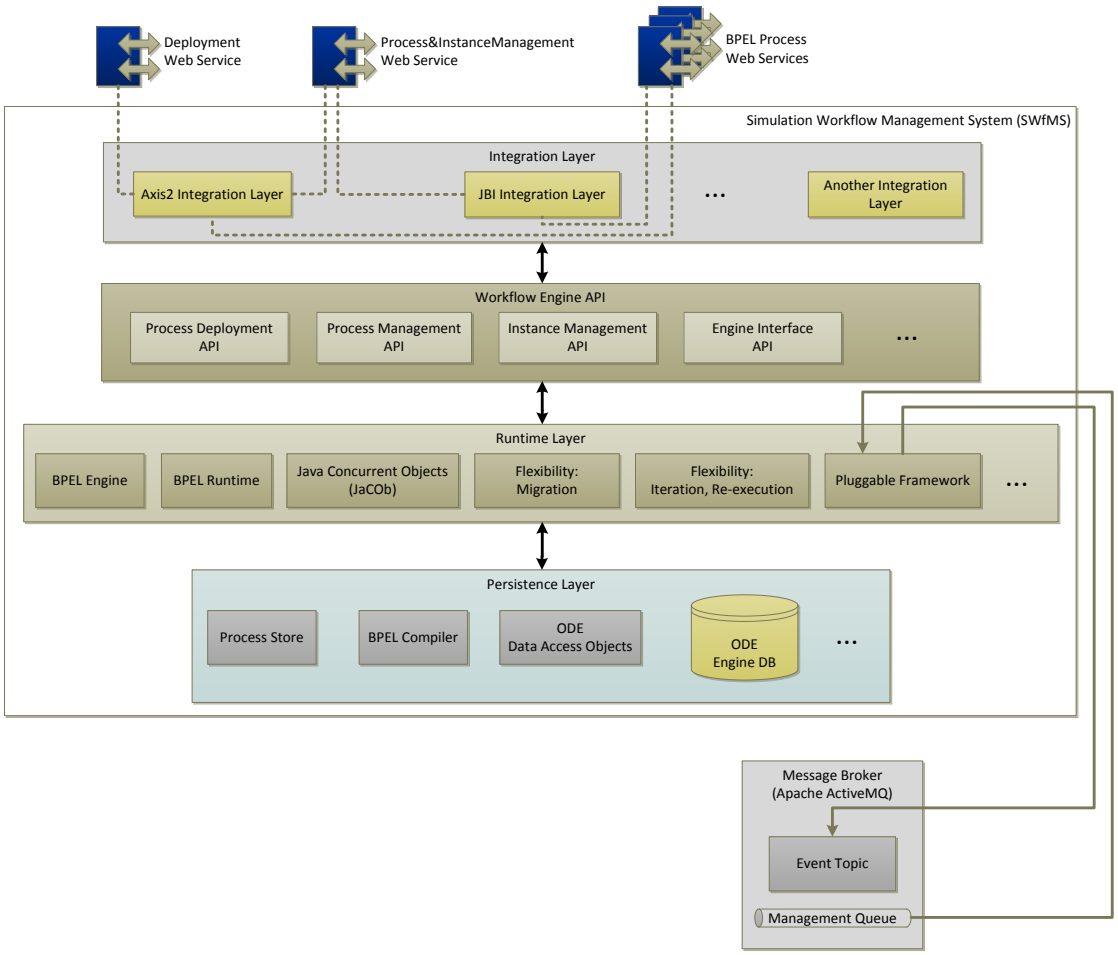

Figure 2.1: Architecture of the extended BPEL Engine *Apache ODE*

The *Integration Layer* provides the engine-internal functionality to the outside and enables communication. ODE provides per default an Axis2 and a JBI Integration Layer which handle the correct routing of message exchanges to the engine-internal objects and provide the APIs of the engine to the outside. The Axis2 Integration Layer provides a Web Service for Deployment, Process and Instance Management and for each deployed BPEL Process. The JBI Integration Layer integrates ODE into a JBI Environment (e.g. an ESB, for example Apache Service Mix) which provides the Web Service Interfaces for Process and Instance Management and the BPEL Processes to the outside. The deployment of new process models is handled by the JBI Environment and therefore no Deployment Web Service exists. By the realization of a new Integration Layer implementation, ODE can be integrated in any other environment as with the two existing layers.

The *Workflow Engine API* consists of a set of APIs which provide generic interfaces for the engine-internal classes. For example, the *Process Management API* provides an interface for the engine-internal management functionality provided by the BPEL Engine component. This interface enables all Integration Layers to forward incoming management requests to the classes of the Runtime Layer which provide an implementation for the interface. Additionally, the *Engine Interface API* is used by the Integration Layer to forward incoming requests send to the BPEL Process Web Services to the Runtime Layer.

The *Runtime Layer* provides the core functionality of the engine. Therefore the *BPEL Engine* component contains all engine-related classes, like a *BpelProcess* class which represents a process model and contains the corresponding functionality to initialize a new process instance. Another important class contained in the BPEL Engine is the *BpelRuntimeContext* which represents a running process instance and contains all related functionality, like the invocation of external services, reading or writing variable values or the correlation of message exchanges to the correct process instance. The *BPEL Runtime* component contains the implementation of all BPEL constructs which are used by the engine to execute a process model. To enable the reliable execution of process instances, the BPEL Runtime relies on *Data Access Objects* (DAO) which are used to store all important runtime data in the engines' database (see Persistence Layer). To enable the persistent and concurrent execution of the BPEL runtime constructs, the *Java Concurrent Objects* framework (JaCOb) is used. It provides an application-level concurrency mechanism and a mechanism for interrupting execution and persisting execution state. For example, this enables the resumption of all running instances after an outage of the engine. The *Pluggable Framework* (PGF) is based on the work of Steinmetz who extended ODE with a generic BPEL 2.0 Event Model and some additional event-based functionality. He realized the propagation of engine-internal events to the outside with the help of messaging and enabled the debugging of the execution of process instances [Ste08]. A messaging topic (*Event Topic*) is used to provide all engine-internal events of the engine wrapped as Event Messages to multiple external subscribers. These subscribers are able to influence or debug the execution of instances by sending messages to a provided queue (*Management Queue*). For example, by sending a corresponding management message, variable values can be set or breakpoints can be resolved by an external application. This initial work was extended by other diploma thesis with flexibility and model migration concepts. Ning added new functionality to dynamically iterate or re-execute parts of a running process instance (*Flexibility: Iteration, Re-execution*) [Nin11]. The difference between iteration and re-execution is, that the former just starts a set of activities again whereas the latter first compensates the corresponding activities in reverse order and then starts them again. In other words the iteration creates a kind of loop cycle where all contained activities are executed once more. For example, if the loop cycle contains a data assignment activity which increases the value of a variable, this value is increased again. The re-execution creates also a loop cycle but first compensates all completed activity contained in the cycle. For example, an increased variable is reset to the value before it was increased. The compensation step resets the whole instance back to the state before any of the activities which should be re-executed are started. Schliemann added a model migration concept which provides the ability to change the underlying model of a running instance by migrating the instance to the changed model (*Flexibility: Migration*) [Sch11]. This enables a user to change the control flow of a running instance and provides a

trial and error like modeling approach for scientists. They can start modeling their maybe complex simulations with simple model templates. These templates are then extended with additional BPEL constructs (e.g. new activities or variables) and immediately executed to try if the model behaves like they expected. If an error occurs or something is missing, the model is adapted and the running instance is migrated to this model. The migration is necessary because simulation workflows are quite often long-running processes and therefore it must be able to adapt the control flow of a running instance without loosing the current status of the instance. The migration functionality is also combinable with the iteration and re-execution flexibility functionality. This provides much more powerful possibilities because it enables the migration of the past of a running instance by just re-executing the completed part, adapt the model and migrate the instance.

The *Persistence Layer* provides the *Engine Database* which holds all deployed process models and all instance-related data like events, message exchanges, variable values or the execution state. As already mentioned, the DAOs are used to mediate the interaction between the Runtime Layer and the Engine Database. ODE provides per default two different DAO implementations which can be used to realize data persistence, Hibernate[3] and OpenJPA[4]. The *BPEL Compiler* converts all files contained in a deployment bundle (e.g. BPEL process files, WSDLs or schemas) into an engine-internal representation which is suitable for the execution. The result of a compilation is an object model similar to the BPEL process structure but with resolved references (e.g. variables or parner links referenced by their names) and some generated default objects, like default fault handlers or compensation handlers. This object model is serialized as a **.cbp* file along with the deployment bundle. The *Process Store* component handles the deployment of new process models and triggers their compilation. It also uses DAOs to persistently store related data, like the set of deployed process models and their state (e.g. active or retired).

2.6 Java Business Integration

The *Java Business Integration* (JBI) defined by the Java Community provides a standards-based architecture for integration solutions [JBI05]. This architecture allows different vendors to "plug in" their components into a standardized infrastructure which enables the decoupling and interoperability between their components on the basis of standards-based messaging. The result is a multivendor "echosystem" of interoperating components [JBI05].

The vendor-specific components therefore communicate over a standardized message exchange format, the so called Normalized Message (NM) format. A NM has three fields. An eXtensible Markup Language (XML) payload, a meta-data and an attachment field which is referenced by the payload. Instead of directly connecting the JBI components – a mediator – the Normalized Message Router (NMR) shown in Figure 2.2, routes the NMs between the components. A component can either be provided as a *Binding Component* (BC) or a *Service Engine* (SE). The

[3]JBoss Community, Hibernate: http://www.hibernate.org/
[4]The Apache Software Foundation, OpenJPA: http://openjpa.apache.org/

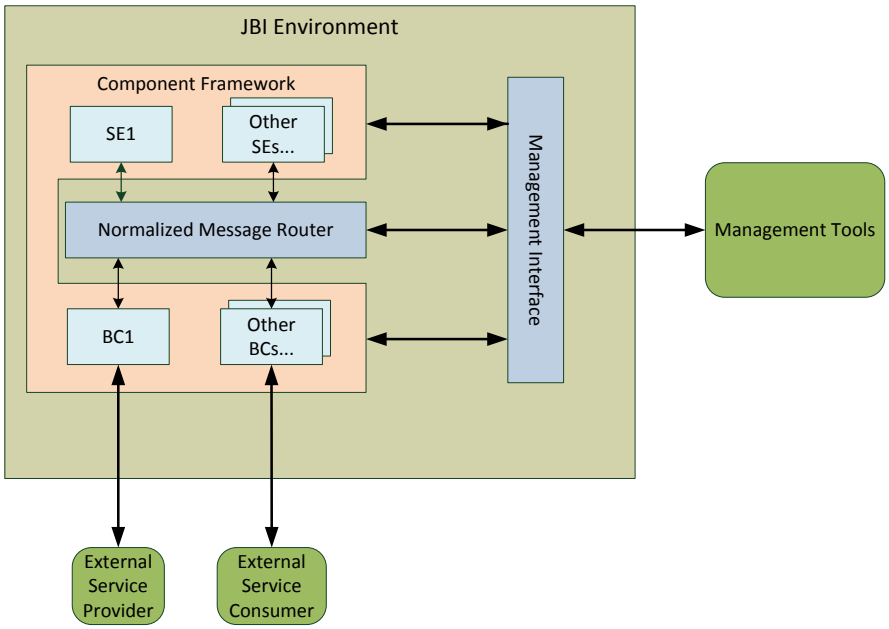

Figure 2.2: Architecture of the Java Business Integration, cf. [JBI05]

former are used to enable the exchange of protocol-specific messages between any connected external service and the JBI environment. Therefore they isolate the JBI environment from any particular protocol by normalizing and denormalizing any incoming or outgoing messages to and from the internal NM format. The latter provide some advanced message processing functionality like message transformation, message routing or the composition of existing services. For example a SCE is such a JBI SE which enables the composition of existing services. Both JBI components can act as service consumers or service providers where the services are described according to the WSDL 2.0 [5] specification. A service provider makes a service available through an endpoint [JBI05]. A service consumer uses a provided service by initiating a message exchange through the endpoint of the target service. All JBI components are able to exchange normalized messages through the NMR. Therefore the components act with the NMR over a *Delivery Channel* which provides a bidirectional delivery contract for message reception and delivery [JBI05]. Different SEs can directly communicate with each other using NMs without the need of normalizing or denormalizing messages from or to any protocol-specific format.

The JBI specification provides four different asynchronous message exchange patterns which enable the communication between the JBI components. They differ in the direction and the reliability of the communication they provide. New component-specific artifacts, so called *Service Units* (SU), can be deployed to any of the installed BCs or SEs of a JBI environment.

[5] W3C, Web Services Description Language (WSDL) Version 2.0: http://www.w3.org/TR/wsdl20/

The JBI specification does not define the contents of such SUs because they are opaque to JBI and the contents are determined by the target JBI component to which they should be deployed. For example, the contents of a HTTP SU which integrates an external service to the JBI environment are completely different as for a SCE SE which contains a process model and other related files. In case that the most integration problems could not be solved by involving a single JBI component, one or more SUs are grouped into a *Service Assembly* (SA). These SAs are usually packaged as ZIP files and contain a JBI deployment descriptor which specifies the components where each of the contained SUs should be deployed to. As shown in Figure 2.2 the JBI environment provides a *Java Management eXtension* (JMX) Interface which enables the installation and life cycle management of JBI components and the deployment of new artifacts to installed components.

2.7 OSGi Framework

The OSGi framework supports the deployment of extensible and downloadable applications known as *bundles* in a Java Virtual Machine (JVM) [OSG12]. It provides loose coupling for modularized applications and encourages dynamic code-loading by resolving the dependencies between the deployed bundles. An OSGi bundle is packaged as a Java Archive (JAR) file which contains a collection of Java classes and meta-data specifying the list of capabilities and requirements of the bundle. A capability is a reference to a Java package for which the bundle provides Java classes for. They provide some functionality to other bundles or an end user. Requirements are references to Java packages the bundle relies on, but that are provided as capabilities by another bundle. If a new bundle is installed to an OSGi framework, the framework uses the meta-data of the bundle to resolve its dependencies by matching the list of required packages with the capabilities of all installed bundles. This enables the reuse of any existing functionality encapsulated by a bundle and the realization of loosely coupled modular applications. Another important aspect of the OSGi framework is the bundle lifecycle management. One of its key features is the installation, update and uninstallation of bundles without requiring a system reboot. Each bundle therefore has to implement OSGi specific interfaces for lifecycle management that allow the OSGi framework to start and stop a bundle after it is installed and resolved.

2.8 Multi-tenant aware Apache ServiceMix

In this diploma thesis the multi-tenant aware version of Apache ServiceMix[6] 4.3.0 realized by Essl, Gómez and Muhler is used to provide and integrate a multi-tenant aware version of SWfMS. The underlying Apache ServiceMix 4.3.0 version is referred to as ServiceMix in this document. Essl evaluates in his work different available ESB solutions and decides to extend ServiceMix with multi-tenancy support by realizing tenant-aware communication [Ess11].

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

Muhler added with his work the required tenant-aware administration and management functionality to ServiceMix [Muh12]. Gómez integrates the work of Essl and Muhler and evaluates the resulting system [Sáe13]. This multi-tenant aware version of ServiceMix is used for this diploma thesis and is further referred to as ESB^{MT}.

The OSGi Framework implementation Apache Karaf[7] builds the kernel layer of ServiceMix. It provides a lightweight container into which various components and applications can be deployed. To install new components like OSGi bundles, JBI components and SAs or to manage the lifecycle of installed components, Apache Karaf provides an extensible management text console. Furthermore, new components can also be deployed using Apache Karaf's hot deployment mechanism by just copying the component packages into a *deploy* folder. If one of the package files is deleted from the deploy folder the corresponding component is automatically undeployed.

ServiceMix includes a JBI container which is fully compliant with the JBI specification. The NMR is based on the open source message broker Apache ActiveMQ[8] and routes the messages between the logical endpoints of the deployed JBI components. An endpoint is either registered as consumer or provider. The former are exposed as services which consume messages that are routed by the NMR to the correct JBI component and the latter provide access to an external service by receiving messages from the NMR and sending them to the external endpoint of the service. ServiceMix is delivered with a collection of already deployed OSGi bundles and JBI components. In the scope of this thesis we will concentrate on the HTTP BC extended by Muhler and Gómez and the Apache Camel[9] SE.

2.8.1 Multi-tenant HTTP Binding Component

The HTTP JBI BC provides HTTP communication support in ServiceMix. The original implementation is extended for ESB^{MT} to enable tenant-aware HTTP communication in [Muh12] and [Sáe13]. Therefore Muhler provides tenant-aware endpoints which are dynamically created in the BC by injecting tenant context data in the generated JBI endpoint URLs [Muh12]. In addition to that, Gómez provides a Normalized Message Format (NMF) which attaches tenant context data as a set of properties to a NM [Sáe13]. The assigned property values are used by the NMR during runtime for the correct routing of the messages. The functionality of the tenant-aware HTTP BC is used in this diploma thesis to provide multi-tenant aware endpoints for the process model Web Services exposed by the SCE. This is further described in Chapter 5.

[7]The Apache Software Foundation, Apache Karaf: http://karaf.apache.org/
[8]The Apache Software Foundation, Apache ActiveMQ: http://activemq.apache.org/
[9]The Apache Software Foundation, Apache Camel: http://camel.apache.org/

2.8.2 Apache Camel

Apache Camel[10] is a powerful open source integration framework based on *Enterprise Integration Patterns* (EIP) [HW04]. It supports the definition of routing and mediation rules in a variety of *Domain Specific Languages* (DSL). Each rule defines a routing or mediation between two or more endpoints. For example a Java-based DSL, Spring-based XML configuration files or a Scala DSL can be used to specify these rules. The Java-based DSL can be used in any class which extends the Apache Camel *RouteBuilder* class and is packaged in a *Plain Old Java Object* (POJO) file. The routing or mediation configuration files must be packaged in a SU to comply with the JBI specification. As already described one or more SUs must be packaged in a SA for deployment. To ease the process of creating SUs and package them in SAs, Apache Camel provides a set of Apache Maven[11] archetypes. Apache Maven is a software project management tool which manages the build of a project based on a *Project Object Model* (POM) file. An archetype is a kind of project template which can be used in Apache Maven to generate an Apache Maven project with a predefined structure. For example in the case of Apache Camel, the "camel-archetype-java" archetype generates a project which contains already an example POJO file. This file provides a class which extends the Apache Camel *RouteBuilder* class and can be directly used to define some rules using the Java-based DSL. The definition of rules in a XML configuration file provides a much faster and easier approach for developers but also restricts the complexity of the routing and mediation rules. The use of the Java-based DSL in a POJO class provides a much more powerful approach but therefore the development complexity increases. For example the developer is able to use the data contained in the header or the body of a NM to dynamically route the message to the correct target endpoint. In the context of this diploma thesis Apache Camel is used to realize some dynamic message routing scenarios which are described in detail in Chapter 5.

2.9 JBIMulti2

Muhler has implemented JBIMulti2 which provides a tenant-aware administration and management layer to enable multi-tenancy awareness for ESB^{MT} [Muh12]. In the context of this diploma thesis, JBIMulti2 is reused and extended in some parts to provide a tenant-aware administration and management layer for a multi-tenant SCE. As already described in Section 2.6, users deploy new component-specific artifacts in form of SAs which consists of a set of SUs to a JBI environment. In a multi-tenant JBI environment these SAs and the SUs they contain, must be deployed in a tenant-aware manner. This means some tenant information has to be provided for the deployment so that for example a deployed endpoint configuration is exposed as a service which is only accessible by the tenant who has deployed the corresponding SA. Muhler solves this lack of tenant-specific data during deployment with JBIMulti2 by injecting tenant context in all SA packages which makes them tenant-aware [Muh12].

[10]The Apache Software Foundation, Apache Camel: http://camel.apache.org/
[11]The Apache Software Foundation, Apache Maven: http://maven.apache.org/

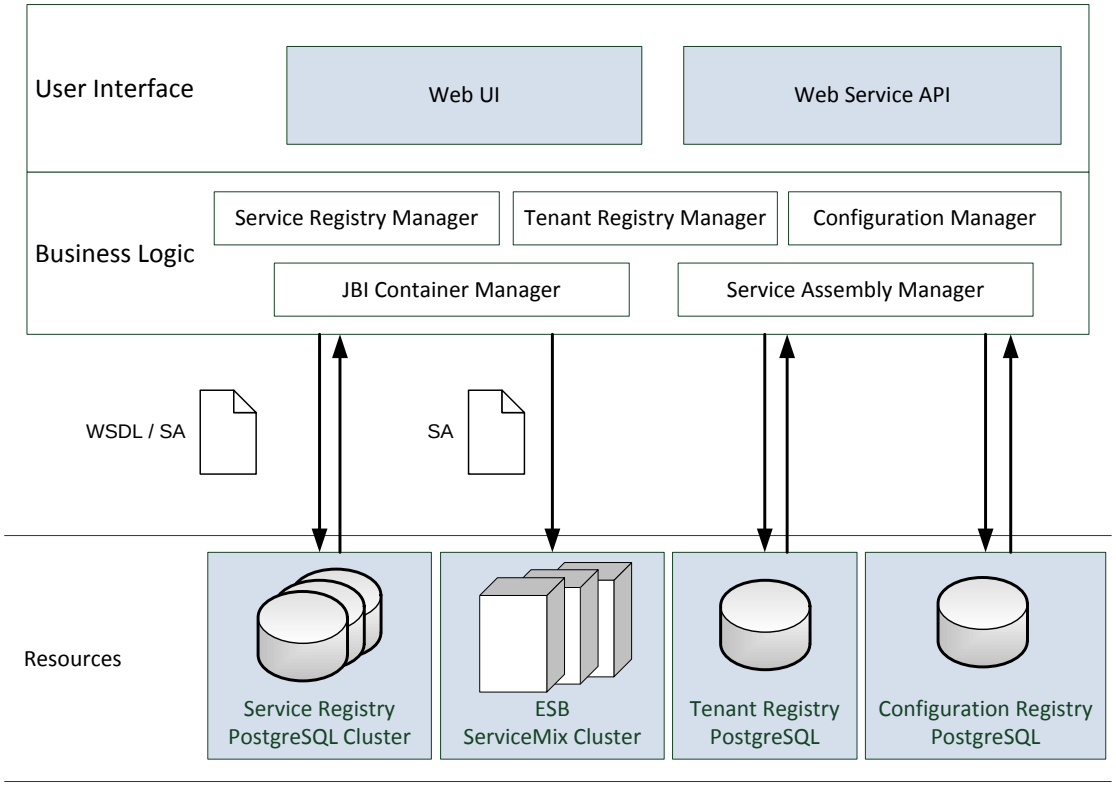

Figure 2.3: JBIMulti2 System Overview, [Muh12]

Figure 2.3 shows an overview of the JBIMulti2 system. It consists of three parts: a user interface, business logic and a set of resources. JBIMulti2 uses three registries to store the data of tenants and their users, their configuration and management data and their deployed SAs. If a new tenant or tenant user is registered at JBIMulti2 an unique identifier is created and stored together with some other tenant data in the *Tenant Registry*. All three registries are designed to persist the data of more than one application and therefore enable sharing of data. This benefit is used in this diploma thesis for the tenant-aware deployment of process models and the management of configuration data. The Tenant Registry is used to store any required tenant information. The *Service Registry* stores SAs in a tenant-isolated manner. All other tenant-related data is stored in the *Configuration Registry*. This contains for example the user roles of each tenant or which SA belongs to which tenant user. In the context of this diploma thesis the database schemas of the Service and Configuration Registry introduced by Muhler are extended to store SCE and process model related data. Furthermore a new *Event Registry* is introduced which stores the event data emitted by a SCE during the execution of process models.

The user interface provides access to the systems business logic. Currently only the Web Service API is implemented. The business logic provides the secure and tenant-aware management of tenants (or tenant users) and their resources (SAs, JBI components, etc.) by providing a role-based access control mechanism. The tenant-aware deployment of SAs into an Apache ServiceMix instance is realized over a JMS topic to which all ServiceMix instances are subscribed to. JBIMulti2 publishes a message to this topic which contains the SA ZIP file and the tenant context it belongs to. In each of the subscribed ServiceMix instances a JMS management service is installed which consumes these messages and deploys the contained SAs in a tenant-aware manner. This is realized by serializing the tenant context information send with the message as a XML file into each SU contained in the SA. These tenant context XML files are then used during runtime by the multi-tenant aware BCs to realize tenant-aware authentication of incoming requests send to a service endpoint. To deploy the SAs, the JMS management service uses the administration functionalities provided in ServiceMix. The communication between JBIMulti2 and the ServiceMix instances is realized unidirectional. This means in case of successful deployment the deployed endpoint is immediately reachable by the tenant and in case of an error an unprocessed management message is send to a dead letter queue.

Detailed descriptions of the functionality of the JBIMulti2 application and how it is installed and used are out of the scope of this thesis and will be provided in the work of Muhler [Muh12] or Gómez [Sáe13].

3 Related Works

This chapter provides a general overview of existing work that deals with the topic of multi-tenancy and the requirements and challenges to realize multi-tenant aware applications or services. Furthermore, the identified challenges, requirements and solution approaches and how they are applicable to this diploma thesis are evaluated.

There exist a variety of different multi-tenancy definitions in literature, for example in [GSH+07], [KMK12] or [WTJ11]. In the context of this diploma thesis we use the definition provided by Strauch et al. They define multi-tenancy as "the sharing of the whole technological stack (hardware, operating system, middleware and application instances) at the same time by different tenants and their corresponding users" [SALM12].

Most of the related works discussed in this chapter are based on enabling multi-tenancy and the realization of isolation and scalability for software hosted in the cloud, for example *Software as a Service* (SaaS) applications. SaaS is a cloud delivery model which enables multiple customers to use an application which is hosted in the cloud [ALMS09]. Customers do not have to care about the underlying infrastructure or the installation of applications or services. On the other side, this service model enables service providers to use their infrastructure and middleware in a shared way to provide such SaaS offerings and therefore to serve a much higher number of customers in parallel. This reduces operational costs and enables new potential customers (e.g. small and medium sized businesses) to use this kind of offerings. On the one hand the architecture of this applications must enable maximized sharing of resources across tenants and on the other hand still be able to recognize which data belongs to which customer [CC06]. The three main attributes to create such SaaS applications which are able to serve hundreds or thousands of customers in a parallel and isolated manner are *scalability*, *multi-tenant efficiency* and *configurability* [CC06].

Chong and Carraro have defined a four-level maturity model where each level adds the support for one of the three attributes defined above [CC06]. Figure 3.1 shows the four levels of maturity. In Level 1 each tenant has its own customized version of the application and runs one instance of it on the provider's servers [CC06]. As a result the provider is able to reduce costs by utilizing and sharing its server hardware in an efficient way to host the application instances. Level 2 adds *configurability* to a SaaS application. In this level each tenant also has its own instance of the application. The main difference is that the customization is realized over configuration and not by customizing the implementation of the application for each tenant. The SaaS application must provide therefore a set of configuration possibilities which enable the tenant to customize its instance of the application. This move to a single code base enables the provider to provision changes of the implementation of the application for all tenants at once [CC06]. But the provider still needs to host a potentially high number

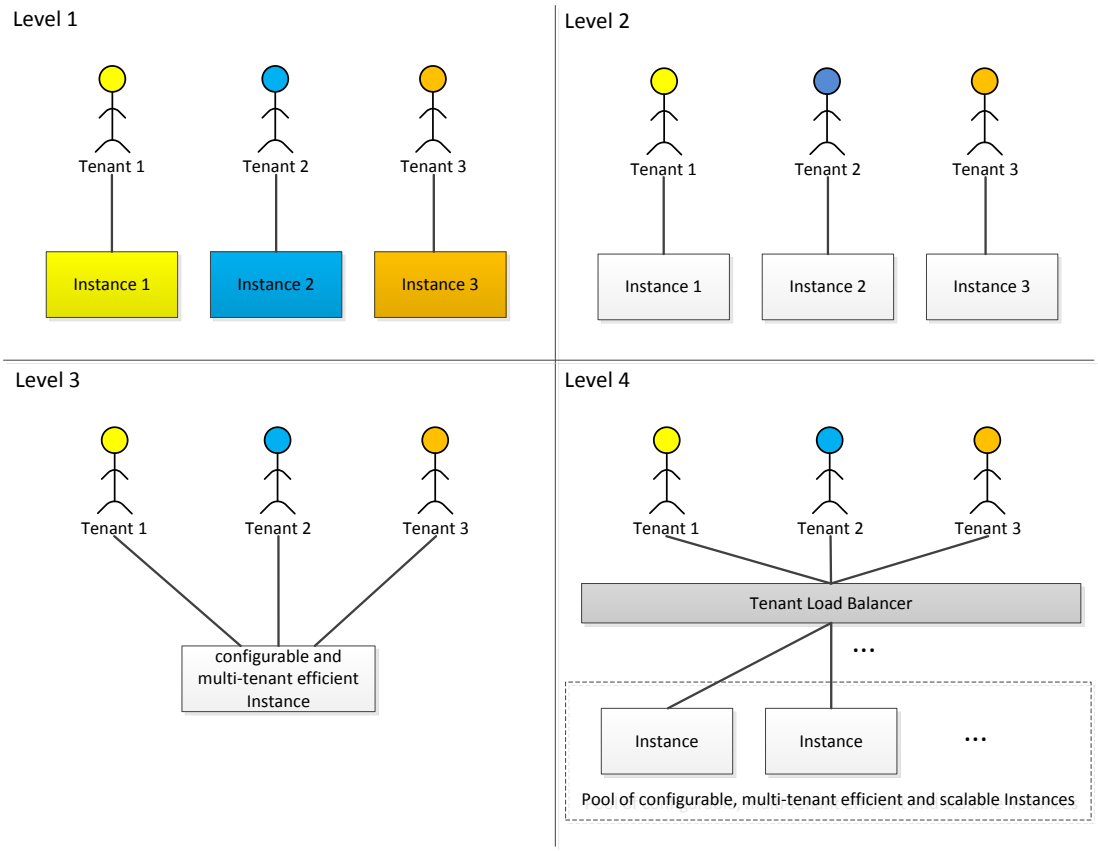

Figure 3.1: Four-level SaaS maturity model defined by [CC06]

of concurrently running application instances. Level 3 adds *multi-tenant efficiency* to the application. This enables the provider to use a single (configurable) instance of the application to serve all tenants [CC06]. On the one hand this level introduces new challenges like the isolation of resources between tenants (data, performance, ...) or security concerns. On the other hand the provider does not need to prepare dedicated server space for each of its tenants to run a separate application instance. This efficient use of computing resources leads to much lower costs [CC06]. The last level, Level 4, adds *scalability* to the application. A Tenant Load Balancer provides scalability by dynamically increasing or decreasing the number of instances based on the number of tenants to serve. Chong and Carraro indicated the four maturity levels as "*a continuum between isolated data and code on one end, and shared data and code on the other*". This is important if you choose the "right" maturity level for an application. Level 4 is not for all scenarios the best case, for example if the customization needs of a customer are not realizable over configuration options.

