Institut für Architektur von Anwendungssystemen

Diplomarbeit Nr. 3614

On-Demand Provisioning of Services

Nedim Karaoğuz

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dipl.-Inf., Dipl.-Wirt. Ing. (FH) Karolina Vukojevic-Haupt
Begonnen am:	08.01.2014
Beendet am:	09.07.2014
CR-Klassifikation:	C.2.4, D.2.11, H.3.5, H.5.3

Abstract

Workflows and service oriented computing (SOC) are an integral part of today's business scenarios. The SimTech project aims to leverage these proven technologies in the context of scientific research. Yet, this field of eScience has different requirements on SOC than their business counterparts. One of these differences is, that services and resources needed by scientists are commonly only required for very specific amounts of time and do not need to follow the always-on principle of traditional SOC. Thus, a means is necessary to make services and resources available when required and also free them again as soon as they are no longer needed. As a solution to utilize SOC in eScience scenarios, SimTech promotes the use of Cloud Technologies to enable the on-demand provisioning of services and their necessary infrastructure.

This diploma thesis is focused on describing different architectural concepts and designs that enable the on-demand provisioning of services and their underlying infrastructure and middleware. These designs and concepts aim to strike a middle ground between abstract high-level architectures and very low-level architectures that focus solely on software specifics. The concepts have been designed in context of a Scientific Workflow Management System. A prototypical implementation that demonstrates the developed concepts concludes this thesis.

Contents

List of Abbreviations

API	Application Programming Interface
Axis2	Apache extensible interaction system 2
BC	Binding Component
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
BPMS	Business Process Management System
BWR	Bootware Remote
CMP	Custom Message Processor
CSAR	Cloud Service ARchive
DAO	Data Access Object
DBMS	DataBase Management System
DSL	Domain Specific Language
EAI	Enterprise Application Integration
EIP	Enterprise Integration Pattern
EP	Endpoint
EPR	Endpoint Reference
ER-Diagram	Entity Relationship Diagram
ESB	Enterprise Service Bus
HTTP	HyperText Transfer Protocol
IA	Implementation Artifact
IaaS	Infrastructure as a Service
IAAS	Institute of Architecture of Application Systems, University of Stuttgart
IL	Integration Layer
IRI	Internationalized Resource Identifier
JAR	Java ARchive
JBI	Java Business Integration
JDBC	Java DataBase Connectivity
JMS	Java Message Service
JMX	Java Management eXtensions
MAR	Axis2 Module Archive
MEP	Message Exchange Pattern
MOM	Message Oriented Middleware
NaaS	Network as a Service
NCName	Non-Colonized Name
NIST	National Institute of Standards and Technology
NMR	Normalized Message Router
ODE	Orchestration Director Engine
ODE-PGF	ODE with Pluggable Framework for extended BPEL behavior
OSGi	Open Service Gateway initiative
PaaS	Platform as a Service
POJI	Plain Old Java Interface
POJO	Plain Old Java Object
QName	Qualified Name

QoS	Quality of Service
SA	Service Assembly
SaaS	Software as a Service
SE	Service Engine
SimTech	Excellence Cluster Simulation Technology
SOA	Service-Oriented Architecture
SOAP	(SOAP is no abbreviation)
SOC	Service Oriented Computing
SPR	Service Package Repository
SR	Service Registry
SU	Service Unit
SWfMS	Scientific Workflow Management System
TOSCA	Topology and Orchestration Specification for Cloud Applications
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
W3C	World Wide Web Consortium
WAR	Web application ARchive
WS	Web Service
WSDL	Web Service Description Language
XML	eXtensible Markup Language
XPath	XML Path Language

1 Introduction

In the cloud computing world the term “on-demand provisioning” or “dynamic on-demand provisioning” is often not tied to the provisioning of services directly but rather to the underlying infrastructure and resources required to operate a service (see also chapter 3 “Related Work”). Although services may be deployed on-demand through means like TOSCA, the deployment process is still triggered by an individual user [TOS13]. This is not a problem in traditional Web Services-SOA, since services are designed to follow the always-on principle and are permanently available to clients after their initial deployment [Pap08]. The same aspect also applies to business processes or workflows, which again, are normally designed once and not altered very often. However when utilizing workflow technology to model an experiment in a scientific environment, processes are redesigned multiple times and services are infrequently used [GSK+11].

The scope of this thesis encompasses the conceptualization of a system to enable an automatic and on-demand provisioning and deprovisioning of services, as well as a prototypical implementation of such a system.

1.1 Motivation

On-demand provisioning of services facilitates the usage of web services by a grand margin. It is no longer necessary to deploy and manage services by hand. Instead, the dynamic binding functionality of a service bus is enhanced by also taking care of provisioning the service and its underlying infrastructure. In the field of eScience, where services are used infrequently in contrast to business scenarios, on-demand provisioning of services with subsequent deprovisioning enables the utilization of Cloud resources for a variety of application and computing scenarios.

1.2 Outline

- Chapter 1: General introduction, motivation and conventions.
- Chapter 2: Introduction of various technologies, standards and software relevant to this thesis.
- Chapter 3: Overview of other scientific works which are related to the general scope of on-demand provisioning of services in various settings and environments.

- Chapter 4: Previous work that forms the basis of this diploma thesis.
- Chapter 5: Modifications and additions to previous work.
- Chapter 6: Core concepts and designs for realizing on-demand provisioning of services.
- Chapter 7: Overview of different aspects of the prototypical implementation.
- Chapter 8: Summary and outlook on future works and subjects still requiring additional research.

1.3 Conventions

The following typographical conventions are used in this thesis:

- *Italic*: Indicates new or important terms and/or terms directly related to a figure or listing in whose context these terms are used.
- `Constant width`: Indicates some kind of source code.

2 Fundamentals

In this chapter basic explanations about concepts and technologies are given that are relevant to this thesis and help to understand the later chapters.

2.1 Cloud Computing

The term Cloud Computing has been a buzzword for several years now. Companies are moving their applications and systems away from running on individual servers and into the Cloud. This enables application scenarios that would have previously required a large amount of resources and money, even for smaller businesses. Cloud service providers offer virtualized computing resources or services that may span over multiple physical machines. Services may be anything between single applications and whole infrastructures that are available over a network. The virtualization process is completely hidden from the client requesting the resources. This process also enables features like automatic scaling. If suddenly more resources are required, they can be allocated dynamically. The client is generally only billed for the actual resources he used, because in times of low resource consumption, these can be utilized by other clients. This results in a near perfectly efficient use of the Cloud provider's infrastructure.

The National Institute of Standards and Technology (NIST) defines Cloud Computing with the five "essential characteristics" [NIS11]. *On-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity* and *measured service*. *On-demand self-service* means that there is no human interaction required to use the provided capabilities of a service provider. *Broad network access* defines that all capabilities are available over a network and that these can be accessed through standard mechanisms. *Resource pooling* is the fact that the provided computing resources are pooled and served to a multitude of consumers. The required physical and virtual resources are dynamically assigned and reassigned depending on consumer demand. The consumer generally has no sense of the exact location of the actual used resources. These resources and capabilities appear to be unlimited to the consumer, since they can be elastically provisioned and released at any time and in some cases automatically. This is defined as *rapid elasticity*. The term *measured service* means that the resource use is automatically controlled and optimized. Additionally it can be monitored, controlled and reported to provide further information for both consumer and service provider [NIS11].

NIST also defines three fundamental service models, offering various degrees of abstraction and how much control a consumer has over the service: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS). Today there exist many more types of service models that generally are a subclass of one of the fundamental service models, e.g. a *Network as a Service* (NaaS) would be a kind of IaaS offering.

The *Infrastructure as a Service* model is the most basic of the three models. This model most closely resembles a traditional server, where the consumer is able to run any kind of software, like operating systems and applications and configure them freely on the provided system but has no control over the underlying cloud infrastructure. For example, IaaS models include virtual machines or storage offerings.

The next level of abstraction is the *Platform as a Service* model. On a PaaS the abstraction starts at an environment that is preconfigured by the provider. The consumer is still able to deploy its own applications, but has no longer control over the operating system or other dependencies like libraries and tools. Real world examples of PaaS offerings are Google App Engine[1] and Microsoft Azure[2], which offer runtime environments or middleware.

A *Software as a Service* model offers the highest level of abstraction. In this model, a consumer directly uses applications offered by the provider, for example through a web browser or other thin client interface. The consumer has only limited control over the application settings and none over its underlying software stack or infrastructure. There are a variety of different SaaS offerings found on the web. For example Google services like Docs[3] or Drive[4].

2.2 Service-Oriented Architecture

Weerawarana et al. define that *Service-Oriented Architecture* (SOA) describes an abstract architectural concept that enables the interaction between different services that are based on the principles of loose coupling and dynamic binding [WCL⁺05]. It should be noted that SOA is not exclusive to a specific technology like web services. The basic principles of SOA are illustrated in the following Figure 2-1.

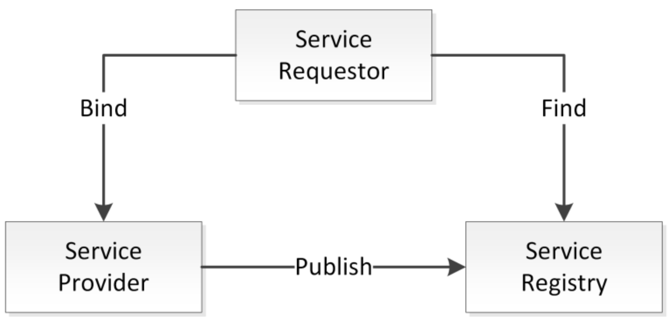


Figure 2-1: The SOA Triangle

[1] https://developers.google.com/appengine/
[2] http://azure.microsoft.com
[3] https://docs.google.com
[4] https://drive.google.com/

A *Service Provider publishes* a definition of his service to a central *Service Registry*. This definition includes functional and nonfunctional capabilities of the service and also information of how a consumer is able to *bind* to this service. A *Service Requestor* instructs the Service Registry to *find* a service that matches his requirements. The Service Registry then tries to match the provided requirements of the requestor with the capabilities of a service. After a service has been found, the *Service Requestor* now possesses all the required information to *bind* himself to the matched service and use it [WCL+05].

Figure 2-2 illustrates how this process can be facilitated, especially for the Service Requestor, by introducing a *Service Bus*. The Service Bus acts as a middleware between the Service Requestor on the one side and Service Provider and Registry on the other side. A requestor simply sends his data and description of the service he intends to use to the Service Bus. This description does not target a specific service. The requestor merely conveys information about his functional and non-functional requirements to the bus. The bus selects the appropriate service, transforms the original request message if necessary and passes the response from the service back to the Service Requestor [WCL+05].

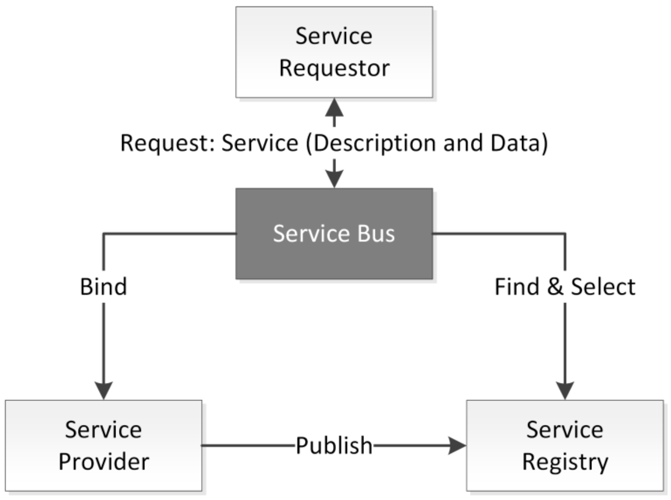


Figure 2-2: SOA Triangle with Service Bus

Chappell describes the Enterprise Service Bus as the “implementation backbone” for a SOA [Cha04]. This approach is followed in this thesis, as an Enterprise Service Bus is the key component of the overall architecture introduced later in this thesis in chapter 4 “Previous Work”.

2.3 Enterprise Service Bus

Chappell in [Cha04] describes in detail different characteristics and features of an Enterprise Service Bus (ESB). An Enterprise Service Bus provides all necessary mechanisms for loosely coupled interactions between different applications. All applications that are connected to the bus no longer need to worry about data transformation or losing messages. If an application wants to communicate with the service bus, only an adapter has to be used to let the application connect to the bus in its native way; meaning that the integration of an existing application does not need to be changed. This is illustrated in Figure 2-3 where different components are integrated by the ESB via adapters that allow various technologies to connect to the Service Bus. Both components on the bottom of Figure 2-3 may for example be Web Services that are communicating over HTTP. So they are both integrated by HTTP-adapters (marked with an X in Figure 2-3) whose endpoints are exposed by the ESB so the Web Services themselves are unaware that they are communicating through a Service Bus. The other component in the top half of Figure 2-3 may be a custom Java application that communicates via POJOs so the corresponding adapter (marked with a Y in Figure 2-3) has to receive the objects and transforms them into the bus.

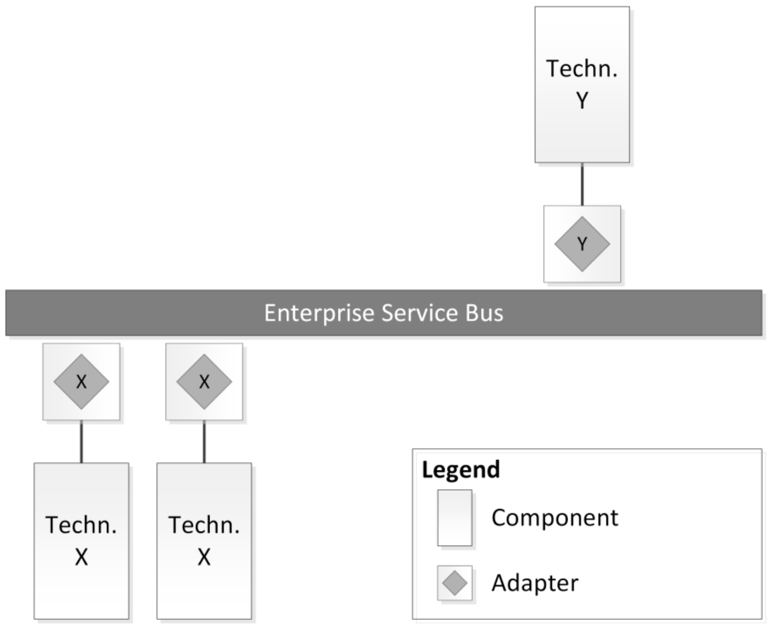


Figure 2-3: ESB integrating components based on different technologies

An ESB both separates application and integration logic and provides a distributed integration. This is in contrast to other integration approaches like traditional EAI, using an Application Server or a Message Oriented Middleware (MOM). While a MOM also provides the ability to connect applications in a loosely coupled and asynchronous fashion in contrast to hub-and-spoke architectures, applications need low-level coding in order to be able to interact with the MOM. This means that the integration logic is hardwired with the application logic. An ESB allows for clear separation of integration and business logic. Furthermore an ESB architecture may span across multiple physical networks by interconnecting multiple buses [Cha04].

Chappell sums up the characteristics of an ESB as being pervasive, meaning that an ESB can be used in a variety of integration situations and projects. Being a highly distributed and event-driven SOA and allowing for a selective deployment of integration components. All these components can take advantage of reliable messaging and other security and reliability features. The data on the bus is allowed to flow across any application or service that is connected to the ESB thus allowing for orchestration of process flows. An ESB has an autonomous yet federated managed environment and allows for incremental adoption even in smaller projects that may later transferred into a larger integration network. Finally an ESB uses XML as its native datatype and allows for real-time insight into live business data [Cha04].

2.4 OSGi

Bartlett describes OSGi (Open Service Gateway initiative) as being "nothing more nor less than the way to build modular applications in Java" [Bar09]. OSGi is targeted towards a multitude of devices that are running Java applications; like set-top boxes, PCs, cars and mobile phones and which require a modular architecture. The standard is maintained by the OSGi Alliance[5] and split into multiple specifications [OSG09]. The core of the OSGi Service Platform Specifications is formed by the framework. Its goal is to provide a general-purpose, secure and managed Java framework that supports the deployment of bundles, which are extensible and downloadable applications. A bundle is a Java Archive (JAR) that defines a unit of modularization. It contains Plain Old Java Objects (POJOs) and Plain Old Java Interfaces (POJIs) as well as additional meta-information about exported and imported packages. For example, an exported POJI provides a service interface to other bundles. These may then in turn provide a service implementation of that interface. The functionality of the framework is divided into multiple layers that are shown in Figure 2-4.

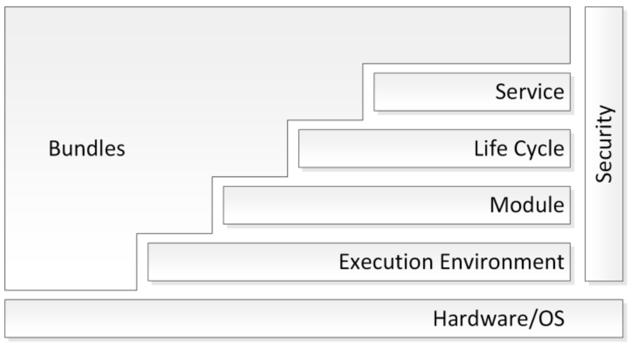


Figure 2-4: OSGi Layer Architecture [OSG09]

Each layer in Figure 2-4 is dependent on its subjacent layer. The lowest layer describes the *Hardware and Operating System* Layer on top of which an *Execution Environment*

[5] http://www.osgi.org

runs, i.e. the Java Virtual Machine. To the right of Figure 2-4 is the *Security* Layer. It handles different security aspects of bundles. The *Module* Layer is tasked with the modularization of Java packages. This means that the Module Layer has a set of rules for sharing Java packages between bundles or hiding them from others. The *Life Cycle* Layer provides an API for life cycle management to bundles. This includes starting and stopping bundles, as well as updating, installing and uninstalling. Furthermore the Life Cycle Layer provides an event API. The *Service* Layer provides developers with a dynamic service selection model and loose coupling. Java interfaces are decoupled from their implementations, allowing a user to bind to services using only their corresponding interface. The whole selection process is handled by the runtime [OSG09].

2.5 Apache ServiceMix

Apache ServiceMix[6] is the realization of a lightweight open source Enterprise Service Bus. It is built on Apache Karaf[7], an enhanced version of the Apache Felix[8] Framework, and supports the Java Business Integration (JBI) standard [JBI05]. Apache Felix is an OSGi framework and compliant to version 4.2 of the OSGi specification [OSG09]. The version of ServiceMix is explicitly referenced, because newer versions of ServiceMix are bundled with different components and no longer support JBI.

In Apache ServiceMix applications can be deployed in several ways and then operate in a loosely coupled manner. Apache Karaf provides the user with a command line tool to manage the ESB environment. It supports various lifecycle operations like deploying and configuring OSGi bundles or JBI components and Service Assemblies (SA). Service Assemblies are collections of deployment artifacts, so called Service Units (SU), and associated metadata. Additionally to deploying applications through the command line, ServiceMix provides a hot deployment directory. Users can simply deploy OSGi bundles or JBI Service Assemblies by adding them to this directory. If the file containing the bundle or assembly is deleted, ServiceMix will automatically undeploy the corresponding application. The core of ServiceMix is the *Normalized Message Router* (NMR) that resides inside the JBI container, see Figure 2-5.

[6] http://servicemix.apache.org/
[7] http://karaf.apache.org/
[8] http://felix.apache.org/

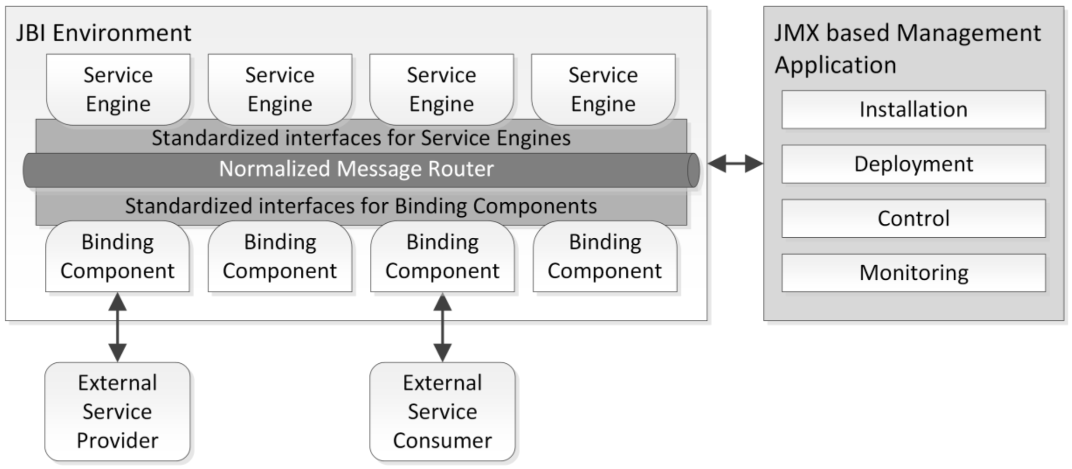

JBI Environment
Service Engine
Service Engine
Service Engine
Service Engine
Standardized interfaces for Service Engines
Normalized Message Router
Standardized interfaces for Binding Components
Binding Component
Binding Component
Binding Component
Binding Component
External Service Provider
External Service Consumer
JMX based Management Application
Installation
Deployment
Control
Monitoring

Figure 2-5: Architecture of the JBI System, based on [NMR08]

The NMR provides the necessary message exchange infrastructure for components that are connected to it. Components come in two categories: As a *Service Engine* (SE) or *Binding Component* (BC). A *Service Engine* provides business logic and transformation services to other components that are connected to the NMR. SEs are shown above the NMR in Figure 2-5. *Binding Components* are displayed below the NMR. BCs provide connectivity to services that are external to the JBI environment. BCs offer support for many different protocols and transform messages accordingly which are going in and out of the JBI environment. Both SEs and BCs can act as a service provider, service consumer or both. A service provider consumes the functionality of an internal JBI service and exposes it through an external endpoint. A service consumer works in the other direction, as it provides the functionality of an external service to other components on the NMR. The *JMX* (Java Management eXtensions) *based Management Application* provides different maintenance functionalities for the JBI environment. The application follows the JMX standard defined in [JMX00].