Guo et al. introduce two main multi-tenancy patterns: *multiple instances* and *native multi-tenancy* [GSH+07]. The former provides one separated instance of an application for each tenant hosted on a shared infrastructure (e.g. shared hardware) [GSH+07]. In contrast to that, native multi-tenancy provides one shared single instance of an application which serves natively multiple tenants. Compared to the described maturity model of Chong and Carraro, *multiple instance* multi-tenancy is provided by the Levels 1 and 2 and *native multi-tenancy* is realized by the Levels 3 and 4.

Anstett et al. investigate the execution of BPEL processes in the cloud based on different delivery models namely infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS) [ALMS09]. They point out a set of requirements and challenges based on the used delivery model. Outsourcing the execution of BPEL processes with the IaaS delivery model has no special requirements since it is just a move of responsibilities for hosting the infrastructure to the cloud provider [ALMS09]. The installation of the BPEL Engine and the configuration of the infrastructure (e.g. security) are not much different, as everything is hosted in their own data centers. Moving one level up the hierarchy by using the PaaS delivery model, the provider also hosts the platform middleware, like the BPEL engine and a database [ALMS09]. This forces new security requirements to ensure the confidentially of the process models and their instances [ALMS09]. Since the PaaS provider might be able to directly access the database and the file system of the operating system on which the BPEL engine runs, the process models and their instance data must be protected (e.g. by encryption) [ALMS09, GSH+07]. The use of the SaaS delivery model transfers all responsibilities to the provider. The underlying infrastructure, middleware and even the BPEL processes are moved to the area of responsibility of the provider. This reduces the complexity for the customer but also forces a loss of flexibility because the provider decides which customization options he offers to its customers [ALMS09]. By the move of responsibilities, the process models become an asset of the SaaS provider and represent no longer an asset of the tenant's enterprise. The SaaS provider can serve multiple customers with the same process model. The SCE and the database, therefore have to support multi-tenancy.

Now we want to have a closer look at some solution approaches discussed in the related works for the (above introduced) new challenges and requirements to realize multi-tenant aware applications.

3.1 Configurability

As discussed above, *configurability* is one of the main requirements to realize a multi-tenant application ([CC06, KMK12, WTJ11]). Krebs et al. describe a configurable application as "one which provides tenant specific behavior or appearance, whereby this behavior is configured without tenant specific code" [KMK12]. The tenant-specific configurations can be specified over configuration files or defined with the help of an administration user interface (UI). The key requirement therefore is, that the change of the configuration of one tenant should not impact other tenants or influence the way the application behaves or appears for other tenants [KMK12, GSH+07]. Furthermore, the configuration of the application should be possible

during runtime to minimize unnecessary downtime and to do not impact other tenants. The configuration of an application is realized by the provisioning of a set of configuration options (e.g. UI elements) which can be used by the tenants to adapt the application to their needs. Possible configuration options for a SCE and its process models are evaluated in Chapter 4.2 and Chapter 4.3.

3.2 Scalability

As described above, scalability is an important requirement for a multi-tenant application. Applications should be able to scale with the number of users. We can distinguish between vertical and horizontal scalability. In the first case, applications can be scaled up by adding more computing resources to the application or by moving the application to a more powerful server. In the second case, applications can be scaled out by running more instances of the application maybe on different servers [CC06]. Scaling out is only possible if the application itself supports scalability. Chong and Carraro introduce some basic guidelines to design an application which can be scaled out. The most important one is that an application should run in a *stateless fashion*. Any user or session specific data should be stored apart from the application (e.g. in a distributed store), so that each transaction can be handled by any of the application's instances [CC06]. In the context of this thesis, scalability is not realized, but the architecture is designed with scalability in mind to enable an easier integration of a scalability approach in a future work.

3.3 Isolation of Tenants

Another important topic for the realization of a multi-tenant aware application is to provide isolation of tenants in nearly all parts of an application [GSH+07, KMK12]. To uniquely identify a tenant or associate resources with a tenant, a *tenant ID* ([GSH+07]) or a so called *tenant context* ([ALMS09, SALM12]) can be used. The following sections provide a short introduction and discussed solutions approaches of the most important isolation requirements: Data Isolation, Communication Isolation, Administration Isolation and Performance Isolation.

3.3.1 Data Isolation

The isolation of the tenants' data is one of the most important requirements when providing a multi-tenant application. Chong et al. introduce three different approaches to realize multi-tenancy at the data layer: *Separate Databases*, *Separate Schema* and *Shared Schema* [CCW06]. Similar to the the four-level maturity model, the three approaches define a continuum between isolation and sharing and the optimal degree between the two extremes depends on the tenants' requirements [CCW06]. The following list provides a short description of each approach and its advantages and disadvantages.

Separate Database The most isolated and simplest approach to realize a multi-tenant data architecture is to store the data of each tenant in a separate database. With the help of meta data (e.g. a tenant context) each tenant can be associated with his database and the security layer of the database provides a reliable access control [CCW06]. This makes it relatively easy to extend the application's data model on the needs of each tenant, but also causes relatively high hardware and maintenance requirements and costs [CCW06]. An advantage is the relatively easy restoring of the tenant's data from a backup in case of a failure. Since a database server can only host a specific number of databases, the number of tenants which can be served is also limited by the database server.

Separate Schema This approach uses a shared database, with each tenant having its own set of tables grouped in a separate schema [CCW06]. As a result, each tenant can then be associated with its schema and the access control is provided by the database security again. A significant drawback of this approach is the much more complicated restore of tenant's data in case of a failure [CCW06]. In fact that the data of all tenants is stored in the same database, one tenant's data can not be restored by just restoring the most recent backup. This would lead to the loss of data for all other (non affected) tenants because their current data is overwritten with the backup data. To restore just the tables of a single schema - the data of a specific tenant, the database administrator has to restore only the tables of the tenant's schema from the backup. This approach is appropriate for applications that use a relatively small number of database tables (about 100 or fewer) and can serve more tenants than the *Separate Database* approach [CCW06].

Shared Schema The most shared and cost effective approach to realize a multi-tenant data architecture is to store the data of different tenants in the same database and the same set of tables. Each table can contain records of all tenants where each record is associated with its tenant over a tenant id. Therefore each table has to be extended with a new column to store the tenant id to which a record belongs. The main advantage of the *Shared Schema* approach are the low hardware and backup costs, because it allows you to serve the largest number of tenants per database server [CCW06]. As a result of sharing a schema between tenants, the access control can not be provided by the database security. This leads to additional development effort because the application has to ensure that tenants can never access the data of other tenants. The backup procedure in case of a failure is similar to that for the *Separate Schema* approach. To restore the data of one tenant, the database administrator has to restore only individual rows in each of the database tables from the backup. This approach is therefore appropriate when the application should serve a large number of tenants with a small set of servers and the customers accept the lower level of data isolation in exchange for the lower costs of this approach [CCW06].

3.3.2 Communication Isolation

Communication Isolation is a special kind of Data Isolation which keeps the message exchanges for each tenant separate [SALM12]. Therefore, a message has to be associated with a tenant

context to uniquely identify the tenant whom the message belongs. This enables the authentication of incoming messages by a SCE based on the tenant context they are associated to.

3.3.3 Administration Isolation

Administration Isolation is provided if no tenant is able to manage or administer resources which belong to another tenant. For example, a SCE may provides functionality for the administration of process models, process instances and auditing data. To enable Tenant Isolation all of these resources are associated to a tenant. This tenant should be the only entity which is able to administer the associated resources. The Administration Interface should also isolate any high-level information (e.g. number of process models deployed, endpoint address of a process model) from any unauthorized tenants.

3.3.4 Performance Isolation

Performance isolation means to ensure that each tenant gets the performance he paid for and that the potentially bad behavior of one tenant does not adversely affect the performance of other tenants [WMTJ12, GSH+07]. The requirements of a tenant are defined over so called *Service Level Agreements* (SLA). Walraven et al. introduce a pluggable middleware framework to enforce performance isolation for multi-tenant SaaS applications [WMTJ12]. A *tenant-aware monitoring* solution is required to associate each request to a tenant and to monitor the resource usage of those tenant-aware requests throughout the whole processing cycle [WMTJ12]. Furthermore, the tenants must be able to specify tenant-specific SLAs to define their requirements as part of their configuration data. These SLAs are then validated during runtime with the help of *performance isolation algorithms* and the resource monitoring data by the pluggable middleware framework. A scheduling mechanism handles all incoming requests and ensures that the requests are processed by the application in an order where all tenant SLAs are met. If a tenant exceeds his quotas his requests are not processed until the application meets all other tenant SLAs and has again some free resources to handle the new requests.

3.4 Existing Multi-tenant SCE Approach

At the end of this chapter, we will have a look at an existing approach to realize a multi-tenant SCE. Pathirage et al. introduce a multi-tenant SCE architecture based on the open source BPEL engine Apache Orchestration Director Engine (ODE)[1] [PPKW11]. They use the WSO2 Carbon[2] platform to enable multi-tenancy in ODE. WSO2 Carbon is an OSGi based platform for building scalable, high performance servers. It provides multi-tenancy support by some

[1]The Apache Software Foundation, ODE: http://ode.apache.org/
[2]WSO2 Inc., WSO2 Carbon: http://wso2.com/products/carbon

adaptations to its underlying execution engine Apache eXtensible Interaction System v. 2 (Axis2). Pathirage et al. reuse the multi-tenancy functionality of WSO2 Carbon in ODE by adapting the Axis2 integration layer of ODE. An integration layer enables the communication with external services and provides the internal services of ODE to the outside by the use of a service middleware (e.g. Axis2, Enterprise Service Bus). This differs from our approach because the use of Axis2 as multi-tenancy enablement layer and the adaption of ODE's integration layer makes the approach solution specific. Other SCE implementations may use another service middleware or do not offer the ability to provide a set of interchangeable integration layer implementations. We try to separate as much of the multi-tenancy functionality from the SCE internal logic as possible to provide an abstract and reusable concept, as discussed in Chapter 4. To make ODE itself multi-tenant aware, Pathirage et al. use one tenant-aware Process Store for each tenant. This enables the logical isolation of all process models inside of ODE on a per-tenant basis and directly identifies all models which belong to one tenant.

4 Requirements and Concepts

As described already multi-tenancy means sharing an application or service between a set of tenants. The three most important points to realize a multi-tenant efficient application are configurability, scalability and isolation. The conception of a multi-tenant SCE requires the identification of multi-tenancy aspects for the SCE itself and also for the process models. This is important because the process models are provided as services by the SCE as described in Chapter 2.4. Considering that, this chapter introduces the different aspects for a SCE and the executed process models and provides some general concepts how both can be extended to become multi-tenant aware. To enable a better understanding of the components of a SCE and what is required to realize multi-tenancy, first of all a general SCE architecture is introduced. This architecture is used in the following chapters to describe the necessary adaptations and extensions.

4.1 General SCE Architecture

There exists no standardized or reference architecture for Service Composition Engines in the literate. Each SCE implementation has its own vendor-specific architecture. For example, Apache ODE[1], Orchestra[2] or the YAWL Workflow Engine[3] are based on very different architectures. This is also one of the main reasons why the WfMC does not make any statements about the architecture of a Workflow Engine in their Workflow Reference Model [Hol95]. We use the descriptions in [LR00] to identify the main components and functionality of which a SCE consists. Figure 4.1 shows the resulting abstract SCE architecture. The layering is just used to organize the different components and their functionality. The components of the architecture are used in the following chapters to reference a specific part or functionality of a SCE. This should enable the mapping of the concepts provided in this thesis into any SCE implementation. A developer can use the descriptions to identify and extend the correct part of its implementation-specific architecture. Now we take a closer look on the abstract SCE architecture.

First of all, the architecture consists of three layers. The *Integration Layer* provides the engine-internal functionality to the outside and enables communication. It contains a *Message Exchange Processor* which handles the correct routing of incoming and outgoing messages to the their corresponding services (process models). For example, if a request is sent to the

[1]The Apache Software Foundation, Apache ODE (Orchestration Director Engine): http://ode.apache.org
[2]OW2 Consortium, Orchestra: http://orchestra.ow2.org
[3]The YAWL Foundation, YAWL (Yet Another Workflow Language): http://www.yawlfoundation.org

service interface of a process model which creates a new instance of the model, the Message Exchange Processor routes the request to the created instance and returns the response message of the instance back to the sender of the request. The *Service Interfaces* component provides the service of each deployed process model to the outside (e.g. as Web Service) and therefore enables sending requests to a process model. The *Management Interfaces* component provides the management functionality of the SCE to the outside (e.g. as Web Service), like querying all deployed models or suspending a running instance.

The *Runtime Layer* contains the logic of the engine. The *Navigator* is responsible for the execution of process instances based on the control flow defined in the process model. Each time an activity completes, the navigator evaluates all outgoing control flow connectors, calculates the next successor activities to execute and starts the corresponding *Activity Runtime* component. The engine provides an Activity Runtime for each activity type of the underlying process model language (meta-model). This runtime provides the functionality of an activity to the engine, like the assignment of data or the invocation of external applications or services. The *Correlator* component is responsible for the correlation of request and response messages. For example, if a process model has multiple concurrently running instances which invoke an external service, the response of this service has to be routed to the instance which sends the corresponding request. The *Process/Instance Manager* component contains the implementation of all process and instance management functionality which is provided over the Management Interfaces. This contains for example, the change of the execution state of an instance (e.g. suspend, resume, terminate), the deployment or undeployment of process models or querying some information from the engines databases, like a list of running instances.

The *Data Layer* provides the persistent stores of the engine which are normally realized with databases. The *Process Database* contains all process-related data, like a persisted version of all deployed process models, the state of each process model or a complete list of all started instances of each process. The *Runtime Database* contains all runtime-related data (instance contexts, event data, ...) in a persistent manner. This database is very important because it holds the instance contexts of all process instances, like variable data, incoming and outgoing messages or activity states. The Navigator and other components of the Runtime Layer use this instance contexts and the Instance Database to provide their functionality. The Process/Instance Manager component also works with both of the databases to provide information to the outside or to manage a process or an instance. For example, undeploying a process model over the Process/Instance Manager causes the deletion of the corresponding process model from the Process Database.

4.2 Multi-tenancy aspects of a Service Composition Engine

The SCE itself is (nearly) stateless because the execution state of process instances is hold in so called instance contexts separated from the status of the engine. These contexts hold all the data of an instance e.g. variable data, activity status or the current instance execution state. This makes it possible to use a single instance of a SCE to serve multiple tenants

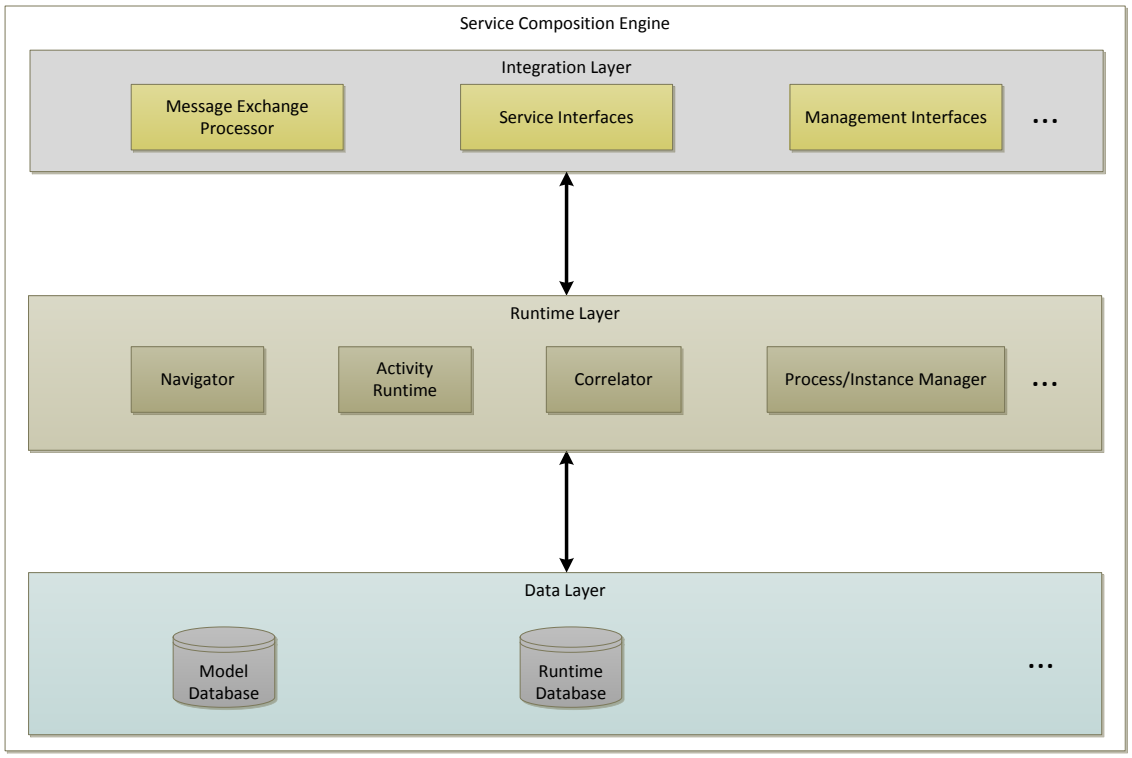

Figure 4.1: General architecture of a SCE

(single instance multi-tenancy/native multi-tenancy). The following sections describe how configurability, isolation and scalability can be realized in a SCE.

4.2.1 Configurability

Configurability is a key requirement for multi-tenancy and therefore the configuration of the single SCE instance on a per tenant basis should be possible.

Figure 4.2 shows the above introduced general SCE architecture with some exemplary configuration options. Tenants are able to use different configurations to adapt some parts in each layer of the shared SCE instance to their needs. All configurations are stored in a *Configuration Database*. To get the correct configuration for each tenant the corresponding data can be queried from the database with a *tenant context*. A tenant context uniquely identifies a tenant and can therefore be used to associate resources to a tenant. Each of the three layers provide some opportunities for configuration based on its functionality. The following descriptions provide just a subset of possible configuration options because they vary based on the used SCE implementation. In case of that, Chapter 5 provides the configuration options which are realized in the context of this thesis based on the extended ODE implementation.

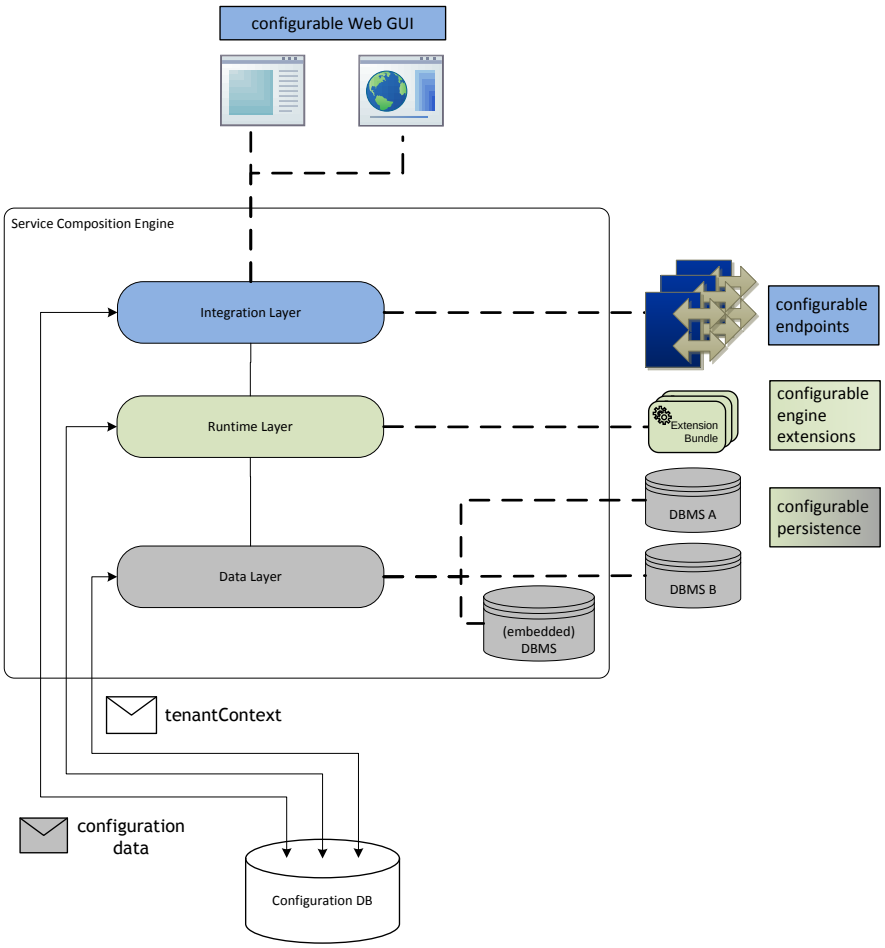

Figure 4.2: Some examples for the configurability of a SCE

As described above, the Integration Layer provides the SCE internal logic to the outside. Tenants might be able to use the following two configuration points:

Configurable Web GUI Tenants can define customized views on their data (e.g. process instance meta-data, runtime event data or statistics) and customize an Administration Web UI. For example, this can be realized with portlets [Por08].

Configurable Endpoints Tenants can specify the endpoints (e.g. HTTP address, JMS address, ...) on which the services of their deployed process models should be reachable. Another possibility would be the definition of a pattern to generate the endpoints, e.g. `http://[ip]:8085/sce/processes/[processServiceName]`. The values surrounded by square brackets are replaced by the SCE during runtime. This enables tenants to separate their

services from other tenants by using different ports and URL paths without the need to specify the endpoint for each process model.

The Runtime Layer provides the core functionality to execute process models. One useful configuration option would be the possibility to register *Extension Bundles* on a per tenant basis. An Extension Bundle provides the runtime implementation of one or more activities. This will enable the injection of tenant-specific code into the SCE which provides a customization opportunity. If the Navigator executes an activity for which an Extension Bundle is registered, the tenant-specific implementation is used instead of the default implementation. The process modeling language BPEL provides already the possibility to define so called BPEL Extension Activities which can be used to model new types of activities. To make these new activities executable an Extension Bundle which contain the runtime logic of the new activities, can be registered at the BPEL engine. This makes it possible that each tenant can define a set of new activity types which provide some tenant-specific functionality and register the corresponding implementation at the engine. Nevertheless it makes also sense that, e.g. the SCE provider registers a set of Extension Bundles which can be used by all tenants. The management of such Extension Bundles can be realized over an extension of the SCE Management API. So that tenants can dynamically activate or deactivate a set of registered Extension Bundles and register or deregister new/existing Extension Bundles.

The Data Layer realizes the persistent storage of the engine's data in one or more databases. The *configurable persistence* option shown in Figure 4.2 enables tenants to specify the database where to store their data of the SCE. A tenant is therefore able to choose whether he will use the default database of the SCE provider or hosts his own (self-managed) database. This is an important configuration option if data privacy and strict Data Isolation are fundamental requirements for a tenant or its users. If the default database of the SCE provider uses a Shared Schema approach to enable Data Isolation, this option enables the tenant to isolate his data in a separate database. The use of a separate database is a trade-off between an increase of isolation and a decrease of performance. As shown in Figure 4.2 the configurable persistence option is also filled with the color of the Runtime Layer. This is because this option also implicates some adaptations of the Runtime Layer to enable the selection of a database.

4.2.2 Isolation

As introduced in Chapter 3 the isolation of tenants has to be provided in nearly all layers of an application. Therefore, we analyze how a SCE can provide the different kinds of isolation based on the architecture shown in Figure 4.1.

Data Isolation The Data Layer should provide Data Isolation. Therefore, the Model Database and the Runtime Database must be realized multi-tenant with one of the Multi-tenant Data Architecture approaches described in 3.3.1. Also the configuration data has to be stored in a multi-tenant database which isolates the tenants' data.

Communication Isolation The tenant context of incoming requests must be forwarded by the Message Exchange Processor to the Runtime and Database Layer. As a result, the

tenant context will be associated to the process instance which handles the request message. This makes it possible to use the tenant context for communication isolation and attach it to every outgoing message which is send by a process instance to an external service or application. The Correlator is also able to use the tenant context associated to the messages if this is required to realize message correlation in an multi-tenant SCE.

Administration Isolation One tenant should not be able to manage the process models or instances of another tenant. Therefore, the Process/Instance Manager should be extended to authenticate request messages send to the Management Interfaces. The authentication can be realized by comparing the tenant context which is associated to a process model or instance and the tenant context which is associated to the incoming message.

4.2.3 Scalability

To enable scalability the SCE should be run in a stateless fashion. This means that any tenant specific data (e.g. process models or configuration data) should not be bound to a single SCE instance and therefore be stored apart from a SCE instance in a distributed store. As a result, this enables the realization of a load balancing mechanism which uses the data of the distributed store to dynamically create new configured instances of a multi-tenant SCE (maybe based in different implementations), deploy the tenants process models and thus serve new incoming requests of the tenant's users in a highly flexible and transparent manner.

4.3 Multi-tenancy aspects of a Process Model

A process model is per default multi-tenant because each incoming request which is send to the service interface of a process model creates a new instance (multiple instances multi-tenancy). The following sections describe how configurability, isolation and scalability can be realized for process models.

4.3.1 Configurability

By enabling the configuration of process models, tenants can adapt some parts of the model or influence the execution of an instance. Figure 4.3 shows some possible configuration options. All configurations are registered in a Configuration Database by the tenants or users. The figure shows two different types of configuration options for process models.

One possibility is to just register a set of runtime data (variable values, partner-link values, ...). These runtime data are loaded during instantiation time of the process model and assigned to the target elements of the instance, like variables or partner-links. This makes it possible that instances of the same model use automatically the correct tenant data without any further work. With the help of this configuration option, tenants are able to specify different callback

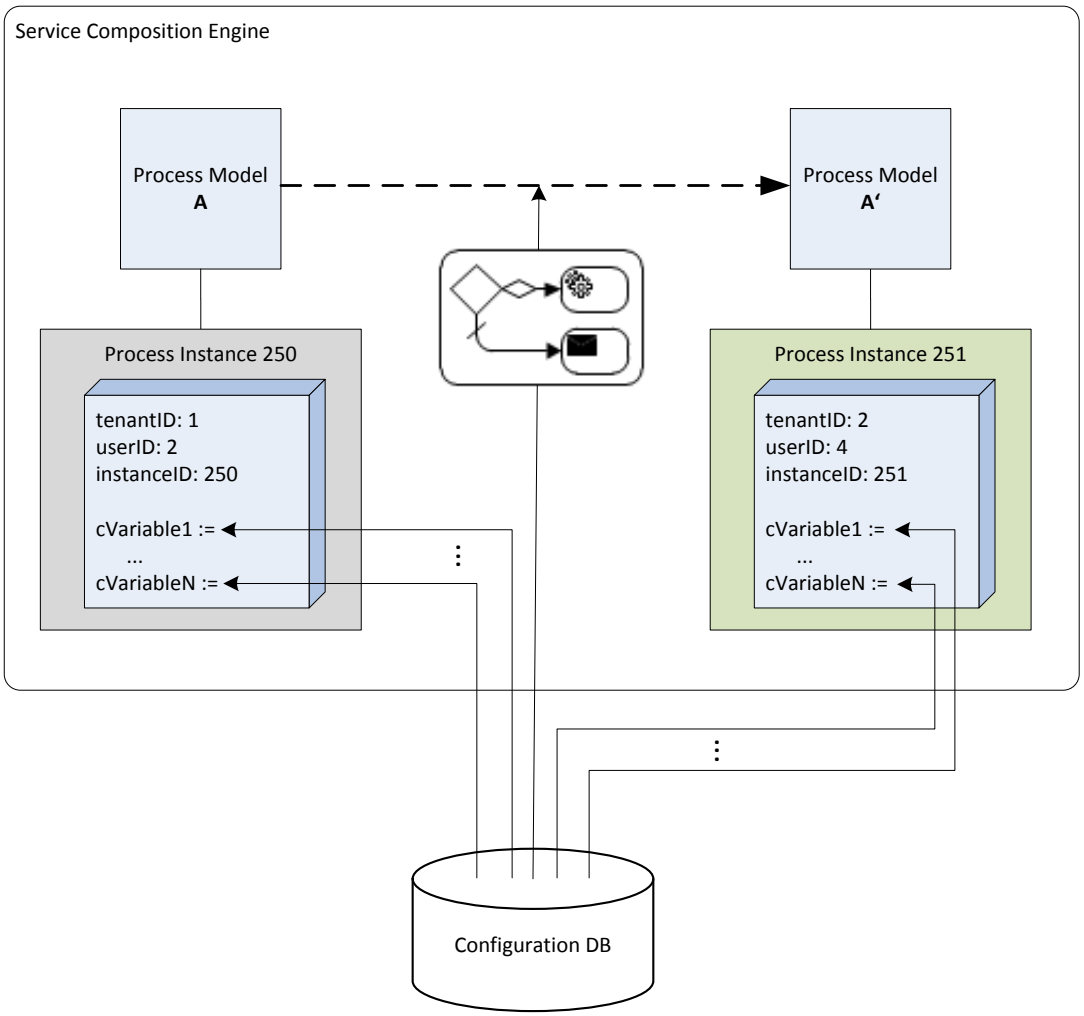

Figure 4.3: Some examples for the configurability of a process model

endpoints over partner-links or use other constant values (e.g. country specific tax rates or simulation constants) in variables without any changes to the process model.

Another option is to register process fragments (control flow snippets). These fragments are dynamically weaved in the process model on instantiation time. This is realized by a re-compilation of the original (template) model (Process Model A, in Figure 4.3) with all referenced process fragments. The result is a new temporary Process Model A' which can then be instantiated. This changes the control flow of the process model for all instances of one tenant. This option is not realized in the context of this thesis, but Chapter 6 provides some more details how the configuration of process models with process fragments can be realized.

For both possibilities, some elements (variables, partner-links, activities, ...) of the process model are enriched with marks by the process modeler. These marks are used by the SCE to load the corresponding configuration data from the database. How far process models can be configured is limited by the process model itself. If there are no configurable (marked) elements modeled a tenant or user is not able to configure anything.

4.3.2 Isolation

The SCE holds the tenant/user specific runtime data in an instance context during the execution of a process instance. This provides Data Isolation on a process model level since each instance has its own context which is by default not accessible from another instance. All other isolation requirements must be fulfilled by the SCE, as described above.

4.3.3 Scalability

A process model provides native scalability because each request is handled per default by a new instance of the model. The number of parallel running instances is only limited by the performance of the SCE on which the instances are running. But this restriction is rescinded by just deploying the process model to more than one SCE instance. For example, with the help of a Load Balancer the customers' requests can then be spread across all SCE instances on which the process model is deployed.

4.4 Behavior of a Multi-tenant aware SCE and Process Models

This section describes how the behavior of the SCE and the process models changes if they become multi-tenant aware and the process models are deployed and executed by a multi-tenant SCE. For that, the four most important procedures will be described: process deployment, process instantiation, service invocation and instance correlation.

4.4.1 Process Deployment

Figure 4.4 shows how tenant-based deployment of process models looks like in a multi-tenant SCE. The yellow components mark extensions or adaptations of the SCE to enable multi-tenancy. All Deployment Bundles which are deployed under tenant context, are associated to a specific tenant or user. "Deployment under tenant context" means that a Deployment Bundle is registered for one specific tenant or user which is uniquely identified by the tenant context. This tenant or user becomes the owner of the contents of the Deployment Bundle and therefore is the only entity who is able to use the Deployment Bundle (e.g. to instantiate a process model). A Deployment Bundle contains a set of process models and any other information which the SCE needs to execute the process models, like service interface descriptions (e.g. WSDL files) or a deployment descriptor. The deployment descriptor is used to provide the

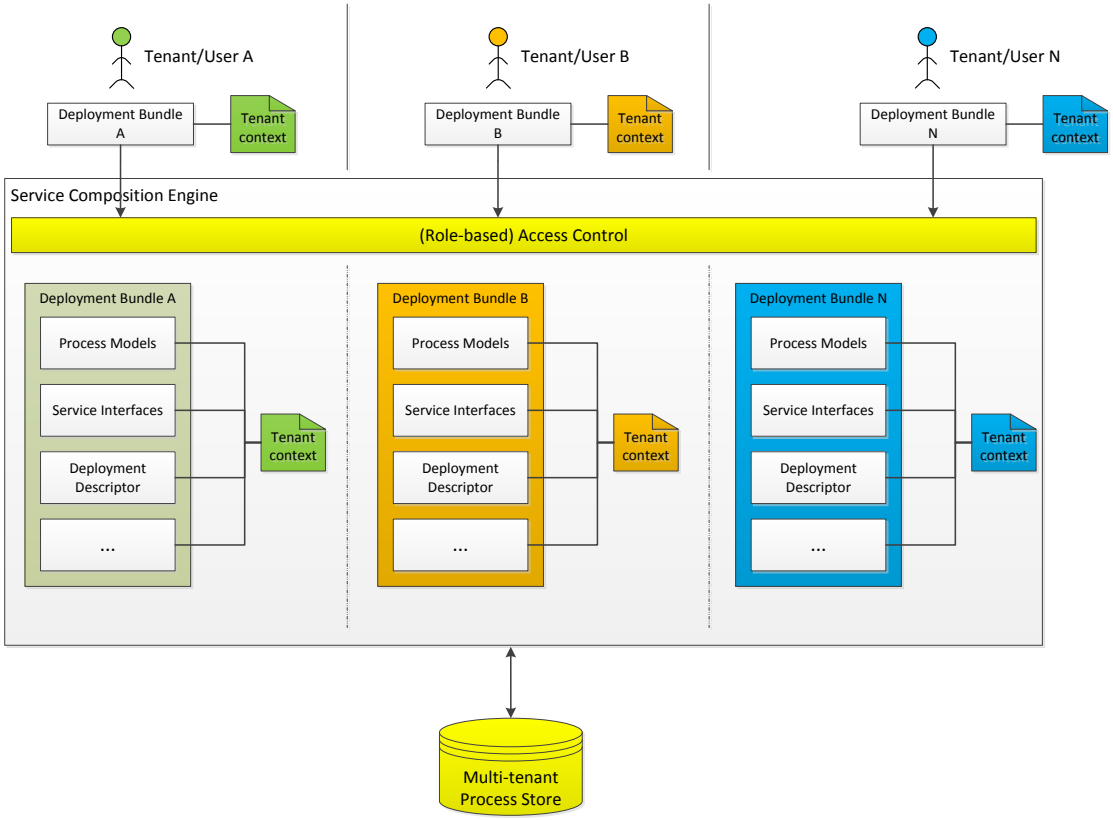

Figure 4.4: Process Deployment with a multi-tenant SCE

SCE some deployment specific information for the process models of a bundle. For example, the endpoint location of used external services or of the provided process services. A new role-based Access Control layer validates if a tenant or user has the corresponding role or access permission to deploy a new bundle. After that, the engine associates the Deployment Bundle and all its contents with the tenant context and persists the bundle in a multi-tenant aware Process Database. This enables the engine to evaluate if a tenant or user is allowed to instantiate a process model or for example query information about process models over the Management Interfaces of the engine (see Chapter 4.1). To loosen the one-to-one relationship between a tenant or user and a process model it should be possible to define a so called deployment style when deploying new bundles. This enables some kind of collaboration and makes it possible that a tenant can deploy process models which can be used by other tenants and their users. The SCE provider is also able to deploy some public process models for all tenants he serves. The different deployment styles and their effects are described in detail in Chapter 4.5.

4.4.2 Process Instantiation

Figure 4.5 shows the process instantiation behavior of a multi-tenant SCE. The colored components mark extensions or adaptations to provide multi-tenancy. First of all, to uniquely identify the tenant or user a message belongs to, all messages should be extended with a tenant context. With the help of the tenant context, the SCE is then able to check if an incoming request send to a process service is valid or not. The instantiation of a process model by a message which contains a tenant context is referred to as "invocation under tenant context". Therefore the SCE compares the tenant context associated to the incoming request with the tenant context which is associated to the process model stored in the Multi-tenant Process Database. In other words the engine has to check if the sender of the request is the owner of the process model (e.g. *owner(ProcessModel1) = UserA?*). If the tenant contexts are equal the engine can forward the request to the service interface of the process model as shown in Figure 4.5 for User A and B. If the tenant contexts are not equal the engine returns a corresponding fault message to the requester as shown in Figure 4.5 for User N. The generated new process instance is directly associated with the tenant context contained in the request. This is necessary to enable the SCE later to authenticate calls to the Management Interfaces for an instance, like suspending or terminating the instance. If a tenant has registered any configuration data for the instantiated process model, these data is dynamically loaded from the Configuration database and assigned to the instance. The runtime data of all process instances is persisted in the Multi-tenant Instance Store.

4.4.3 Service Invocation

The invocation of engine internal services (process models) and external services is shown in Figure 4.6. An instance of Process A invokes an external service and instantiates another process model (Process B). As described in the previous section, a tenant context is used to identify the tenant and user to whom a message belongs. For the invocation of services the tenant context plays another important role. If the invoked service is also multi-tenant aware like Process B, the tenant context is used to check if the referenced user has the permissions to use the service or not. For that purpose, the tenant context must be forwarded by the SCE to any invoked service whether it is multi-tenant or not. This is no problem because if the invoked service is not multi-tenant aware it just ignores the tenant context send with the request message. In both scenarios the service could respond with a fault message. The reason for the fault response might be an internal exception of the service which is thrown to the outside or a tenant context which references a user without the correct permissions to use the service. For example, those fault messages can then be handled by the SCE with the fault handling mechanisms of the underlying process modeling language (e.g. BPEL).

4.4.4 Correlation of Process Instances

The correlation of asynchronously communicating process instances shown in Figure 4.7 does not change in case of enabling multi-tenancy. An instance of Process A (*Instance A*) creates a

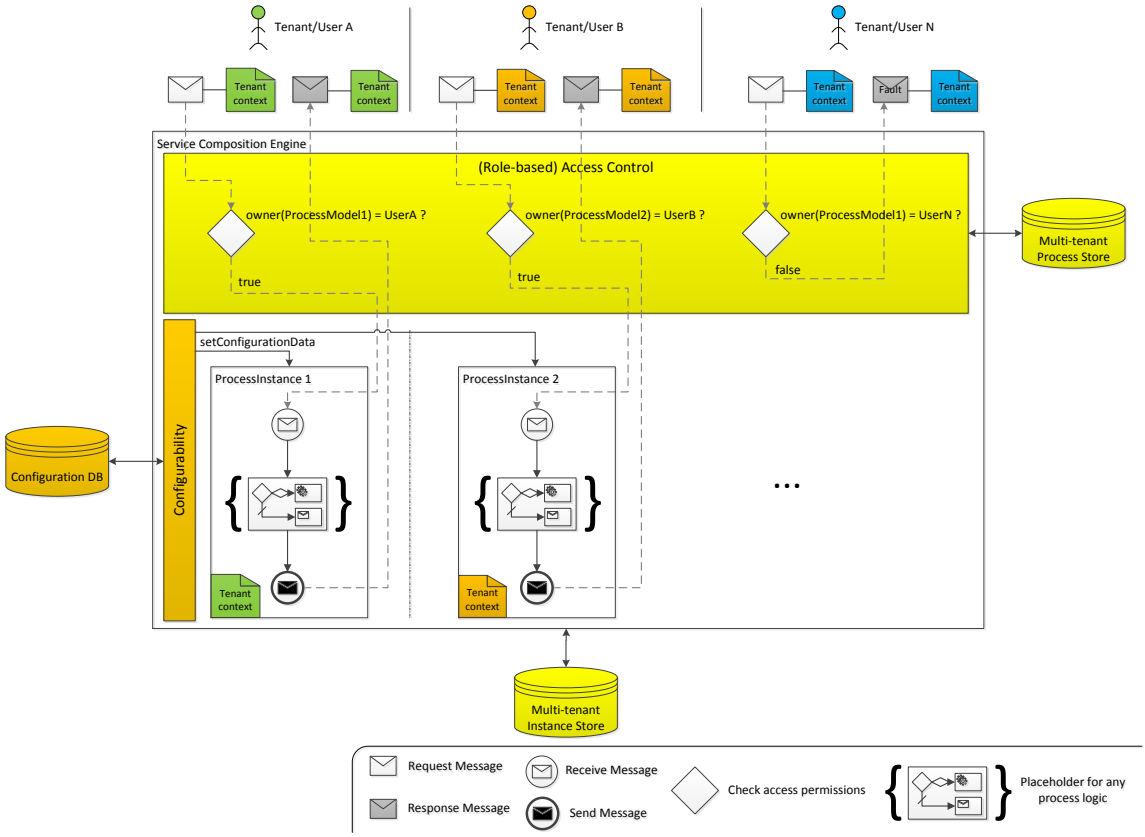

Figure 4.5: Process Instantiation with a multi-tenant SCE

new instance of Process B (*Instance B*) during the execution of the modeled Invoke activity by sending a request message to the service interface of Process B. Instance A is suspended until it receives a corresponding response of Instance B. After the execution of some process logic, Instance B sends a response message back to Instance A and is also suspended until it receives an additional message from Instance A. Right after that, Instance A executes some process logic and then sends another message to Instance B. Since the message should be routed to the same process instance as the first request message, a correlation context has to be attached to the message. This correlation context is used by the SCE to identify the correct instance of Process B (Instance B) to which the message should be forwarded. Instance B also attaches a correlation context to enable the routing of the response message to Instance A. As described in the section above, the tenant context will be forwarded in all message exchanges. The only adaption might be the use of the tenant context to realize the correlation of messages between instances by a SCE. But this would depend on the used SCE implementation.

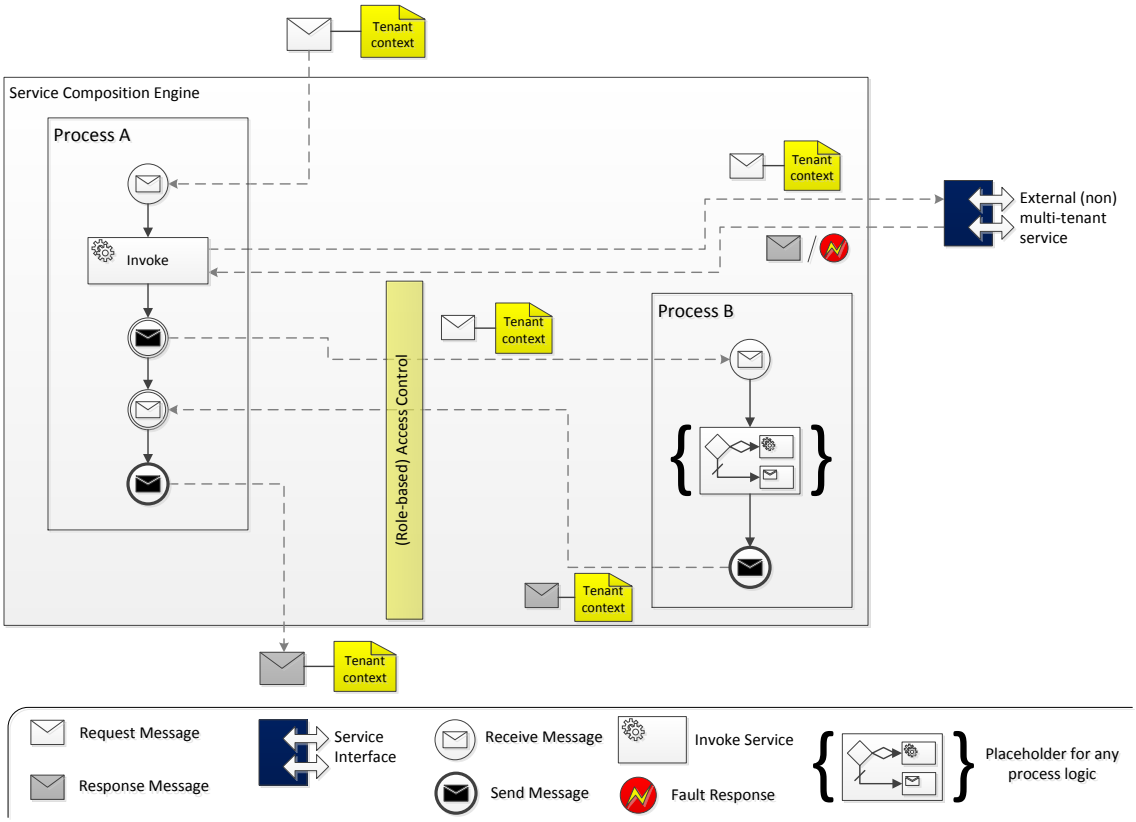

Figure 4.6: (External) Service invocation with a multi-tenant SCE

4.5 Collaboration Aspects of a Multi-tenant SCE

Especially for the SimTech project (see chapter 1.1.1), where a variety of scientists are working together to realize multi-scale, multi-physics and multi-domain simulations, collaboration is an important aspect. Simple multi-tenancy realizes the isolation of resources between tenants. To enable collaboration in a multi-tenant environment this isolation must be loosened in some areas. The two Figures 4.8 and 4.9 illustrate the differences between “collaborative” multi-tenancy on process model or process instance level. The main thing to enable collaboration between tenants and their users is that access permissions for process models and instances must be easily assignable across tenants’ boundaries. The possibility to dynamically define fine-grained access permissions during the runtime of the SCE and its process instances would make the collaboration more flexible but is outside of the scope of this thesis. Chapter 6 provides some initial ideas how to enable the dynamic assignment of fine-grained access permissions on a process model and process instance level.

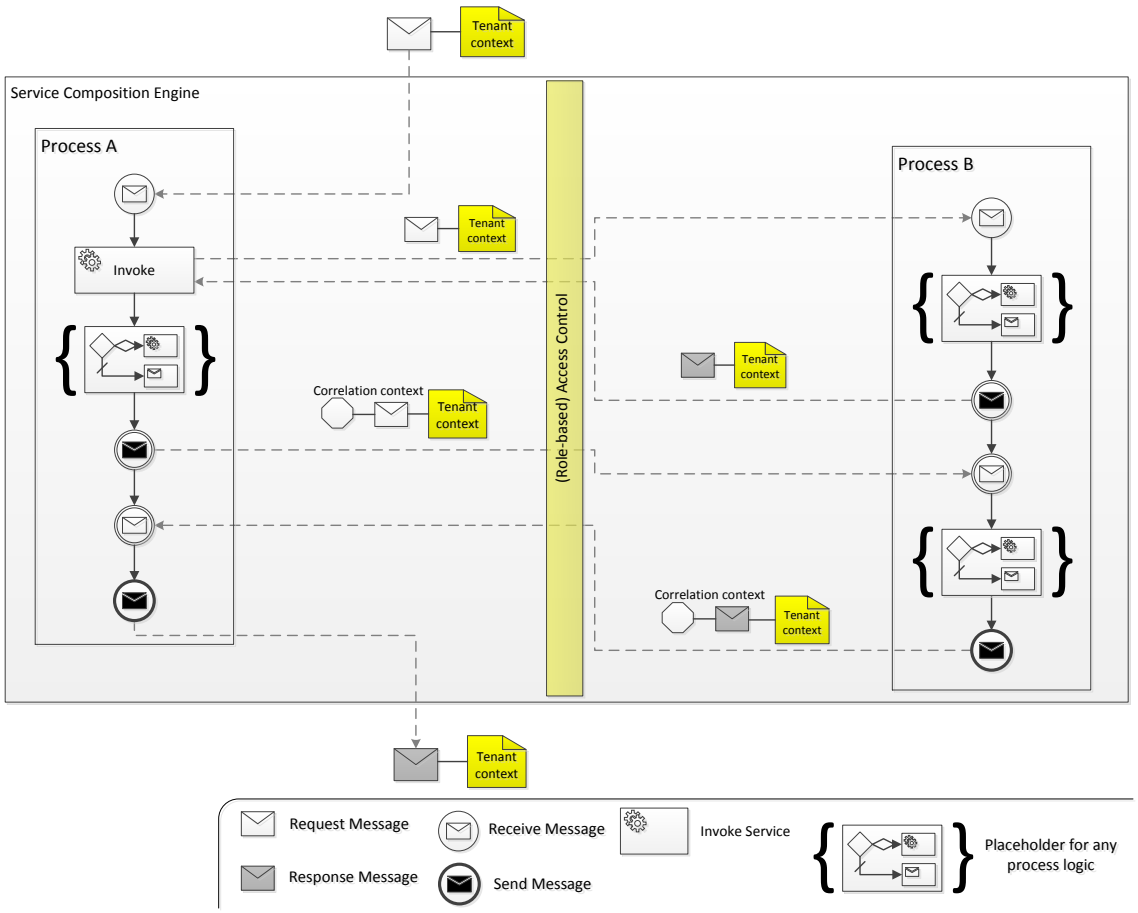

Figure 4.7: Correlation of process instances with a multi-tenant SCE

Deployment Styles to enable Collaboration

A first step to enable the collaborative work with process models is to provide a mechanism to share them across tenant boundaries. Therefore, a process model is deployed with a specific *deployment style* constant. The SCE uses this constant to select the level of equality which should be used for the authentication of incoming requests. For example, the strictest deployment style only authenticates requests with a tenant context fully equal to the one which is associated to the process model. In contrast to that, the loosest deployment style authenticates all incoming requests whether they specify a tenant context or not. In the following three basic deployment style constants are described which provide different access possibilities and influence the way the engine authenticates incoming requests.

Public Process models which are deployed under a *Public* deployment style do not specify any access permissions. This means in contrast to the description in Chapter 4.4.1, the corresponding Deployment Bundle is not associated with any tenant context. Any incoming request send to the Service Interface of a public process model is forwarded without any authentication. These process models can be used by any user of any tenant without any restrictions. The fact that a process model is public means not that the created instances of it are also accessible by any user. If a public process model is instantiated under tenant context, the instance is directly associated with the corresponding tenant context as described in Chapter 4.4.2. This secures all tenant-specific instances and their data from any unauthorized entities. If a public process model is instantiated without a specified tenant context, the instance is also public. In case of that, everyone is able to manage these instances over the Management Interfaces of the SCE. This deployment style enables for example the SCE provider to deploy a bunch of process models which can be used by all of its tenants and may provide some useful domain-specific functionality. It also provides backward compatibility because non multi-tenant aware applications or services are able to send requests to public deployed process models.

Tenant-private The *tenant-private* deployment style enables the deployment of process models which can be used by all users of one tenant. The process models are deployed as usual under tenant context. The SCE only forwards requests with a tenant context which references the same tenant as the target process model. For example, if the process model is deployed for tenant A only requests of users of tenant A are forwarded, any other requests are rejected. The referenced user of the tenant context is irrelevant because the users would not be compared by the engine to authenticate an incoming request. Requests without a tenant context are also rejected. As described for the public deployment style the created instances of a tenant-private model also belong to a single entity – identified by the tenant context of the initial request.

User-private Process models which are deployed under a *user-private* deployment style are only accessible by a single user. The process models are deployed as usual under tenant context. The SCE only forwards requests with a tenant context which references the same tenant and user as the target process model. Requests without a tenant context are also rejected. As described for the public deployment style the created instances of a user-private model also belong to a single entity – identified by the tenant context of the initial request. This is the default deployment style and if no style constant is specified during the deployment of a new process model, the engine behaves like for user-private models.

Table 4.1 provides an overview of the three deployment styles and the behavior they determine. Therefore for each deployment style all possible authentication scenarios are listed. This includes request messages without a tenant context, with a tenant context where only the tenantId matches or both the tenantId and the userId are equal. A “+” indicates that the request message is forwarded and a “-” that it will be rejected by the SCE. Chapter 5.3.6 describes in detail how the deployment styles influence the authentication process of request messages during the process instantiation.

Deployment Style	Messages with		
	no tenant context	a tenant context with matching	
		tenantId	tenantId&userId
Public	+	+	+
Tenant-private	-	+	+
User-private	-	-	+

Table 4.1: Overview of the three deployment styles

Collaborative Multi-tenancy

Figure 4.8 shows a simple collaborative multi-tenancy scenario which enables collaboration on a process model level. *Tenant 1* has deployed one tenant-private process model (*Process Model A*) which can be instantiated by all users of the tenant (Users a and b). In contrast to that, *User b* (blue user, see Fig. 4.8) has deployed a user-private process model (*Process Model B*) which can only be instantiated by *User b*. The two deployed process models are provided as services by the SCE and can be instantiated over a corresponding service call. The multi-tenant SCE has to ensure that only tenants and users which have access permissions are able to call the operations of the provided services. As an example, *User b* has started an instance of both process models (Process Instances $A_{2,b}$ and $B_{1,b}$) and *User a* (yellow user) is only able to start an instance of *Process Model A* (Process Instance $A_{1,a}$). All started process instances are under the control of one specific user during their runtime.

Figure 4.9 shows a complex collaborative multi-tenancy scenario which additionally enables collaboration on a process instance level. As a result, users (of different tenants) are able to work together on the same process model or process instance. In this scenario an additional tenant and user are introduced. *User c* (red user, see Fig. 4.9) belongs to Tenant 2. He is for example a professional in Business Process Management (BPM) and perhaps a scientist too. As a result *User c* is able to help other scientists if they have some problems to realize or run their simulations by using BPM and the workflow technology. *Tenant 2* provides a global *Process Model C*. The global process model is accessible by all (other) tenants and their users as described above. The three process models and their instances show a subset of a variety of possible collaboration scenarios. The three different scenarios are described from left to right. *Process Model A* illustrates a scenario where *User b* starts a new instance (Process Instance $A_{3,bc}$) which runs not as expected. The collaboration aspect enables *User b* to dynamically assign (fine-grained) access rights for instance $A_{3,bc}$ to get help from a BPM professional like *User c*. With corresponding access rights *User c* is now able to monitor the execution of the instance or analyze stored event data and solve the execution problems. Another scenario is shown by *Process Model B*. *User b* enables *User c* to instantiate *Process Model B* which was originally deployed as a user-private process model of *User b*. Process instance $C_{2,ab}$ shows another kind of collaboration. In this case *User a* and *User b* work together on the same instance. For example, instance $C_{2,ab}$ is a multi-domain simulation where one part is provided by *User a* and the other part comes from *User b*. Both users are able to monitor the

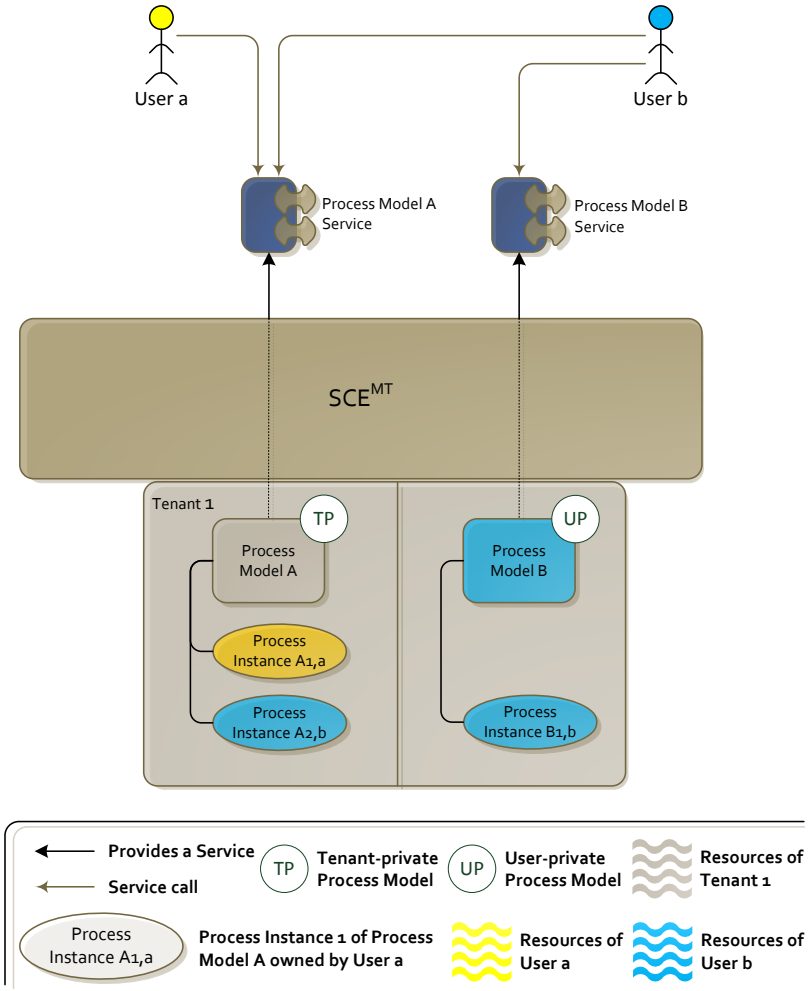

Figure 4.8: Example for a simple collaborative multi-tenancy scenario with a multi-tenant SCE

execution or influence the instance behavior based on the access permissions they have. The collaborative multi-tenancy on a process instance level is not realized in the context of this diploma thesis. Chapter 6 provides some initial ideas how collaboration on a process instance level can be realized.

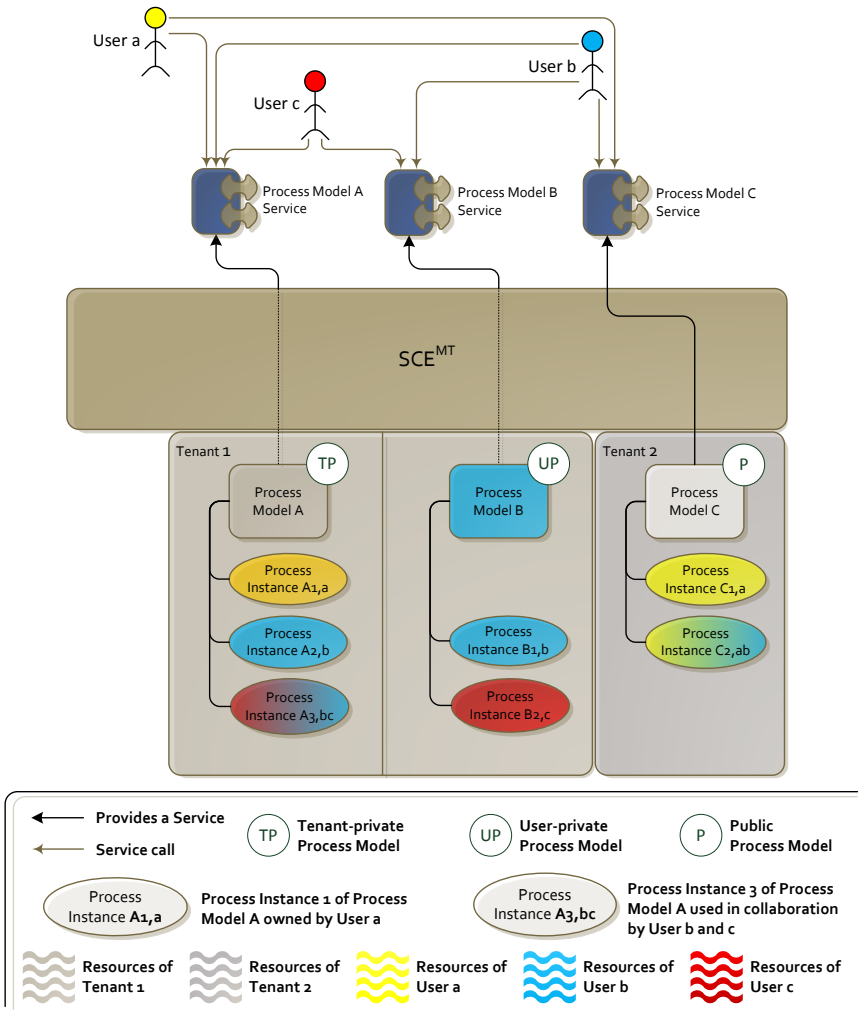

Figure 4.9: Example for a complex collaborative multi-tenancy scenario with a multi-tenant SCE

4.6 Multi-tenancy Requirements

Based on the descriptions and the introduced requirements in the previous chapters, this chapter provides a set of functional and non-functional requirements which should be fulfilled to realize a multi-tenant SCE and process models.

4.6.1 Functional Requirements

Tenant awareness A Service Composition Engine (SCE) must be able to identify multiple tenants and their associated resources (e.g. deployment bundles, process instances, messages, event data). For example, a tenant-based identification and an access control for tenants and their users must be supported.

Tenant-based deployment and configuration The SCE should provide tenant-based configuration options for both the engine and the process models. Furthermore, the SCE must enable the deployment of process models on a per-tenant basis.

Tenant-specific interfaces A set of customizable interfaces must be provided, which enable the tenant-based management of SCE resources, like process models, instances or configuration data. For that, both GUIs and Web services interfaces should be provided.

Shared registries Since the SCE solution will be integrated into an environment with other multi-tenant components demanding similar information, the SCE must work with a set of shared registries which contain data about tenants/users, services and configurations.

Backward compatibility The SCE solution should be able to work with applications and services which are not multi-tenant aware and conversely should be able to be used seamlessly and transparently by such applications and services.