The ServiceMix distribution entails several components that are pre-deployed in OSGi or JBI respectively. Of these components the Apache ODE[9] and Apache Camel[10] are of high relevance to this thesis. Apache ODE is introduced in the following chapter 2.7 and Apache Camel in chapter 2.9.

2.6 ESB^{MT}

Strauch et al. describe in [SAL⁺12] and [SAG⁺13] how to enable multi-tenancy in Enterprise Service Buses. Multi-tenancy means that a single instance of a software application serves multiple clients (tenants). This, together with virtualization, is an important feature in Cloud Computing as it enables Cloud providers to utilize their infrastructures to their full extent. The abbreviation ESB^{MT} therefore stands for multi-

[9] http://ode.apache.org/
[10] https://camel.apache.org/

tenant aware Enterprise Service Bus. Strauch et al. identify and analyze different requirements a multi-tenant aware ESB has to fulfill, describe an ESB architecture that meets these requirements and realize a prototype implementation of the proposed architecture, called ESB^{MT} [SAL^{+}12]. ESB^{MT} is an ongoing research project at IAAS[11]. The implementation is based on Apache ServiceMix 4.3.0 and its JBI environment. ESB^{MT} is used in the implementation part of this thesis.

2.7 WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL or just BPEL) is an extensible workflow-based language that enables choreographing and aggregating service interactions. BPEL is defined in [WSB07]. It is XML based and was created to address the requirements of composition in a service-oriented computing environment without being locked into static environments with proprietary tools and languages [WCL^{+}05].

BPEL was designed to layer on top of other WS-* specifications so that it can use WSDL interfaces to define the functionality a process provides as well as the requirements of services that are part of the composition and the interactions between them. Since BPEL is XML based, data access and manipulation is possible via using XPath expressions [XPA99] [WCL^{+}05].

The standard was designed to meet the following requirements that have been identified for a composition model in such a SOA environment: *Flexible integration*, *recursive composition*, *separation and composeability of concerns*, *stateful conversations and lifecycle management*, and *recoverability*. *Flexible integration* defines, that the model must be able to express business scenarios that different partners might exchange. Additionally, the model must be able to adapt rapidly to possible changes to the services of the composition. *Recursive composition* is the ability of the model to provide different views on a composition. Interaction between multiple workflows as well as scalability and reuse is met by offering a process as a standard Web Service. In using the Web Services Framework, the specific business logic should be decoupled from the underlying or supporting stack, like protocols, messaging frameworks or quality of service. This is called *separation and composeability of concerns*. *Stateful conversations and lifecycle management* means, that a lifecycle model of a workflow should be clearly defined and able to support multiple long running conversations with interacting services. Especially long running business processes have to provide fault handling and compensation mechanisms in order to deal with expected errors that might appear during process execution time. This is summarized under the term *recoverability* [WCL^{+}05].

BPEL is used in this thesis to implement certain parts of the ESB Control Flow (see also chapters 5.7.2 and 5.7.4), e.g. commissioning the Provisioning Manager with provisioning or deprovisioning of services.

[11] http://www.iaas.uni-stuttgart.de/esbmt/

2.8 Apache ODE

The Apache Orchestration Director Engine (ODE) is an open source WS-BPEL Process Engine and Business Process Management System (BPMS). It executes business processes that are following the WS-BPEL standard [WSB07]. ODE handles all web service calls, message sending and receiving, as well as data manipulation and error recovery that are defined in a business process. It supports both synchronous and asynchronous processes and is thus able to cope with long running process executions. The ability of managing long running business processes as well as providing a reliable, compact and embeddable component, were key goals in the development of Apache ODE [ODA13].

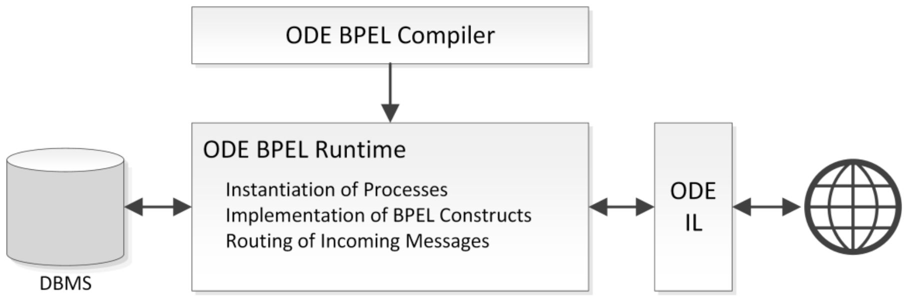

ODE BPEL Compiler
ODE BPEL Runtime
Instantiation of Processes
Implementation of BPEL Constructs
Routing of Incoming Messages
ODE IL
DBMS

Figure 2-6: Apache ODE Architecture [ODA13], simplified

In Figure 2-6 a simplified view on the high-level architecture of Apache ODE is shown. On top is the *ODE BPEL Compiler* which is responsible for converting source BPEL artifacts, like BPEL process documents and WSDLs, into a compiled format that is ready for processing by the *ODE BPEL Runtime*. The runtime executes the compiled processes and creates new instances of processes as necessary. The ODE BPEL Runtime is also able to determine to which instance an incoming message should be delivered and implements a Process Management API that can be used to interact with the engine.

Data Access Objects (DAOs) are used for the interaction between the runtime and a *Database Management System (DBMS)* acting as a data store to achieve persistence. Normally the DBMS is a JDBC relational database. The DAOs include active instances, message routing information, variables, partner links and information about the process execution state.

The ODE BPEL Runtime is unable to exist on its own, because the runtime itself lacks the ability of interacting with the outside world. Thus it relies on an ODE *Integration Layer* (IL). An Integration Layer embeds the runtime in an execution environment and provides channels to communicate with external components. Apache ODE is available with an Axis2 IL or a JBI IL; both variations are of relevance to this thesis. The Axis2[12] Integration Layer allows ODE to communicate through Web Service interactions, while the JBI IL connects ODE to the Normalized Message Router of JBI.

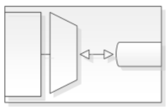
[12] http://axis.apache.org/axis2/java/core/

2.9 Enterprise Integration Patterns

Hohpe and Woolf noticed that many integration problems and their respective solutions are quite similar. They organized and cataloged them as patterns which are published in [HoW10]. Patterns are best practices that have been developed over time. They offer solutions that have been proven and validated in numerous real-world scenarios. Each pattern has a unique icon and is presented in the form of a problem statement, possible solutions, which may have some kind of drawback, and the proposed solution. The following Table 2-1 illustrates various EAI patterns by Hohpe and Woolf that are also used in the different architectural concepts introduced in chapter 6. On the left side the icon that is associated with this pattern is depicted and on the right the name of the pattern including a problem statement and proposed solution are shown. A summary and further information about Enterprise Integration Patterns is also available under [EIP12].

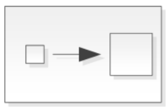
Aggregator
How do you combine the results of individual, but related messages so that they can be processed as a whole?

Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages..



Channel Adapter
How can you connect an application to the messaging system so that it can send and receive messages?

Use a Channel Adapter that can access the application's API or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.

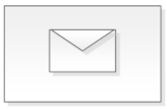


Content Enricher
How do you communicate with another system if the message originator does not have all the required data items available?

Use a specialized transformer, a Content Enricher, to access an external data source in order to augment a message with missing information.

Content Filter
How do you simplify dealing with a large message, when you are interested only in a few data items?

Use a Content Filter to remove unimportant data items from a message leaving only important items.

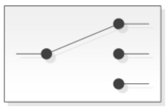


Envelope Wrapper
How can existing system participate in a messaging exchange that places specific requirements on the message format, such as message header fields or encryption?

Use a Envelope Wrapper to wrap application data inside an envelope that is compliant with the messaging infrastructure. Unwrap the message when it arrives at the destination.

Message Endpoint
How does an application connect to a messaging channel to send and receive messages?

Connect an application to a messaging channel using a Message Endpoint, a client of the messaging system that the application can then use to send or receive messages.

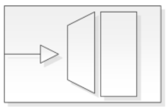


Message Router
How do you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions.

Message Store
How can you report against message information without disturbing the loosely coupled and transient nature of a messaging system?

Use a Message Store to capture information about each message in a central location.



Messaging Gateway
How do you encapsulate access to the messaging system from the rest of the application?

Use a Messaging Gateway, a class that wraps messaging-specific method calls and exposes domain specific methods to the application.

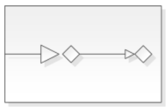
	Multicast[13] [14] How can you route a message to a number of endpoints at the same time? Use a Multicast that allows you to route the message to a number of endpoints and process it in different ways.
	**Service Activator** How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques? Design a Service Activator that connects the messages on the channel to the service being accessed.

Table 2-1: EAI Patterns [HoW10]

2.10 Apache Camel

Ibsen and Anstey describe Apache Camel as a tool that has been created to ease integration for users [IbA11]. Camel allows a user to specify routes between different endpoints and how messages are processed along those routes. Apache Camel is available as a standalone application as well as part of Apache ServiceMix, as it is used in this thesis. Camel is a lightweight open-source integration framework that has been in active development since 2007 and focuses on simplifying application integration [IbA11].

Its core component is a routing-engine builder that allows the user to define his own routing rules as well as how to process, accept and send messages. These can be described via a Domain Specific Language (DSL) in Java, Spring[15], Scala[16] or others. It is also possible to use a combination of the DSL in multiple languages. Therefore Camel offers high-level abstractions to interact with different systems by using a simple API. Camel makes no assumption about the data format of the messages, thus there is no need to convert your messages for routing. Even though it already has a set of built in converters and allows the configuration of custom converters if a message has to be transformed between endpoints. Although Camel supports message routing, transformation and orchestration, Camel in itself is not an Enterprise Service Bus because it lacks ESB specific features like a container or a reliable message bus. But Camel can be deployed inside an ESB. The following example (Listing 2-1) illustrates a simple Camel Route implemented via the Java DSL [IbA11].

[13] Multicast is a pattern defined by Apache Camel and not part of the Enterprise Integration Patterns published by Hohpe and Woolf. In the context of this thesis Multicast is symbolized by the Recipient List icon, because of its similarity.
[14] http://camel.apache.org/multicast.html
[15] http://spring.io/
[16] http://www.scala-lang.org/

```
1   from("direct:d")
2      .choice()
3         .when(header("grilled").isEqualTo("potato"))
4            .to("jms:queue:potato"))
5         .when(header("grilled").isEqualTo("cheese"))
6            .to("jms:queue:cheese"))
7         .otherwise()
8            .to("file:grill/other");
```

Listing 2-1: Example Camel Route in Java DSL

The example routes incoming messages depending on a specific header value to different Java Messaging System (JMS) queues. This route defines an endpoint (line 1) which in this case is a Camel internal endpoint that may be targeted by another route. The lines 2 to 8 describe a conditional expression with three cases. In line 3 and 5 the value of the header field *grilled* is evaluated and the message is either routed to the *potato* JMS queue or the *cheese* queue depending on the header value. If the header evaluates neither to potato nor to cheese the message is saved as a file at the location specified in line 8. This example also illustrates the Content Based Router Pattern of the Enterprise Integration Patterns (EIP) by Hohpe and Woolf integrated through Camel. The Camel framework is heavily based on EIPs and all of them can be easily implemented with Camel [IbA11]. Moreover Camel has defined some additional patterns, like the Multicast[17] or the Delayer[18] Pattern.

2.11 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) standard defines a metamodel for defining services [TOS13]. TOSCA provides a standardized description model of how to manage a service and the service's structure. Management tasks of a service include, e.g. deployment, operation and termination. The structure of a service is called topology. In TOSCA a topology is a graph of typed nodes and directed typed edges. Nodes define individual components of the service and edges describe relationships between these nodes. An example of a TOSCA application topology is shown in Figure 2-7.

[17] http://camel.apache.org/multicast.html
[18] http://camel.apache.org/delayer.html

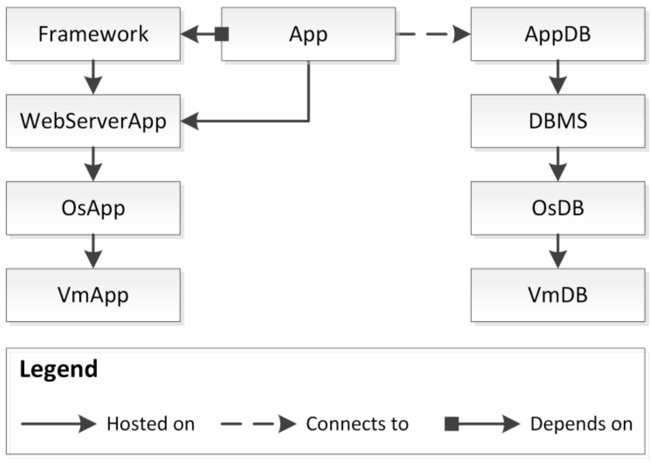


Figure 2-7: TOSCA Example Topology

The example shows a topology consisting of different nodes with three different kinds of relationships. At the bottom level are two virtual machine (VM) nodes representing two different virtual machines, *VmApp* and *VmDb*. An operating system is *hosted on* both machines (see *OsApp* and *OsDB* nodes). On the right, a Databank Management System (*DBMS*) is running on which in turn a database is hosted. On the left, a web server (*WebServerApp*) is installed on which a *Framework* is configured that the main application (*App*) *depends on*. This means that even though both the framework and the application are hosted on the web server, the framework has to be installed first. As the last step, the application has to be configured to *connect to* the database.

Additionally TOSCA defines an archive format for Cloud applications in which all necessary artifacts and information of a service are bundled together. This archive format is called Cloud Service Archive Format (CSAR). A CSAR is a container and usually a zip file with a specified directory structure and defined location of metainformation about the service [TOS13].

2.12 OpenTOSCA

Binz et al. introduce OpenTOSCA[19] in [BBH+13]. OpenTOSCA is an open source TOSCA runtime environment that has been developed at IAAS. OpenTOSCA supports imperative processing of CSARs. This means, that the deployment and management logic of applications has to be provided by plans. Plans are workflow models, for example defined via BPEL. Plan processing is supported by the Plan Engine. The Plan Engine is a modular component, that follows a plugin architecture. This ensures that additional workflow languages can be supported by providing additional plugins to the Plan Engine. Different management operations for nodes and relationships are provided in either one of two ways. The Cloud provider, to which the application is being deployed, offers web services that provide management functionality which can be used in a plan. The other possibility

[19] http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php

is using Implementation Artifacts (IAs) contained in the CSAR. In this case the Implementation Artifact Engine runs these artifacts. The Implementation Artifact Engine's architecture is similar to that of the Plan Engine, as its functionality can be extended by providing additional plugins. For example, an IA can be a web service packaged as a Java Web Application Archive (WAR) that provides certain management functions. To deploy this web service and utilize its functions, a plugin is required that understands where and how to deploy WAR files [BBH+13].

3 Related Work

In contrast to the business world, there is no widespread adoption of cloud computing platforms for scientific computing applications, as Vecchiola et al. point out in [VCK⁺12]. They recognize that the introduction of grid computing was the first step away from using local computing infrastructure of research institutions and provide a solution to the ever growing demand for higher computational power. However the resources provided by a grid may be not enough at certain peak times and be hardly even used at other times. Vecchiola et al. also raise attention to another problem that arises in grid computing and scientific applications. After running for some time the initially reserved resources for an application might prove insufficient to achieve its deadline. To counter this, the authors propose the additional on-demand acquisition of cloud based resources. They envision a hybrid cloud setting in which an application uses both grid and cloud based resources. To achieve this Vecchiola et al. introduce Aneka [VCB09], a software platform for constructing and managing distributed systems. Yet, their focus lies only within the dynamic and on-demand provisioning of resources and not the services themselves.

On-demand provisioning has a very broad interpretation in SOA. In [VKL13] the authors introduce new binding strategies on which this thesis is primarily based (more on this in chapter 4.1). These binding strategies explicitly target a Scientific Workflow Environment in which the necessary software stack and services have yet to be provisioned in a cloud environment.

Albeit not in a Scientific Workflow Environment, [GTS⁺09] and [GTK⁺10] introduce an architecture to enable automatic discovery and provisioning of real-world services. Such services are often found in small embedded devices which, in contrast to traditional SOA, might not always be available or disappear regularly and reappear later on.

In [CBC⁺05] the authors see on-demand SOA as a means to achieve responsiveness and a high degree of flexibility for business processes, where the required services are already deployed, so a business can react quickly and accordingly to new situations that may arise. Thus the authors focus more on a high level view on how to achieve this goal.

4 Previous Work

This chapter introduces the different concepts and strategies published by Vukojevic-Haupt et al. in [VKL13] on which this thesis is primarily based on. In that paper the focus lays on the various requirements eScience applications and scientific workflows have at a SOA in contrast to traditional business applications. Namely these are the on-demand provisioning of services including the provisioning of the underlying software stack required by the service. The authors' approach includes a middleware architecture providing the necessary ecosystem. Additionally different binding strategies have been classified; one of which has been newly developed. The authors show how a generic Workflow Management System (WfMS) can be extended to support the on-demand provisioning and deprovisioning of services and their infrastructure. Vukojevic-Haupt et al. demonstrate their approach via a Scientific Workflow Management System – the SimTech SWfMS[20] [VKL13].

4.1 Binding Strategies

In this chapter different binding strategies are explained. The binding strategies are: *Static Binding*, *Dynamic Binding*, *Dynamic Binding with Service Deployment* and *Dynamic Binding with Software Stack Provisioning*. These strategies need to be integrated with the middleware, i.e. the ESB, to allow for correct service discovery and selection, and subsequent provisioning and deprovisioning of services.

A *Static Binding* occurs if the endpoint of the desired service is already known by the caller. The concrete endpoint is provided to the ESB which in turn directly invokes the specified service and returns its response.

In the case of a *Dynamic Binding* the requestor provides the desired operation including potential parameters but no specific endpoint. Additionally the requestor specifies Quality of Service (QoS) which are functional and non-functional requirements the service must fulfill. Here, the ESB performs a service discovery and selection by consulting a service registry to find and select a service matching the required capabilities specified by the requestor. Afterwards the ESB invokes the service and returns its response to the requestor.

Dynamic Binding with Service Deployment means that the service has to be deployed in addition to the dynamic binding processing steps the ESB needs to perform. From the requestor's point of view it is still a normal dynamic binding strategy.

[20] http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/

The *Dynamic Binding with Software Stack Provisioning* strategy involves the provisioning of the infrastructure, middleware and other applications required by the selected service. The ESB has to trigger the provisioning logic and provide a reference to a service package containing the service topology, among other artifacts required by the service. After the necessary middleware has been provisioned, the service can be deployed and subsequently invoked by the ESB.

4.2 Overall Architecture

Figure 4-1 shows the general architecture of the middleware, its components and services. This architecture is following the SOA concept introduced in chapter 2.2 including a central Enterprise Service Bus. Components in the lower half run locally on the user's machine while components in the top half run in a cloud. As illustrated on the bottom of the figure the architecture is separated into three lifecycle phases: the *Modeling Time* phase, the *Middleware Runtime* phase and the *Service Runtime* phase.

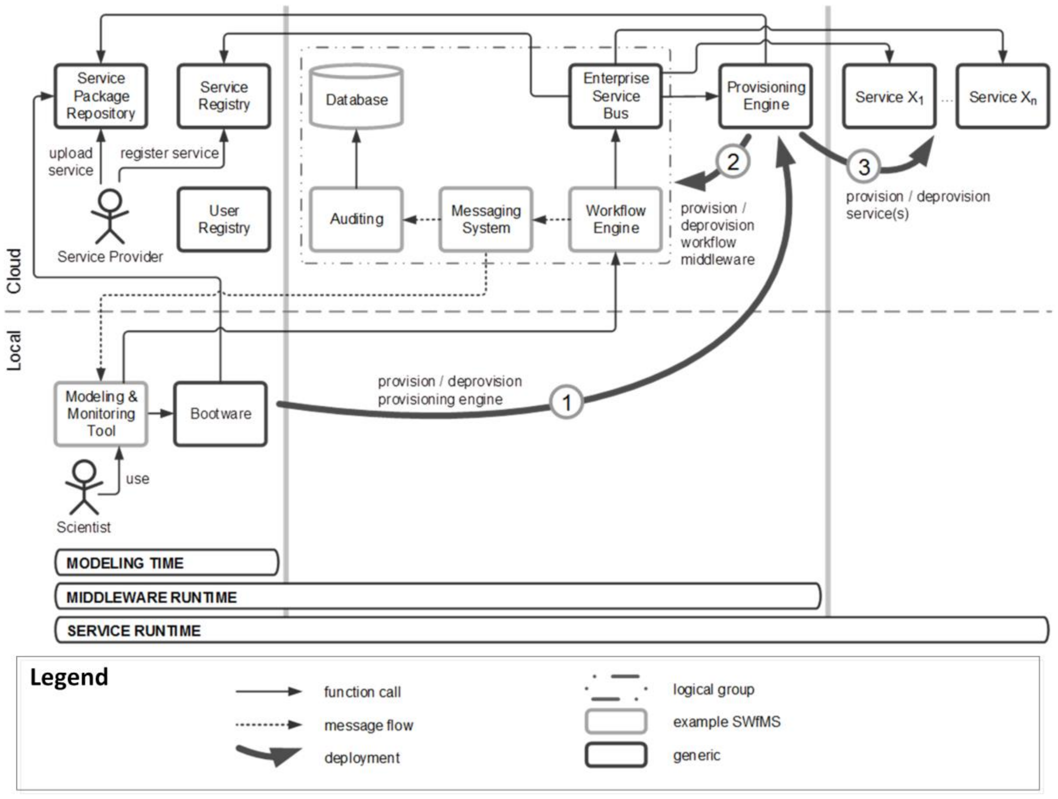


Figure 4-1: Overall System Architecture [VKL13]

During the Modeling Time phase a scientist develops his workflow model using the local *Modeling and Monitoring Tool*. During the execution of his workflow the scientist can use this tool to also monitor the status of running workflow instances. When the scientist triggers the execution of his workflow model, the *Bootware* is started which initiates the

Middleware Runtime phase. This phase includes the on-demand provisioning of the WfMS and its underlying infrastructure. If the ESB detects that a service call from the Workflow Engine targets a not-provisioned service, the Service Runtime phase is triggered. During this phase, the required service is provisioned on-demand along with its underlying infrastructure and middleware. The Service Runtime phase ends with the deprovisioning of the service. After the scientist's workflow model has finished executing, the Middleware Runtime phase ends and the WfMS is deprovisioned.