Collaboration The SCE solution should support collaborative work with provided process models and process instances between different tenants and their users.

4.6.2 Non-functional Requirements

Tenant Isolation Tenants must be isolated on all layers of the SCE to prevent them from gaining access to resources (e.g. data or computing resources) of other tenants.

Reusability & extensibility The multi-tenancy enabling mechanisms and the underlying concepts should not be specific for one Service Composition Engine implementation or depend on specific technologies. The defined mechanisms, concepts and the realized components should therefore be extensible/adaptable and reusable in different Service Composition Engines.

Transparent integration A tenant/user should be able to use the SCE in the same way, whether it has been integrated into an ESB or not. For example, the deployment of new process models on the SCE should be always handled with a single call. Any additional operations should therefore be executed invisibly in the background.

Scalability To enable horizontal scalability (scale out) the SCE should be run in a stateless fashion. Any tenant specific data (e.g. process models or configuration data) should therefore be stored in a distributed store and only be buffered by the SCE. A load balancing mechanism is then able to use the distributed store to dynamically create new configured instances of the multi-tenant SCE, deploy the tenants process models and thus serve new incoming requests of the tenant's users in a highly flexible and transparent

manner. The horizontal scalability is out of the scope of this diploma thesis, but the underlying concept and its implementation is realized with scalability in mind.

4.7 Multi-tenant SCE Architectures

This section introduced two different architectures for the realization of a multi-tenant SCE. At the end of this chapter one of these approaches is selected, which is then further elaborated in Chapter 5 and prototypically implemented within the context of this thesis. The two different extended architectures are based on the general SCE architecture shown in Figure 4.1.

Figure 4.10 shows one possible high-level architecture to enable multi-tenancy. This architecture approach in the following is referenced as "Architecture A". It contains a new *Multi-tenancy Enablement Layer (MT-Layer)* which provides an intermediate layer between the Integration Layer (Service interfaces) and the Runtime Layer with the internal logic. The new layer contains a set of loosely coupled modules which provide the necessary functionality to enable multi-tenancy. Therefore, some of the SCE components must be extended or adapted, for example to enable the Configurability component of the MT-Layer to configure the SCE. In the following all new or changed components of the general SCE architecture – which are marked green in Figure 4.10 – are described. Since the descriptions are based on an abstract SCE architecture, the SCE components which have to be changed vary based on the used SCE implementation.

The Message Exchange Processor must be extended to enable tenant-aware communication. This means the tenant context of all incoming messages should be forwarded to the underlying layers. On the other side, each outgoing message has to be associated with the tenant context it belongs to. In addition to that, the Model and Instance Database should be extended to provide Data Isolation and data privacy for the tenant-specific data. Therefore one of the Data Isolation approaches introduces in Chapter 3.3.1 can be used. The Application Layer must be adapted to support tenant-based configurations of the SCE. To enable the correlation of tenant-aware message exchanges and provide Communication Isolation, the Correlator has to be extended. The management of tenants and their users is realized by an external Tenant Registry which should be connected to the MT-Layer. The external Configuration Registry provides the tenant-based configuration data to the SCE and is also connected to the MT-Layer. The management of the configuration data can be realized by an external application or through the extension of the SCE Management Interfaces and the Configurability module of the MT-Layer.

The MT-Layer itself contains a set of modules where each module complies to a different requirement to enable a multi-tenant SCE. The *Security* module contains all security related functionality and enables Communication Isolation, like the authentication of incoming messages to process models or the Management Interfaces. Furthermore, it authenticates all calls to the Management Interfaces and provides with that Administration Isolation. It may also provide some encryption and decryption functionality, for example to secure the stored process models from any direct access (e.g. by the SCE provider). The *Configurability* module handles the management of tenant-based configurations. For example, on process instantiation it

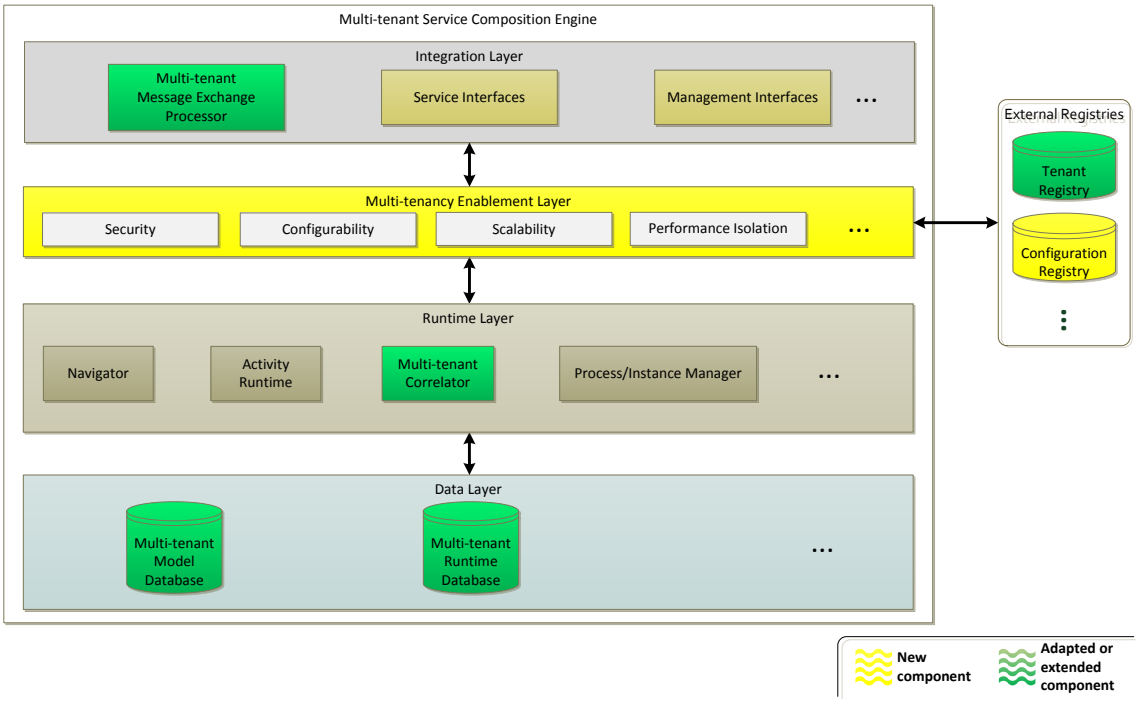

Figure 4.10: Possible architecture of a multi-tenant SCE with an integrated multi-tenancy enablement layer (Architecture A)

collects all necessary configuration data referenced by the calling tenant from the Configuration Registry and sets the data in the corresponding components of the SCE. The *Scalability* module provides functionality to enable scaling out the SCE and the *Performance Isolation* module handles the performance which the SCE provides to each of the tenants. These two modules are out of the scope of this diploma thesis and will not be further described in Chapter 5. The MT-Layer is designed extensible and can therefore be extended with additional modules to provide other multi-tenancy related functionality to the SCE in the future.

Now we have a look on some of the advantages and implications of Architecture A.

Advantages:

- The SCE is itself multi-tenant aware and does not need any external application or service.
- Any SCE component or internal data can be directly (re)used for the realization of the MT-Layer.

Implications:

- Integration of multi-tenancy enablement functionality into the SCE makes the result implementation-specific.

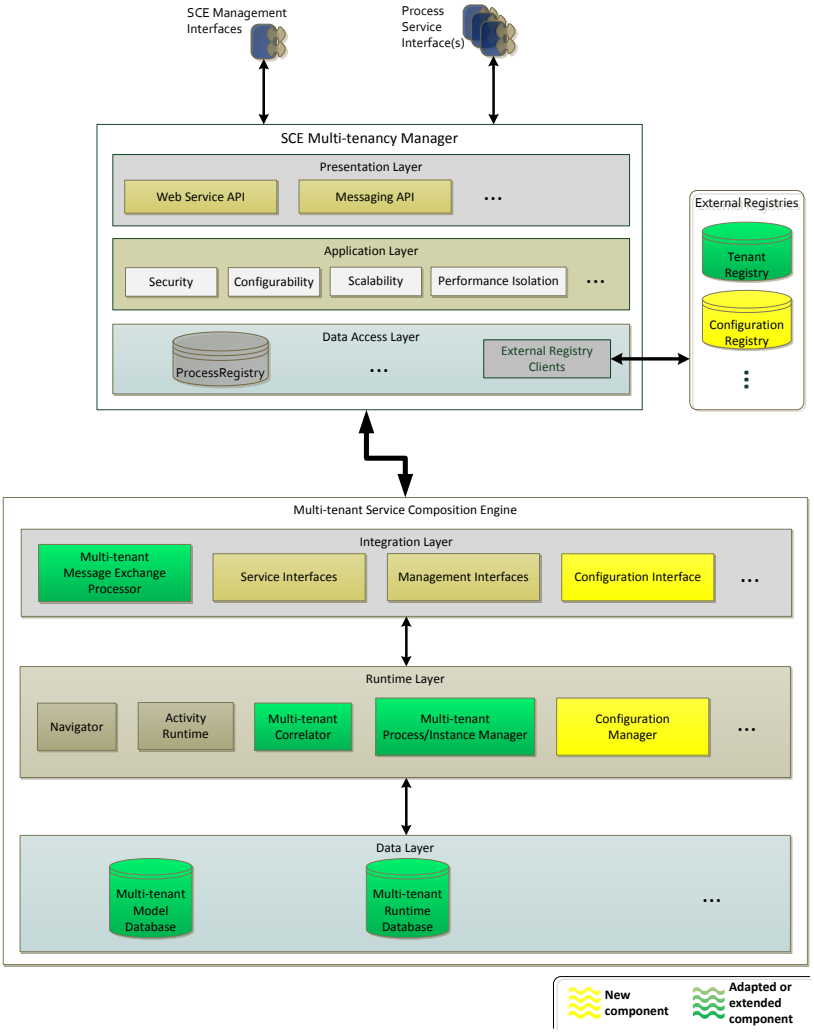

Figure 4.11: Possible architecture of a multi-tenant SCE with an outsourced multi-tenancy enablement layer (Architecture B)

- MT-Layer has to be (re-)designed and (re-)implemented for each SCE implementation.
- Realization of Scalability inside the SCE implementation is more complex as with an external component (e.g. a Load Balancer).
- Any extension or change of the MT-Layer is a change of the SCE implementation.

Figure 4.11 shows the second architecture, which in the following is referenced as "Architecture B". In contrast to Architecture A, as much as possible of the multi-tenancy functionality has been outsourced to a standalone application, the *SCE Multi-tenancy Manager*.

The *SCE Multi-tenancy Manager* (SCE-MT Manager) acts as the Multi-tenancy Enablement Layer introduced in Architecture A. The main difference is, that the multi-tenancy functionality is not bound to the SCE implementation. The SCE implementation must only be adapted to integrate it with the SCE-MT Manager. This provides a much easier approach because the SCE-MT Manager has to be realized only once and is then able to be used by any adapted SCE implementation. The SCE-MT Manager acts like a surrounding container for a set of maybe different SCE implementations. All requests send to one of the SCE interfaces are consumed by the SCE-MT Manager to authenticate and reroute them to one of the SCE interfaces. This is necessary because the SCE-MT Manager provides the multi-tenancy functionality and therefore needs to act as an intermediate layer between the outside and the connected SCE instances. For example, the SCE-MT Manager may provide the SCE interfaces over a Middleware Container like an ESB to the outside. The authentication of all incoming requests send to the Process Service Interfaces can then be handled over the ESB message routing mechanisms. On successful authentication the message is forwarded to the corresponding SCE service and in any other case the message is directly rejected. Messages send to the SCE Management Interfaces are just forwarded to the SCE because the authentication of management requests requires some SCE-internal data like which instances belong to a tenant. The SCE-MT Manager consists of a three layer architecture – Presentation Layer, Application Layer and Data Access Layer.

The Presentation Layer provides a Web Service and a Messaging Application Programming Interface (API) which provide interfaces to communicate with the manager and the interfaces of all connected SCE instances. For example, a SCE is able to register itself at the manager by sending a message to the Messaging API or calling a Web Service operation. The SCE-MT Manager also uses this APIs to communicate with all registered SCE instances, for example to send configuration data to a SCE if a new tenant is registered at the Tenant Registry.

The Data Access Layer provides a new *Process Registry* which is used to independently store the Deployment Bundles of all tenants. By default a Deployment Bundle and the contained process models are stored by a SCE instance. This might become a bottleneck in case of scalability and availability. If a new SCE instance is created to scale out, first of all the process models must be extracted from the SCE instance they are currently deployed, to be able to deploy them on the new SCE instance. A similar problem emerges if an SCE instance becomes unavailable as a result of a server outage. If the process models are not stored independently from the SCE instance by which they are executed, there is no chance to handle new incoming requests send to these models. Therefore, the Process Registry is realized to store process models independent of any SCE instance. This enables the dynamic deployment of one or more process models on a dynamic changeable set of SCE instances. If one SCE instance becomes unavailable or in terms of scalability a new SCE instance should be created, the SCE-MT Manager is able to deploy any process model stored in the Process Registry to the new SCE instance. The Data Access Layer also contains a set of database clients (*External Registry Clients*) which are used to access external registries, like the Tenant Registry or the Configuration Registry.

The Application Layer contains a set of modules where each module complies to a different requirement to provide multi-tenancy to all integrated SCE implementations. The *Security*

module contains all security related functionality and enables Communication Isolation, like the already described authentication of incoming messages. Furthermore, it realizes Administration Isolation by the authentication of any access to the Configuration Registry or the Process Registry. It may also provide some encryption and decryption functionality, for example to secure all stored process models in the Process Registry from any unintended direct access. The *Configurability* module handles the management of tenant-based configurations. For example, on process instantiation it collects all necessary configuration data referenced by the calling tenant from the Configuration Registry and sends the data to the *Configuration Interface* of the corresponding SCE instance. The *Scalability* module provides functionality to enable scaling out like the dynamic creation of new SCE instances described above and maybe can provide some kind of load balancing mechanism. The *Performance Isolation* module provides some functionality to analyze the performance of all registered SCE instances (e.g. based on auditing data) and maybe reject new incoming requests of a tenant if he exceeds his quotas. These two modules are out of the scope of this diploma thesis and will not be further described in Chapter 5. The SCE-MT Manager is designed extensible and can therefore be extended with additional modules to provide other multi-tenancy related functionality in the future.

As described for Architecture A the databases (Model and Instance Database), the Message Exchange Processor and the Correlator of the SCE implementation must be adapted to become multi-tenant aware. Furthermore, the Process/Instance Manager component has to be extended to realize the authentication of incoming calls to the Management Interfaces. This has to be done inside the SCE because the authentication depends on context-related data like the instance or process model for which a management operation is invoked. It may be also possible to outsource this to the SCE-MT Manager, but then all necessary data to handle the authentication has to be synchronized between the SCE instance and the SCE-MT Manager. Another problem is, that this approach might become very complex if scalability (multiple SCE instances managed by one SCE-MT Manager) and fine-grained access permissions come into play in the future. Since the advantages of this approach are much lower as its effort, the authentication of requests send to the Management Interfaces are handled by the SCE implementation itself. The new *Configuration Manager* component provides the functionality to configure the SCE and its process models on a tenant basis. It contains all implementation specific code and is provided over a new *Configuration Interface* to the outside.

Now we have a look on some of the advantages and implications of Architecture B.

Advantages:

- The multi-tenancy functionality is separated from the SCE implementation as much as possible. As a result the SCE-MT Manager can be used in combination with different SCE implementations.
- New multi-tenancy functionality can be integrated into the SCE-MT Manager without any changes to the SCE implementation.
- Parts of a maybe complex access control are outsourced to the SCE-MT Manager.
- Scalability can be efficiently realized apart from the SCE implementation.

- The powerful messaging and integration functionality of an ESB can be used to provide parts of the SCE-MT Manager functionality (discussed in the next section).
- SCE^{MT} without the SCE-MT Manager can also be used as "non multi-tenant" workflow engine (in some scenarios certainly useful, e.g. for testing or if only a few users of one department work with it).

Implications:

- The SCE is only multi-tenant aware in combination with the SCE-MT Manager.
- The design and implementation of the SCE-MT Manager application is more complex than just realizing everything inside the SCE implementation.
- Routing of messages through the SCE-MT Manager may reduce the overall performance.

Based on the advantages of Architecture B and the requirement to integrate the solution of this thesis in a multi-tenant aware ESB, this architecture is used as the basis for the realization of a multi-tenant SCE (SCE^{MT}) and is further referenced as SCE^{MT} Architecture. The next sections describe how SCE^{MT} can be integrated into an ESB and describe the advantages and implications of the different integration possibilities.

4.8 Integration of SCE^{MT} into an ESB

As a result of this thesis, a multi-tenant SCE should be realized and integrated in an existing multi-tenant ESB (ESB^{MT}). Therefore, this section provides two different ways how a SCE can be integrated into an ESB over a JBI container. One solution is to integrate applications over one or more JBI Binding Components. The other solution is to integrate an application as a JBI Service Engine into an ESB. This section describes these two possibilities and their advantages or implications based on the SCE^{MT} architecture defined in the previous section.

4.8.1 Integration of SCE^{MT} over Binding Components

As described in Chapter 2.6, (external hosted) services can be integrated with the ESB over Binding Components. A Binding Component (BC) enables the definition of consumer or provider endpoints for a specific protocol (e.g, HTTP or JMS). A consumer endpoint is used by the ESB to reroute incoming messages to an ESB-internal hosted service (Service Engine) or another provider endpoint which consumes the incoming messages. A provider endpoint is used by the ESB to send messages to the correct (external or internal) service interface which processes the messages. Figure 4.12 shows this solution to integrate an application or service to the ESB. To integrate the SCE over Binding Components a provider endpoint and a corresponding consumer endpoint is registered at the ESB for all service interfaces of the SCE. If a message is sent to one of these endpoints it is rerouted to the correct SCE service by the ESB as shown in Figure 4.12.

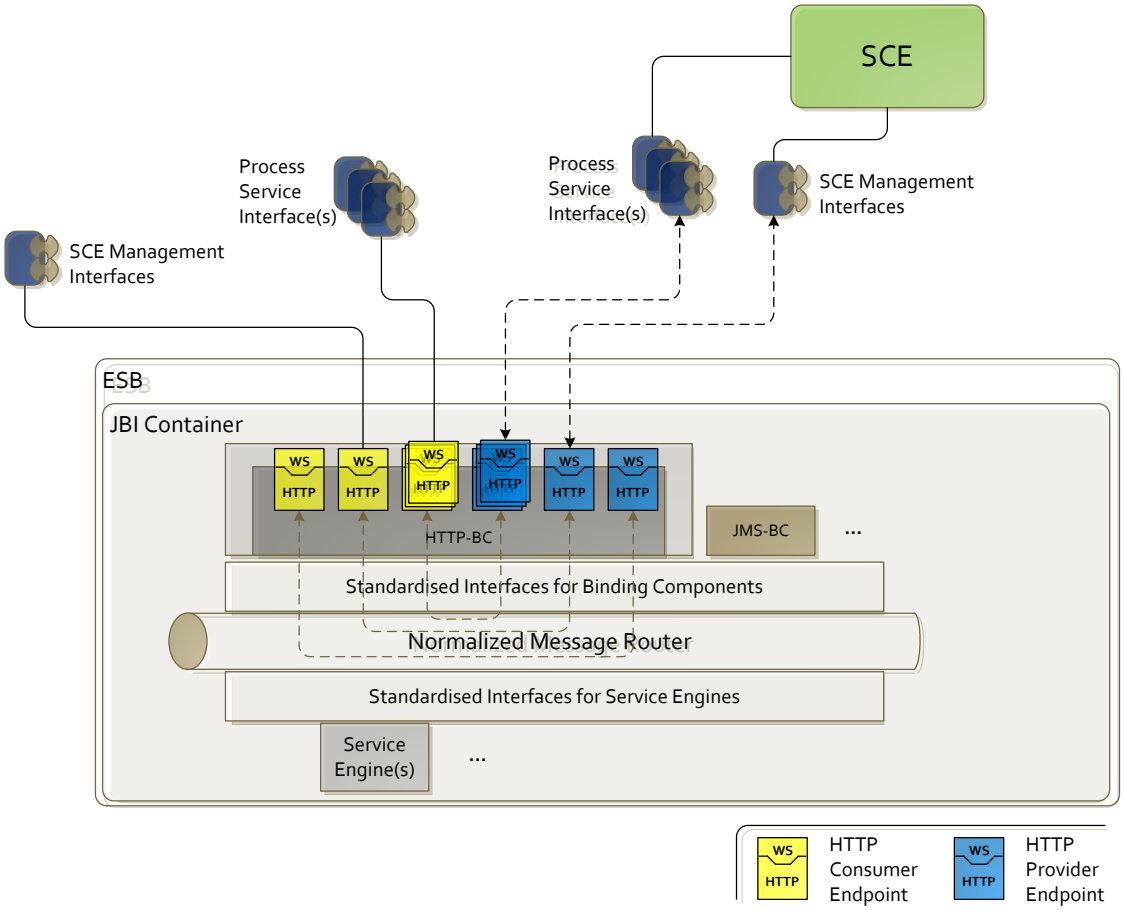

Figure 4.12: Integration of a SCE over Binding Components

Figure 4.13 shows how the integration of SCE^{MT} looks like. The SCE-MT Manager is integrated into the ESB as Service Engine to be able to use the ESB functionality inside the manager. All SCE Service Interfaces are provided by the SCE-MT Manager to the outside over a set of registered HTTP consumer endpoints. The SCE-MT Manager consumes all incoming messages send to these endpoints as described above. All messages send to the Process Service endpoints are authenticated and then forwarded to the correct HTTP provider endpoint. In case of that, all valid requests are send to the correct SCE Process Interfaces. Messages send to the Management Interfaces endpoints are just passed through the SCE-MT Manager which reroutes them to the correct SCE provider endpoint without any authentication.

The advantages of integrating SCE^{MT} over Binding Components are

- A SCE can be integrated over its existing interfaces into the ESB without any changes to the underlying implementation.

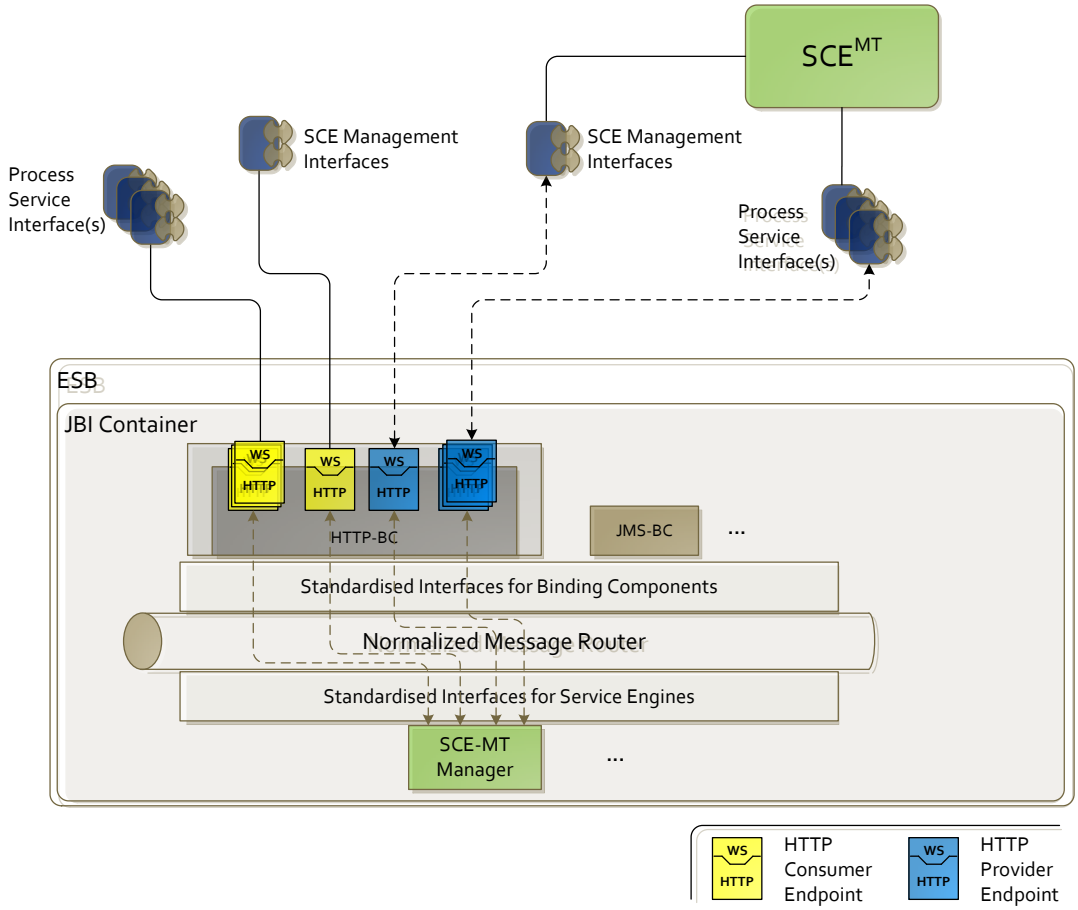

Figure 4.13: Integration of SCE^{MT} over Binding Components

- Message Authentication and Routing can be realized most efficient with the ESB.
- Parts of the SCE-MT Manager functionality can be implemented by using the ESB, e.g. using a set of rule-based message routings or a Context-based Message Router to realize message authentication.

The integration approach has the following implications:

- Messaging overhead: Every message that is forwarded to the SCE has to be normalized and de-normalized again.
- Management overhead: For each of the Process Services provided by the SCE, a new consumer and provider endpoint must be registered at the ESB by the SCE-MT Manager.
- Realization of the access control over the ESB allows unauthorized (direct) calls to the external provided SCE Service Interfaces.

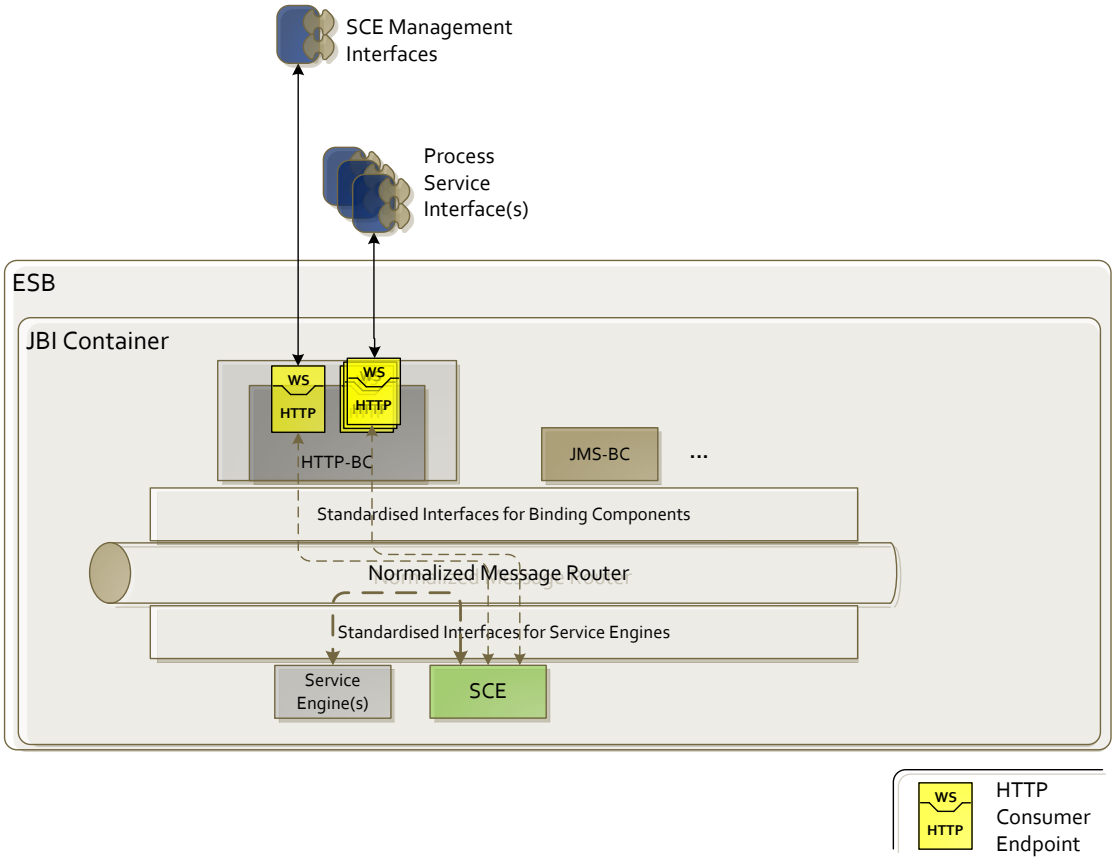

Figure 4.14: Integration of a SCE as Service Engine

4.8.2 Integration of SCE^{MT} as Service Engine

The second approach is to integrate the SCE as a Service Engine (SE) into the ESB. This provides some advantages as described in Chapter 2.6. For example, that the SCE can communicate with other SEs directly over the Normalized Message Router without any message (de-)normalization as shown in Figure 4.14. The Process Service and the SCE Management Interfaces are provided over the JBI HTTP BC to the outside. For each of the services a HTTP Consumer Endpoint is deployed to the BC which routes the messages over the NMR to the SCE. This contains the normalization of all protocol-specific request messages into the NMF to route them over the NMR. As well as the denormalization of all response messages provided in the NMF by the NMR back into protocol-specific response messages. As a result that the SCE directly communicates over NMs, it can exchange messages with all components plugged to the NMR without the need of (de-)normalizing those messages.