The *Service Package Repository* stores service packages. A service packages contains all artifacts needed to provision a service, e.g. information about its topology, underlying middleware and infrastructure. This information is used by the *Provisioning Engine* on how to provision, deploy and run the service in a cloud environment. A Service Provider publishes his service packages to the Service Package Repository, for example as a CSAR and also registers the service at the *Service Registry*.

The Service Registry holds information about all available services including functional and nonfunctional properties of services that are available through the Service Package Repository as well as of those that are made available and provided by a third party.

The *Bootware* is software responsible for deploying a Provisioning Engine on a cloud so that this engine may in turn provision the rest of the workflow execution middleware and its underlying cloud infrastructure. More details about the Bootware component are provided in [Rei14].

The authors distinguish between two kinds of services. Provisioned services are classic web services in terms of being always up and ready to use and that their implementation, underlying middleware and infrastructure are a black box to the user. For these types of services a static or dynamic binding is used. Not provisioned services are those that are available through the Service Package Repository and have to be provisioned before they may be utilized. A dynamic binding with software stack provisioning has to be used here. In addition, a not provisioned service can be one of two variants, dedicated or shared.

If a service is a dedicated service, only one service call may be active at the same time. Even if multiple service calls originate from the same user, each call has to have its own separate instance that needs to be provisioned with its underlying middleware and infrastructure. This is due to dedicated services not providing any form of elasticity, which is in contrast to shared services that do provide elasticity functionality. A shared service instance can handle multiple service calls from the same user. Thus a new instance with underlying middleware and infrastructure is only provisioned if the calling user does not already have an active instance of this service.

During the Middleware Runtime phase the ESB and the Provisioning Engine interact with the components used in the previous Modeling Time phase and the remaining Simtech SWfMS components depicted in Figure 4-1. The Workflow Engine forwards the service calls to the ESB to invoke the services requested by workflow activities. If the ESB detects that a not provisioned service is called it triggers the Provisioning Engine which gathers all necessary artifacts, e.g. the service package from the Service Package Repository and provisions the service including its underlying middleware and infrastructure. After the provisioning has completed, the ESB sends the request to the newly provisioned service.

The Service Runtime phase is the part in which a service has received a request and performs its functionality. All components shown in Figure 4-1 have been provisioned and are running. When the service has finished its computation, it sends its result back to the ESB which forwards the response back to the Workflow Engine.

Since the service has performed the specific task and reported back the result, the ESB can now trigger the deprovisioning of the service, as the resources required to operate the service are no longer needed. The ESB issues the Provisioning Engine with the deprovisioning of a particular service. If the service is a shared service it is first checked if this service is still processing other requests. Only if this is not the case, the service will be deprovisioned. The remaining parts of the SWfMS are deprovisioned by the Provisioning Engine as soon as all running workflows have finished. The Provisioning Engine is in turn deprovisioned by the Bootware.

In [Sch13] it is suggested to add an additional component, the Provisioning Manager, which is used to hide the complexity of interacting with multiple Provisioning Engines from the ESB and therefore to provide a stable interface. This idea has been adopted and incorporated into the different concepts developed in the scope of this thesis (see also chapters 5.1 and 5.2 for more details).

4.3 ESB Control Flow

In Figure 4-2 below it is described how the ESB should manage and coordinate the on-demand provisioning and de-provisioning of services.

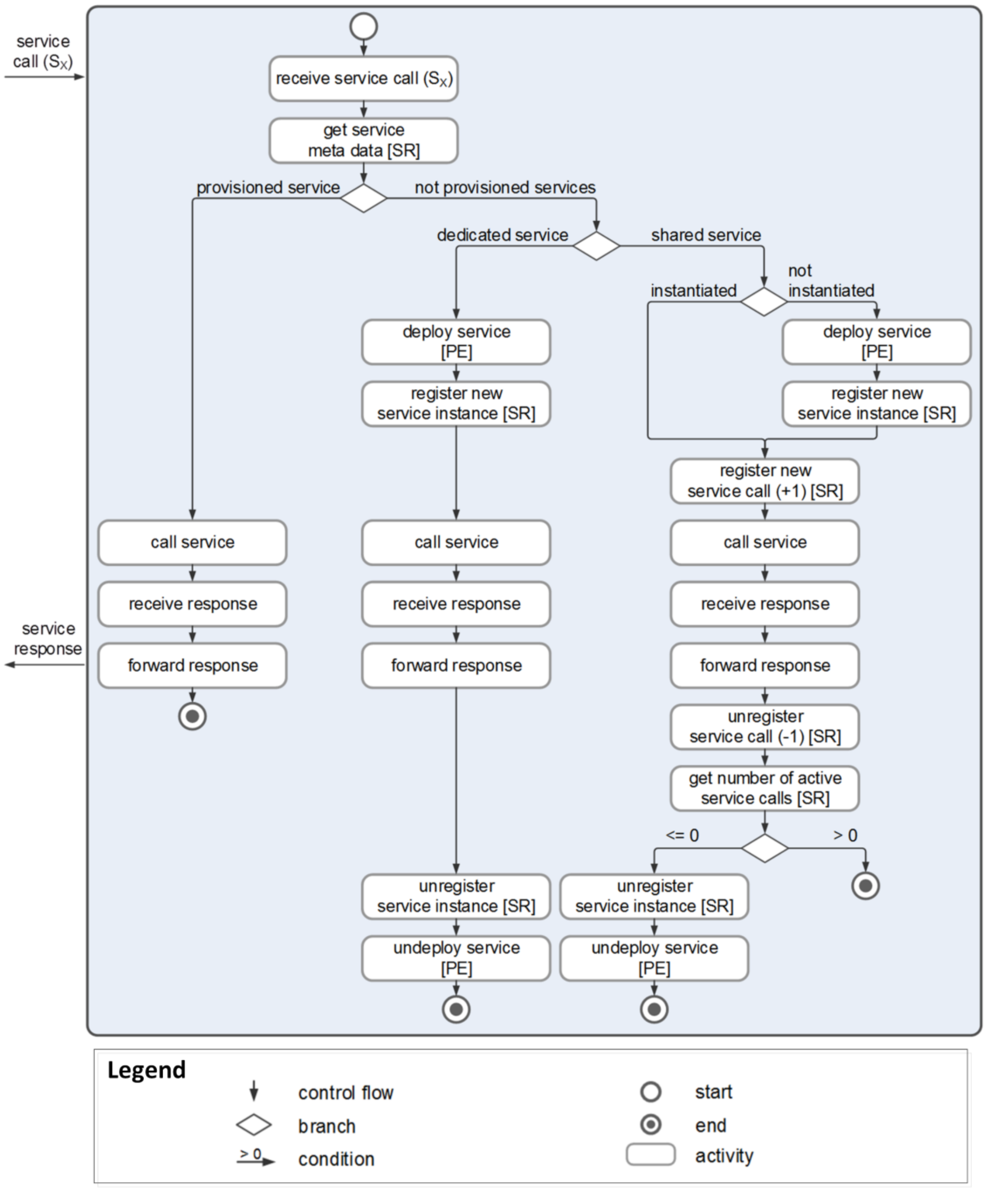


Figure 4-2: ESB Control Flow [VKL13]

The ESB Control Flow is only triggered if a dynamic binding strategy has to be used. If the ESB receives a service call that uses a static binding, the call is directly forwarded to the service and the ESB Control Flow is not activated.

In case the ESB detects that the required service is a provisioned service, the service call is sent to the specific service and its response returned to the Workflow Engine.

If the service call concerns a not provisioned service, the ESB has to distinguish between a dedicated or shared service. In case of a dedicated service, the ESB issues the Provisioning Engine with the provisioning of a new service instance. For this, the ESB passes a reference to the specific service package from the Service Package Repository to the Provisioning Engine. The Provisioning Engine uses this reference to get the actual service package, provisions the service including its underlying middleware and infrastructure and returns the endpoint of the service back to the ESB. Afterwards the ESB registers the newly provisioned service in the Service Registry and forwards the original service call to the service. Once the service has finished processing the service call it reports the result to the ESB, which passes it on to the Workflow Engine. Afterwards the ESB removes the service instance from the Service Registry and orders the Provisioning Engine to deprovision the service with its underlying middleware and infrastructure.

In case of addressing a shared service, the ESB consults the Service Registry if an instance of this service is already running for this user. If not, a new instance is provisioned and the ESB proceeds as described in the previous case. When the ESB identifies that a shared service instance has no more active service calls it triggers the service's deprovisioning.

5 Extending the Current Architecture

In this chapter, along with chapter 6 “Architectural Concepts and Designs”, the core work of this diploma thesis is described. It is analyzed which components of the architecture from chapter 4.2 “Overall Architecture” have been extended or altered and which have been newly added. This includes components that have been developed as part of this thesis, as well as components that fall in the scope of related parallel work at IAAS. It is also discussed how this parallel work influenced different design decisions made in this thesis.

This forms the basis for a more detailed view on the architecture, which is necessary to enable the development of different approaches for the realization of on-demand provisioning of services and their underlying infrastructure. These approaches are discussed in chapter 6 “Architectural Concepts and Designs”.

5.1 Provisioning Manager as an Internal/External Component

To hide the idiosyncrasies of different Provisioning Engines, the *Provisioning Manager* (PM) was introduced to the architecture. As mentioned at the end of chapter 4.2, this component has previously been proposed by Schneider in [Sch13]. Schneider suggests two possibilities. One being, that the PM is placed inside the *Enterprise Service Bus*. The other possibility is placing the Provisioning Manager outside of the ESB as an external service. Yet, Schneider did not explicitly endorse one of his propositions. In this chapter both ideas are introduced and their different advantages and disadvantages are explained in depth. Based on these, one of the propositions is chosen and incorporated in the different overall architectural concepts of the system that are the subject of chapter 6 “Architectural Concepts and Designs”.

Figure 5-1 shows a section of the overall architecture, which is depicted in Figure 4-1, with new, modified and extended components. The figure only depicts components that are relevant for the on-demand provisioning of services and their underlying infrastructure.

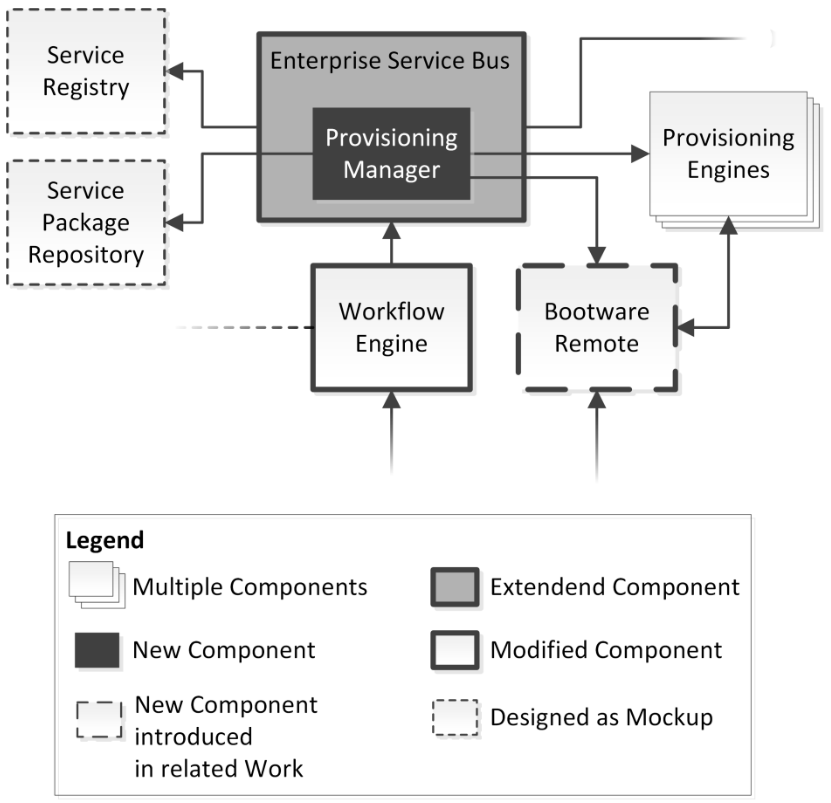


Figure 5-1: Provisioning Manager as an Internal Component

On the left part of Figure 5-1 the *Service Package Repository* and the *Service Registry* are shown. Both components were already part of the overall architecture that was previously introduced in chapter 4.2. To allow for a prototypical implementation (see chapter 7), the Service Package Repository and the Service Registry have been designed and implemented as mockups, providing only a very basic functionality. The following chapters 5.3 and 5.4 give a more detailed view on the design of the Service Registry and Service Package Repository mockups.

To the right, Figure 5-1 depicts multiple *Provisioning Engines*. This is in contrast to the previous architecture shown in Figure 4-1 “Overall System Architecture [VKL13]” in which only a single Provisioning Engine was shown. Having multiple Provisioning Engines is expected since the various service packages from the Service Package Repository might be of different formats. The overall architecture was designed with the goal to be as generic as possible; this implies that an exact format for a service package has not been specified and is not intended [VHK⁺14]. Thus, a service package may be available in a repository in many different formats such as TOSCA, Puppet[21] or Chef[22]; and each format has to have its own matching Provisioning Engine that can deal with the corresponding format specifics. Figure 5-2 shows an example of a Service Package Repository containing multiple service packages in different formats.

[21] http://puppetlabs.com/
[22] http://www.getchef.com/

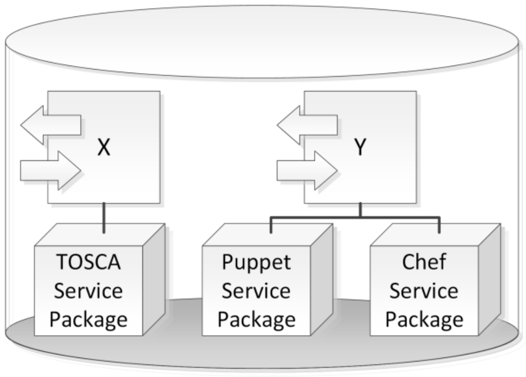


Figure 5-2: Service Package Repository with different service packages [VHK+14]

Interface X is only supported by one service package in TOSCA format; while interface Y is supported by both a Puppet and a Chef service package. If, for example, a workflow requires two services, one implementing interface X and one implementing interface Y, two different Provisioning Engines are needed. A closer look on this matter is provided in [VHK+14].

As shown in Figure 5-1 the Provisioning Manager is placed inside the *Enterprise Service Bus.* From there it communicates with a potential multitude of Provisioning Engines and the *Bootware Remote* (BWR) component. The internal ESB Control Flow is decoupled from dealing directly with different Provisioning Engines. The Provisioning Manager realizes an additional layer of abstraction by providing the ESB Control Flow with a stable interface to trigger the on-demand provisioning of services and their underlying infrastructure. The ESB Control Flow can initiate the provisioning by passing a reference of the desired service package to the Provisioning Manager, which then takes care of the actual provisioning and sends a response back to the ESB Control Flow as soon as the desired service is up and running. When the Provisioning Manager gets called to provision a service, the format of the referenced service package might require a Provisioning Engine that is not yet available to the Provisioning Manager. In this case, the Provisioning Manager first needs to instruct the BWR component to provision the required Engine.

The *Bootware Remote* component is the result of parallel work at IAAS by Reinfurt [Rei14]. As already mentioned, the Provisioning Manager needs a means to deploy Provisioning Engines. To allow this, the PM can acquire a list of all running Provisioning Engines from the BWR. If no appropriate engine is running, the Provisioning Manager passes a reference to the format of the required Provisioning Engine to the BWR, which in turn replies with the endpoint address of the newly provisioned engine after it is done processing the request. Furthermore the BWR takes care of starting the initial provisioning of the ESB and other components of the Scientific Workflow Management System. In-depth information and details on the BWR and its other components can be found in [Rei14].

The *Workflow Engine* component has been modified slightly, partially because of specific software idiosyncrasies. However the main issue was, how to route the messages from the Workflow Engine to the ESB. Since these messages are originally either targeted at a specific service endpoint or have no specific destination at all. More details and a solution are given in chapter 5.5 "Routing Messages to the ESB" and chapter 5.5.3 "Using Handlers to Reroute and Mediate Messages in Axis2".

The preceding Figure 5-1 shows the architecture in case the Provisioning Manager is located inside the ESB environment. As was mentioned previously, the other possibility is placing the Provisioning Manager outside the ESB as an external service. This approach is depicted in Figure 5-3 below.

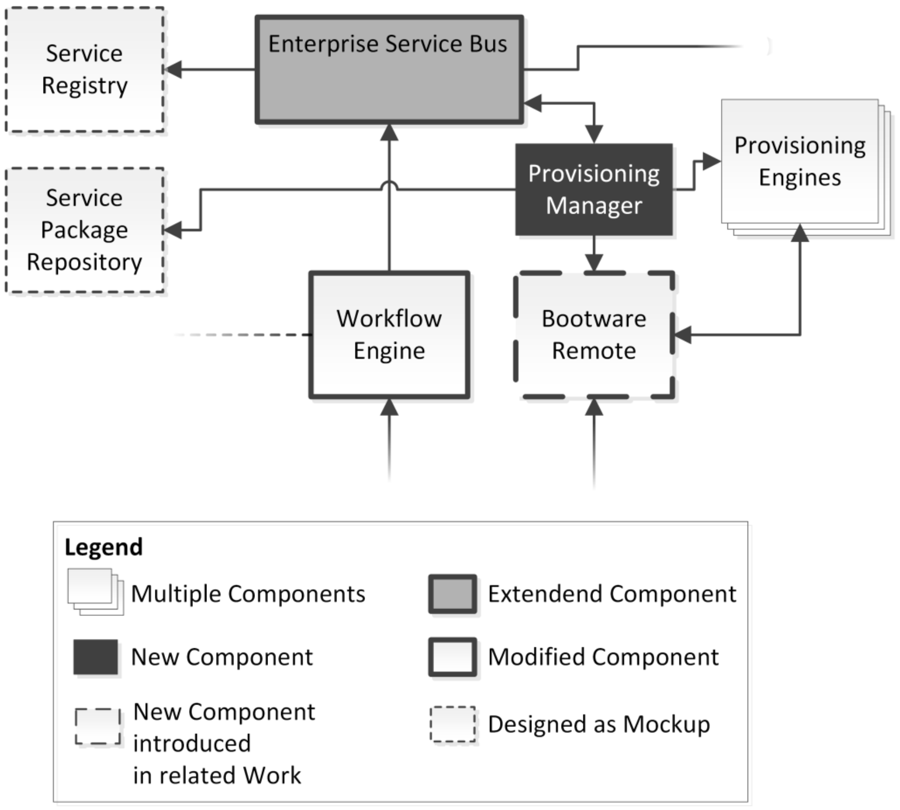


Figure 5-3: Provisioning Manager as an External Component

The internal architecture of the Provisioning Manager and its application logic remain the same in both approaches, see also chapter 5.2 "Provisioning Manager". The Provisioning Manager still provides a stable interface for the provisioning of services and their underlying infrastructure and interacts with the BWR when an additional Provisioning Engine is required. Yet there are certain advantages and disadvantages in both architectural approaches.

It is possible to combine the logic of both the Provisioning Manager and the Bootware Remote into one component, see also chapter 8 "Summary and Future Work". This would require that this new combined component is separated from the ESB, since otherwise the bootstrapping process of the entire system would not be possible. The BWR has to be available and operational before provisioning the ESB.

From an architectural point of view of the overall architecture, it might seem that having the Provisioning Manager inside the ESB violates a traditional SOA approach. In a classic Service-Oriented Architecture the Bus takes care of service selection, by interacting with a service registry, and routing messages to their correct destinations. A service package repository and provisioning of services are not part of a traditional SOA.

The Provisioning Manager and the BWR are based on a plug-in architecture, see also chapter 5.2 “Provisioning Manager”. Thus OSGi, as a well-established standard for a modularization-framework, was used as a base for the Provisioning Manager and the Bootware Remote. As was already mentioned in chapter 2.5 “Apache ServiceMix”, ServiceMix is built on Apache Karaf which supports Apache Felix as an OSGi implementation. Therefore the Provisioning Manager can be easily integrated and use the OSGi environment provided by the ESB. If the Provisioning Manager was to be realized as an external component, this component would need to provide the OSGi framework implementation for the Provisioning Manager. The BWR realizes this approach, since it has to be a separate component due to how the bootstrapping process of the system works. The Bootware Remote wraps the Apache Felix OSGi framework and the application logic as a web service. This means that the approach shown in Figure 5-3, where the Provisioning Manager is realized as an external component, would lead to the deployment of three separate OSGi frameworks in total: One in the ESB, one in the Provisioning Manager and one in the BWR component. Moreover, additional routing logic would be required to connect ESB and PM.