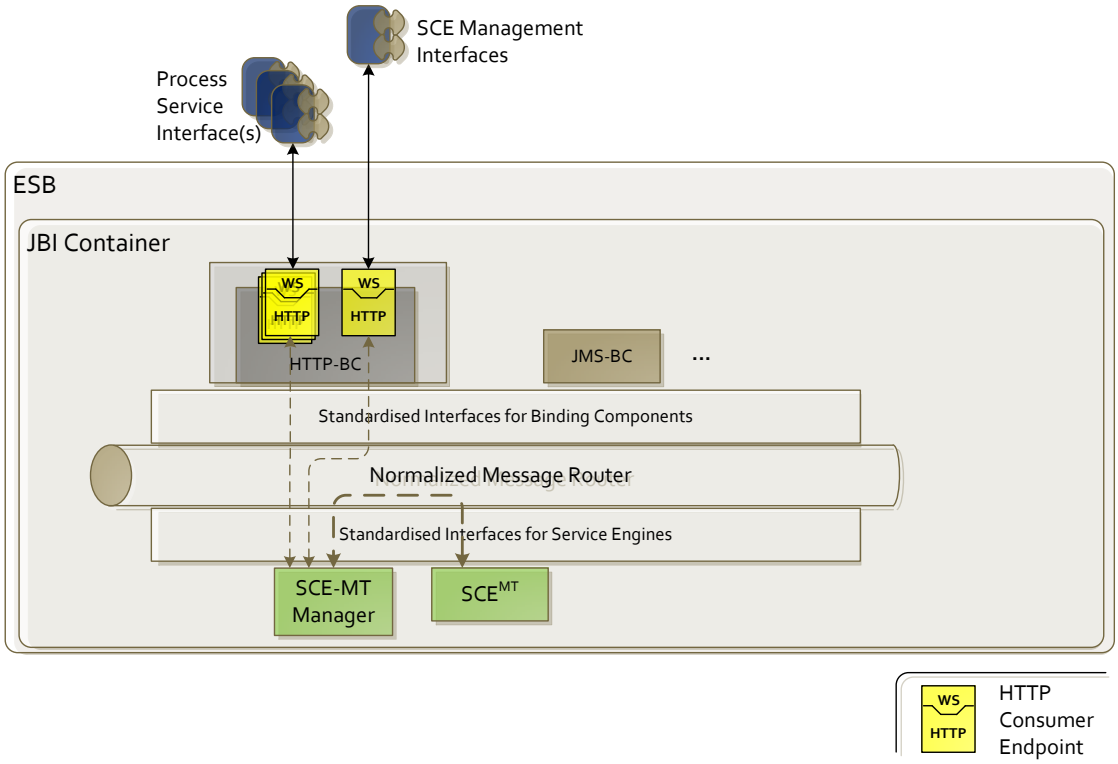

Figure 4.15: Integration of SCE^{MT} as Service Engine

Figure 4.15 shows how the integration of SCE^{MT} looks like. The SCE-MT Manager is again integrated as Service Engine. All SCE Service Interfaces are provided by the SCE-MT Manager to the outside over a set of registered HTTP consumer endpoints. The SCE-MT Manager consumes all incoming messages send to these endpoints. All messages send to the Process Service endpoints are authenticated and then forwarded over the Normalized Message Router (NMR) to the correct SCE Service Interface. In case of that, all valid requests are send to the correct SCE Process Interfaces. Messages send to the Management Interfaces endpoints are just passed through the SCE-MT Manager which reroutes them over the NMR to the correct SCE Management Interface without any authentication.

Advantages:

- The SCE is available, if the ESB is available.
- Message Authentication and Routing can be realized most efficient with the ESB.
- Existing components and functionality of the ESB can be (re)used and/or extended.

- Parts of the SCE-MT Manager functionality can be implemented in a flexible manner by using the ESB, e.g. registering a set of rule-based message routings or define a Context-based Message Router to realize message authentication.
- Messaging infrastructure of the ESB can be used to realize an asynchronous and reliable communication between one SCE-MT Manager and a set of SCE^{MT} instances.
- SCE-MT Manager and all connected SCE^{MT} instances directly communicate over Normalized Messages, no normalization and de-normalization is required.

Implications:

- Setup and administration of an ESB makes infrastructure a bit more complicated.
- To integrate the SCE as a Service Engine into the ESB the underlying implementation maybe has to be changed.

5 Implementation

This chapter describes how the specified SCE^{MT} architecture can be realized with the *SWfMS* and how the resulting components will be integrated with ESB^{MT} and JBIMulti2. Furthermore, the most important procedures will be specified in detail, like the deployment of process models or the registration of configuration data. First of all the overall architecture of the whole system is introduced as an entry point for the detailed descriptions of the components and their collaboration.

5.1 Overall Architecture of the Realization Approach

Figure 5.1 shows the high-level architecture of all realized, extended and integrated components to provide a multi-tenant SCE as a JBI Service Engine. These components are the SCE-MT Manager, a multi-tenancy enabled SCE implementation (SWfMS^{MT}), the ESB^{MT}, the JBIMulti2 application and a set of databases.

Muhler realizes the JBIMulti2 application which provides an administration and management interface with authentication and authorization for ESB^{MT} [Muh12]. Since the SCE-MT Manager also needs a secure administration and management interface, for example to register SCE configuration data, the JBIMulti2 application is therefore reused and extended in the scope of this diploma thesis. As a result the JBIMulti2 application additionally communicates with the SCE-MT Manager and not only with the *JMSManagementService* as in the original setup. The set of registries is also extended with a new *EventRegistry*. This registry is used to store all event messages emitted by any SWfMS instance in a tenant-isolated manner. The event data is stored in a separate database because the information provided by the events could be used in various ways in the future. For example, to analyze the resource consumption or overall performance of the execution of a process model as a basis for the realization of Performance Isolation (see Chapter 3.3.4). Another possible use of the event data could be some kind of Complex Event Processing, like for the realization of Business Activity Monitoring [EB09]. The existing *ConfigurationRegistry* and *ServiceRegistry* are extended to store some new SCE and process model related data. The registries are also used to integrate the SCE-MT Manager with the JBIMulti2 application without having any duplicated data or the need to synchronize data between two separated sets of databases. This is important because the SCE-MT Manager writes some data to the registries which are used by JBIMulti2 to respond to user requests and vice versa the SCE-MT Manager needs access to the data which is inserted to the registries by JBIMulti2. For example, the SCE-MT Manager needs access to SCE configuration data of a tenant which was registered over JBIMulti2. JBIMulti2 needs therefore access to the event data stored by SCE-MT Manager to provide users a list

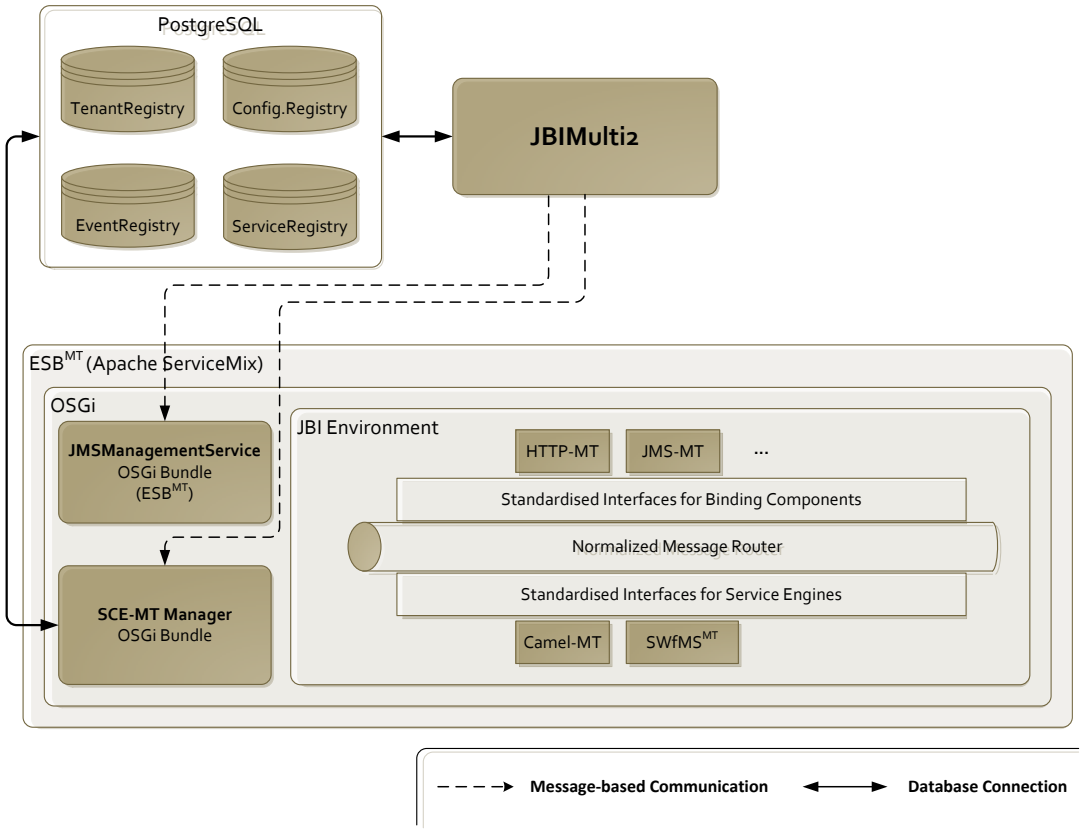

Figure 5.1: Overall architecture of ESB^{MT}, JBIMulti2 and the SCE^{MT} realization

of their running process instances over its Web Service API. As a result, JBIMulti2 and the SCE-MT Manager write and read data to and from the shared registries without the need to send the data to each other. The only required communication between JBIMulti2 and the SCE-MT Manager consists of sending status updates which are just simple event messages to inform the SCE-MT Manager that something has changed in the registries. For example if a new tenant is registered at JBIMulti2, the SCE-MT Manager has to create a new topic for the tenant's event data and store the endpoint of the topic in the ConfigurationRegistry. The new SCE administration functionalities and the status forwarding of JBIMulti2 are described in detail in Section 5.3.3. The new or extended database schemas of the different registries are described in detail in Section 5.2. The extended architecture of the JBIMulti2 application is described in Section 5.7.

The SCE-MT Manager is realized as an OSGi bundle and is installed to the underlying OSGi platform on which Apache ServiceMix is built on. The SCE-MT Manager is realized with scalability in mind and therefore provides a management layer for a set of SCE instances. Therefore, a tenant does not need to know on which SCE instance(s) its process models are deployed or on which SCE instance the corresponding process instances are executed. Also the

messaging infrastructure of all SWfMS^{MT} instances is managed which are registered to the SCE-MT Manager. The messaging infrastructure contains an event queue and a management queue. The event queue is required to securely route the event messages of the execution of a process instance to the correct tenant. The management queue is used to receive engine management messages like for the registration of process breakpoints. As already mentioned the SCE-MT Manager cooperates therefore with JBIMulti2 and also uses the powerful message routing functionality of the ESB by the installation of a set of message routes. Furthermore, the SCE-MT Manager is responsible for the tenant-isolated storage of all event messages in the new EventRegistry. The architecture of the SCE-MT Manager realized in the scope of this diploma thesis is described in Section 5.6. The interaction between the SCE-MT Manager, ESB^{MT} and SWfMS^{MT} is described in Section 5.3.

As mentioned previously, the SWfMS^{MT} workflow engine is integrated into the JBI environment as a Service Engine. SWfMS is therefore adapted based on the SCE^{MT} Architecture described in Chapter 4.7. The resulting architecture which will be implemented in the scope of this thesis is described in detail in Section 5.4. The realized SWfMS^{MT} implementation will initially provide two configuration options which are described in Section 5.5.

5.2 Database Schemas

This section provides the database schemas of the shared registries used by JBIMulti2 and the SCE-MT Manager. The schema of the TenantRegistry is not adapted and therefore will not be explained in this thesis. Interested readers can find it in the diploma thesis of Muhler [Muh12]. Furthermore, only the adaptations and new components of the schemas introduced by Muhler are described in this section. Muhler uses the *(Min, Max) Notation* defined by Abrial (see [Abr74]) to model cardinality constraints for relations between entities. A (min, max) constraint defines how often an entity participates in a relation and not the cardinality of the entities. Therefore, the constraints must be interchanged while reading a relation. For example, the relation [ProcessModel]-(0,n)-**has**-(1,1)-[ProcessInstance] defines that a process model has zero to infinitely many process instances and a process instance belongs to exactly one process model.

Figure 5.2 shows the extended database schema of the ServiceRegistry. The *ServiceAssembly* entity type is extended with a new *type* and *deploymentStyle* attribute which are marked yellow in the figure. The type is used to be able to identify *Process Service Assemblies* (PSA) as a special kind of *Service Assemblies* (SA). The deploymentStyle attribute holds one of the deployment style constants (Public, Tenant-private or User-private) defined in Chapter 4.5. The JBIMulti2 application is extended to set these two attributes if a PSA is deployed over a new *deployProcessServiceAssembly() Web Service* (WS) operation. With the help of the type information, the SCE-MT Manager is able to identify all SAs which contain process models. The deployment style constant is used during the authentication of requests send to a process model HTTP endpoint (see Sect. 5.3.6).

Figure 5.3 shows the extended database schema of the ConfigurationRegistry where all new elements are marked yellow. The *SCE* entity type represents a registered SCE instance and is

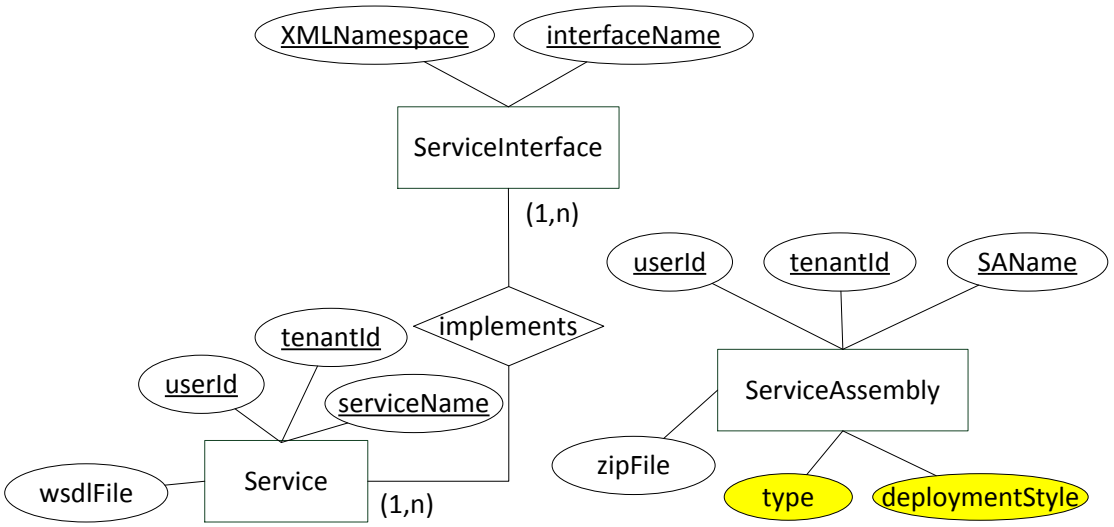

Figure 5.2: Extended entity-relationship diagram of the Service Registry using (Min,Max) Notation, cf. [Muh12]

defined as a specialization of the *JBIComponent* entity type. This is required because the SCE-MT Manager has to store additional information for each registered SCE instance. The key attribute *sceInstanceId* is used to uniquely identify a SCE instance and to associate data which belongs to this instance. For example, this makes it possible to store by which SCE instance a specific process instance is executed. Furthermore, the sceInstanceId is used as a credential for the authentication of connections and message exchanges between a SCE instance and the SCE-MT Manager (see Section 5.3.2). The *managementServiceEndpoint* attribute holds the HTTP address of the Management WS of a SCE instance. This HTTP address is used by the SCE-MT Manager to install a corresponding message route at the ESB which is described in Section 5.3.8. The attributes *eventQueueEndpoint* and *managementQueueEndpoint* hold the JMS endpoint addresses of the corresponding SWfMS Message Queues. These two addresses are also used by the SCE-MT Manager to install message routes at the ESB which is described in detail in Section 5.3.1). The *status* attribute holds the current status of the SCE instance, like if it is available to process requests or is shutdown or maybe in a maintenance mode. The *SCEConfiguration* entity type represents the tenant-based configuration of one or more SCE instances (see Fig. 5.3). To isolate the configuration data of different tenants, the SCEConfiguration entity type has a *tenantId* key attribute. The *configurationId* key attribute is used to uniquely identify a configuration and can be used in the future to select one configuration out of a set of maybe multiple interchangeable configuration sets specified by a tenant. A SCEConfiguration entity contains a collection of *SCEConfigurationEntry* entities. These are defined as weak entities because if a SCEConfiguration is deleted all contained SCEConfigurationEntry entities should be also deleted from the database. To provide extensibility for future versions which support more or other SCE configuration options, the *ExtensionBundle* entity type is defined as a

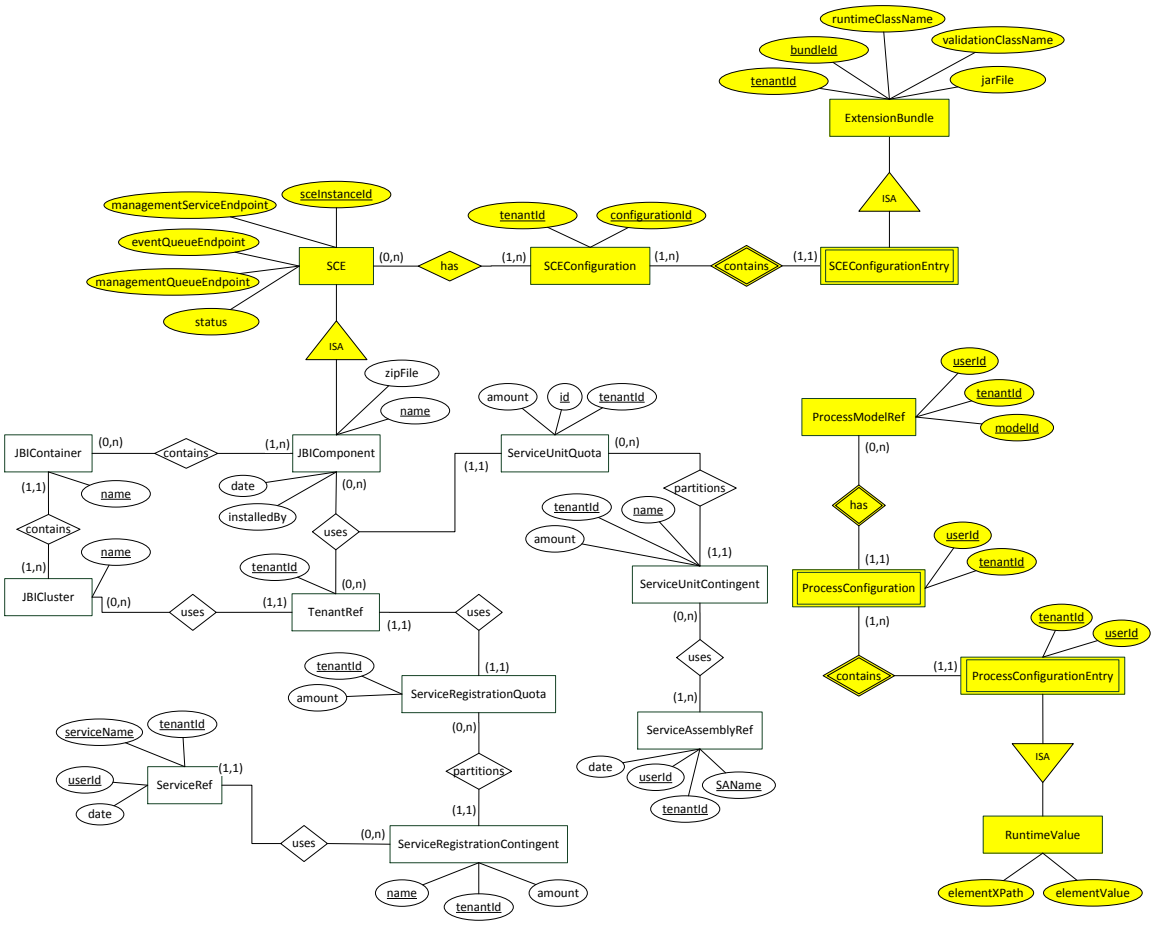

Figure 5.3: Extended entity-relationship diagram of the Configuration Registry using (Min,Max) Notation, cf. [Muh12]

specialization of a SCEConfigurationEntry. The ExtensionBundle entity type provides the tenant-aware registration of Extension Bundles (see Sect. 5.5). The tenantId key attribute is used again to specify the tenant to which the configuration entry belongs. The *bundleId* key attribute is an unique identifier for a ExtensionBundle entity type. The *runtimeClassName*, *validationClassName* and *jarFile* attribute contain the required data which is used by SWfMS^{MT} for the installation of the bundle to the engine (see Sect. 5.5).

The *ProcessModelRef* entity type is used to reference a *ProcessModel* entity stored in the EventRegistry shown in Figure 5.4. The key attributes *tenantId* and *userId* enable the tenant-isolated storage and the *modelId* key attribute uniquely identifies a process model. The storage of process model configuration data is handled the same way as for SCE configuration data. The *ProcessConfiguration* entity type has an additional *userId* key attribute because also users and not only tenants are able to specify configuration data for a process model. It contains a collection of *ProcessConfigurationEntry* entities which again have a tenantId and userId

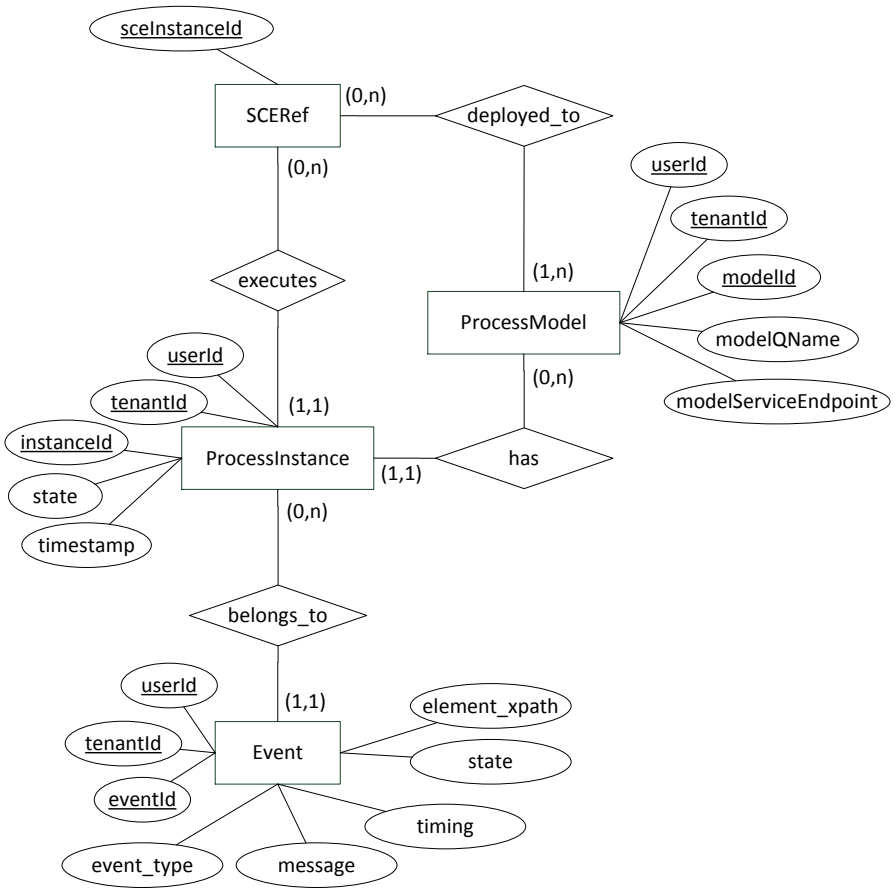

Figure 5.4: New entity-relationship diagram of the Event Registry using (Min,Max) Notation

attribute to enable data isolation and are realized in an extensible fashion by the use of specialization. The *RuntimeValue* entity type is a specialization of the ProcessConfigurationEntry and enables the storage of process model configuration data. The *elementXPath* attribute holds the XPath expression of the target element of a process model. For example, if a runtime value for the third variable of the process model should be registered, the XPath expression of the target element looks like this: `/process/variables[1]/variable[3]`. The *elementValue* attribute holds the data which should be assigned to the target element in terms of configuration. The data can be a simple literal value (e.g. a constant) or any structured XML data (e.g. the endpoint data of a partner link).

Figure 5.4 shows the database schema of the new EventRegistry. To enable Data Isolation between tenants and their users the Shared Schemas approach described by Chong et al. (see Chapt. 3.3.1) is realized by adding a *tenantId* and *userId* key attribute to all defined entity types. The *SCERef* entity type is used to reference a SCE entity contained in the ConfigurationRegistry. The unique sceInstanceId attribute is used therefore to reference a SCE

instance. If a new process model is deployed to a SCE instance, a new *ProcessModel* entity is created and associated with the SCE instance over a SCERef entity. The ProcessModel entity type has a globally unique *modelId* key attribute which is used to identify a process model. The *modelQName* attribute holds the qualified name of the process model and the *modelServiceEndpoint* attribute specifies the endpoint address of the process model service (e.g. the HTTP address of the process model WS). The *ProcessInstance* entity type represents a created process instance of a process model. Therefore it is associated with its process model so that the process instance knows its model and vice versa a process model knows all of its instances. The ProcessInstance entity is also associated to an SCERef entity to identify the SCE instance on which a process instance is executed. This is important to enable scalability in the future if process instances of one process model are executed on more than one SCE instance. Then the SCE-MT Manager and the ESB have to know which SCE instance executes which process instance to be able to route management requests to the correct SCE instance. A *ProcessInstance* entity type has a unique *instanceId* key attribute, a *state* and a *timestamp* attribute. The state attribute holds the current state of the process instance, for example "executing", "completed" or "suspended". The timestamp attribute contains a timestamp when the process instance was started. Each ProcessInstance entity is associated with a collection of *Event* entities. The Event entity type has a *eventId* key attribute to uniquely identify an event. The *event-type* attribute holds the type of the event, for example "Activity_Completed" or "Loop_Condition_Evaluated". The *message* attribute is stored as a Binary Large Object (BLOB) and contains a serialized copy of the event message to which the event belongs. The *timing* attribute holds a timestamp when the event was emitted by a SCE instance. The *element_xpath* attribute references the element of a process model which is the source of the event. The *state* attribute holds the current runtime state of the element which is referenced by the event. The runtime states are based on the BPEL 2.0 Event Model defined by Steinmetz. Therefore additional information can be found in his diploma thesis [Ste08].

5.3 Interaction of JBIMulti2, ESB^{MT}, SCE-MT Manager and SWfMS^{MT}

This section provides some detailed descriptions for the most important interaction scenarios between all introduced components, like the deployment of new process models, the authentication of the instantiation of a process model or the routing of event messages by the ESB.

5.3.1 Overall Messaging Infrastructure

Figure 5.5 shows the overall messaging infrastructure of ESB^{MT} with installed SCE-MT Manager. Sending management messages from a JBIMulti2 application to one or more subscribed JMSManagementServices is realized over the *Management Messages.topic* as described in [Muh12, Sáe13]. If the processing of a management message in a JMSManagementService causes an exception, the JMSManagementService sends a corresponding fault message to the

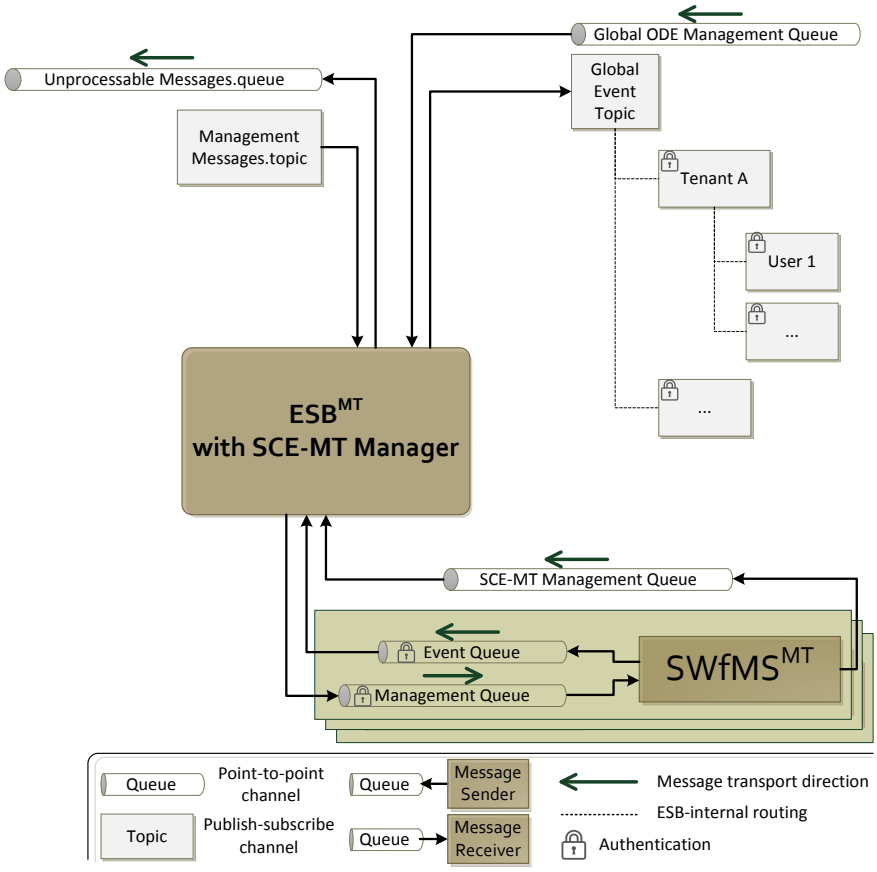

Figure 5.5: Messaging infrastructure of ESB^{MT} with installed SCE-MT Manager

Unprocessable Message.queue. To send status messages to a SCE-MT Manager, JBIMulti2 is connected to the *SCE-MT Management Queue*. The forwarding of this status messages from JBIMulti2 to a SCE-MT Manager is described in Section 5.3.3. These three channels provide the communication between JBIMulti2 and the components installed to the ESB (JMSManagementService and SCE-MT Manager).

The *Global ODE Management Queue* is provided by the SCE-MT Manager as a static endpoint to send ODE management messages to a set of SWfMS instances without the requirement to know the endpoints of their Management Queues. The SCE-MT Manager uses the powerful message routing functionality of the ESB to reroute all messages send to this queue to the correct set of SWfMS^{MT} *Management Queues*. This is an important feature in terms of scalability because a process model can be deployed to a set of SCE instances and the dynamic routing of the messages realizes the distribution of one management message to all required Management Queues. Section 5.3.8 provides some further descriptions how this is realized.

The *Global Event Topic* and all its sub-topics are also used to decouple external tools (e.g. monitoring software) from the SWfMS^{MT} *Event Queue*s and will provide a more flexible approach. The SCE-MT Manager again installs corresponding message routes to the ESB which realize the dynamic routing of event messages from a Event Queue to one or maybe a collection of Tenant or User Event Topics like shown in Figure 5.5. Section 5.3.7 provides some more details how the routing of event messages is realized. The topic hierarchy and the Global ODE Management Queue provide the communication between a set of SWfMS^{MT} instances and any external tools.

The communication between the SCE-MT Manager and a SWfMS^{MT} instance is realized over the *SCE-MT Management Queue*, the Global ODE Management Queue and the SWfMS Management Queue. A SWfMS^{MT} instance registers itself at the SCE-MT Manager by sending a corresponding message to the SCE-MT Manager Queue (see Sect. 5.3.2). The SCE-MT Manager uses the existing SWfMS Management Queue to respond to a registration request message. After a SWfMS^{MT} instance is registered correctly, the SCE-MT Manager can use the Global ODE Management Queue to send management message to the correct set of SWfMS^{MT} instance. For example to register configuration data for a process model or the SCE. This is further described in Section 5.3.4.