Thus it was decided to focus on the approach having the Provisioning Manager as an internal component of the ESB. If in future Provisioning Manager and Bootware Remote are consolidated into one component, no modification of the internal application logics should be required, since both components operate in a loosely coupled manner and do already exist as OSGi services.

5.2 Provisioning Manager

The Provisioning Manager was designed as having a plug-in architecture, similar to the architecture of the Bootware Remote component. This was a joint decision, since both components offer the provisioning of certain entities, i.e. services and engines respectively, and have to support multiple platforms. The Provisioning Manager has to support a variety of Provisioning Engines whose quantity is unknown. Thus supporting modularity was a key requirement. It was decided to use the OSGi framework for both the Provisioning Engine and the BWR. Figure 5-4 shows the plugin architecture of the Provisioning Manager.

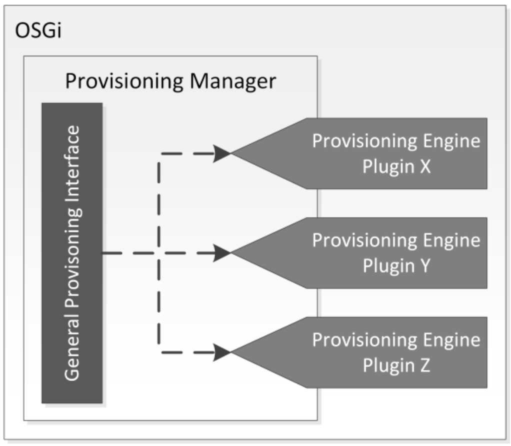

OSGi
Provisioning Manager
General Provisioning Interface
Provisioning Engine Plugin X
Provisioning Engine Plugin Y
Provisioning Engine Plugin Z

Figure 5-4: Provisioning Manager Architecture

The *Provisioning Manager* and *Provisioning Engine Plugins* reside inside an *OSGi* container. The PM provides a *General Provisioning Interface* that can be utilized to provision arbitrary services by referencing a service package and providing the address to a running Provisioning Engine. After receiving a call, the Provisioning Manager selects a suitable plugin. When a new plugin registers itself to the OSGi container, the Provisioning Manager is notified about its availability. Likewise the Provisioning Manager is informed if a plugin gets undeployed. This architecture implies that the Provisioning Manger can operate with any number of plugins; even without any plugin at all, which would albeit limit its functionality severely.

All plugins implement the same interface that is specified by the Provisioning Manager. A plugin provides the necessary logic to provision a service via a specific Provisioning Engine.

To determine whether a Provisioning Engine that matches the referenced service packages is available, the Provisioning Engine consults the Bootware Remote. The BWR holds information about all its deployed Provisioning Engines. If no suitable engine is available, the Provisioning Manager instructs the BWR to deploy such an engine.

5.3 Service Registry

A thorough design and implementation of the Service Registry was out of scope of this thesis and thus it was only designed and implemented as a mockup that would provide sufficient functionality for the on-demand provisioning of services, see also chapter 8 “Summary and Future Work” on how this component will be extended in the future. Figure 5-5 below shows the metamodel of the Service Registry mockup as an ER-Diagram in Chen’s notation.

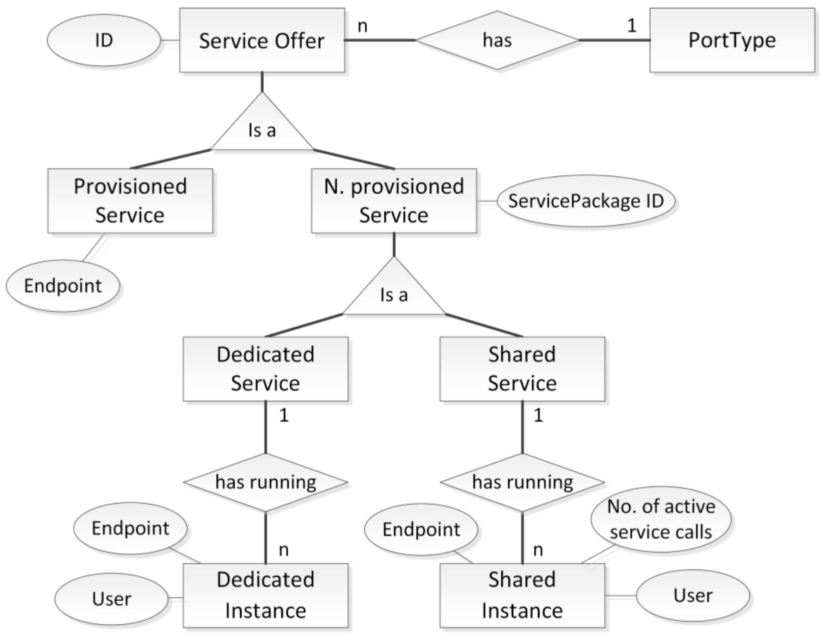


Figure 5-5: ER-Diagram of Service Registry Mockup, based on [VHK+14]

The Service Registry holds multiple *Service Offers*. As only functional requirements are taken into account for a service discovery, a Service Offer has exactly one *PortType* and an attributed *ID*. The port type is the only functional property that is used for service discovery in this thesis. Each Service Offer is one of two types: *Provisioned Service* or *Not Provisioned Service*. A Provisioned Service has only one attribute, the *Endpoint* at which this service is available. In case of a Not Provisioned Service, a *Service Package ID* is specified that identifies the service's corresponding service package in the Service Package Repository. A not provisioned service can either be a *Dedicated Service* or a *Shared Service*. A dedicated service may have multiple *Dedicated Instances* of this service. Each instance has an attributed *Endpoint* and *User*. The endpoint attribute stores the URL of this particular service instance. Shared services may have any number of *Shared Instances*. A shared service instance has three attributes: *Endpoint*, *No. of active service calls* and *User*. The endpoint attributes points to the service's location. The difference between dedicated and shared services is that a shared service instance can have multiple concurrent calls from the same user, while a dedicated service instance only serves one call. This is why the total number of active service calls, this shared service instance has, is stored.

5.4 Service Package Repository

As was the case with the Service Registry, the Service Package Repository was only designed as a mockup to provide very basic functionality to dependent components. The metamodel, as an ER-Diagram in Chen's notation, of the Service Package Repository mockup is described in Figure 5-6.

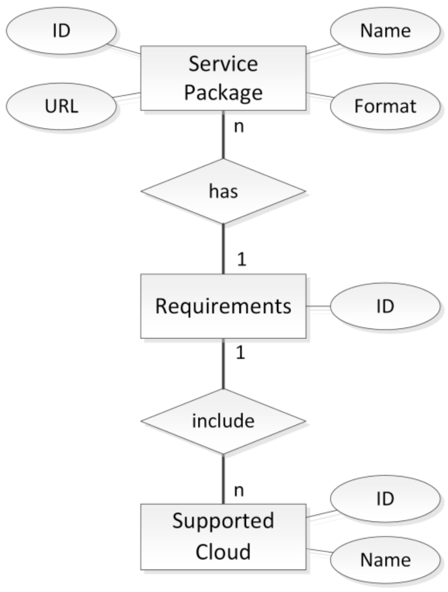


Figure 5-6: ER-Diagram of Repository Mockup

The Service Repository holds information about multiple service packages. Each *Service Package* has four attributes: *ID* identifies the service package by a unique identifier. *URL* stores the URL which points to the actual location of the service package archive. *Name* holds a non-unique name that has been given to this service package. The last property describes this service package's *Format*. For example, the format of a service package can be TOSCA. A Service Package has *Requirements*. Requirements are a set of technical requirements that must be met to deploy a particular service package. These should not be confused with functional requirements of a service which are stored in the Service Registry, see chapter 5.3. Since this is only a mockup, Requirements only includes one or more *Supported Clouds*. A Supported Cloud is attributed with an *ID* and *Name*. For example, a service package may be geared towards being deployable on Windows Azure or Google App Engine only.

5.5 Routing Messages to the ESB

As previously shown in Figure 4-1 all message-interactions should be run through the ESB. Thus all service calls from the Workflow Engine need to be routed to the Bus. To achieve this, the target endpoint of a message sent from the Workflow Engine has to be changed. In chapter 5.5.1 it is first discussed what possibilities do already exist to route messages from the Workflow Engine to the ESB. In chapter 5.5.2 the messaging architecture of Axis2 is introduced. This is relevant as ODE-PGF[23] is later used for the Workflow Engine component. ODE-PGF is based on Apache ODE, which runs inside an Axis2 Integration Layer (IL). A concept that enhances the messaging capabilities of ODE-PGF's Axis 2 IL is introduced in chapter 5.5.3.

[23] http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/

5.5.1 Using existing Technologies

Apache ODE allows a user to configure property files to alter the behavior of a deployed process [ODE13]. With these files it is possible to overwrite the original target endpoint of a service. Unfortunately this method was not an option, since with each deployed process a matching property file would have needed to be deployed to reroute the requests to the called services. Additionally these files would have been needed to be generated and deployed automatically because the scientist executing the process cannot be bothered to place these files inside the corresponding directory on the server running ODE-PGF; apart from the fact that the scientist would probably not have access to it. Furthermore Apache ODE only checks roughly every 30 seconds if a property file has been changed, meaning that after updating the property file the corresponding process needs to be stalled for at least that amount of time. Additionally the message needs to be enriched with information about functional requirements that a service must fulfill so that the ESB may perform a proper service discovery. These functional requirements are the port type that is specified in a WSDL and the actual address of a desired service in case a static binding should be used.

The first approach was to utilize the WS-Addressing [WSA07] standard to convey all information necessary for on-demand provisioning to the ESB. WS-Addressing defines multiple additional SOAP headers which are especially useful in asynchronous message exchanges [WAC06]. These headers are:

- To: An absolute IRI defining the address of the intended receiver of this message.
- From: An Endpoint Reference (EPR) from where this message originated.
- ReplyTo: An EPR defining the receiver of replies to this message.
- FaultTo: An EPR defining the receiver of faults that are related to this message.
- Action: An absolute IRI uniquely identifying the semantics implied by this message.
- MessageID: An absolute IRI that uniquely identifies this message.
- RelatesTo: A pair of values. Each value is an absolute IRI. One value of a pair defines the relationship; the other identifies the corresponding message by its MessageID.

- ReferenceParameters: Any kind of information represented as any type that is provided for the intended receiver.

An Endpoint Reference is an element that carries additional information like parameters and/or metadata besides the actual address of the endpoint. Unfortunately *To* is not an EPR but only a URI or IRI. This means that additional information about the destination of the message must be provided either through the *ReferenceParameters* or the *Action* element. ReferenceParameters is an optional element and may already hold information for the desired service when it leaves the Workflow Engine. Additionally the information held by the ReferenceParameters property is not intended for some intermediary, like the ESB, but rather for the intended receiver. So inserting additional information for service discovery that are of no interest to the originally intended receiver contradicts the intended purpose of this element. The other interesting property, Action, is mandatory and may or may not convey information that is suitable for a service discovery. It is recommended by the W3C "that the value of the [action] property is an IRI identifying an input, output, or fault message within a WSDL interface or port type." [WAC06]. As this is only a recommendation by the W3C, the Action element cannot safely be used for a service discovery process.

This does not mean that WS-Addressing is not utilized in the context of this thesis. In fact, WS-Addressing is required for correctly routing response messages from a called service back to the Workflow Engine. A detailed explanation about this matter is given in chapter 6.4 "Routing Response Messages".

Figure 5-7 shows how an additional SOAP header block was introduced that carries the necessary functional requirements to enable dynamic binding and service discovery by the ESB.

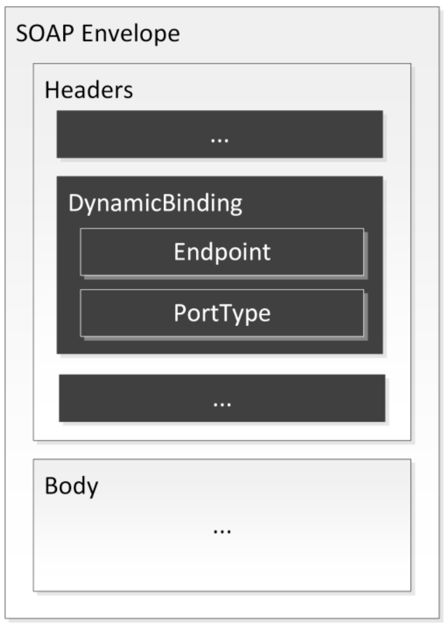


Figure 5-7: DynamicBinding header block in SOAP message

The DynamicBinding header block includes two fields: *Endpoint* and *PortType*. The Endpoint field indicates to the ESB which binding to use. If it is empty no endpoint is specified and thus a dynamic binding has to be used. In case of a static binding the ESB needs to know which destination the message needs to be routed to. Thus the already known endpoint of the service needs to be specified in the SOAP request message. The PortType field refers to the actual port type from the WSDL of the desired service, since this information is not part of a normal SOAP request message. This information is required to provide the ESB with the functional requirement needed to find a matching service.

Due to these circumstances an alternative way to using ODE property files had to be found to reroute messages. As will be described in chapter 5.5.2, handlers in Axis2 are well suited for such a task.

5.5.2 Message Flows inside Apache Axis2

Figure 5-8 shows a simplified message flow inside Apache Axis2. Omitted are Fault-Flows, security related phases. Potential user defined phases are summarized and symbolized by the dashed "User Defined" phase.

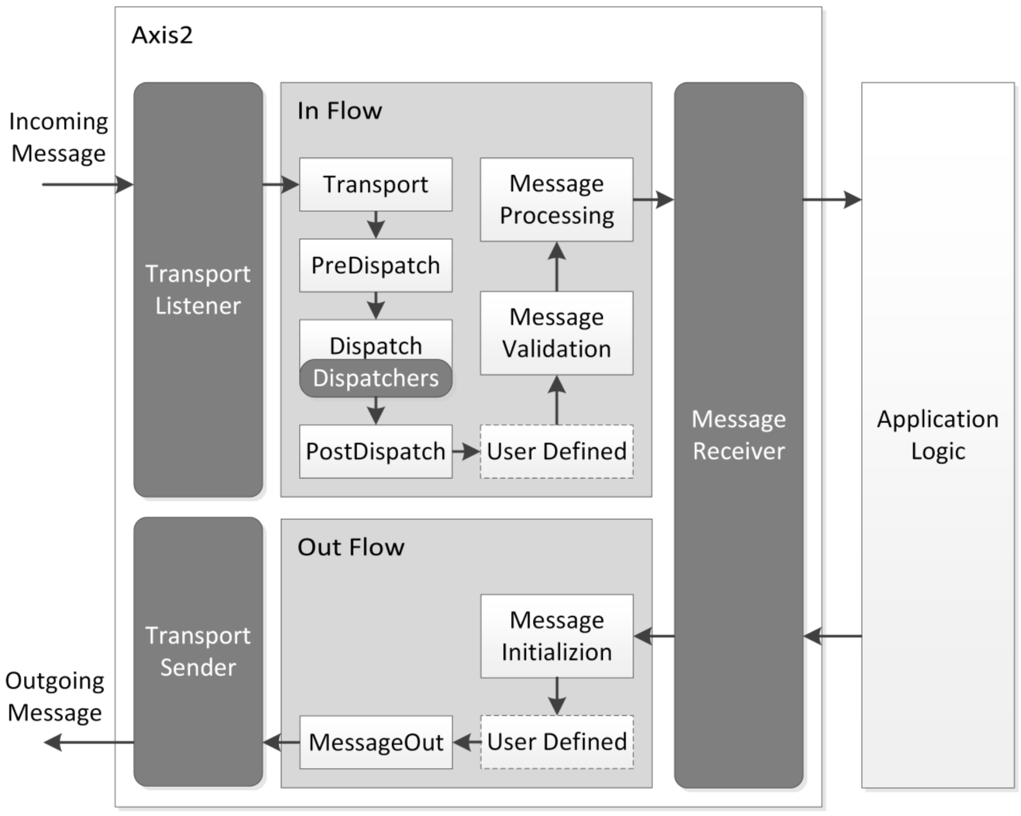


Figure 5-8: Axis2 Message Flows with predefined phases [AAG12] [IBM05], simplified

When a message is received by Axis2 it travels along the *In Flow* (or In Pipe) until it can be consumed by the application logic. Likewise when the application logic dispatches an outgoing message it will travel through the *Out Flow* (or Out Pipe). To allow for mediation of messages along the message path, Apache Axis2 supports the use of so called handlers which act as intermediaries and intercept messages going in and out of the Axis2 Engine. Each handler is associated with a specific phase which marks the point on the message path at which it processes a message. Additionally to the predefined phases by Axis2, being *Transport*, *PreDispatch*, *Dispatch*, *PostDispatch*, *Message Validation*, *Message Processing*, *Message Initialization* and *MessageOut*, one may define own phases after the PostDispatch or Message Initialization phase.

Thus an incoming message is first processed by a so called Transport Listener which detects the message. It is then passed on through the different phases along which the message may be processed by different handlers. For example, if the message is based on WS-Security [WSS06] a handler may be used to decrypt the message. Such a handler may for example be placed in a user defined phase that is associated with dealing with security aspects. All handlers are organized in modules which can be plugged into an Axis2 system and its phases. After the message has passed the PreDispatch phase the dispatchers take care of finding the matching service and operation for the message. If they are unable to find a service, a “service not found” error will be thrown. The Message Validation Phase ensures that the message has been processed correctly and hands it over to the Message Processing Phase. There the message flow always ends with the Message Receiver passing the message to the application (i.e. Application Logic in Figure 5-8).

An outgoing message follows the Out Flow whose composition is much simpler than the In Flow. This is due to the fact that, in contrast to processing an inbound message, there is no need to find a service and operation matching the information given in the message, since these are already known when the application initiates the Out Flow. The message first passes the Message Initialization phase before engaging potential custom phases and the final Message Out phase. This phase marks the end of the Out Flow and passes the message to the Transport Sender, dispatching the message to its target endpoint [AAG12].

5.5.3 Using Handlers to Reroute and Mediate Messages in Axis2

Figure 5-9 shows a custom Mediator Module inside the Apache Axis2 Integration Layer of ODE-PGF.

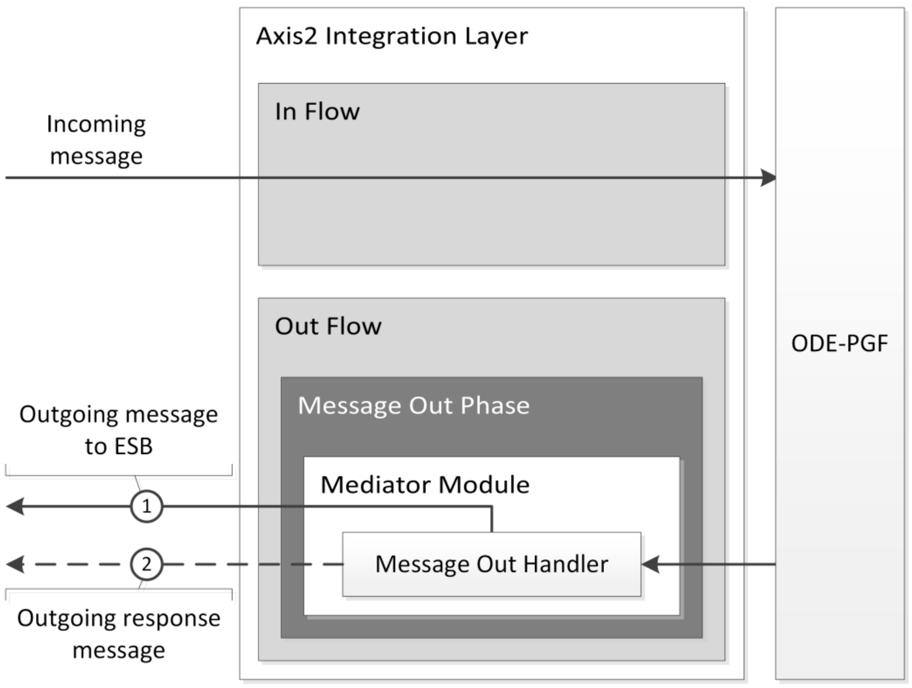


Figure 5-9: Custom handler in Axis2 Integration Layer of ODE-PGF

The handler is placed in the Message Out Phase and intercepts every message ODE-PGF dispatches. The handler has to distinguish between two cases of messages. A service call that has to be forwarded to the ESB (route ①) and secondly a response message back to the scientist's client which must not be rerouted (route ②).

Additionally in case ① the SOAP header block of the message has to be extended with information about the service's endpoint and port type to enable for a dynamic service selection by the ESB (see also Figure 5-7). This information is part of the message context created by ODE-PGF that is passed through the pipe. The ESB's endpoint that is used as the new target endpoint for these messages could, for example, be acquired from a configuration file.

5.6 Message Store

A message store is part of the different architectural concepts introduced in later chapters; namely 6.1, 6.2 and 6.3. This message store fulfills multiple purposes, depending on the different architectures. Yet, first and foremost its task is to provide a central and persistent storage location for all messages the ESB receives from the Workflow Engine. As was the case with the Service Registry and Service Repository, the

Message Store is only meant as a mockup in the scope of this thesis and may be heavily modified or stripped and replaced entirely in future works. Figure 5-10 shows the metamodel of the Message Store.

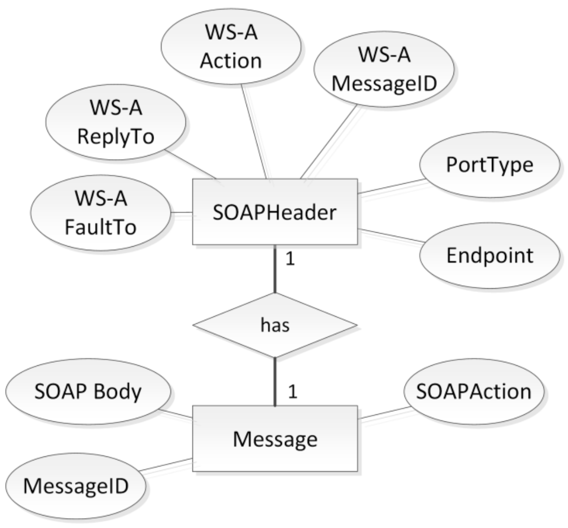


Figure 5-10: ER-Diagram of Message Store Mockup

Two entities are defined in this model: *Message* and *SOAPHeader*. Message has the following attributes:

- SOAPAction: *SOAPAction* indicates the general purpose of the message to the intended receiver. In case of a SOAP-over-HTTP binding, SOAPAction is part of the HTTP header of the message. This attribute is needed as it is part of the service call that has to be forwarded to a service, later.

- SOAPBody: This attribute stores the complete body of the SOAP envelope. This is the main payload that has to be conveyed to the service.

- MessageID: This is the ID that gets automatically assigned to every message that is received by the ESB and uniquely identifies it.

One Message entity relates to, i.e. *has*, exactly one SOAPHeader entity which has the following attributes. Attributes with a *WS-A* prefix refer to attributes that are part of the WS-Addressing specification [WAC06].

- WS-A FaultTo: Holds the EPR to which fault messages should be sent.

- WS-A ReplyTo: The callback EPR to which the intended receiver of this message should send its reply.

- WS-A Action: Contains the WS-Addressing Action property IRI.

- WS-A MessageID: Holds the WS-Addressing Message ID that was assigned by the caller.

- PortType: The port type that is specified in the dynamic binding header block. See also chapter 5.5.

- Endpoint: The endpoint this message should be sent to. Either already specified in case of a static binding or dynamically resolved by the ESB.

As may have been noticed, not all possible WS-Addressing attributes are part of this model. This is due to them not being relevant in the context of this thesis. Yet in future, it might be useful to store other attributes as well, for example Reference Parameters.

5.7 Partitioning the ESB Control Flow

While it would have been possible to realize the entire ESB Control Flow (shown in Figure 4-2) as a single workflow model, this is not desirable. Several parts of the ESB Control Flow are more suited for a realization as workflow models, while others are better fit for routing via EIPs. Consequently the ESB Control Flow has been split up in multiple parts to facilitate the processing of requests, some parts being workflow models and others routing models. In Figure 5-11 it is shown how the ESB Control flow has been divided in different parts.

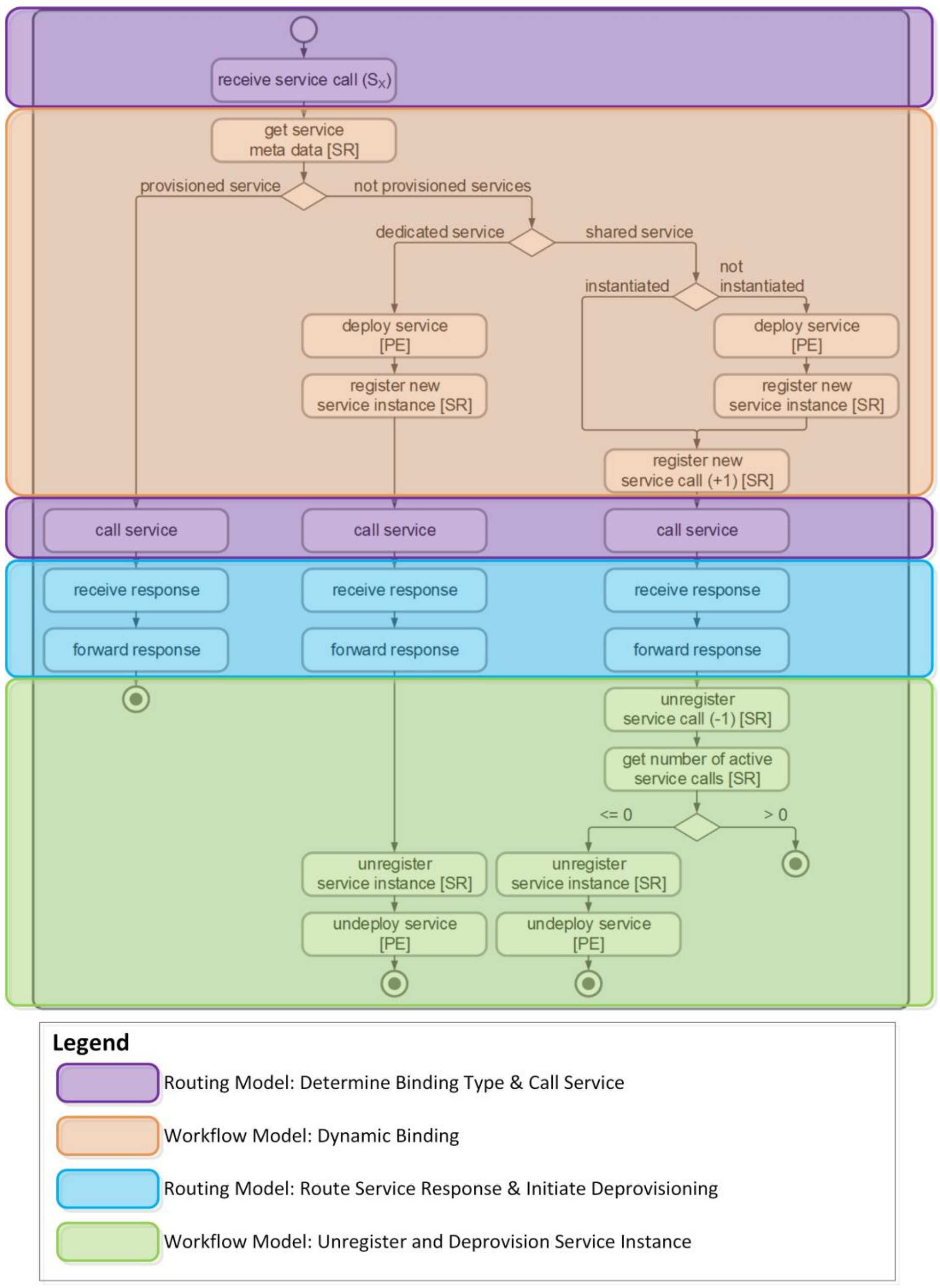

receive service call (S_X)
get service meta data [SR]
provisioned service
not provisioned services
dedicated service
shared service
instantiated
not instantiated
deploy service [PE]
register new service instance [SR]
deploy service [PE]
register new service instance [SR]
register new service call (+1) [SR]
call service
call service
call service
receive response
forward response
receive response
forward response
receive response
forward response
unregister service call (-1) [SR]
get number of active service calls [SR]
<= 0
> 0
unregister service instance [SR]
undeploy service [PE]
unregister service instance [SR]
undeploy service [PE]
Legend
Routing Model: Determine Binding Type & Call Service
Workflow Model: Dynamic Binding
Routing Model: Route Service Response & Initiate Deprovisioning
Workflow Model: Unregister and Deprovision Service Instance

Figure 5-11: Partitioned ESB Control Flow

Each colored section represents a separate model; chained together they form the complete ESB Control Flow. The purple and orange parts are responsible for routing the original service call to a service, which might need to be provisioned first. They are the subject of chapters 5.7.1 and 5.7.2. The blue and green parts are subjects of chapters 5.7.3 and 5.7.4. They are required to receive and forward service responses, as well as to update the Service Registry and if required, initiate the deprovisioning of a service.

5.7.1 Determination of Binding Type - Routing Model

As seen in the overall ESB Control Flow in Figure 4-2, it has first to be determined whether a static binding or a variation of dynamic binding has to be used. To do this, it has to be checked whether a static or dynamic binding has to be used. As this is a procedure that is not very complex, it has been realized via a routing model. How this routing model is then transformed into a more detailed concept is shown in the chapters 6.1, 6.2 and 6.3 when EIPs are applied. Nevertheless the coarse process is the one depicted as a BPMN model in Figure 5-12 below.

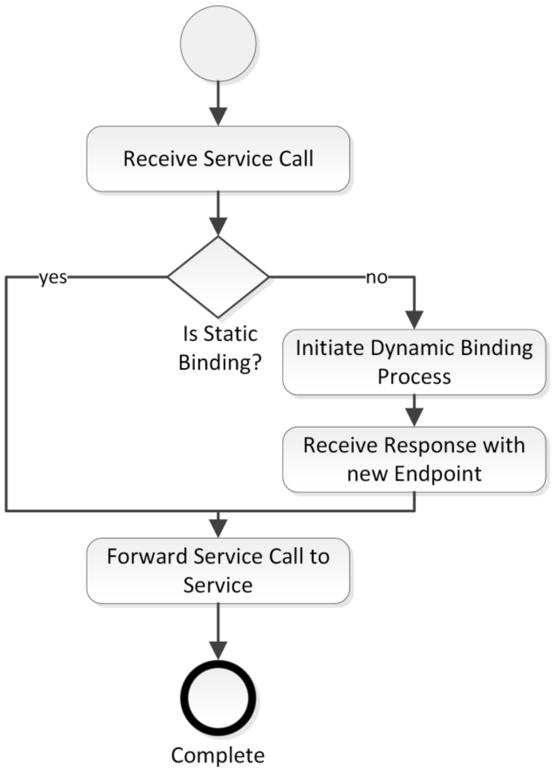

Receive Service Call
yes
no
Is Static Binding?
Initiate Dynamic Binding Process
Receive Response with new Endpoint
Forward Service Call to Service
Complete

Figure 5-12: Routing Model: Determine Binding Type & Call Service

The incoming request message from the Workflow Engine is received and it is determined whether a static binding can be used or not. If it can be used, the request is directly forwarded to the service. Otherwise a dynamic binding has to be used and the next part of the overall ESB Control Flow is initiated. As shown in Figure 5-11 this is a workflow model whose task is to provision a service if need be. Upon receiving the new endpoint of the service, the original request is then forwarded to the service.

5.7.2 Dynamic Binding of Services - Workflow Model

Figure 5-13 below shows in BPMN notation the provisioning workflow model that is part of the ESB Control Flow.

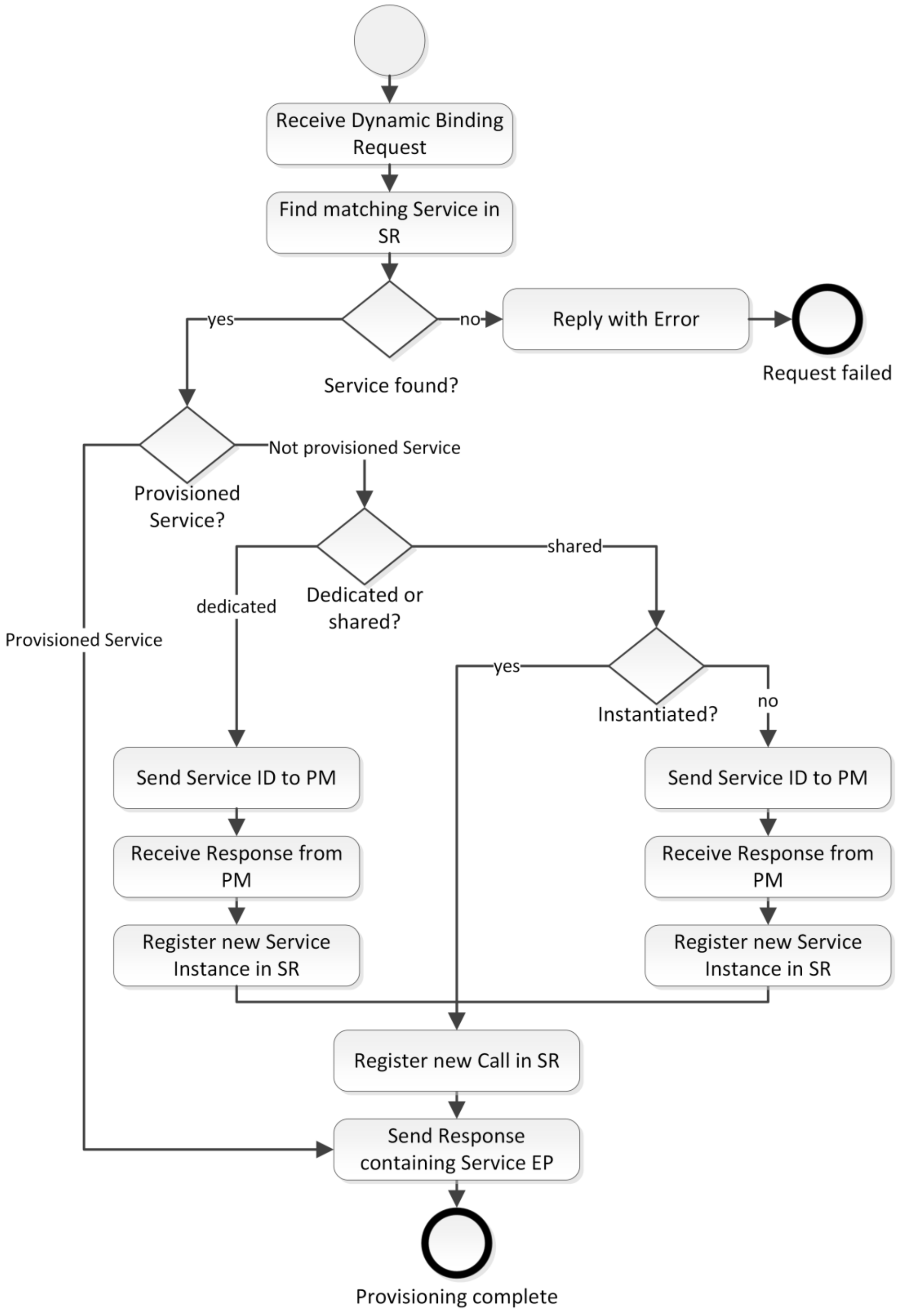


Figure 5-13: Workflow Model: Dynamic Binding

The instance of the workflow model receives a request to perform a dynamic binding and find a service that matches the supplied functional properties. Afterwards the Service Registry is consulted whether such a service is available. If not, the workflow replies with an error. After this point the process differs depending on the type of service. First it is determined whether the specific service is a provisioned or not-provisioned service. In case of a provisioned service, the corresponding endpoint is already known and fetched from the registry.

If a service is found, but it is not-provisioned and a dedicated service the Provisioning Manager is called with a reference ID to the selected service. This differs from the original ESB Control Flow in Figure 4-2. There, the Provisioning Engine is called directly since that architecture lacks the Provisioning Manager component. After receiving a response from the Provisioning Manager that the specific service has successfully been provisioned, the Service Registry is updated with a new active instance of this service which contains the new endpoint.

In case of a shared service it is first checked whether it has already been instantiated. If it is, the Service Registry is updated that this service instance has now an additional active service call. Otherwise, if the service has not yet been instantiated, the same process as if the service would have been a dedicated one applies. The Provisioning Manager is called, its response received and a new service instance is registered in the Service Registry.

A new active service call is registered for any not-provisioned service. In the original ESB Control Flow, service calls were only tracked for shared services. By registering service calls for every not-provisioned service instance, the deprovisioning logic is simplified (see also Figure 5-15 “Workflow Model: Unregister and Deprovision Service Instance”).

In all cases, whether it is a provisioned, dedicated or shared service, a response message containing the endpoint of the service is sent back to the route depicted in Figure 5-12 before this process completes.

5.7.3 Routing Service Replies and initiating Deprovisioning of Services

As was the case in the provisioning part of the ESB Control Flow, the deprovisioning part also first begins with a route, see Figure 5-14 below. The model shows the coarse process while the workflow model in Figure 5-15 depicts the concrete steps necessary for the deprovisioning of a service.

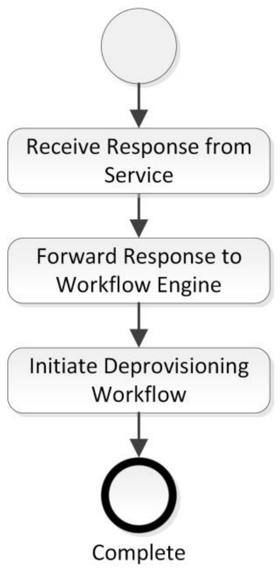

Receive Response from Service
Forward Response to Workflow Engine
Initiate Deprovisioning Workflow
Complete

Figure 5-14: Routing Model: Routing Service Replies

In the first step, the route receives a response from a previously called service. Afterwards this response message is forwarded to the Workflow Engine and as the final step the workflow process is called, which is responsible for dealing with the possible deprovisioning of the specific service. Further details about the forwarding of response messages are provided in chapter 6.4 “Routing Response Messages”.

5.7.4 Deprovisioning of Services - Workflow Model

Whether a service actually needs to be deprovisioned, is decided in a workflow model. There are several cases in which a service is not deprovisioned. The metamodel of this process is shown in Figure 5-15.

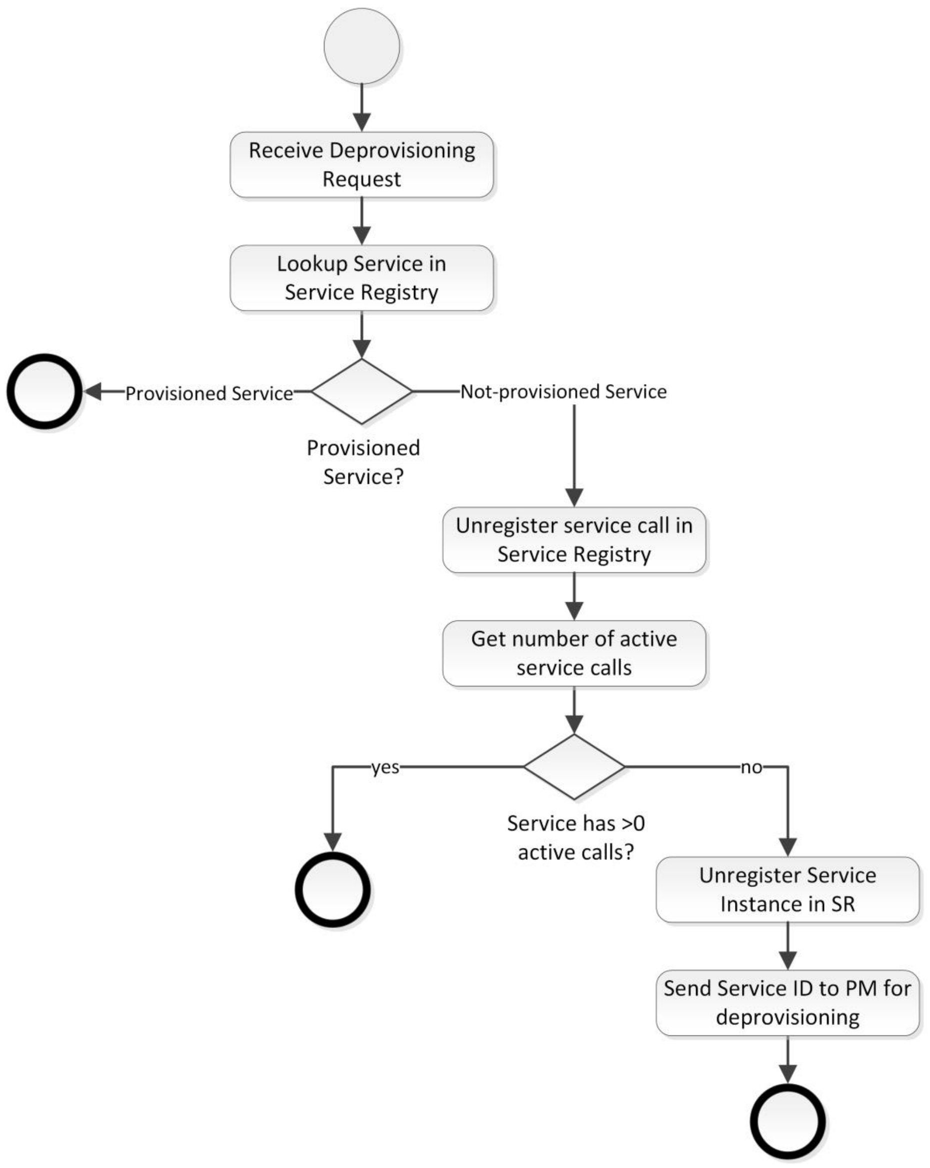

Receive Deprovisioning Request
Lookup Service in Service Registry
Provisioned Service
Not-provisioned Service
Provisioned Service?
Unregister service call in Service Registry
Get number of active service calls
yes
no
Service has >0 active calls?
Unregister Service Instance in SR
Send Service ID to PM for deprovisioning

Figure 5-15: Workflow Model: Unregister and Deprovision Service Instance

After receiving the deprovisioning request from the route shown in Figure 5-14, the corresponding service is looked up in the service registry. If it is a provisioned service, no further processing is required. If it is a not-provisioned service, the number of active service calls is reduced by one. It then depends on the number of active service calls whether the service is going to be deprovisioned or not. A shared service instance might have additional active service calls and is thus not deprovisioned. Otherwise the shared service instance will be deprovisioned. If it is a dedicated service, it will have no more active service calls, since a dedicated service can at most have one active service call. Thus, it will be deprovisioned.

6 Architectural Concepts and Designs

In this chapter three different architectural concepts are introduced in chapters 6.1, 6.2 and 6.3. Each concept formulates a different approach for the realization of on-demand provisioning of services and their underlying infrastructure. Every concept is discussed in-depth with its different advantages and disadvantages. Based on these, one concept has been chosen that is then prototypically implemented, which is the topic of chapter 7.

Subject of chapter 6.4 is the forwarding of response messages from services back to the Workflow Engine. This includes both synchronous and asynchronous response messages.