5.3.2 Registration of SWfMS^{MT} instances at SCE-MT Manager

Figure 5.6 shows the registration process of SWfMS^{MT}. This process is initalized when a new SWfMS^{MT} instance is installed to the ESB. Apache ServiceMix provides some different possibilities to install a new component. Since in the scope of this thesis only a single SWfMS^{MT} instance is used, it is installed over the ServiceMix console by hand. In a future version the installation can be realized by using an internal ServiceMix API, similar to the JBIMulti2 approach which uses the JMSManagementService to deploy SAs to the ESB over the `org.apache.servicemix.jbi.deployer.AdminCommandsService` class. The SCE-MT Manager could use the ServiceMix JMX interface or the class `org.apache.karaf.features.FeaturesService` of the OSGi implementation Apache Karaf[1], on which ServiceMix is build on, to install SWfMS^{MT}. This enables the installation of SWfMS^{MT} directly from a corresponding Maven repository without the need to store or transfer the installation data. The SCE-MT Manager has only to register the Maven repository which contains the SWfMS^{MT} installation files and can then initiate the installation of all necessary features to ServiceMix.

In case the installation of SWfMS^{MT} is executed by hand or by the SCE-MT Manager in the future, SWfMS^{MT} registers itself at the SCE-MT Manager to avoid any additional administration effort. How the registration is realized is shown in Figure 5.6. First of all a new SWfMS^{MT} instance is installed. This new instance automatically connects to the SCE-MT Management Queue and sends a corresponding registration message to it (see Fig. 5.6, Point 1). If the SWfMS^{MT} instance was already registered at the SCE-MT Manager and is just started again after a shutdown, the SWfMS^{MT} instance sends only an event message to the

[1]The Apache Software Foundation, Apache Karaf: http://karaf.apache.org/

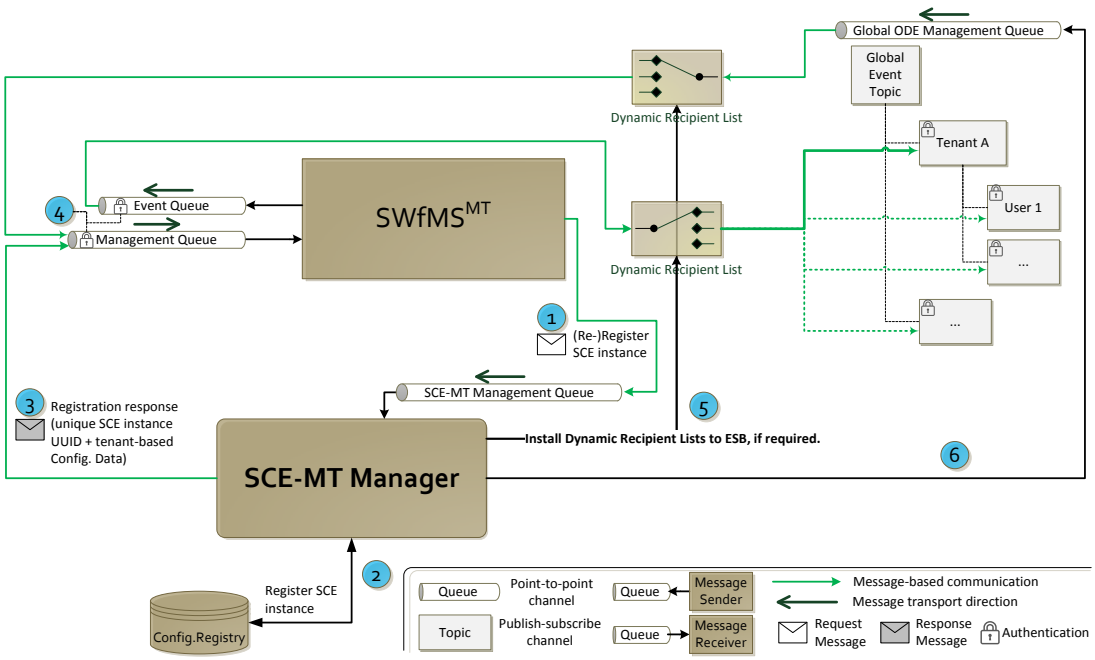

Figure 5.6: Registration of a new SWfMS^{MT} instance at the SCE-MT Manager

queue. This event message signals the SCE-MT Manager that the engine instance is available again and can be used to serve new requests. If a registration message or an event message should be send by SWfMS^{MT} is evaluated by checking if an sceInstanceId is assigned already to the engine instance. The registration message contains all required data of the SCE instance, like the endpoint addresses of its queues and services. The SCE-MT Manager consumes the registration message, generates a *Universally Unique Identifier* (UUID) for the SCE instance and registers the data of the new SCE instance with the UUID at the ConfigurationRegistry (see Fig. 5.6, Point 2). After the SCE instance is registered, the SCE-MT Manager sends a response message to the Management Queue of the SWfMS^{MT} (Point 3). This message contains at least the sceInstanceId and an optional set of SCE configuration data in the case that the SWfMS^{MT} instance is just re-registered. Subsequently, SWfMS^{MT} consumes this response message, persists the sceInstanceId into its internal database and uses the private sceInstanceId as a credential to secure its messaging channels (see Fig. 5.6, Point 4). In the context of this thesis the messaging channels are secured with simple username/password security provided by the Apache ActiveMQ[2] messaging server which hosts all required messaging channels. The sceInstanceId is therefore used as the password and the username is just a constant value like "ode". At the end of the registration process the SCE-MT Manager installs a message route for the Event Queue and the Management Queue. The event messages published over the

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

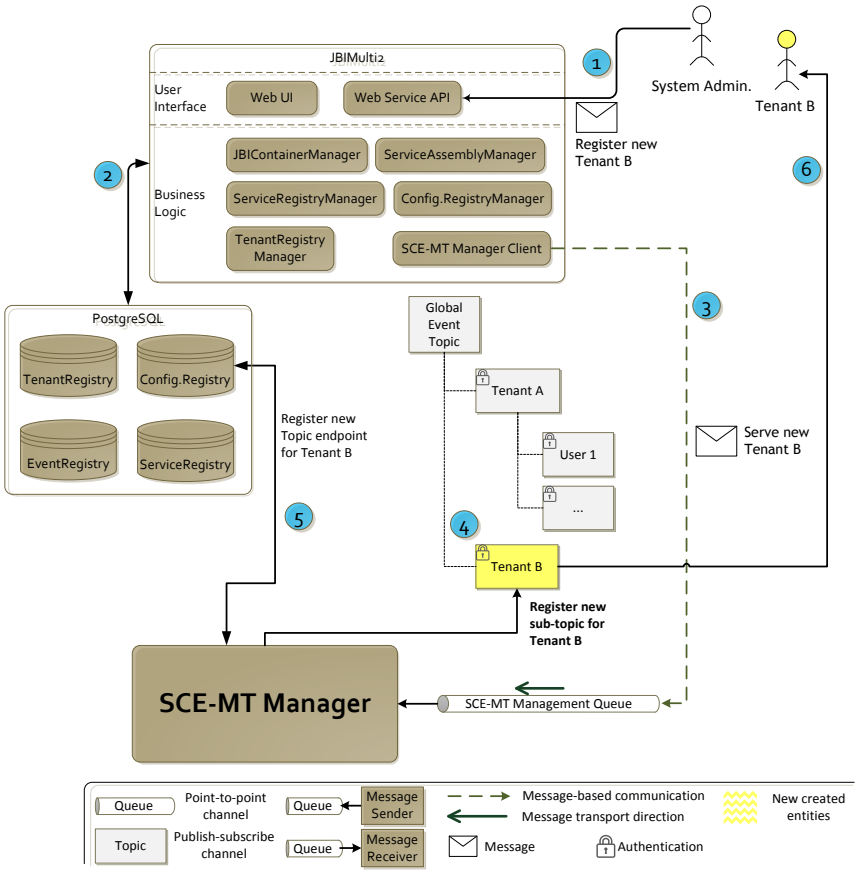

Figure 5.7: Example of status forwarding if a new tenant is registered at JBIMulti2

Event Queue are routed with the help of a Dynamic Recipient List to the correct subset of all registered topics. Another Dynamic Recipient List is installed to route the messages send to the Global ODE Management Queue to all Management Queues of the correct subset of all registered SWfMS^{MT} instances. As a result, the SCE-MT Manager can also use the Global ODE Management Queue to send management messages to any registered SWfMS^{MT} instance (see Fig. 5.6, Point 6). The routing of SWfMS^{MT} management messages is described in detail in Section 5.3.8 and the routing of event messages is further described in Section 5.3.7.

5.3.3 Tenant-aware Administration over JBIMulti2 and Status Forwarding

In this section we take a closer look how JBIMulti2 is used as an administration and management layer for the SCE-MT Manager. As already mentioned, the authentication and authorization functionality of JBIMulti2 is reused to provide a secure connection to the management functionality of the SCE-MT Manager. Figure 5.7 shows therefore an example scenario where

a new tenant is registered at JBIMulti2 and the SCE-MT Manager is informed about that with a status message. First of all, at Point 1 an authenticated System Administrator registers a new Tenant over a call to the corresponding Web Service API operation. The JBIMulti2 application creates a new tenant entity and inserts all provided data of Tenant B in the TenantRegistry at Point 2. Next the JBIMulti2 application should forward the change of the TenantRegistry to the SCE-MT Manager. In Point 3 therefore the *SCE-MT Manager Client* sends a corresponding status message to the *SCE-MT Management Queue* of the SCE-MT Manager. The SCE-MT Manager Client is created in the scope of this thesis to enable sending messages from JBIMulti2 to a SCE-MT Manager. The status message only contains the tenantId of the new tenant. With the help of the tenantId the SCE-MT Manager is able to query some additional information from the database, like the name of the tenant. In Point 4 the SCE-MT Manager registers a new event topic for Tenant B. Subsequently the SCE-MT Manager stores the JMS endpoint address of this new topic in the ConfigurationRegistry (Point 5). At the end, Tenant B is now able to connect to its event topic and receive event messages. The required extensions to the JBIMulti2 application are described in detail in Section 5.7. All other SCE-MT Manager related management tasks are handled in the same way as the example scenario described above. The registration of configuration data is such another management task which is described in detail in Section 5.3.4.

5.3.4 Tenant-based Configuration of SCE Instances and Process Models over JBIMulti2

Figure 5.8 shows how configuration data for registered SCE instances or process models can be registered over the JBIMulti2 application. An authorized user – an authenticated user with the corresponding access permissions to specify configuration data – registers new configuration data at JBIMulti2 over its Web Service API. JBIMulti2 inserts this data in the ConfigurationRegistry and associates it with the specified target SCE instance or process model. Subsequently a status message is created and send to the SCE-MT Management Queue to inform the SCE-MT Manager that new configuration data is registered. The SCE-MT Manager queries the new configuration data from the ConfigurationRegistry and creates management messages out of it. Each of these messages contains the tenantId and an optional userId of the tenant or user which has registered the configuration data at JBIMulti2. Furthermore the modelId of the target process model is added to all messages which contain configuration data for a process model. Those management messages are then send to the Global ODE Management Queue by the SCE-MT Manager. The installed Dynamic Recipient List uses the inserted ids to take care that the management messages are forwarded to all required SWfMS^{MT} instances. If a management message has to be forwarded to an engine instance or not is decided by the Dynamic Recipient List with the help of the data stored in the EventRegistry. Process configuration data has to be forwarded to all SCE instance on which the model is deployed and SCE configuration data has to be forwarded to all SCE instances on which one or more process models of a tenant and its users are deployed. How the detailed routing of management messages is realized is described in Section 5.3.8. After the management messages are routed to the Management Queue of the correct SWfMS^{MT} instance, the contained configuration data is process by SWfMS^{MT} as described in Section 5.5. Each SWfMS^{MT} instance has its own subset of all available configuration data stored in an engine internal database. The reason

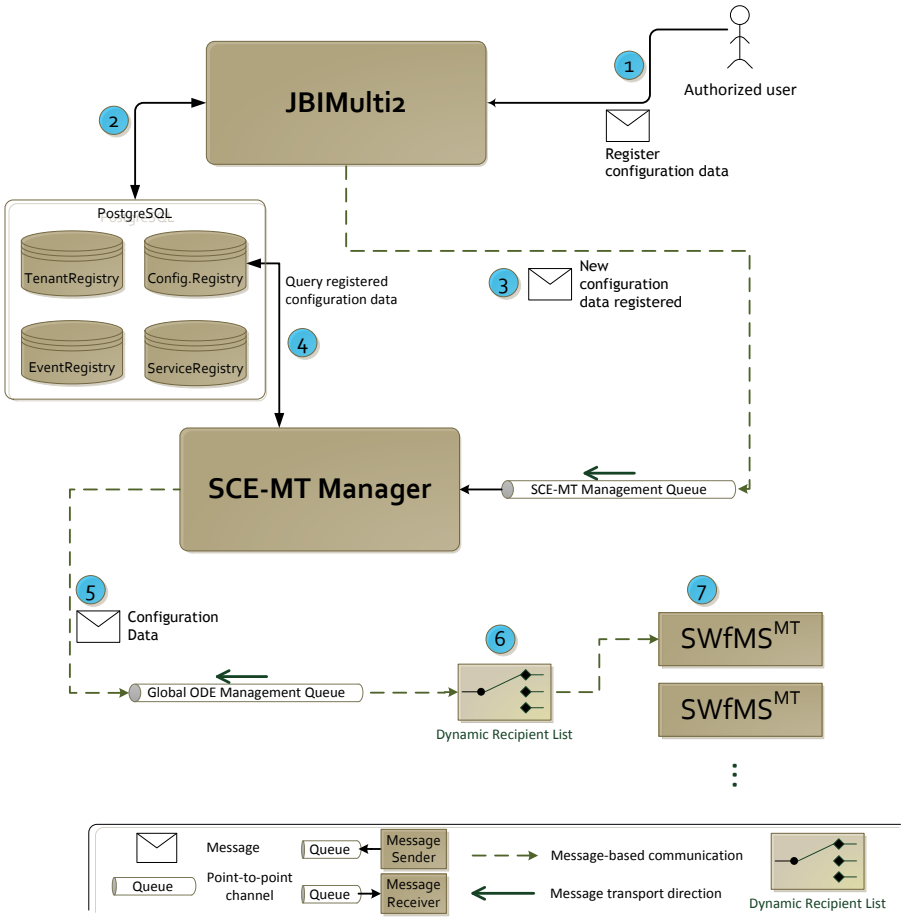

Figure 5.8: Complete process of the registration of configuration data over JBIMulti2

for this realization approach is that the configuration data is less often changed as it is used by SWfMS^{MT}. For example, registered process configuration data is required during each instantiation of the process model. Therefore forwarding and buffering the configuration data inside the SCE provides a better performance and less messaging effort in contrast to querying the configuration data from the shared ConfigurationRegistry on demand. Furthermore the SWfMS^{MT} instances are decoupled from the shared registries and the SCE-MT Manager and can therefore run isolated which is a huge advantage in case of an outage of the registries or the SCE-MT Manager.

5.3.5 Tenant-based Deployment of Process Models over JBIMulti2

Before we have a look how the deployment of process models to SWfMS^{MT} looks like in a JBI environment, the structure and contents of a *Process Service Assembly* (PSA) shown in Figure

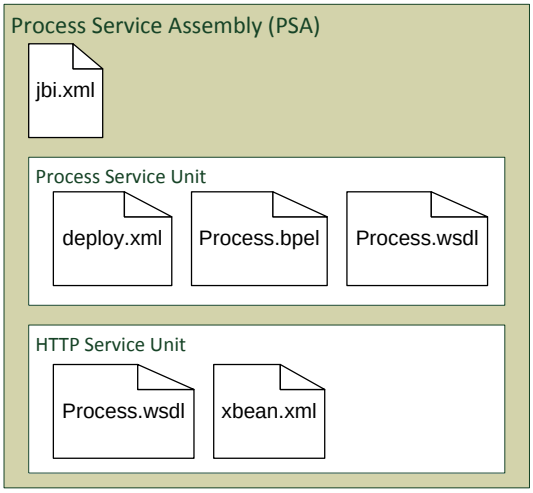

Figure 5.9: Example of a Process Service Assembly and its contents

5.9 are introduced. As described in Chapter 2 to deploy new components to a JBI environment they have to be packaged as *Service Units* (SU) and multiple service units have to be packaged as *Service Assemblies* (SA). As shown in Figure 5.9 we need a Process Service Unit to bundle the resources which should be deployed to the SWfMS^{MT} SE and another HTTP Service Unit to bundle the resources which should be deployed to the multi-tenant HTTP BC of ESB^{MT}. The jbi.xml file shown in Figure 5.9 is used by the JBI deployment mechanism to deploy each SU to the correct target JBI component. In the case of a PSA, the Process SU has to be deployed to the *OdeBpelEngine* which then handles the engine-internal deployment of the process models contained in the Process SU. The HTTP SU has to be deployed to the multi-tenant HTTP BC (*servicemix-http-mt*) which will provide the Web Service interface of a process model to the outside by the registration of a corresponding multi-tenant HTTP consumer endpoint for the process model WS. Listing 5.1 shows the contents of such a jbi.xml file. All values surrounded by ${...} mark properties which have to be changed for each PSA.

The Process SU just packages all process model related files which are required by ODE as shown in Figure 5.9. The *deploy.xml* file is a ODE Deployment Descriptor document which contains some deployment related information used by ODE to deploy and execute the referenced process models. The *Process.bpel* file is the BPEL process model XML document which should be deployed, compiled and later executed by ODE. A Process SU can contain more than one BPEL model file. The *Process.wsdl* file is a WSDL document which provides the service interface the BPEL process implements. A Process SU should contain the WSDL file for each service interface implemented or invoked by any contained BPEL process model.

The HTTP SU contains a copy of all WSDL files contained in the Process SU which provide a service interface of a BPEL process model. In the example HTTP SU shown in Figure 5.9,

Listing 5.1 Example JBI descriptor XML document of a PSA

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>${SA-Name}</name>
      <description>${SA-Description}</description>
    </identification>
    <service-unit>
      <identification>
        <name>${ProcessSU-Name}</name>
        <description>${ProcessSU-Description}</description>
      </identification>
      <target>
        <artifacts-zip>${processSUfilename}.zip</artifacts-zip>
        <component-name>OdeBpelEngine</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>${HttpSU-Name}</name>
        <description>${HttpSU-Description}</description>
      </identification>
      <target>
        <artifacts-zip>${httpSUfilename}.zip</artifacts-zip>
        <component-name>servicemix-http-mt</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

the HTTP SU therefore contains a copy of the Process.wsdl file. The *xbean.xml* file is used during the deployment of the HTTP SU at the multi-tenant HTTP BC. Listing 5.2 shows the contents of the xbean.xml file. The defined HTTP SOAP Consumer endpoint is registered at the ESB and enables the message exchange with the BPEL process model by specifying its service interface and the target service and endpoint name. Furthermore, a specialized HTTP marshaler is referenced over the *http:marshaler* element. The marshaler class provides the authentication functionality for the multi-tenant HTTP endpoint as described in [Sáe13]. In the context of this diploma thesis an extended *ProcessHttpSoapConsumerMarshaller* is implemented which supports the different deployment styles introduced in Chapter 4.5. Now that the structure and contents of a PSA are clear, we want to have a look at the deployment of these SA over JBIMulti2.

Figure 5.10 shows how a PSA is deployed over JBIMulti2 to ESB^{MT} and an installed SWfMS^{MT} SE. The deployment process itself is nearly identical as for any other type of SA as described in the work of Muhler and Gómez [Muh12, Sáe13]. To enable the specification of a deployment style constant under which a PSA should be deployed, the JBIMulti2 WS API is extended with a new `deployProcessServiceAssembly()` operation. This operation extends the existing `deployServiceAssembly()` operation with an additional parameter to specify a deploy-

Listing 5.2 Contents of a xbean XML file to provide a HTTP consumer endpoint over the ESB

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:http="http://jbimulti2.iaas.uni-stuttgart.de/http/1.0"
       xmlns:${prefix}="${targetNamespace of Process.wsdl}">

       <http:soap-consumer targetService="${prefix}:${ProcessServiceName}"
                                          targetEndpoint="${ProcessEndpointName}"
                                          wsdl="classpath:Process.wsdl">
              <http:marshaler>
                     <bean class="de.unistuttgart.iaas.simtech.
                         httpSoapMarshaller.ProcessHttpSoapConsumerMarshaller" />
              </http:marshaler>
       </http:soap-consumer>
</beans>
```

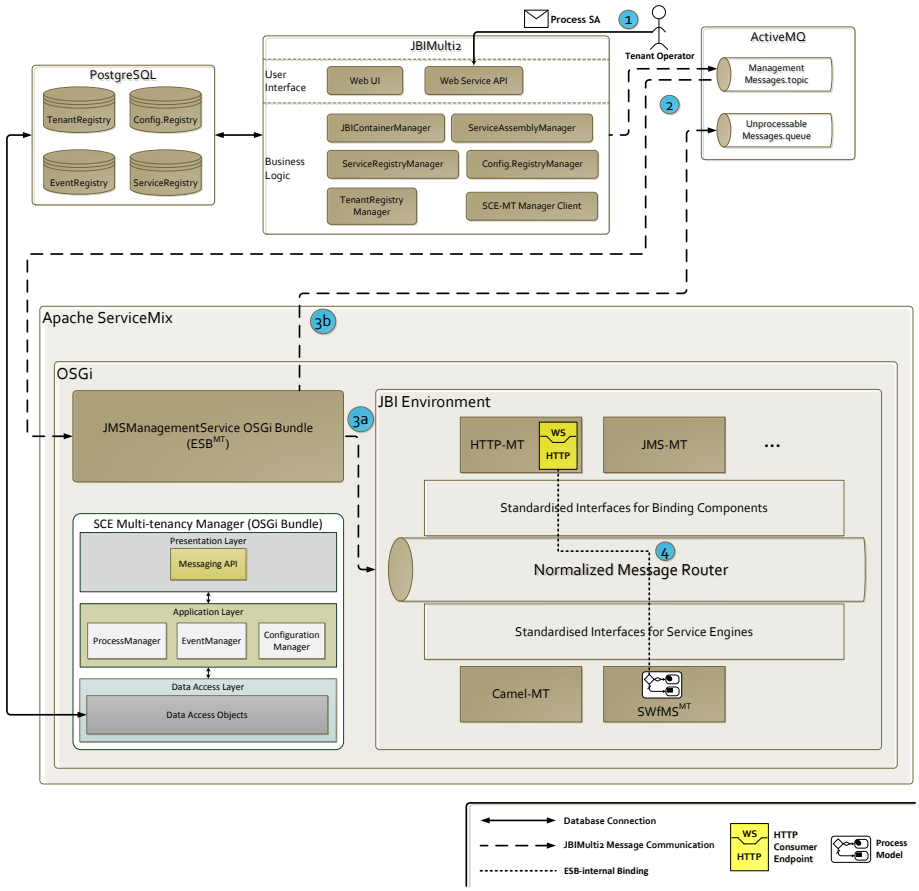

Figure 5.10: Deployment of Process Service Assemblies to ESB^{MT} with installed SWfMS^{MT} over JBIMulti2

Listing 5.3 Tenant context with an optional entry to specify the deployment style of a PSA