All concepts and designs that have been developed as part of this thesis try to be as general and uncoupled as possible from concrete implementations of certain technologies. Nevertheless these concepts and designs are not as high-level as the architecture of the overall system described in chapter 4.2. Thus they have to take into account certain limitations, features and idiosyncrasies of software that is used to realize the on-demand provisioning of services and their underlying infrastructure in the context of the Simtech SWfMS. The following software has been used for the different components: For the ESB component Apache ServiceMix 4.3.0 is used, which utilizes Apache Camel 2.6 and Apache ODE 1.3.5 that are packaged as part of ServiceMix. ODE-PGF is used for the Workflow Engine. ODE-PGF is an enhanced open-source version of Apache ODE that runs inside an Axis2 Integration Layer on an Apache Tomcat Server.

6.1 Concept I: Using an Apache Camel Aggregator

In this chapter the first of three concepts that have been developed in the context of this thesis is explained. The concept that is introduced in this chapter relies heavily on different EIPs that are already integrated in Apache Camel. This chapter is split in two parts, chapter 6.1.1 “Architectural Overview” and chapter 6.1.2 “Routing Logic”. In chapter 6.1.1 a general overview of the different components that enable the on-demand provisioning of services is given. This also includes the interaction between the different components. In chapter 6.1.2 the routing logic that controls the internal message flow of the ESB is explained in detail.

6.1.1 Architectural Overview

Figure 6-1 provides an overview of the architecture that has been developed as part of this concept.

This figure only depicts the message flow and components involved for routing a service call from ODE-PGF to a specific service. How to return reply messages is subject of chapter 6.4.

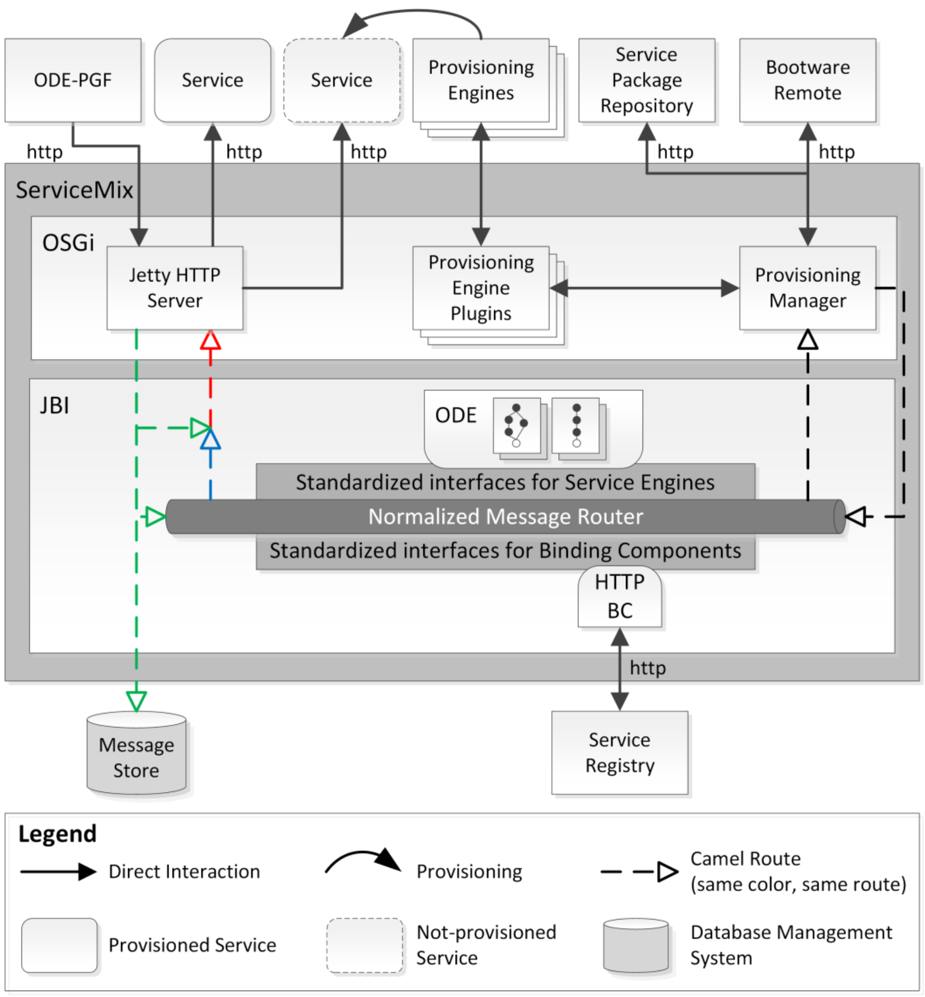


Figure 6-1: Concept I – Architectural Overview

The figure shows the ESB, i.e. Apache *ServiceMix*, along with its internal components. *OSGi* and *JBI* are the two frameworks that are running inside ServiceMix. A *Jetty HTTP Server*[24], various *Provisioning Engine Plugins* and the *Provisioning Manager* are running inside the *OSGi* environment of ServiceMix. The *Normalized Message Router*, Apache *ODE*, and a *HTTP Binding Component*, that connects the external *Service Registry* with the NMR, are part of the *JBI* environment. Apache ODE is the internal workflow engine of ServiceMix and runs as a service engine that is connected to the NMR. Different workflow models are deployed on ODE that implement certain aspects of the ESB Control Flow (see also the previous chapters 5.7.2 and 5.7.4).

[24] http://www.eclipse.org/jetty/

Depicted above and below ServiceMix are several external components, namely *ODE-PGF*, multiple *Provisioning Engines*, *Service Package Repository*, *Service Registry*, *Message Store* and *Bootware Remote* that interact with the ESB and its internal components. Service Registry, Service Package Repository and Bootware Remote all offer web service interfaces that are accessible via SOAP over HTTP. The way of communicating between a Provisioning Engine Plugin and its corresponding engine is dependent on the particular implementation of the Provisioning Engine. Thus, Figure 6-1 does not show a particular protocol for this interaction. Furthermore two *Services* are displayed. One is an already provisioned Service, while the other one is not-provisioned and needs to be deployed by a Provisioning Engine. The Message Store is designed as an external database hosted on a DBMS. Apache Camel routes are used to connect the different components that are split over JBI, OSGi and external platforms. Camel routes are shown as dashed lines. Each color represents a different Camel route. There are four different routes that are colored in green, red, blue and black. More details and explanations about the different Camel routes are provided in chapter 6.1.2 "Routing Logic".

An incoming service call from ODE-PGF is received by the internal Jetty Server. From there, different Camel Routes control the internal message flow. The incoming message is also saved in the Message Store. Depending on the required type of binding, the message is routed directly to the specified service or through a series of intermediaries that take care of different dynamic binding strategies (green and red route in Figure 6-1). This includes ODE and the Service Registry that are connected to the NMR; and the Provisioning Manager if a service has to be provisioned (black route in Figure 6-1).

When ODE receives a message through the NMR it starts an instance of the workflow model that is responsible for triggering the registration and provisioning of a service (see also Figure 5-13). Once the Service Registry has to be called to determine whether a service matching the functional requirements is available, ODE sends a service call into the NMR. The NMR routes this service call to the Service Registry's matching binding component that is registered on the Normalized Message Router. This binding component forwards the service call from ODE to the Service Registry and subsequently returns its response back to the NMR. If a service has to be provisioned, a Camel route (black route in Figure 6-1) receives the call from the workflow model instance on the NMR and invokes the Provisioning Manager.

The architecture of the Provisioning Manager was previously shown in Figure 5-4. The Provisioning Manager consults the Bootware Remote about available Provisioning Engines and chooses a matching Provisioning Engine Plugin that is registered on the OSGi framework. The Provisioning Manager acquires a reference to the service package from the Service Package Repository, which is then passed on to the chosen plugin along with the address of the Provisioning Engine. The plugin orders a Provisioning Engine with the provisioning of a service instance, including its underlying middleware and infrastructure. After the provisioning is complete, the result containing the endpoint of the newly provisioned service is passed back to the NMR through a Camel route (black route in Figure 6-1). From there, the NMR forwards the response to ODE and its workflow model instance that commissioned the provisioning of this service. Now that the endpoint is known, ODE publishes a result message containing the new service endpoint to the NMR. This message is received by the blue route on the NMR for further processing. This processing is explained in the following chapter 6.1.2. As soon as the original service call has been enriched with the service's endpoint, the Jetty server is used again to forward the message to the specified service.

6.1.2 Routing Logic

Figure 6-2 shows the routing model in this approach which is comprised of different EIPs, message flows and the NMR. The different colors correspond to the Camel routes shown in Figure 6-1.

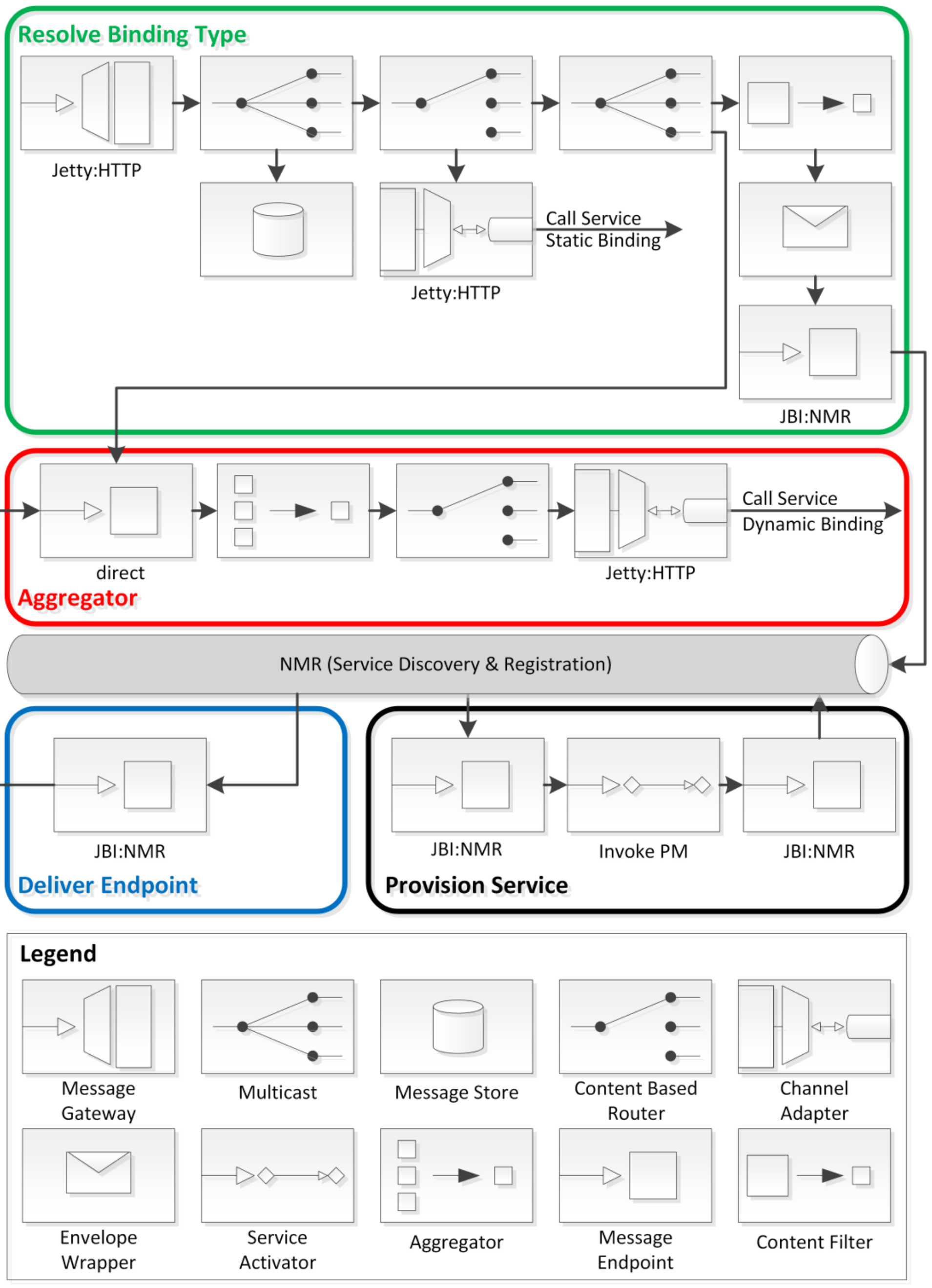


Figure 6-2: Concept I – Routing EIP Diagram

Incoming messages from the Workflow Engine are received by the *Resolve Binding Type* route (green) through a Message Gateway (see upper left part of Figure 6-2), which is an endpoint on a Jetty HTTP Server. This endpoint is exposed by Camel under a specified URL and port. The first step after receiving the message is to store it persistently. A Multicast sends a copy of the message to a Message Store where the message is saved. Additionally the message is forwarded to the next step. A Content Based Router determines whether a static binding or a dynamic binding has to be used for this specific service call. This means that the Content Based Router checks whether an endpoint has been specified in the message to which it should be sent. If an endpoint is specified, a static binding is used. The Content Based Router then uses the Jetty Server as a Channel Adapter and forwards the service call directly to the service. If no endpoint has been specified, a dynamic binding has to be used because the URL of the service is unknown. Thus, the message is forwarded to another Multicast, sending a copy of the message to the endpoint of the *Aggregator* route and another copy to a Content Filter. This component is necessary because the message has to be mediated in order to activate a workflow model instance on Apache ODE. This workflow model implements a part of the ESB Control Flow that deals with dynamic binding strategies (see also Figure 5-13). Unimportant parts are stripped from the message, leaving only information that is necessary for a service discovery in the message. These are the functional properties that the service has to provide. Afterwards the Envelope Wrapper wraps necessary parts of the original service call as a new message that is used to invoke the dynamic binding workflow model that is running on ODE. This is done by having this route's endpoint registered on the NMR. The format of this message is defined by the workflow model's WSDL. The NMR picks up the message and finds a suitable recipient, i.e. the workflow model running on Apache ODE. This concludes the *Resolve Binding Type* route that corresponds to Figure 5-12 which showed this section of the overall ESB Control Flow.

The message flow continues on the Normalized Message Router. If a service needs to be provisioned the *Provision Service* route (black) is activated. This route has an endpoint on the NMR from where it receives calls from the workflow model instances (see also Figure 6-1). Afterwards the Provisioning Manager is invoked and its result returned to the NMR. The result containing the new endpoint of a service is routed via the *Deliver Endpoint* route (blue) from the NMR to the *Aggregator* route.

The *Aggregator* route (red) shows a separate route containing an Apache Camel aggregator. The route has a Camel specific endpoint that can be targeted by other routes. Apache Camel comes with its own aggregator component[25]. The aggregator in Apache Camel provides persistence support for messages with an integrated HawtDB[26] acting as a message store. HawtDB is a lightweight key value database. This aggregator combines different messages and sends out a new message, when certain criteria are met. After the aggregator receives the message containing the endpoint of the service from the *Deliver Endpoint* route it combines the message of the original service call with the message containing the new service endpoint. To allow this, a correlation property has to be specified so that the aggregator has a means to know which messages relate to one another. In this scenario the message ID of the original service call is used to identify

[25] http://camel.apache.org/aggregator2.html
[26] http://hawtdb.fusesource.org/

which messages belong together. The message containing the endpoint of the newly provisioned service also encloses the original service call's message ID. The aggregator can then look up the corresponding original service call and combine the information of both messages. This means that the aggregator builds a new message containing the original service call and the new endpoint of the service, so the destination of the service call is now explicitly defined. This is followed up by a Content Based Router. This router analyzes the message, and routes it depending on the specified destination endpoint. Again Jetty serves as a Channel Adapter that sends the message to the specified service.

The main disadvantage that this concept has lies within the Apache Camel aggregator component. In general, Apache Camel components can be configured to work with different message exchange patterns (MEP). Namely *Request-Reply* and *Event Message*, which are also part of the EAI patterns defined by Hohpe and Wolf in [HoW10]. The Request-Reply message exchange is a two way communication, meaning that the sender of a message expects a more or less immediate reply to his request. In Camel this MEP is also defined as an *InOut* exchange. There are the following two approaches to a Request-Reply pattern [HoW10]:

- Synchronous Block: The requestor sends his request message and blocks this thread until he receives a reply and then processes it.

- Asynchronous Callback: The requestor waits on a callback for the reply message to his request and is thus non-blocking. This means the requestor is able to continue other processing while waiting for the reply. The callback is specified in the request message so the called entity knows where to send its reply.

In contrast, an Event Message exchange is only a one way communication. For example, a sender only informs a recipient about some incident without expecting any reply message in return. This one way communication is also called *InOnly* in Apache Camel.

Unfortunately the Camel aggregator only functions in an InOnly manner. This means that it always treats any incoming message exchanges in an asynchronous manner. Yet the original service call from the Workflow Engine might be targeted at a synchronous or an asynchronous web service. Hence in case of a synchronous service call the aggregator, upon obtaining the message through the routes, would receive the message and the exchange would end. Camel would reply that it has successfully delivered the message and this reply would then be passed back to ODE-PGF. But since it expects a synchronous reply message that contains the result to the initial request message, ODE-PGF would throw an error. This is unacceptable as the ESB has to support service discovery for both synchronous and asynchronous service calls without limitations.

6.2 Concept II: Using a Custom Message Processor

The second of the three concepts tries to eliminate the problem faced with in Concept I (chapter 6.1). The Camel aggregator of Concept I cannot be used, due to its lack of supporting synchronous message exchanges. This concept introduces a custom processor to enable the correct processing of synchronous service calls.

6.2.1 Architectural Overview

Figure 6-3 provides an overview of the architecture that has been developed as part of this concept. It slightly deviates from the architecture of Concept I that was shown in Figure 6-1. As was the case in Concept I, this figure only depicts the message flow and components involved for routing a service call from the Workflow Engine to a specific service. How to return reply messages is subject of chapter 6.4.

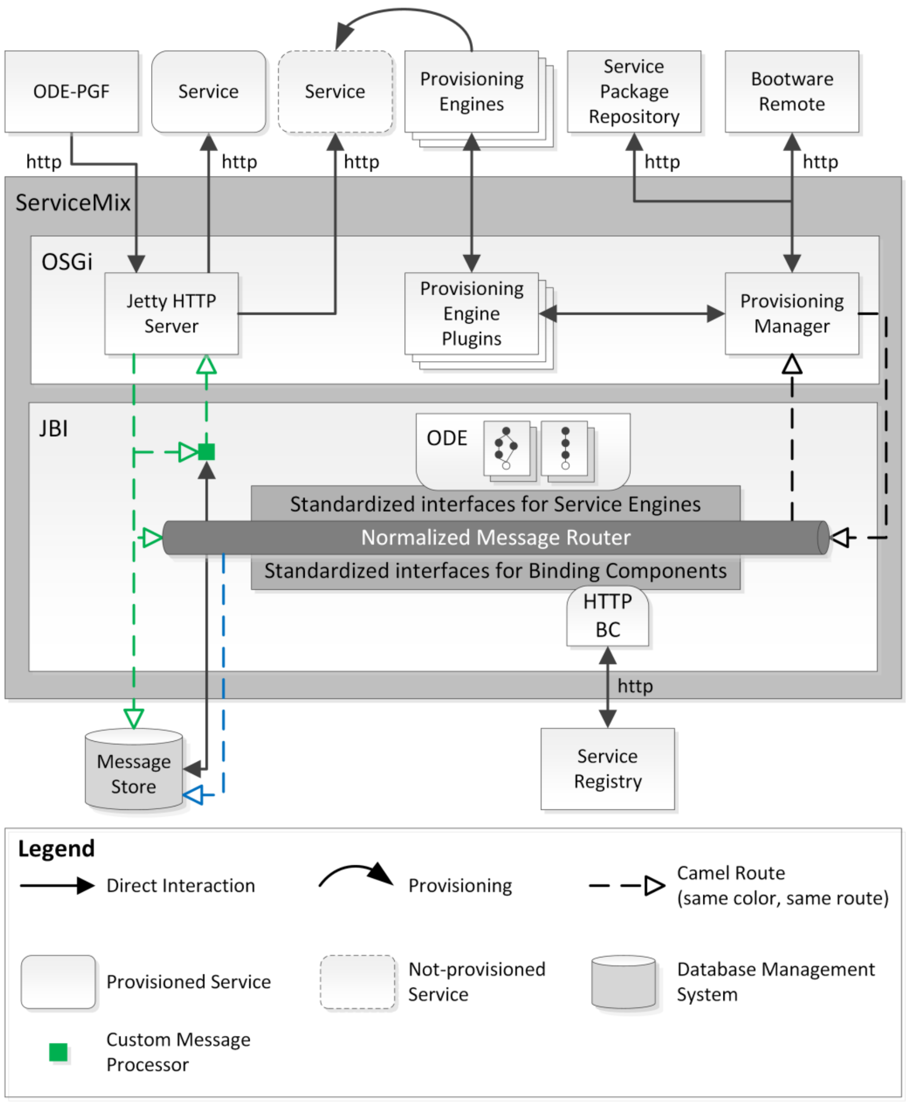


Figure 6-3: Concept II – Architectural Overview

In comparison to Concept I all components remain the same, as well as their locations. Figure 6-3 introduces only one new component, a *Custom Message Processor* (CMP). In addition the routing logic has changed. Further details about this subject are given in chapter 6.2.2. The focus in this chapter lies on describing the differences between Concept I and Concept II in context of the architectural overview.

A message from ODE-PGF is received through the Jetty HTTP Server on which a Camel route (green) has published an endpoint. As was the case in Concept I, this route is responsible for saving the message in a *Message Store*, determining the binding type, and triggering further processing, i.e. service discovery. The new Custom Message Processor also resides on this route.