```
<ctxjmu2:TenantContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context">
        <ctxjmu2:TenantId>tenantId</ctxjmu2:TenantId>
        <ctxjmu2:UserId>userId</ctxjmu2:UserId>
        <ctxjmu2:OptionalEntry>
               <ctxjmu2:Key>deploymentStyle</ctxjmu2:Key>
               <ctxjmu2:Value>{Public|Tenant-private|User-private}</ctxjmu2:Value>
        </ctxjmu2:OptionalEntry>
</ctxjmu2:TenantContext>
```

ment style (see Chapter 4.5). The deployment of a PSA is started by a call to the new `deployProcessServiceAssembly()` WS API operation by an authenticated and authorized user, like a Tenant Operator as shown in Figure 5.10. The PSA has to be provided as a zip file which is then transfered to JBIMulti2. This zip file is immediately stored associated to the tenantId and userId of the deploying tenant user into the ServiceRegistry. Furthermore, JBIMulti2 marks the registered SA as a PSA over the type attribute of the corresponding ServiceAssembly entity and persists the specified deployment style constant into the deploymentStyle attribute of the ServiceRegistry (see database schemas in Sect. 5.2). After everything is stored to the ServiceRegistry, JBIMulti2 creates a *DeployServiceAssemblyCommand* message to deploy the PSA to ESB^{MT}. This message contains the binary data of the PSA and a tenant context. The tenant context contains the tenantId and userId of the user to which the PSA belongs to. Furthermore, it contains the specified deployment style constant as an optional entry as shown in Listing 5.3. This message is then published to the Management Messages.topic as shown in Figure 5.10. A subscribed JMSManagementService consumes the command message and uses the tenant context to realize the tenant-aware deployment of all Service Units of the PSA. Therefore the JMSManagementService adds the tenantId and userId to the service assembly name, all service unit names and updates all JBI deployment descriptors. This makes it possible that different tenant users can deploy equally named service assemblies and service units. The JMSManagementService also adds the tenant context send with the command message as an XML file to each of the service units contained in the PSA. This is important because the tenant context XML file is later used to authenticate process instantiation requests as described in Section 5.3.6. After all modifications are finished, the JMSManagementService uses the `org.apache.servicemix.jbi.deployer.AdminCommandsService` to deploy the PSA to the JBI environment. If any exception occurs during the deployment, the JMSManagementService sends a corresponding fault message to the *Unprocessable Messages.queue* (see Fig. 5.10, 3b). The AdminCommandsService deploys each SU contained in the PSA to the correct JBI target component with the help of the JBI descriptor document contained in the PSA. In case of a PSA, this means the HTTP endpoint is registered at the multi-tenant HTTP BC and the Process SU is deployed to the SWfMS^{MT} SE. After that the deployed process models are ready to be instantiated by sending a request message to their HTTP endpoint.

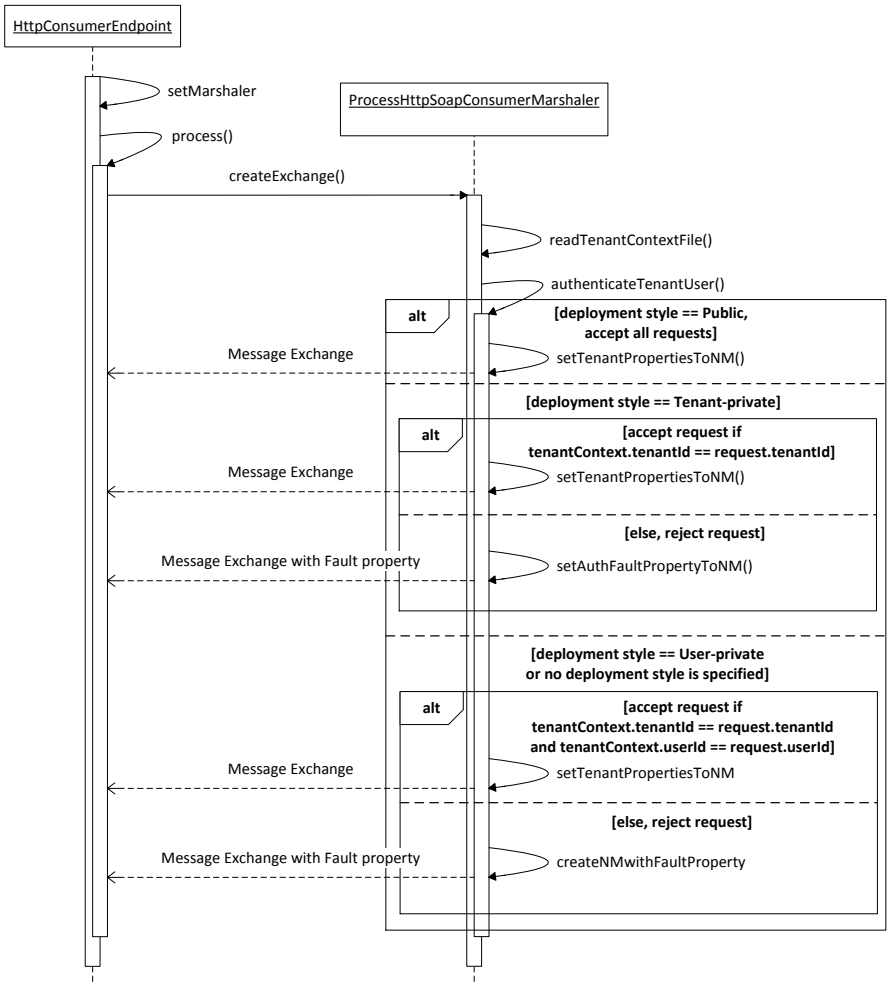

Figure 5.11: Sequence diagram of the authentication of incoming requests at a multi-tenant HTTP endpoint

5.3.6 Tenant-aware Process Instantiation with ESB^{MT}

As already described, the authentication of requests send to the WS interface of a process model is realized over ESB^{MT} with the endpoint-based authentication functionality introduced by Gómez [Sáe13]. Figure 5.11 is a *Unified Modeling Language* (UML) Sequence diagram which shows how the authentication process looks like. The *HttpConsumerEndpoint* class represents a deployed HTTP consumer endpoint and is therefore responsible to process any requests send to the endpoint address. If a PSA is deployed an instance of this class is created during the deployment of the process WS HTTP consumer endpoint based on the provided xbean.xml file (see Sect. 5.3.5). All values specified in the xbean.xml file are set to

the created HttpConsumerEndpoint instance, like the target service name or the location of the WSDL file. Furthermore, the marshaler class which is referenced in the *http:marshaler* element of the xbean.xml document is set to the HttpConsumerEndpoint instance over its `setMarshaler()` method. The HTTP SU of a PSA will contain a specialized *ProcessHttpSoapConsumerMarshaler* which extends the multi-tenant aware *HttpSoapConsumerMarshaler* base class. As described above, the ProcessHttpSoapConsumerMarshaler provides an extended authentication functionality by supporting the introduced deployment styles. If a new request message is send to the deployed HTTP consumer endpoint, the `process()` method of the HttpConsumerEndpoint instance is invoked as shown in Figure 5.11. Subsequently, the HttpConsumerEndpoint starts the authentication of the incoming requests by a call to the `createExchange()` method of the ProcessHttpSoapConsumerMarshaler. Next the marshaler retrieves the tenant context information stored in the tenant context file of the deployed HTTP SU (`readTenantContextFile()`) and compares it with the tenant context information contained in the incoming request message (`authenticateTenantUser()`). How the tenant context and the tenant data contained in the incoming SOAP request are compared is based on the deployment style specified in the tenant context file. Before we have a look at the different authentication cases based on the deployment styles, the general response of the `createExchange()` operation is described. In any case a new *MessageExchange* object is created which contains a *Normalized Message* (NM) representation of the incoming SOAP request. This NM is created with the data contained in the SOAP request. When the request is successful authenticated and therefore accepted the tenant context information is added in form of a set of property maps to the NM. If the authentication is not successful and therefore the request should be rejected a corresponding fault property is set to the NM. This enables the HttpConsumerEndpoint to decide if an incoming request should be forwarded to the *Normalized Message Router* (NMR) and therefore be processed by SWfMS^{MT} or a fault message should be responded immediately.

Now we want to have a closer look at the different authentication cases shown in Figure 5.11. The following descriptions are based on the definition of the three deployment styles provided by Chapter 4.5. The *Public* deployment style accepts any incoming requests whether they contain a tenant context or not. If the *Tenant-private* deployment style is specified, all incoming requests which contain the same tenantId as the one stored in the tenant context file are accepted. All other requests are rejected, like requests without any tenant data or a different tenantId. The *User-private* deployment style is the most restrictive one and is used as the default behavior if no deployment style is specified. It accepts only requests which contain the same tenantId and userId as specified in the tenant context file of the HTTP SU. All other requests are rejected.

5.3.7 Tenant-aware Event Messaging and Event Message Routing

As introduced above the event messages will be routed to the correct collection of event topics with the help of the routing functionality of the ESB. Therefore, a corresponding message route

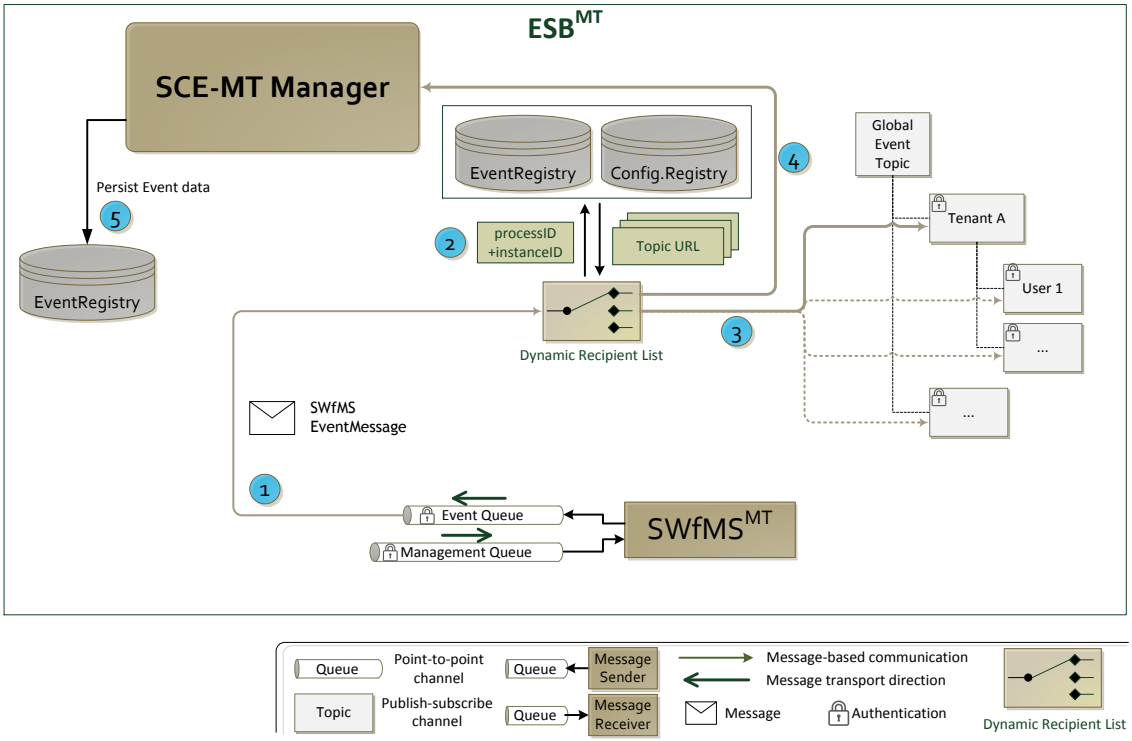

Figure 5.12: Routing of event messages with the ESB

is deployed to the Apache Camel [3] SE of the ESB. Apache Camel provides the implementation for a set of *Enterprise Integration Patterns* (EIP)[4] which can be used to process messages. Figure 5.12 shows how the routing of event messages is realized by deploying a so called *Dynamic Recipient List* (DRL). This special kind of message router enables the routing of incoming messages to a dynamically calculated list of recipients like a collection of queues or topics. The calculation of the recipients is based on some data contained in the message and can use also some external data which is maybe stored in a database. The main advantage is that one message can be routed to more than one endpoint and therefore a copy of the event message can be provided to each tenant user which is authorized to receive it. For each installed SWfMS^{MT} instance one corresponding DRL is deployed. This DRL consumes all event messages from the Event Queue of the SWfMS^{MT} instance and reroutes them to the correct target destination. Figure 5.12 shows the message path each event message flows through. First of all, SWfMS^{MT} puts the event message on its Event Queue. The listening DRL consumes this message and starts to calculate the recipient list. Therefore, it extracts the processId and instanceId value of the event message and uses the EventRegistry to find

[3]The Apache Software Foundation, Apache Camel: http://camel.apache.org/
[4]Apache Camel, Enterprise Integration Patterns: http://camel.apache.org/enterprise-integration-patterns.html

out to which tenant users the corresponding process instance belongs. This is important to realize process instance based collaboration in the future because the shared registries contain the required data which tenant users collaborate on a specific process instance and therefore should get the event messages. The prototype realized in the context of this thesis does not support collaboration on the process instance level and therefore each process instance belongs to one single tenant user. After the DLR calculates the list of all tenant users to which the event message should be send, it uses the ConfigurationRegistry to get their event topic endpoint addresses. If a process instance belongs to no tenant users it is published to the Global Event Topic. In case that one process instance emits a maybe huge number of event messages at frequent intervals, the DRL provides some caching mechanism. The cache is a map between instanceIds and a collection of topic endpoints. Therefore the DRL has only calculate the recipient list for the first event message of a process instance, saves the data to the cache and can then use the cached values for all succeeding event messages of the process instance. After the list of recipients is clear the event messages are copied and published to the corresponding set of event topics as shown in Figure 5.12. An additional copy of each event message routed by any deployed DRL is send to the SCE-MT Manager. The SCE-MT Manager is responsible to persistently store all emitted event messages to the EventRegistry and extracts some required data out of them. This is really useful because the event messages provide some important information like the status of all process instances of one tenant users. Therefore the SCE-MT Manager consumes them, processes some of them and stores all contained data in the EventRegistry. One case where the event data is processed by the SCE-MT Manager to get some required information is the deployment of a new process model. The data contained in the corresponding *Process_Deployed* event message is used to associate the referenced process model with the SCE instance it is deployed to. This information is stored in the EventRegistry and can later be used for example to find out where the process models of a tenant user are deployed.

5.3.8 Routing of SWfMS^{MT} Management Messages

SWfMS provides two distinct management endpoints. The Process&InstanceManagement Web Service provides an interface for the engine-internal Process ManagementAPI and Instance Management API (see Fig. 2.1). It enables the management of process models and process instances like suspending an instance or changing the state of a process model. In contrast to that, the Management Queue provides an interface for the PGF and enables therefore the use of its functionality from the outside. For example, the debugging of the execution of a process instance or setting variable or partner link values of a process instance during its execution. In this section the routing of messages send to both of these interfaces is described. The underlying idea is to provide one static endpoint for each of the interfaces and then route the messages to the corresponding endpoints of the correct SWfMS^{MT} instance. As a result that one process model is maybe deployed to a set of SWfMS^{MT} instances to enable scalability in future versions, the corresponding received management messages have to be sent to all of those engine instances. Another difference in contrast to the routing of event messages is that for most of the management messages SWfMS^{MT} produces a corresponding response message. In the case a request message is maybe send to a number of recipients, the returned response

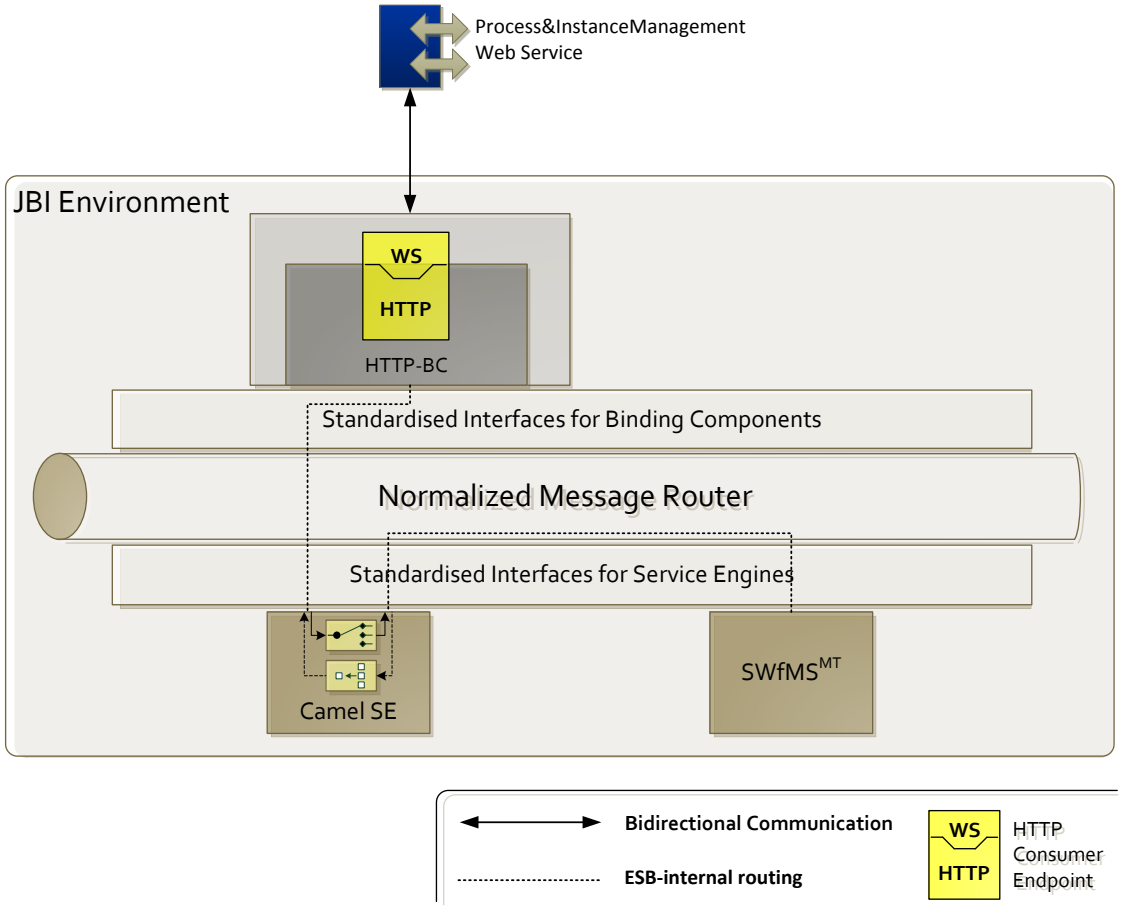

Figure 5.13: Routing of message exchanges for the Process&InstanceManagement Web Service with Apache Camel

messages have to be aggregated into a single message and returned to the initial sender of the request. This routing pattern is known as Dynamic Scatter-Gather EIP and consists of a Dynamic Recipient List and an Aggregator. The DRL is used to route a request message to a dynamically calculated set of SWfMS^{MT} instances to which the request should be forwarded based on the contents of the request message. The Aggregator is used to re-aggregate all response messages back into a single message which can then be returned to the initial sender. This pattern is used to decouple the tenant users from the set of installed SWfMS^{MT} instances and therefore enables scalability. A tenant user does not have to know the list of SWfMS^{MT} instances to which its management messages should be send and also does not have to care about the aggregation of the multiple response messages and the data they contain. In the following the used Dynamic Scatter-Gather EIPs are described for both of the management interfaces.

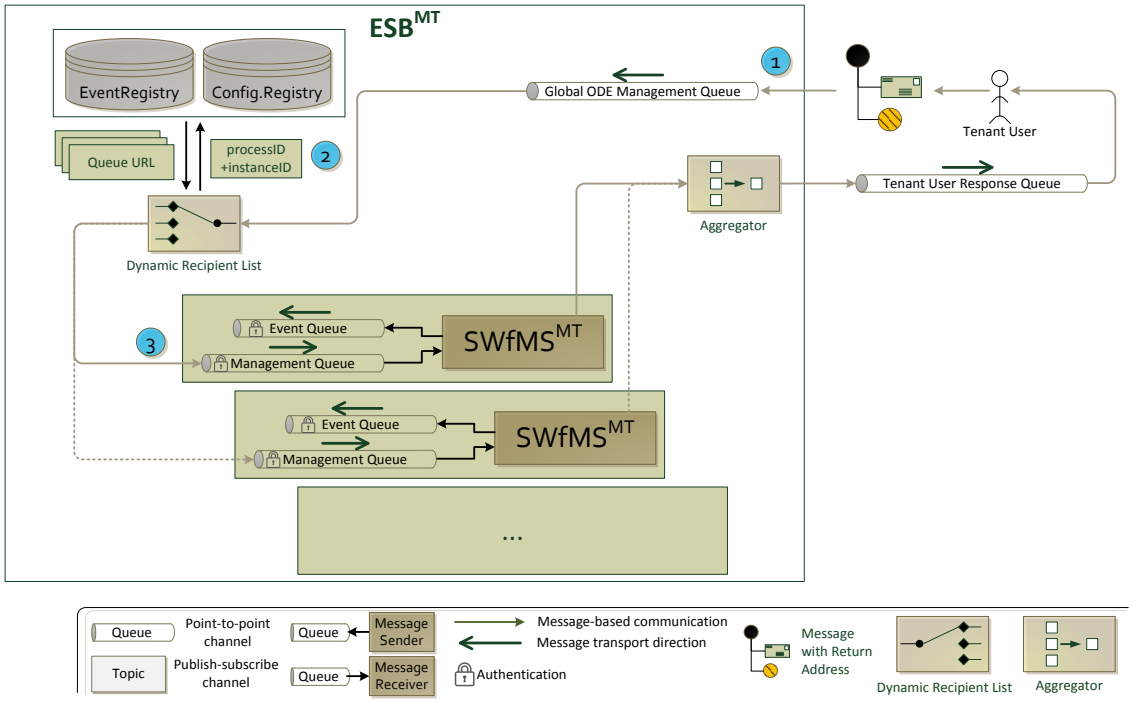

Figure 5.14: Routing of SWfMS management messages with Apache Camel

Figure 5.13 shows the routing of message exchanges with the Process&InstanceManagement Web Service. This service interface is used to deploy a static HTTP consumer endpoint which is provided to the outside. The WSDL file of the Process&InstanceManagement Web Service and an xbean.xml file which contains the HTTP consumer endpoint definition are therefore packaged as a SU and deployed to the HTTP BC. The Process&InstanceManagement service interfaces of all installed SWfMS^{MT} instances are not provided to the outside and therefore are only accessible inside the ESB as JBI endpoints. The internal endpoints are stored together with other SCE instance related data in the ConfigurationRegistry (see Fig. 5.3). They are used as the target endpoints to reroute the incoming messages send to the HTTP consumer endpoint by the DRL. Since in the context of this diploma thesis only a single SWfMS^{MT} instance is used, the Aggregator just reroutes the original return message of the SWfMS^{MT} instance back to the HTTP consumer endpoint. This single internal return message becomes therefore the response message of the initial HTTP request.

Figure 5.14 shows the routing of PGF management messages. This is realized quite similar to the routing of the HTTP requests but through the asynchronous nature of messaging each message requires a Return Address to send the response to. This return address has to be specified by the sender of the request. For example, a tenant user specifies the address of its queue (*Tenant User Response Queue*, see Fig. 5.14) to asynchronously receive the responses for its requests. A request message with a specified return address can then be send to the

Global ODE Management Queue which provides the static endpoint for the PGF management functionality. The DRL of the deployed Dynamic Scatter-Gather EIP consumes all messages send to this queue and reroutes them to the correct target destination. Therefore, it uses the data contained in the request message like a processId or instanceId to find out which SWfMS^{MT} instances are associated with the referenced process model or process instance. For example, if a breakpoint registration message should be routed, the DRL uses the contained processId to query all sceInstanceIds from the EventRegistry. Each of these sceInstanceId identifies a SWfMS^{MT} instance to which the process model referenced in the request message is deployed to (see Fig. 5.4). The list of sceInstanceIds is then used by the DRL to query the required list of Management Queue endpoints from the ConfigurationRegistry (see Fig. 5.3). After the recipient list is calculated the DRL routes the management message to all referenced Management Queues. Finally the Aggregator re-aggregates all response messages of the SWfMS^{MT} instances and sends the final response message to the Tenant User Response Queue. Since in the context of this diploma thesis only a single SWfMS^{MT} instance is used, the Aggregator just reroutes the original return message of the SWfMS^{MT} instance to the Tenant User Response Queue.

5.4 Multi-tenant SWfMS Architecture

Figure 5.15 shows the extended SWfMS (SWfMS^{MT}) architecture based on the general SCE^{MT} architecture described in Chapter 4.7. The architecture of the SCE-MT Manager will be described separately in Section 5.6. The figure shows only an abstracted version of the SWfMS architecture on a module basis because a detailed diagram which contains all classes would be to complex. In case of that, only some details of the most important modules and their internal communication are shown in the figure. Since SWfMS^{MT} should be integrated into an ESB as JBI Service Engine, only the extensions of the existing JBI Integration Layer are described. An extension of the Axis2 Integration Layer is out of the scope of this diploma thesis. All extended or adapted components are marked green and all new components are marked yellow in Figure 5.15. In the following the architecture and the connections between the different modules are described from top to bottom.

The JBI Integration Layer contains all modules to integrate SWfMS into a JBI environment. The *OdeSUManager* class provides some management functionality (e.g. deploy, start or init) for *OdeServiceUnit*s and connects the JBI management functionality with the ODE-internal deployment mechanisms. If a new Process *Service Unit* (SU) is deployed to the JBI Environment, the OdeSUManager creates a new OdeServiceUnit instance and then triggers the ODE-internal deployment process over the *Process Deployment API* with the help of the *ProcessStore*. This internal deployment process should be extended to associate the deployed process models with the teant data contained in the tenant-context.xml file of the underlying Process SU (see Sect. 5.3.5). The *DynamicMessageExchangeProcesser* class handles the message exchange for the *Process Management API* and the *Instance Management API*. The Process&InstanceManagement Web Service interface is therefore provided by the ESB which then reroutes all incoming messages to SWfMS as described in Section 5.3.8. The DynamicMessageExchangeProcesser consumes these messages and forwards them to

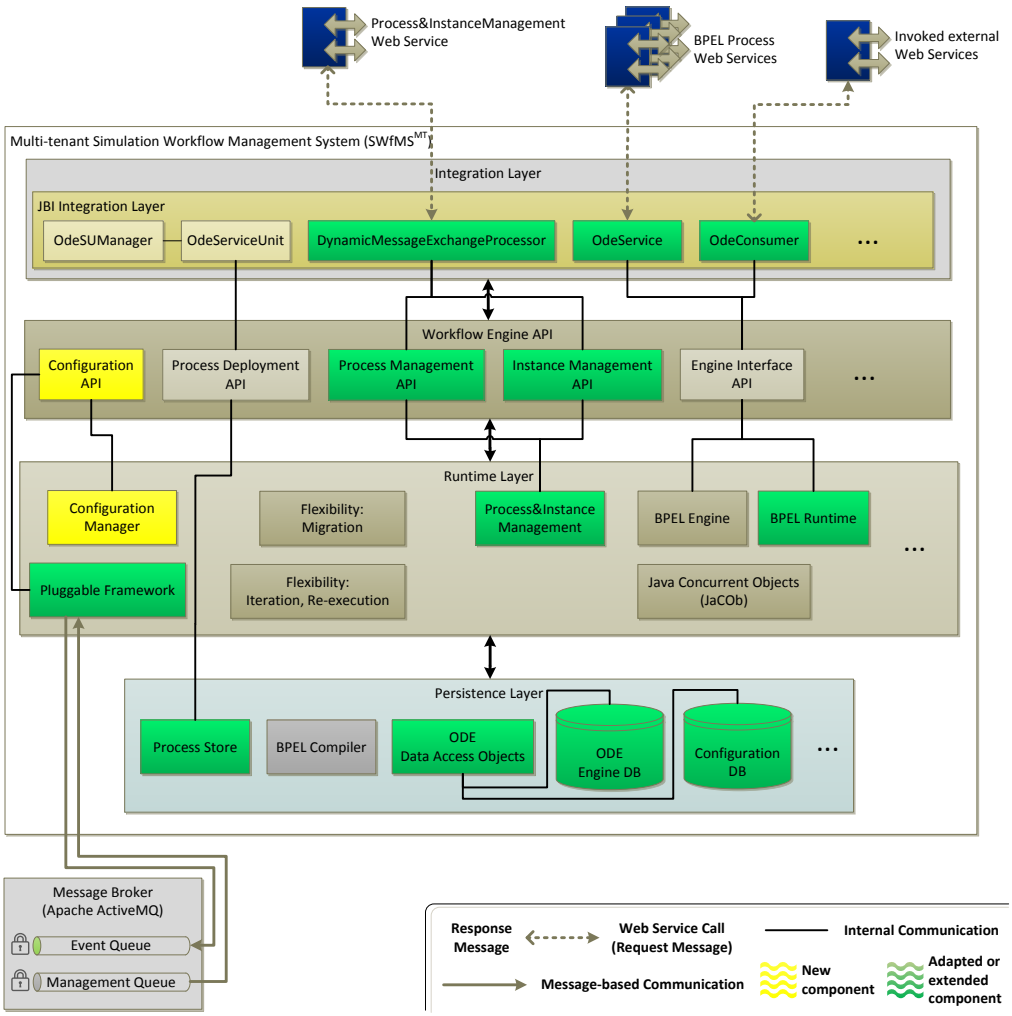

Figure 5.15: Multi-tenant aware SWfMS Architecture

the correct Management API. To enable tenant-aware communication for the Management APIs the DynamicMessageExchangeProcesser should be extended to forward the tenant data contained in the incoming request messages to the Management APIs. Each instance of the *OdeService* class provides one JBI provider endpoint for a specific Process Model to the JBI environment. This provider endpoint is used by the ESB to reroute the messages send to the HTTP consumer endpoint of a process model to SWfMS^{MT}. The consumer endpoint is provided over the HTTP-MT BC to the outside as described in Section 5.3.5. The OdeService class should also be extended to forward the tenant data contained in incoming requests to the engine internal logic. This forwarded tenant data is used to realize "invocation under tenant context") which means that the created process instance is immediately associated to the tenant user specified by the incoming request message (see Chapt. 4.4.2). An instance of

the *OdeConsumer* class defines a JBI consumer endpoint for an external Web Service which is invoked during the execution of a Process Model. This consumer endpoint is used by the ESB to reroute all messages to the correct HTTP provider endpoint which may provide a Web Service hosted inside the ESB (e.g. another Process Web Service) or an external hosted Web Service to the JBI environment. The OdeConsumer class should also be extended to forward the tenant data associated to the process instance which invokes the external service. This is important if the external service is also multi-tenant aware, for example if another process model is invoked by a process instance.

The Workflow Engine API and its components should be extended to forward the tenant information of incoming requests to the components of the Runtime and Persistence Layer as already described. Furthermore, to enable engine and process model configurations, a new *Configuration API* should be provided.

The Runtime Layer should be extended to provide some new multi-tenant functionality, like authentication of management requests or the configuration of the engine on a per-tenant basis. The Pluggable Framework should be adapted to provide two secure messaging queues with at least simple username&password authentication (see Sect. 5.3.1). All event messages are enriched with additional data (e.g. a tenantID and userID) to enable tenant-based isolation of event data. The event messages are not any longer directly published to a topic since the SCE-MT Manager handles with the help of the message routing capabilities of the ESB the correct distribution of event messages now. The details of the new event publishing mechanism are described in Section 5.3.7. The Pluggable Framework is also used to handle the communication between a SWfMS instance and the SCE-MT Manager, for example to register a new SWfMS instance at the SCE-MT Manager. It also provides a Messaging Interface for the Configuration API over the Management Queue. A new messaging listener consumes all configuration messages send to the Management Queue and calls the corresponding Configuration API methods. The *Process&Instance Management* must be extended to provide Administration Isolation and the authentication of incoming management requests. This is implemented inside the Runtime Layer to avoid the implementation of the authentication and isolation functionality for each Integration Layer. The extension of the Process&Instance Management class provides also a more extensible and powerful approach because it will be possible to provide the ability to specify fine-grained access permissions in the future. For example, that a user is only able to manage his process models and instances but is not able to use the flexibility functionality. The authentication of incoming management requests is realized by the comparison of the tenant data contained in the request message and the tenant data associated to the process model or process instance which is referenced in the management request. The new *Configuration Manager* class implements the Configuration API and handles the management of configuration data. Therefore some new *Data Access Objects* (DAO) are created which are used to persist the configuration data in the new engine-internal Configuration Database. Furthermore, the Configuration Manager is responsible to execute the configuration of the engine and the process models based on the stored tenants' configuration data. For example, if a tenant has specified some configuration data for a process model, the Configuration Manager must assign the configuration data on process instantiation to any created instance of the process model which belongs to this tenant. The provided configuration options are described in Section 5.5 and how these options can be registered is described in Section 5.3.4. The *BPEL Runtime* component

should be extended to associate the tenant context referenced by an instance to each message of any message exchange between the instance and another service like for the (a)synchronous invocation of an external Web Service.

The components of the Persistence Layer should be extended to provide Data Isolation. Therefore, the *ODE Engine Database* and the new *Configuration Database* must be realized by using one of the approaches to realize a multi-tenant data architecture described in Chapter 3. We will use the Shared Schema approach to serve a maximum number of tenants with a single shared SCE instance. As a result each process model, instance context and message exchange is associated with a tenant context to uniquely identify the tenant to which this resources belong. The associated tenant contexts can then be used by the Process&Instance Management to realize the authentication of incoming requests by checking the equality of the tenant data. The *ODE Data Access Objects* are extended with new classes to store the configuration data and the tenant contexts in a persistent manner. To enable referencing tenant contexts and configuration DAOs in the corresponding process, process instance and message DAOs, these DAO classes are also extended. The *ProcessStore* should be extended to be also tenant-aware and enable the tenant-based deployment of process models which is described in detail in Section 5.3.5.

5.5 Configurability of SWfMS^{MT}

The SWfMS^{MT} implementation realized in the context of this diploma thesis will provide two of the configuration possibilities described in Chapters 4.2 and 4.3. To prototypically realize the tenant-based configuration of engine instances, the registration of BPEL Extension Bundles on a per-tenant basis is realized. As described in Chapter 2, BPEL provides a language extension construct which enables the modeling of processes with new custom activities (Extension Activities). To execute these new Extension Activities a corresponding runtime implementation has to be registered on the engine side. Therefore, ODE provides the possibility to register so called Extension Bundles which contain the runtime implementation and an optional validation implementation of one or more BPEL Extension Activities. These Extension Bundles must be packaged as *Java Archive* (JAR) files, copied to the classpath of ODE and are registered in the ODE configuration file which is shown in Listing 5.4. ODE reads the configuration file and extracts the qualified name (package and class name) of the main class of all specified Extension Bundles from the corresponding properties. This main class must extend the abstract class *AbstractExtensionBundle* and contains the namespace of the Extension Bundle and a map of BPEL Extension Activity names and implementations (activity name mapped with the implementing class). As already mentioned, there are two different types of Extension Bundle classes. To provide the runtime logic of a BPEL Extension Activity, the *ExtensionOperation* interface must be implemented. Additionally ODE provides the possibility to validate the XML element of an Extension Activity during the compilation of a process model which contains a corresponding Extension Activity construct. To provide the validation of Extension Activities to the BPEL Compiler the *ExtensionValidator* interface must be implemented. Extension Bundles which contain ExtensionOperation classes are registered over the *ode-jbi.extension.bundles.runtime* property and bundles which contain ExtensionValidator classes

Listing 5.4 Extract of a ODE configuration file

```
[...]
# Database Mode ("INTERNAL", "EXTERNAL", "EMBEDDED")
# What kind of database should ODE use?
#   * "EMBEDDED" - ODE will create its own embbeded database (Derby)
#             and connection pool (Minerva).
#   * "EXTERNAL" - ODE will use an app-server provided database and pool.
#                  The "ode-jbi.db.ext.dataSource" property will need to
#                  be set.
#   * "INTERNAL" - ODE will create its own connection pool for a user-
#                  specified JDBC URL and driver.
ode-jbi.db.mode=EMBEDDED