The CMP, which is the centerpiece of this architectural concept, is a blocking component. This means that as soon as the CMP receives a message, it will prevent any further processing on that route until it unblocks the route. The CMP takes advantage of the automatically generated ID that gets assigned to every message that is received by ServiceMix and that was previously stored in the Message Store. When a message is received by the CMP, it will periodically check if the database entry corresponding to this message has been updated with information about the endpoint. When the service discovery, and possible provisioning, process has finished, a Camel route (blue) updates the original service call in the Message Store. After the Custom Message Processor notices that the corresponding entry in the Message Store has been updated, it extends the service call message it held back with the new endpoint and forwards it to the correct service through the Jetty Server.

The route (black) that connects the Provisioning Manager with the NMR is identical to that of Concept I (see also Figure 6-1). It receives calls from ODE on the NMR, forwards them to the Provisioning Manager and subsequently transfers the reply back to the NMR.

6.2.2 Routing Logic

Figure 6-4 shows the routing model of Concept II which is comprised of different EIPs, message flows, the NMR and a Custom Message Processor. The different colors correspond to the Camel routes shown in Figure 6-3.

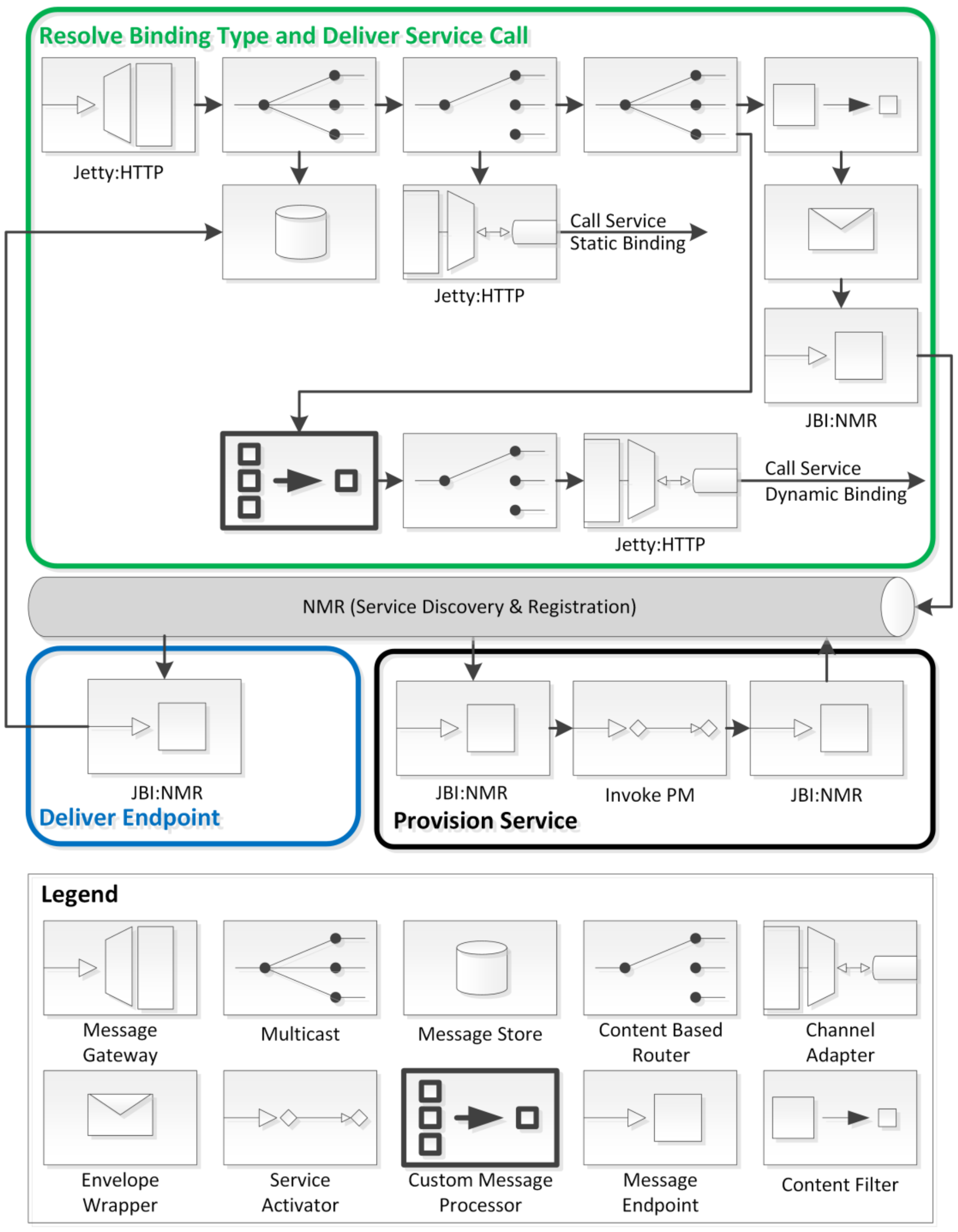


Figure 6-4: Concept II – Routing EIP Diagram

Incoming messages from the Workflow Engine are received by the *Resolve Binding Type and Deliver Service Call* route through a Message Gateway on Jetty (see top left corner of Figure 6-4). The following steps of saving the message to a Message Store and directly routing the service call to a service in case of a static binding are identical to Concept I.

The difference in this route is that it also encompasses the logic for routing service calls for which a dynamic binding is required. In Figure 6-2 of Concept I another route that contained an aggregator was activated. This was explained in the previous “Architectural Overview” chapter, as the CMP blocks the route until it has acquired information about the message’s endpoint. If a dynamic binding strategy has been detected, the message is send to the CMP as well as through several intermediaries to start the necessary workflow model in ODE which is connected to the NMR. The *Provision Service* route is identical to the one in Concept I.

The *Deliver Endpoint* route differs from the one in Concept I. A Camel endpoint on the NMR receives the messages from ODE. These messages contain the endpoint of a service and are sent out by an instance of a workflow model that implements the dynamic binding part of the ESB Control Flow, see also Figure 5-13. Afterwards this route updates the database entry that contains the previously stored original service call with the new endpoint, so that this service call is now ready to be forwarded to its intended receiver. This change is registered by the CMP, it unblocks the route and the message is forwarded to the service via a Content Based Router and Jetty serving as a Channel Adapter.

By introducing a blocking component to the architecture it is ensured that in the event of a synchronous request from ODE-PGF the routing of the service call stays synchronous while other parts of the ESB Control Flow can still interact in an asynchronous manner. This approach is also the base of the implementation that is the subject of chapter 7.

6.3 Concept III: Leveraging the power of the NMR

This third concept follows a different approach than the previous two concepts. It focuses on JBI with the Normalized Message Router and their capabilities and less on Apache Camel routes. The NMR is a powerful component that can automatically route messages to an endpoint that provides the necessary capabilities required by a message. The architectural approach is depicted in Figure 6-5. As was the case in the previous discussed concepts, Figure 6-5 only shows how to route a service call to a matching service. Additionally this concept requires a different kind of realization of the ESB Control Flow than the previous two concepts. The most part of the ESB Control Flow would need to be implemented via workflow models, as Camel routes play only a very little role in this concept. This is because this concept relies heavily on the routing capabilities of the NMR.

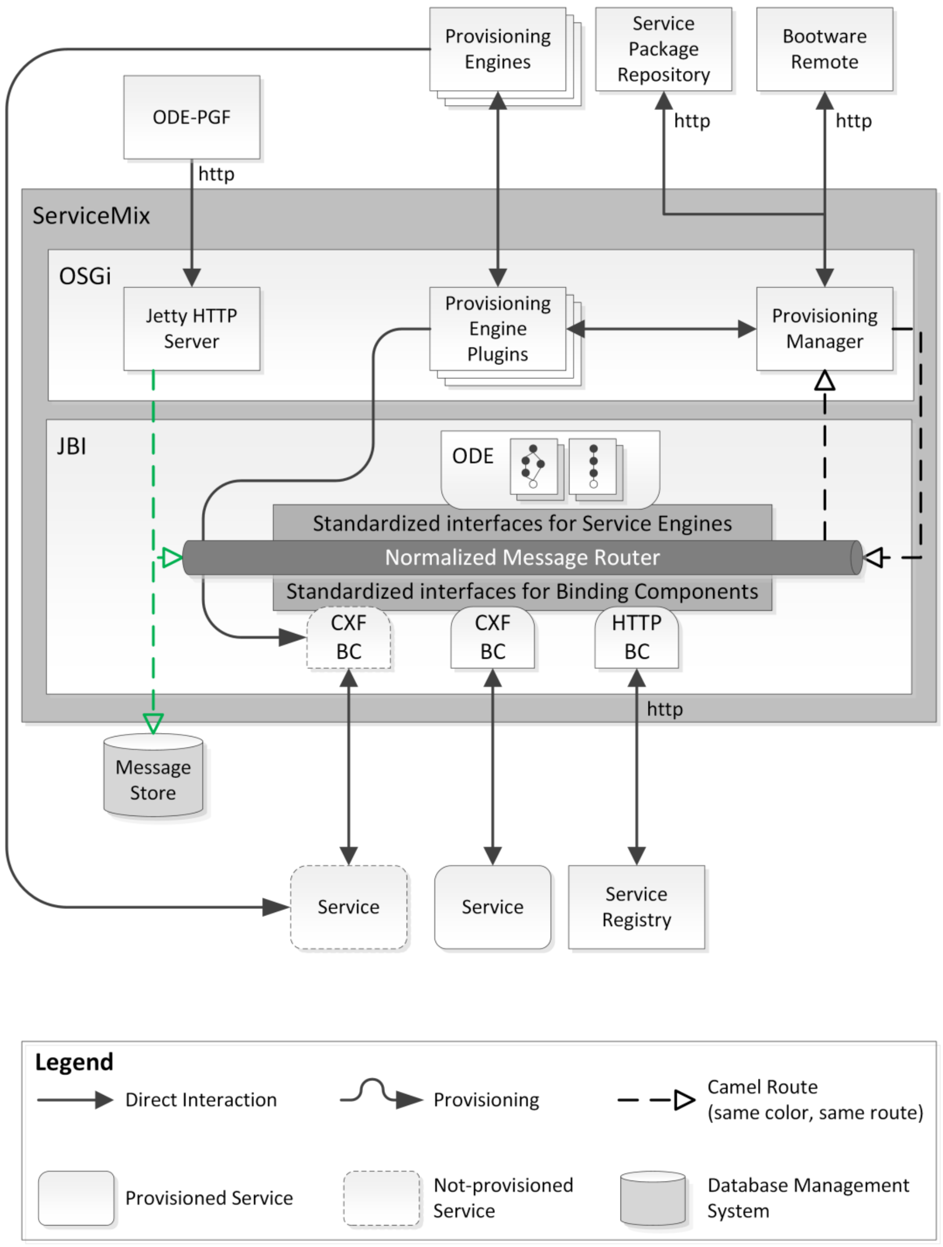


Figure 6-5: Concept III – Architectural Overview

As in the previous concepts, ServiceMix as the implementation of an ESB is the central component along with its OSGi and JBI environment. *ODE-PGF*, *Provisioning Engines*, *Service Package Repository* and *Bootware Remote* are shown above ServiceMix, while *Message Store*, a not-provisioned and provisioned *Service* and *Service Registry* are displayed below ServiceMix. A key difference to the previous approaches is that services are connected to the NMR via binding components (BC).

A Camel route receives the service calls from ODE-PGF via a Jetty HTTP Server, saves the message in a Message Store and delivers it to the NMR (green route). ODE, i.e. a deployed workflow model, receives the message. If a static binding is used ODE sends the message back on the NMR which then routes the message to a matching BC that connects the service to the NMR. If some kind of dynamic binding is required, ODE consults the Service Registry about available services and, if necessary, orders the Provisioning Manager to provision this service (black route).

Since a service must be connected to the NMR, its binding component has to be created and deployed dynamically when a service gets provisioned. In Figure 6-5 a Provisioning Engine Plugin supports this functionality. The binding component has to be deployed after the service is provisioned, because otherwise the endpoint of the service might be unknown.

After both the service and its corresponding binding component are provisioned, ODE sends the service call back into the NMR which then takes care of delivering the message to the right BC and service.

The great advantage this approach provides is that by using binding components protocol independence is achieved. For example, a service might be only available via JMS. This service cannot be engaged through either the approach of Concept I or II, as both require a service to be available over HTTP. Binding components provide a means to easily connect external services with the NMR, independent of specific protocols. A disadvantage is that additional provisioning logic is required to deploy BCs dynamically on the NMR.

6.4 Routing Response Messages

The previously discussed architectural concepts were about how to forward the original service call to the actual service. The subject of this chapter is how to route replies that the ESB receives from called services back to the Workflow Engine. There are two scenarios which have to be taken into account; an asynchronous callback and a synchronous reply from the service.

The approach introduced in this chapter is heavily dependent on WS-Addressing [WAC06]. As was already mentioned in chapter 5.5 “Routing Messages to the ESB”, WS-Addressing defines multiple SOAP headers that are useful for defining a message’s origin, destination, intent and more. This makes it mandatory that the Workflow Engine implements WS-Addressing and uses it on all outgoing messages. Services that provide asynchronous message exchanges also must support WS-Addressing. To identify the type of message exchange that should be used to communicate with a service, the ReplyTo header field of WS-Addressing is used. The ReplyTo header field indicates the intended receiver of replies to this request. In an asynchronous message exchange this would be an endpoint reference, identifying the address of the intended receiver of replies. In Listing 6-1 a snippet of the SOAP headers of a request message is shown that is part of a synchronous message exchange.

```
1  <soapenv:Header>
2  ...
3  <wsa:ReplyTo>
4     <wsa:Address>
5        http://www.w3.org/2005/08/addressing/anonymous
6     </wsa:Address>
7  </wsa:ReplyTo>
8  ...
9  </soapenv:Header>
```

Listing 6-1: WS-Addressing – Synchronous Message Exchange

As can be seen in line 5 of Listing 6-1, the value of the *Address* property in *ReplyTo* is not an endpoint. If no explicit endpoint has been specified for the response message, the predefined URI “http://www.w3.org/2005/08/addressing/anonymous” is used as a value for the Address property [WAC06]. This is the case in synchronous message exchanges as the initial requestor is also the intended receiver of the reply message. Thus this property is used to identify the kind of MEP a request message is part of.

6.4.1 Forwarding Synchronous Replies

A synchronous message exchange is of blocking nature. At the requestor’s side the same thread that sent out the initial request waits until it receives a reply from the service. In case of a synchronous service call, a route is established that links the Workflow Engine, i.e. ODE-PGF, with the synchronous service and the ESB as an intermediary. Figure 6-6 illustrates this approach.

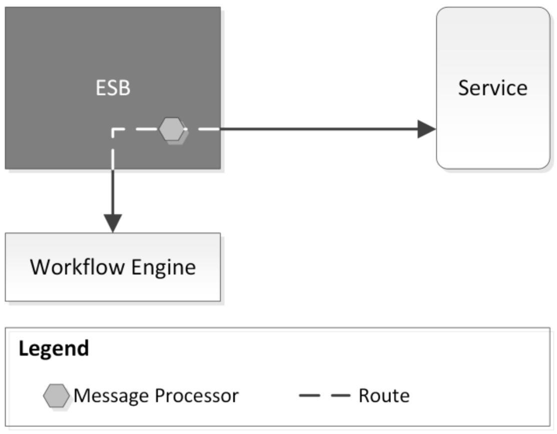


Figure 6-6: Bridging Connection between Workflow Engine and Service

A *Message Processor* is part of a route that connects the *Workflow Engine* via the *ESB* with a *Service*. Every message that travels along this route passes the Message Processor. This component is used to detect an incoming response message in a synchronous message exchange. When the called service sends its response through the route, the Message

Processor initiates the deprovisioning workflow model (see Figure 5-15), which deals with the possible unregistration and deprovisioning of the service instance.

6.4.2 Forwarding Asynchronous Callbacks

In case of an asynchronous callback the service sends its reply message to an endpoint provided by the ESB. In Figure 6-7 it is shown how to make this endpoint known to the service.

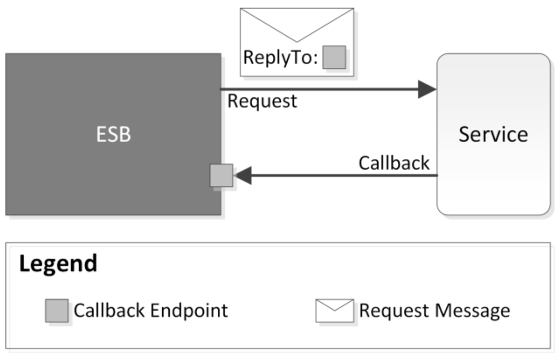


Figure 6-7: Informing a Service about a Callback Endpoint

This endpoint needs to have been specified in the request message, the ESB sent to the service. This is done by setting the WS-Addressing *ReplyTo* header's value of the request message to target this endpoint on the ESB.

After the callback has been received by the ESB it needs to be forwarded to the original callback address that was specified by the Workflow Engine. The metamodel of forwarding the callback message of a service from the ESB to the Workflow Engine is shown in Figure 6-8 below. This model is a more detailed version of Figure 5-14 "Routing Model: Routing Service Replies" of chapter 5.7 "Partitioning the ESB Control Flow".

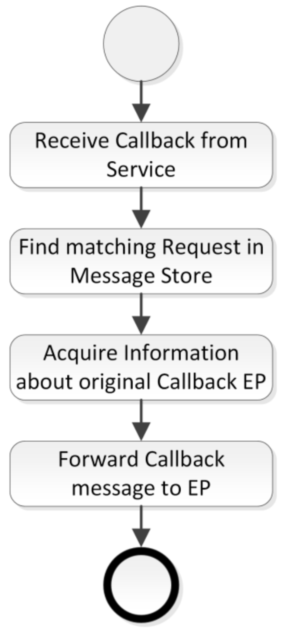

Receive Callback from Service
Find matching Request in Message Store
Acquire Information about original Callback EP
Forward Callback message to EP

Figure 6-8: Forwarding callbacks to the Workflow Engine

After the callback message has been received, the original request message that preceded this callback has to be looked up in the message store. This is done by utilizing WS-Addressing properties. In Listing 6-2 a snippet from a request message is shown that includes the message's unique identifier (see line 4).

```
1  <soapenv:Header>
2  ...
3  <wsa:MessageID>
4     uuid:hqejbhcnphr98b8gtx7b79
5  </wsa:MessageID>
6  ...
7  </soapenv:Header>
```

Listing 6-2: WS-Addressing – Request Message

MessageID is an optional attribute, but it is required in the context of this thesis that the Workflow Engine specifies a unique value for this attribute in every message it sends out. An example reply message is shown in Listing 6-3. WS-Addressing defines that a reply message must reference the related message's ID [WAC06]. This means that every response must contain a reference to its preceding request message.

```
<soapenv:Header>
 ...
 <wsa:RelatesTo>
    uuid:hqejbhcnphr98b8gtx7b79
 </wsa:RelatesTo>
 ...
</soapenv:Header>
```

Listing 6-3: WS-Addressing – Reply Message

This message would indicate that it is the reply to the message of Listing 6-2. This means that the original request message can be easily identified in the message store. The next step in Figure 6-8 is to acquire the callback address defined in the original request message. After this information has been obtained, the callback can be forwarded to the intended location. As was shown in Figure 5-15 "Workflow Model: Unregister and Deprovision Service Instance" the Service Registry must be updated and the service potentially deprovisioned. In Figure 6-9 an architectural overview of this process is given. The process consists of receiving asynchronous callback messages from services and subsequently routing them back to the Workflow Engine, i.e. *ODE-PGF*, as well as deprovisioning these services.

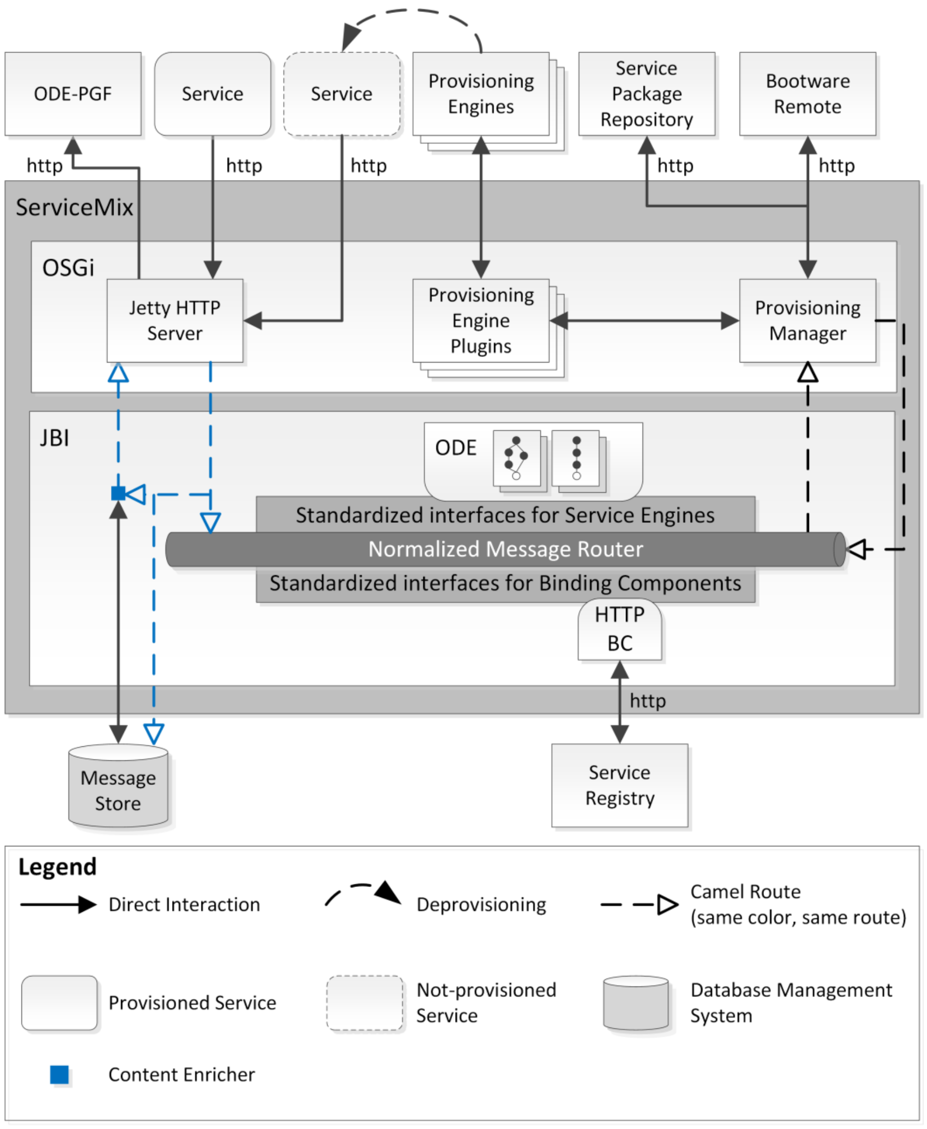


Figure 6-9: Returning Asynchronous Replies – Architectural Overview

All components are identical to the architecture of Concept II (see also chapter 6.2.1). In the center ServiceMix, an implementation of an ESB, along with its OSGi and JBI environments is shown. Around the edges of ServiceMix are various external components that communicate with different internal components of ServicMix. Camel Routes are displayed as colored dashed lines. The difference to the architecture of Concept II is that in Figure 6-9 only routes are shown that are relevant in the context of returning callbacks. In Figure 6-3 “Concept II – Architectural Overview” only routes that are necessary for a successful on-demand provisioning of a service and its infrastructure along with its subsequent invocation were depicted.

Central point for in and outgoing messages is the *Jetty HTTP Server*. The blue Camel route is responsible for saving incoming callbacks persistently in the *Message Store*, forwarding the callback back to ODE-PGF, and invoking a workflow model instance on *ODE* that is tasked with unregistering the service call from the *Service Registry* and possibly ordering the deprovisioning of the service (see also Figure 5-15). The *Content Enricher* extends the result message from the service with the correct callback address that was specified in the initial request message from ODE-PGF.

If the service needs to be deprovisioned, the *Provisioning Manager* is ordered to undeploy the service along with its underlying infrastructure and middleware (black route). The Provisioning Manager selects a suitable *Provisioning Engine Plugin*, which in turn commissions a *Provisioning Engine* to undeploy the service and its infrastructure.

In Figure 6-10 a model of the internal logic of the different Camel routes is shown. The routing model consists of different EIPs, message flows, and the NMR. The different colors correspond to the Camel routes shown in Figure 6-9.

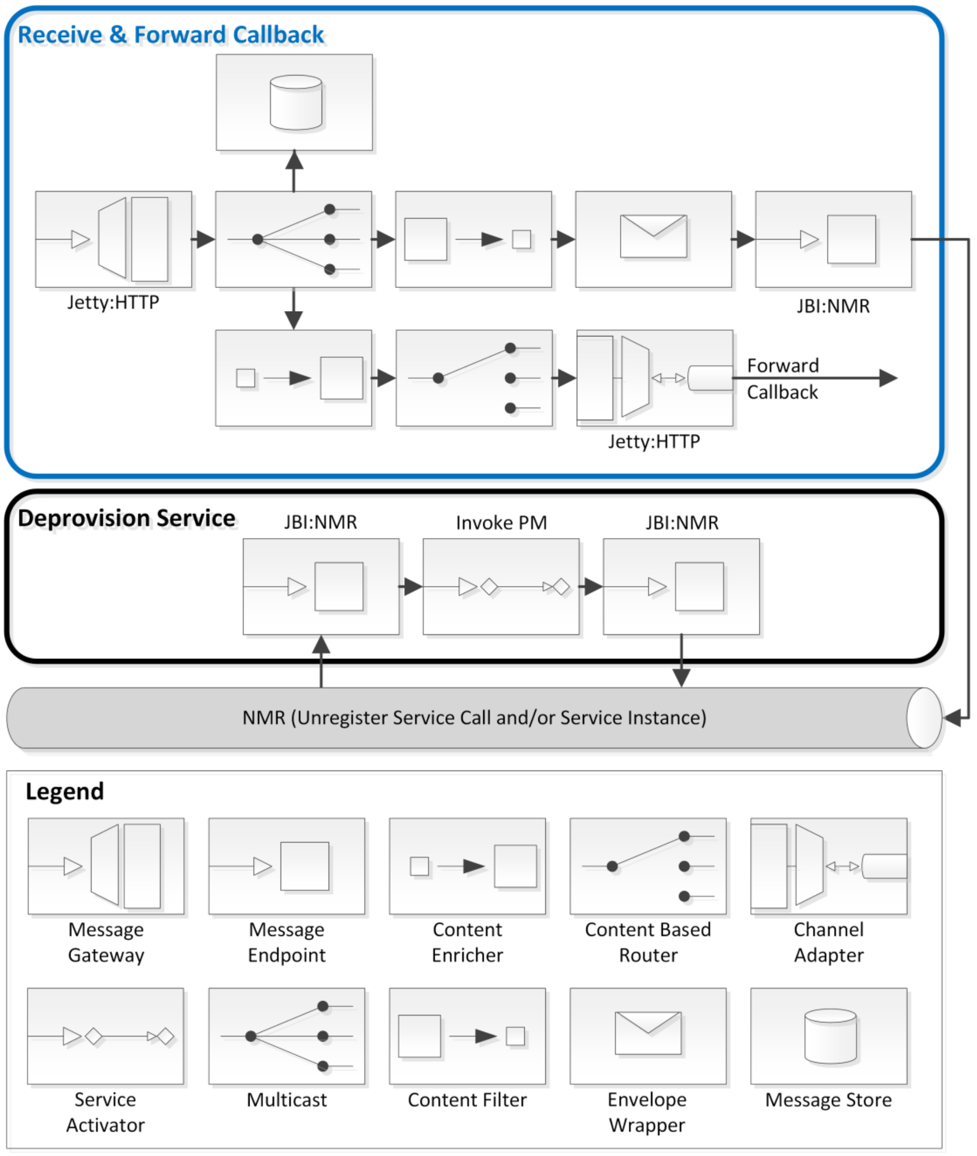


Figure 6-10: Returning Asynchronous Replies – EIP Diagram

The blue *Receive & Forward Callback* route receives a message on a *Message Gateway* through Jetty. This is a HTTP-endpoint that is defined as the callback address for all of the ESB's outgoing request messages. Afterwards a Multicast sends a copy of the message to the *Message Store*, as well as to a *Content Filter* and *Content Enricher*.

The Content Enricher looks up the matching request message to this callback in the message store and extends the callback message with the correct callback address that was specified in the initial request message. Then the callback is forwarded via a Content Based Router and a Channel Adapter on Jetty.

The *Content Filter*, along with the following *Message Wrapper*, prepares the message to invoke a workflow model instance on ODE in the JBI environment that takes care of updating the Service Registry and potentially issuing a deprovisioning of the service.

The *Deprovision Service* route is similar to the Provision Service route in Figure 6-4 of Concept II. An endpoint on the NMR receives a request from a workflow model instance and invokes the deprovisioning logic of the Provisioning Manager. The result from the Provisioning Manager is returned back on the NMR.

7 Implementation

For the implementation part of this thesis the concept introduced in chapter 6.2 “Concept II: Using a Custom Message Processor” was used. This is due to the fact that concept “Concept I: Using an Apache Camel Aggregator” is unable to handle synchronous service calls and “Concept III: Leveraging the power of the NMR” was out of scope of this thesis.

The following software and technologies were used for the implementation. Here, only a quick overview is given as most of them have already been introduced in chapter 2 “Fundamentals”.

- ESB^{MT}: A multi-tenant enabled open source implementation of an ESB. Based on Apache ServiceMix 4.3.0.
- Apache Camel 2.6: A Routing engine and bundled with ServiceMix.
- Apache Karaf with Apache Felix OSGi Framework and bundled with ServiceMix.
- ODE-PGF inside an Axis2 Integration Layer.

The implementation very closely follows the proposed architecture in Figure 6-3 and Figure 6-4 of chapter 6.2 “Concept II: Using a Custom Message Processor”. In this chapter an overview over different interesting aspects of the implementation is given. Additionally to the implementation of the concept, a plugin for the OpenTOSCA Provisioning Engine has also been developed as part of this thesis. This enables a complete demonstration of the on-demand provisioning of a service and its underlying infrastructure and middleware; starting from the ESB receiving the original service call, to provisioning the service, forwarding the service call and returning its reply back to the Workflow Engine, and the subsequent deprovisioning of the service.

All classes shown in class diagrams of this chapter only depict methods. Additionally, to increase readability all methods are displayed without their respective parameters.

7.1 Bug Fixing and Workarounds

Apache ServiceMix 4.3.0 was released in March 2011 which was at the date of this writing more than three years ago. This means that a lot of bug fixes that have been introduced in the meantime are missing in the version that is used in this thesis.

The internal ODE of Apache ServiceMix has a bug that results in ODE being unable to connect to its internal database after a restart of ServiceMix[27]. This prevents the execution of any workflow on ODE. It is possible to reinstall ODE in ServiceMix after the restart and subsequently redeploy all workflow models. But this cumbersome procedure would need to be repeated after every restart. As this is unacceptable, this bug had to be fixed by hand, as newer versions of ODE would not run on ServiceMix 4.3.0. Another solution is to configure an external database to be used with ODE, as this bug only manifests when using ODE's internal database.

A bug in Apache Camel in ServiceMix 4.3.0 prevents the establishment of routes between the NMR of JBI and services in OSGi[28]. In this thesis a simple workaround is used to establish routes between JBI and OSGi (see Listing 7-1 and Listing 7-2).

```
from("jbi:endpoint:http://localhost/osgi/input")
.to("vm:osgi");
```

Listing 7-1: Apache Camel – JBI to OSGi Workaround Example

In Listing 7-1 an example route is shown that connects a Camel endpoint in JBI with a Camel VM endpoint. This type of Camel endpoint has the ability to communicate across different CamelContext instances, amongst other features [CVM14]. The route is defined via the Java DSL of Apache Camel. The above example would be packaged as a Service Unit inside a Service Assembly which is then deployed in the JBI container of ServiceMix.

In Listing 7-2 an example Camel route is defined via OSGi blueprint which is specified in [OSE10]. Blueprint allows a developer to define OSGi services by means of an XML configuration file. Apache Camel supports this notation and thus it is possible to define routes through Blueprint. The XML file can then be simply dropped into the hot-deployment directory of ServiceMix.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
   <route>
      <from uri="vm:osgi" />
         <setBody>
            <method ref="localhost.MyService" method="echo"/>
         </setBody>
      <to uri="vm:foo" />
   </route>
</camelContext>
</blueprint>
```

Listing 7-2: Apache Camel – OSGi Blueprint Route

[27] https://issues.apache.org/jira/browse/ODE-302
[28] https://issues.apache.org/jira/browse/SMXCOMP-877

The route in Listing 7-2 receives messages on the vm:osgi endpoint (line 4). Afterwards it calls a method of a service with the message body as a parameter (line 6) and passes the result to another endpoint (line 8). Both example routes of Listing 7-1 and Listing 7-2 are now able to work together. Thus messages can be passed from the JBI endpoint defined in Listing 7-1 (line 1) to the service referenced in line 6 of Listing 7-2. This creates a bridge for messages between the JBI and the OSGi container of ServiceMix.

7.2 Provisioning Manager and Plugins

In chapter 5.2 the plugin architecture of the Provisioning Manager was discussed. Its main feature is to provide a stable interface for provisioning and deprovisioning of services. Due to its modular architecture the Provisioning Manager along with its plugins is realized as an OSGi service.

In Figure 7-1 a UML class diagram of the Provisioning Manager bundle and its plugin bundles are shown.

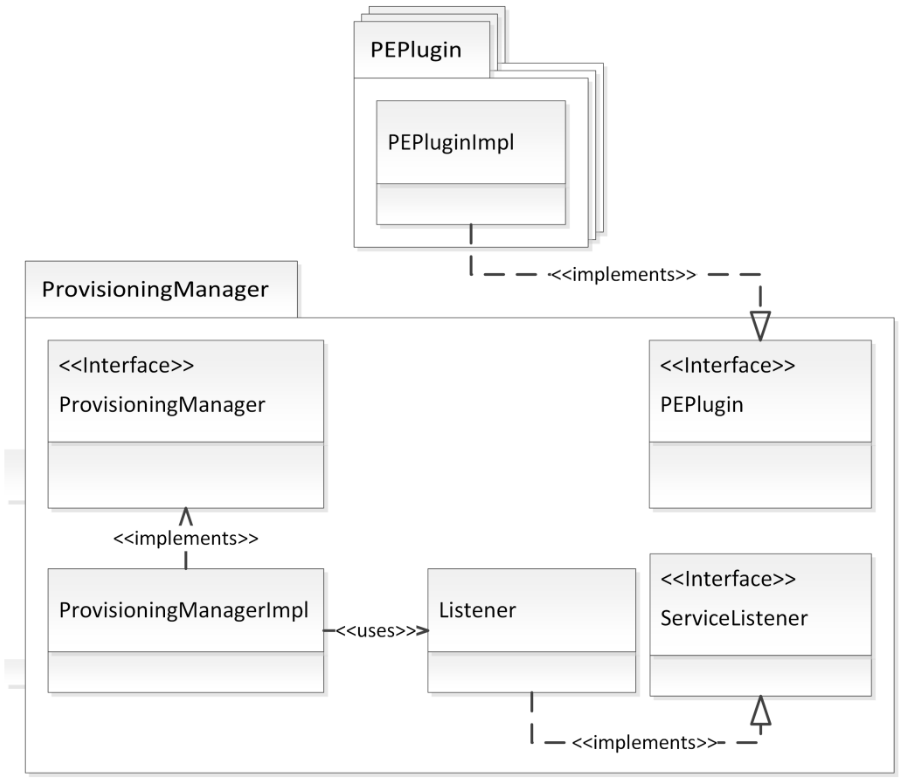


Figure 7-1: Provisioning Manager and Plugins – Class Diagram

The diagram shows important interfaces and classes of the Provisioning Manager and Plugins for Provisioning Engines. The OSGi service the Provisioning Manager offers, is defined via the *ProvisioningManager* interface which has two public methods, *provisionService* and *deprovisionService*. This interface is implemented by the

ProvisioningManagerImpl class. The main application logic of the PM resides in this class, including communication with the Bootware Remote and the selection process to find a matching plugin for a provision/deprovision request. To enable this selection process, the Provisioning Manager uses the *Listener* class. Listener implements the ServiceListener[29] interface, which is defined by the OSGi framework. The Listener class registers itself with the OSGi framework and is configured in such a way, that it is notified whenever a service implementing the PEPlugin interface changes its state. These state changes include the registration of new plugins and the unregistration of plugins that will be no longer available. By this means the Provisioning Manager has an always up to date list of all available Provisioning Engine Plugins.

PEPlugin is an interface, the Provisioning Manager provides, which needs to be implemented by Provisioning Engine Plugins. The interface defines two methods *deploy* and *undeploy*. Plugin developers than provide different implementations of this interface as separate bundles (PEPlugin). By providing this interface with the Provisioning Manager, it is ensured that all plugins have to conform to this stable interface.

7.3 OpenTOSCA Plugin

The OpenTOSCA plugin is a Provisioning Engine Plugin that enables the Provisioning Manager to deploy suitable CSAR service packages via OpenTOSCA. A class diagram displaying important aspects of the plugin is shown in Figure 7-2.

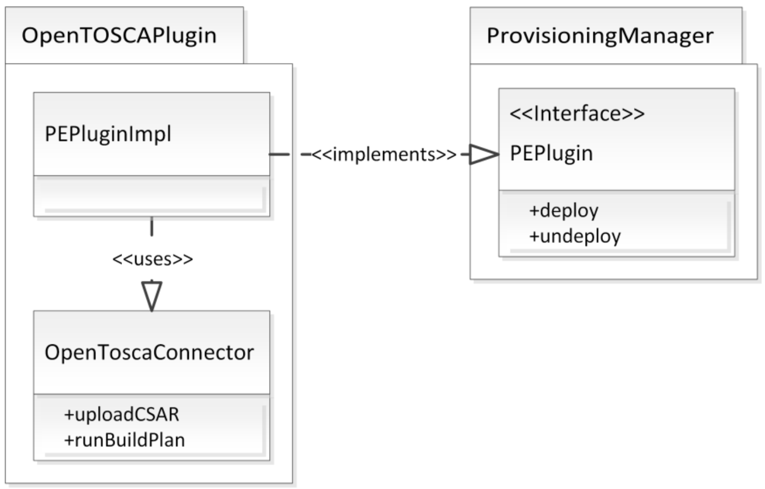


Figure 7-2: OpenTOSCA Plugin – Class Diagram

The *PEPluginImpl* class implements the mandatory service interface provided by the Provisioning Manager. Methods that are specific to OpenTOSCA are bundled in the *OpenToscaConnector* class. OpenTOSCA requires that a CSAR is uploaded to its system prior to starting the actual deployment. This is realized by the *uploadCSAR* method. The

[29] http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/ServiceListener.html

method uses the URL to the referenced service package, which is provided by the Provisioning Manager when calling the plugin, to upload the CSAR to OpenTOSCA. Afterwards the *runBuildPlan* method starts the build plan execution of the CSAR in OpenTOSCA (see also chapter 2.12 "OpenTOSCA").

7.4 Dynamic Binding Axis2 Module

In Figure 7-3 the class diagram of the Axis2 module is shown, that is used to route messages from ODE-PGF to ServiceMix.

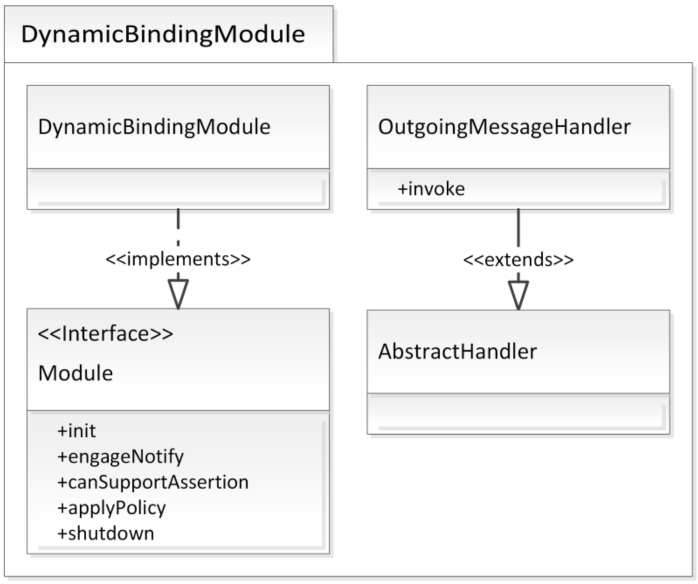


Figure 7-3: DynamicBinding Module – Class Diagram

The module follows the architecture proposed in chapter 5.5.3. It needs to be able to route messages to the ESB, but not interfere with messages that are sent back to the client from ODE-PGF. Every Axis2 module must provide an implementation of the *Module*[30] interface. This implementation is provided by the *DynamicBindingModule* class. The methods defined by the interface can be used to further define the processing of messages in this module, e.g. depending on policies. *OutgoingMessageHandler* extends the *AbstractHandler*[31] class of Axis2. It provides only one method, invoke, which is called on every message that passes through this handler. The logic of this method determines if the message should be rerouted or not and extends the SOAP header bock of the message with additional headers that are required to perform a successful dynamic binding by the ESB (see also chapter 5.5 "Routing Messages to the ESB").

[30] http://axis.apache.org/axis2/java/core/api/org/apache/axis2/modules/Module.html
[31] http://axis.apache.org/axis2/java/core/api/org/apache/axis2/handlers/AbstractHandler.html

Furthermore a module must provide metainformation in form of an XML configuration file (module.xml). In Listing 7-3 the module.xml file of the Dynamic Binding module is shown.

```
1   <module name="DynamicBindingModule"
2           class="org.simtech.axismodule.DynBindingModule">
3      <Description>
4         Module that inserts Dynamic Binding specific headers.
5      </Description>
6      <OutFlow>
7         <handler name="OutgoingMessageHandler"
8                 class="org.simtech.axismodule.MessageOutHandler">
9            <order phase="MessageOut"/>
10        </handler>
11     </OutFlow>
12  </module>
```

Listing 7-3: Axis2 Module – module.xml

The content of line 1 and 2 defines a name for this module and point to the class implementing the Module interface. In line 3 a general description about this module is provided. This information is later displayed in the web admin console of Axis2. Afterwards the specific flow and order of each handler contained in this module is defined (lines 6-11). In this case the Dynamic Binding module encompasses only one handler, *MessageOutHandler*, which is referenced in line 8.

This module is then embedded into the Axis2 Integration Layer of ODE-PGF. This provides a loosely coupled solution, as the module can be easily deployed inside other Axis2 environments.

8 Summary and Future Work

In chapter 2 of this thesis an overview of various technologies and their implementations that are useful in the context of on-demand provisioning of services was given. Mostly these are software that is used in this thesis, e.g. Apache ServiceMix, Apache Camel and OpenTOSCA. Previous work in the scope of SimTech has been analyzed in chapter 4that formed the basis of this thesis. This included an overall architecture of a SWfMS and what binding strategies are used for the on-demand provisioning of services and their underlying infrastructure and middleware. In chapter 5 different architectural and design decisions were introduced and analyzed. This included an overview of which components of the overall architecture have been modified and a closer look on each of these components. Afterwards it was explained how the internal control logic of the ESB has been divided into several models. Different architectural approaches were the subjects of chapters 6.1