# BPEL Extension Bundles
# Uncomment the following to register extension bundles.
ode-jbi.extension.bundles.runtime = de.ustutt.simtech.extensions.SimTechExtensionBundle
ode-jbi.extension.bundles.validation = de.ustutt.simtech.extensions.SimTechExtensionBundle
[...]
```

are registered over the *ode-jbi.extension.bundles.validation* property. As shown in Listing 5.4 the *SimTechExtensionBundle* provides new runtime and validation class and therefore the same bundle has to be registered over both properties.

The ODE configuration file is only read once at the startup of the engine. Since the configuration should be also able during runtime and not influence other tenants, it is not applicable to shutdown the engine, change the configuration file and start the engine again to just register a new Extension Bundle. Therefore, the Configuration Manager is used to enable the management of Extension Bundles on a per-tenant basis during the runtime of the engine. Figure 5.16 shows how the registration of Extension Bundles can be realized with the Configuration Manager also during runtime. The Configuration Manager provides all registered ExtensionValidator implementations to the ProcessStore. If a new model is deployed the ProcessStore forwards the validators registered for the tenant to which the process model belongs to the *BpelCompiler* and initiates the compilation of the model by calling the *compile()* method. If the compiler finds an Extension Activity element, it uses the list of registered ExtensionValidator implementations to instantiate the correct class and start the validation by calling its *validate()* method. After the model is compiled, the engine-internal representation (a new *BpelProcess* object) is created. The Configuration Manager sets all available ExtensionOperation implementations registered by the same tenant to which the process model belongs to this new BpelProcess object by calling its *setExtensionRegistry()* method. If a new request message is send to the process service of the BpelProcess, a new *BpelRuntimeContext* object is created over *createRuntimeContext()*). The BpelRuntimeContext represents a process instance in ODE and its execution is started by calling the *execute()* method. If the underlying process model contains an Extension Activity, a new instance of the *EXTENSIONACTIVITY* class is created which provides the runtime logic to invoke the registered ExtensionOperation implementation based on the qualified name of the Extension Activity. A call to the *createExtensionActivityImplementation()* method of the BpelRuntimeContext class returns a new instance of the corresponding ExtensionOperation class. The BpelRuntimeContext therefore retrieves the correct Extension Bundle with the help

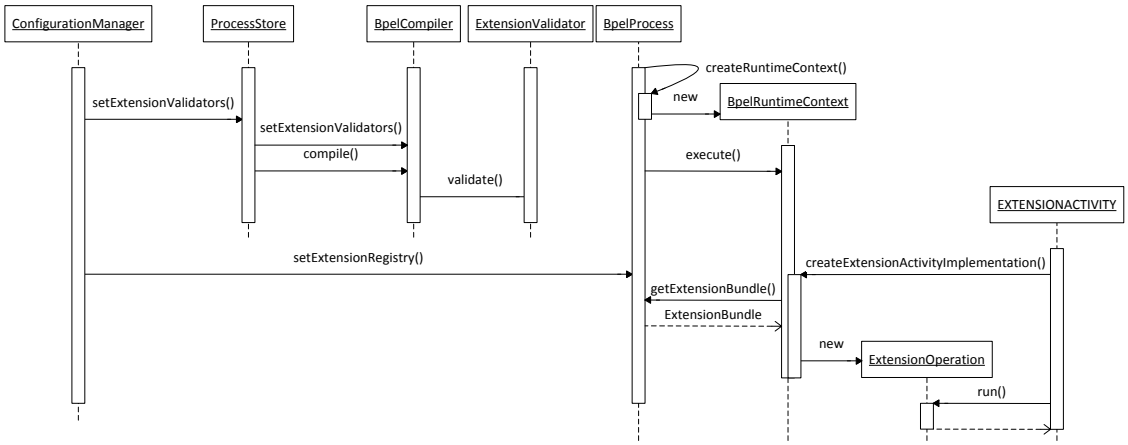

Figure 5.16: Sequence diagram of Extension Bundle configuration with the Configuration Manager

of the namespace of the Extension Activity from the ExtensionRegistry set to the BpelProcess (*getExtensionBundle()*). The class of the required ExtensionOperation can then be resolved over the Extension Bundle with the name of the Extension Activity. This class is then used to create a new instance by using the *newInstance()* method of the *Java Reflection API*. After that the EXTENSIONACTIVITY executes the functionality of the ExtensionOperation by invoking the *run()* method of the created instance. If a tenant registers new Extension Bundles during runtime, the Configuration Manager only has to update the ExtensionValidators at the ProcessStore and the ExtensionRegistry of all BpelProcess objects which belong this tenant.

The other configuration option will provide a prototypical realization for the tenant-based configuration of process models by enabling the registration of runtime data, like values of variables or partner links. The target elements of the process to specify configuration data for, are referenced over their XPath based on the underlying BPEL Process XML file. For example, to specify configuration data for the first variable defined on the process level of a model, the variable can be referenced with the following XPath expression "/process/variables[1]/variable[1]". The Configuration Manager provides a corresponding method which fetch any registered configuration data for a given XPath out of the Configuration Database. Figure 5.17 shows how the configuration of a Process Model is realized with the help of the Configuration Manager. If a new process instance is created by a call of *invokeProcess()* a new BpelRuntimeContext object is generated (*createRuntimeContext()*) and immediately processed over its *execute()* method. The configuration data can not be set on process instantiation because the process model maybe contains a BPEL Assign activity which handles the initialization of any required constructs (e.g. variables, partner links) with default data and therefore overwrites the assigned configuration data. This default initialization is required because if a tenant just want to use a process model as it is without specifying any configuration

data, the model needs the default values to work properly. On the other side, if a tenant specifies configuration data for a process model, this data must not overwritten with any default values. Therefore, the *writeVariable()* and the *writeEndpointReference()* methods of the BpelRuntimeContext class are extended. These two methods handle the writing of variable and partner link values during runtime. The idea is, that the first call of one of these methods initialize the corresponding variable or partner link. So all we have to do, is to check if the variable or partner link has already a value and therefore is initialized or not. In case the variable or partner link is not initialized, their XPath is used to fetch maybe registered configuration data from the Configuration Manager. If the Configuration Manager returns a value, the value passed as method parameter is ignored and the returned configuration data is set as the new value of the variable or partner link. In all other cases, if no configuration data is registered or if the variable or partner link is initialized already, the value passed as method parameter is assigned to the variable or partner link. This enables the correct configuration of instances of a process model with the latest registered configuration data provided by the Configuration Manager.

The overall process how configuration data is registered for a specific tenant at SWfMS^{MT} over JBIMulti2 and the SCE-MT Manager is described in detail in Section 5.3.4.

5.6 Architecture of SCE-MT Manager

Figure 5.18 shows the architecture of the SCE-MT Manager implementation. The *Messaging API* provides the internal logic to the outside and realizes the communication between the SCE-MT Manager, the JBIMulti2 application and all registered SWfMS^{MT} instances. Therefore, a set of *javax.jms.MessageConsumer* and *javax.jms.MessageProducer* classes are created to be able to send and receive messages of all relevant channels of the Messaging Infrastructure shown in Figure 5.5. The Security component introduced in Chapter 4 is realized by the authentication functionality of the multi-tenant aware BCs of ESB^{MT}. How the authentication of requests send to a BPEL Process Web Service is realized by the ESB is described in detail in Section 5.3.6.

The *Application Layer* provides initially four components which contain the business logic of the SCE-MT Manager. The *SCEManager* handles the registration of new SCE instances at the SCE-MT Manager as described in Section 5.3.2 and is responsible for the installation of any required message routes to the ESB. The *ProcessManager* realizes the tenant-aware management of any data related to process models. For example, it associates a process model with its SWfMS^{MT} instance in the EventRegistry during the deployment of the model (see Sect. 5.3.7). The ProcessManager is also responsible to associate the SWfMS^{MT} instances with the process instances they are executing in the EventRegistry. This information is required by the installed message routers for the routing of management messages as described in Section 5.3.8. The *EventManager* realizes the tenant-isolated and persistent storage of all PGF event messages emitted by any connected SWfMS^{MT} instance as described in Section 5.3.7. The *ConfigurationManager* handles the distribution of any SCE and process model configuration data registered by JBIMulti2. As described in Section 5.3.4, JBIMulti2 inserts

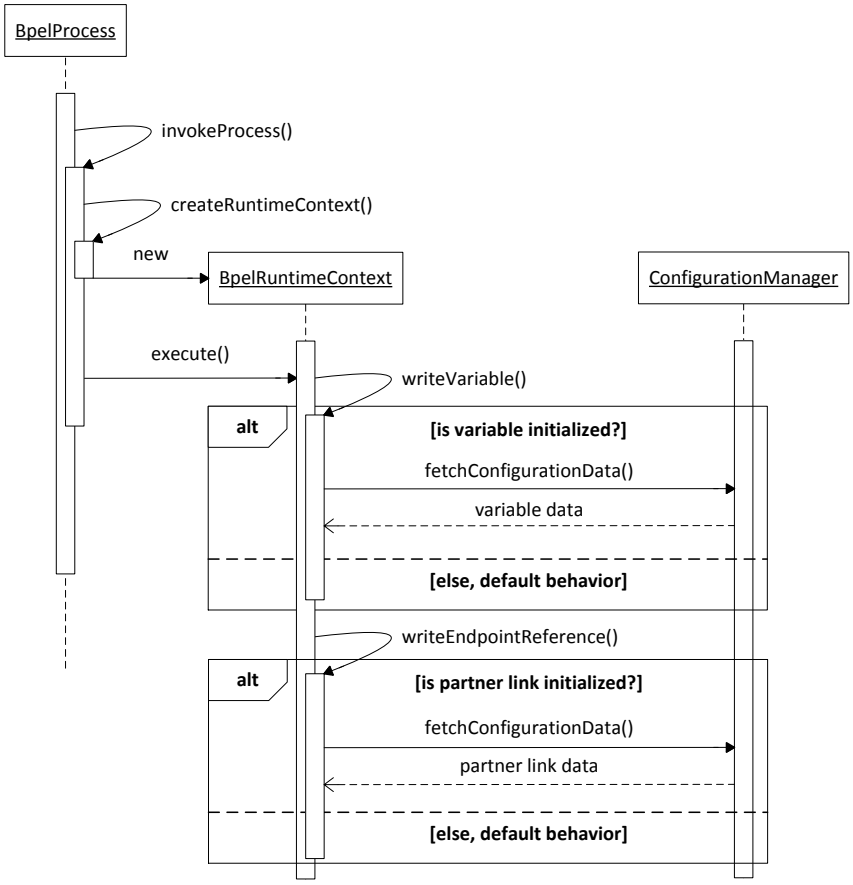

Figure 5.17: Sequence diagram of process model configuration with the Configuration Manager

this data in the ConfigurationRegistry and associates it with the specified target SCE instance or process model. The ConfigurationManager creates corresponding management messages for the registered configuration data and forwards them to the correct set of SWfMS^{MT} instances by sending the messages to the Global ODE Management Queue.

The *Data Access Layer* contains a set of *Data Access Objects* (DAO) classes which realize the communication between the Application Layer components and the external, shared databases.

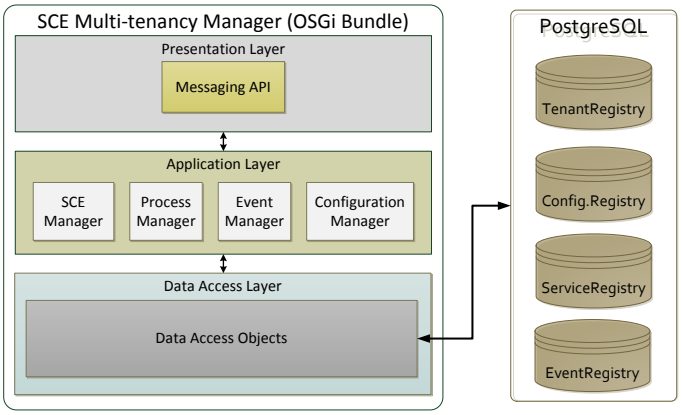

Figure 5.18: Architecture of SCE-MT Manager

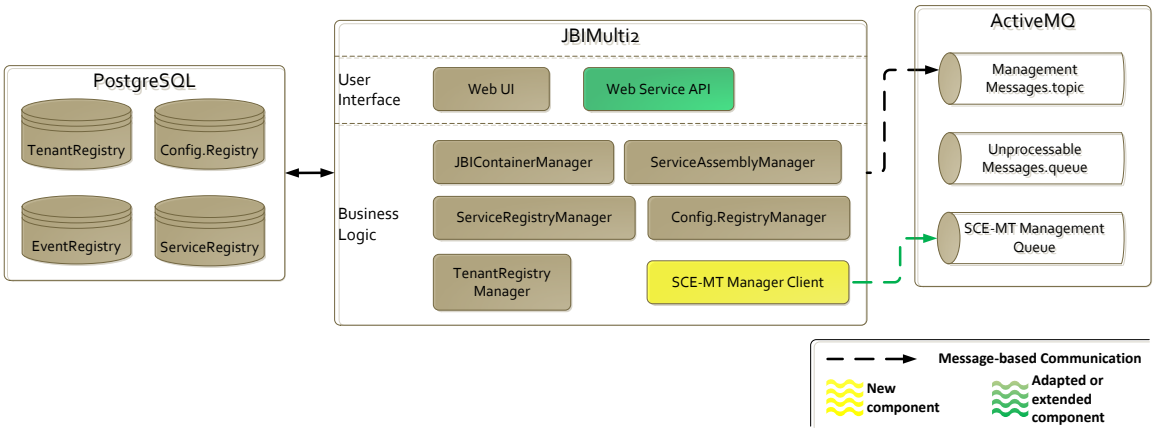

Figure 5.19: Extended JBIMulti2 application

5.7 Extensions of the JBIMulti2 application

Figure 5.19 shows the extended architecture of the JBIMulti2 applications. In the context of this diploma thesis the Web Service API will be extended with some SCE and process model specific operations as described in Section 5.3. For example, two new operations to register configuration data for a SCE instance or process models. The use of these new operations is authorized with the existing user roles and the functionality of JBIMulti2 as described by Muhler [Muh12]. Additionally a new component is added to JBIMulti2. The already introduced *SCE-MT Manager Client* sends status messages to a SCE-MT Manager over its SCE-MT Management Queue. Therefore, the client is listening to a subset of the existing and new WS API operations. If one of these operations is called, the SCE-MT Manager Client sends a corresponding status message to the SCE-MT Manager.

6 Conclusion and Future Work

To leverage the full potential of Cloud computing, multi-tenancy awareness is one of the key requirements for applications. This diploma thesis provides corresponding concepts and an implementation approach to realize a multi-tenant aware SCE. In Chapter 2 all needed fundamentals which are used in this diploma thesis are introduced. After that, in Chapter 3 some related works are introduced. Furthermore the different facets of multi-tenancy like the isolation of tenants or tenant-based configurability are analyzed. The outcomes of Chapter 3 are used as the basis for the following chapters. In Chapter 4 different multi-tenancy aspects of a SCE and its process models are introduced with the focus on configurability, isolation and scalability. After that the necessary behavior of SCEs and process models which provide multi-tenancy support is described. Furthermore some collaboration aspects are introduced which loosen the isolation of tenants in some areas and therefore enable collaboration between tenant users. For example the introduced deployment styles enable tenant users to use process models in a collaborative way. After that, two possible solution approaches to realize a multi-tenant SCE and the underlying functional and non-functional requirements are defined. At the end of the chapter one of the introduced solution approaches is chosen and its integration into an ESB is described. Chapter 5 describes how the abstract SCE^{MT} architecture defined in Chapter 4 can be implemented. Therefore an overall architecture is introduced which consists of the SWfMS with multi-tenancy support, the SCE-MT Manager, ESB^{MT} and JBIMulti2. The usage and possible extensions of each of the components and how they interact with each other are further described. This contains for example the extension of the SWfMS, how ESB^{MT} is used to provide tenant-aware services or the reuse of JBIMulti2 as a management layer for the SCE-MT Manager. The following sections provide some initial ideas for future work and possible extensions of the defined concepts and the realized SCE^{MT} implementation provided by this diploma thesis.

Possible SimTech Architecture with SWfMS^{MT} and ESB^{MT}

Figure 6.1 shows a possible SimTech Architecture with SWfMS^{MT}, JBIMulti2 and ESB^{MT}. First of all a multi-tenant aware JBIMulti2 Management plug-in should be realized for Eclipse. This plug-in should provide a secure and user-friendly graphical interface for the administration and management functionality of JBIMulti2. Users should be authenticated over a single-sign-on mechanism at JBIMulti2 to check their roles and access permissions. In this way the registration of new tenants, configuration data or the deployment of SAs would be much easier for the tenants and their users. The registration of configuration data for a SCE or process models over JBIMulti2 should be possible out of the same tool which is used to define the process models. This is really important because it would ease the configuration process. For

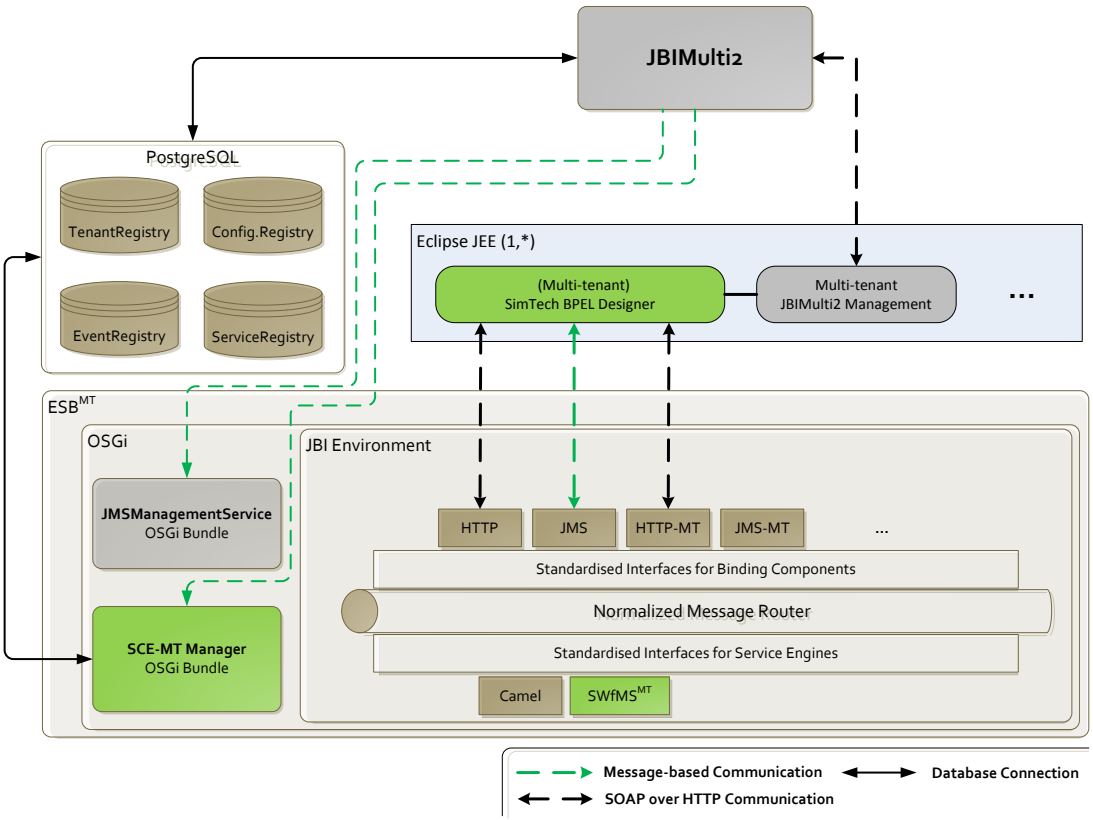

Figure 6.1: Possible SimTech Architecture with ESB^{MT}

example all technical details like the XPath of an variable can be removed from the user. He can just use the SimTech BPEL Designer to point on the variable for which he wants to register configuration data. Additionally the SimTech BPEL Designer has to be extended to support the multi-tenant SWfMS and to use the new communication infrastructure provided by the ESB and SWfMS^{MT}. For example, the deployment of new process models is now handled over JBIMulti2 and not directly by the SWfMS. Therefore the SimTech BPEL Designer can use the functionality of the JBIMulti2 plug-in as shown in Figure 6.1. As described in Chapter 5.3.8 the process and process instance management functionality of SWfMS is provided over a general HTTP endpoint and a JMS queue. The former is deployed to the HTTP BC and the latter is deployed to the JMS BC of the ESB. The tenant-aware HTTP endpoints of the process model Web Service are provided over the HTTP-MT BC as described in Chapter 5.3.6. Therefore the SimTech BPEL Designer should be aware of how the corresponding endpoint URLs are generated based on the tenant context referencing the tenant to which a process model belongs. Another solution would be the retrieval of the correct endpoint URLs over the SCE-MT Manager with the use of messaging. This approach can be also used to get the dynamically created endpoint URL of the Event Topic of a tenant user as described in Chapter 5.3.7.

Furthermore it must be investigated how and if the flexibility functionalities of SWfMS can be used in the new environment. For example the Process Instance Migration functionality is build on top of the Axis2 Integration Layer of Apache ODE. But SWfMS^{MT} uses the JBI Integration Layer and as a result of that, the migration of process instances is not possible with the provided prototype.

Advanced Process Model Configuration

The configuration of process models described in Chapter 4.3 should be extended by separating the specification of possible configurable process elements from the underlying process model. Instead of marking some process elements during modeling time for later configuration, an authorized user should be able to specify the configurable elements of a process model over JBIMulti2 during runtime without the need to change the underlying process model. Therefore the database schema of the ConfigurationRegistry has to be enriched with a new *ConfigurableElements* entity type which will be referenced by a ProcessModel entity. Each process element which should be marked as configurable is specified over its XPath expressions and an optional default value. The registered default values are used during runtime if a tenant user has not specified any configuration data for a configurable process element. The collection of all XPath expressions identifies then all elements of a process model which could be configured by a user. During the registration of process model configuration data over JBIMulti2, this collection of XPath expressions can be provided to the user as a list of possible target elements he can register configuration data for. The user only has to select an XPath expression and register a corresponding value for it.

The registration of process fragments described in Chapter 4.3 should also be realized in a future version. Process fragments would enable a more flexible and powerful configuration of process models for tenant users. As a result that a process fragment directly influences the control flow of a process model by adding new activities, the process modeler should be able to define the valid insertion points for fragments inside the model. This can be realized with a new BPEL Extension Activity which extends the original BPEL Empty activity with the ability to mark the activity as valid insertion point. The resulting process model becomes therefore some kind of process model template which can be refined and extended by any tenant user. The tenant users are able to register their customized logic modeled in process fragments with one of the BPEL Empty activities marked as insertion point. Therefore the registration can be realized again over the corresponding XPath expressions of the target BPEL Empty activity. The BPEL Empty activities are used to mark insertion points because if no process fragment is registered for an insertion point the workflow engine just does nothing. This makes it also possible that tenant users are able to register process fragments for only a subset of the defined insertion points of a process model. Furthermore the SCE implementation has to be extended to dynamically weave in all registered process fragments into the process model by re-compiling the composed process model before it is instantiated.

Providing Dynamic Collaborative Multi-tenancy and Fine-Grained Access Permissions

As introduced in Chapter 4.5, the possibility to dynamically define access permissions on a process model and process instance level would make the collaboration between tenant users more flexible. Therefore tenant users should be able to specify and change access permissions during the runtime of the SCE and its process instances.

The defined deployment styles already enable the static specification of different collaboration possibilities on a process model level during the deployment time. The owner of the process model is not aware of who actually uses his model. Therefore tenant users should be able to dynamically define tenant-based access permission for process models over JBIMulti2 during the whole lifecycle of a model. The registered access permissions are stored in the ConfigurationRegistry and can then be used for the authentication of tenant requests during process instantiation. Since the authentication of incoming requests is handled in the multi-tenant aware endpoints of the services, the corresponding *ProcessHttpSoapConsumerMarshaler* implementations have to be adapted (see Chapter 5.3.6). For each incoming request the associated tenant context has first to be compared with the one in the tenant context XML file of the HTTP SU. If the tenant contexts do not match, all registered tenants with access permissions for the process model must be queried from the ConfigurationRegistry over the unique processId of the model. After that, the responded list of tenant contexts can be compared to the tenant context of the incoming request. If the list contains a matching tenant context the request is forwarded to the SCE. In any other case the request must be rejected by the endpoint.

To enable the dynamic collaboration between tenant users on a process instance level, the definition of fine-grained access permissions should be possible. An example scenario how the collaboration on a process instance level looks like is described in Chapter 4.5. The fact that all management messages are authenticated by the SCE itself makes it possible to support fine-grained access permissions on a method level. This means tenant users are able to specify which method of the Management Interface a tenant user can invoke. The registration of access permissions for process instances should also be possible over JBIMulti2. Therefore the same mechanism as introduced for the forwarding of configuration data in Chapter 5.3.4 can be used to register fine-grained access permissions over JBIMulti2. These access permissions are then forwarded over the SCE-MT Manager to the engine-internal ConfigurationDB of the SCE instance. The engine could then use this data to authenticate and authorize incoming management requests based on the data stored in the ConfigurationDB. Furthermore JBIMulti2 should provide some predefined user roles which can be used by tenant users to define access permissions for a process instance. For example an “Instance Administrator” role which combines all available access permissions and is allowed to assign access permissions for a process instance to other tenant users. As a result each tenant user which has the Instance Administrator role can execute all methods of the Management Interface of the SCE.

Bibliography

[Abr74] J.-R. Abrial. Data Semantics. In *IFIP Working Conference Data Base Management*, pp. 1–60. 1974. (Cited on page 69)

[ALMS09] T. Anstett, F. Leymann, R. Mietzner, S. Strauch. Towards BPEL in the Cloud: Exploiting Different Delivery Models for the Execution of Business Processes. In *Proceedings of the International Workshop on Cloud Services (IWCS 2009) in conjunction with the 7th IEEE International Conference on Web Services (ICWS 2009), Los Angeles, CA, USA, July 10, 2009*, pp. 670–677. IEEE, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-41&engl=0. (Cited on pages 29, 31 and 32)

[BPE07] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (Cited on pages 18 and 19)

[CC06] F. Chong, G. Carraro. Architecture Strategies for Catching the Long Tail, 2006. URL http://msdn.microsoft.com/en-us/library/aa479069.aspx. (Cited on pages 29, 30, 31, 32 and 107)

[CCW06] F. Chong, G. Carraro, R. Wolter. Multi-Tenant Data Architecture, 2006. URL http://msdn.microsoft.com/en-us/library/aa479086.aspx. (Cited on pages 32 and 33)

[Cha04] D. A. Chappell. *Enterprise service bus*. Theory in practice. O'Reilly, Beijing and Cambridge, 2004. (Cited on pages 15 and 16)

[EB09] M. Eckert, F. Bry. Complex event processing (CEP). *Informatik-Spektrum*, 32(2):163–167, 2009. (Cited on page 67)

[Ess11] S. Essl. *Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support*. Masterarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-3166&engl=0. (Cited on pages 9 and 24)

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

Conference on, pp. 551–558. 2007. doi:10.1109/CEC-EEE.2007.4. (Cited on pages 29, 31, 32 and 34)

[Hol95] D. Hollingsworth. *Workflow Management Coalition: The Workflow Reference Model*. The Workflow Management Coalition, 1995. (Cited on pages 17, 18 and 37)

[Hot10] S. Hotta. *Ausführung von Festkörpersimulationen auf Basis der Workflow Technologie*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3029&engl=0. (Cited on page 10)

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004. (Cited on page 26)

[JBI05] Java™Business Integration (JBI) 1.0, Final Release, 2005. URL http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html. (Cited on pages 22, 23 and 107)

[KDS+12] D. Karastoyanova, D. Dentsas, D. Schumm, M. Sonntag, L. Sun, K. Vukojevic. Service-based Integration of Human Users in Workflow-driven Scientific Experiments. In *Proceedings of the 8th IEEE International Conference on eScience (eScience 2012)*, pp. 1–8. IEEE Computer Society Press, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-39&engl=0. (Cited on page 10)

[KMK12] R. Krebs, C. Momm, S. Kounev. Architectural Concerns in Multi-tenant SaaS Applications. In *Proceedings of the 2nd International Conference on Cloud Computing and Service Science (CLOSER'12)*, pp. 426–431. 2012. (Cited on pages 29, 31 and 32)

[LR00] F. Leymann, D. Roller. *Production workflow: concepts and techniques*. Prentice Hall PTR, 2000. (Cited on pages 17 and 37)

[MG11] P. Mell, T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology (NIST), 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. (Cited on pages 16 and 17)

[Muh12] D. Muhler. *Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3226&engl=0. (Cited on pages 9, 25, 26, 27, 28, 67, 69, 70, 71, 73, 81, 98 and 107)

[Nin11] B. Ning. *Iteration und wiederholte Ausführung von Aktivitäten in Workflows*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3096&engl=0. (Cited on pages 10 and 21)

[OSG12] The OSGi Alliance. OSGi Core Release 5, 2012. URL http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf. (Cited on page 24)

[Por08] Java^{TM}Portlet Specification 2.0, 2008. URL http://jcp.org/aboutJava/communityprocess/final/jsr286/index.html. (Cited on page 40)

[PPKW11] M. Pathirage, S. Perera, I. Kumara, S. Weerawarana. A Multi-tenant Architecture for Business Process Executions. In *Web Services (ICWS), 2011 IEEE International Conference on*, pp. 121–128. 2011. doi:10.1109/ICWS.2011.99. (Cited on page 34)

[RBKK12] M. Reiter, U. Breitenbücher, O. Kopp, D. Karastoyanova. Quality of Data Driven Simulation Workflows. In IEEE, editor, *2012 8th IEEE International Conference on eScience*. IEEE Computer Society, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-42&engl=0. (Cited on page 10)

[Rut09] J. Rutschmann. *Generisches Web Service Interface um Simulationsanwendungen in BPEL-Prozesse einzubinden*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2895&engl=0. (Cited on page 10)

[Sáe13] S. G. Sáez. Integration of Different Aspects of Multi-Tenancy in an Open Source Enterprise Service Bus. Studienarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2394&engl=0. (Cited on pages 9, 25, 28, 73, 81 and 84)

[SALM12] S. Strauch, V. Andrikopoulos, F. Leymann, D. Muhler. ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'12)*, pp. 456–463. IEEE Computer Society Press, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-46&engl=0. (Cited on pages 9, 11, 12, 13, 29, 32, 33 and 107)

[Sch11] T. Schliemann. *Unterstützung des "Model-as-you-go"-Ansatzes durch Modell-Versionierung und Instanzmigration*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3121&engl=0. (Cited on pages 10 and 21)

[SCLGK10] M. Sonntag, N. Currle-Linde, K. Görlach, D. Karastoyanova. Towards Simulation Workflows With BPEL: Deriving Missing Features From GriCoL. In R. Alhajj, V. Leung, M. Saif, R. Thring, editors, *Proceedings of the 21st IASTED International Conference on Modelling and Simulation (MS 2010), 2010.* ACTA Press, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-26&engl=0. (Cited on page 10)

[SOA13] The Open Group. Service-Oriented Architecture, 2013. URL http://www.opengroup.org/subjectareas/soa. (Cited on page 15)

[Ste08] T. Steinmetz. *Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2729&engl=0. (Cited on pages 21 and 73)

[Tol11] A. Tolev. *Aufruf und visuelle Korrelation von wissenschaftlichen Workflows in einem Workflow Modellierungswerkzeug.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3227&engl=0. (Cited on page 10)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more.* Prentice Hall, 2005. (Cited on page 15)

[WMTJ12] S. Walraven, T. Monheim, E. Truyen, W. Joosen. Towards performance isolation in multi-tenant SaaS applications. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, p. 6. ACM, 2012. (Cited on page 34)

[WTJ11] S. Walraven, E. Truyen, W. Joosen. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In F. Kon, A.-M. Kermarrec, editors, *Middleware 2011*, volume 7049 of *Lecture Notes in Computer Science*, pp. 370–389. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-25821-3_19. URL http://dx.doi.org/10.1007/978-3-642-25821-3_19. (Cited on pages 29 and 31)

All links were last followed on November 7, 2013.

List of Figures

List of Listings

Decleration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature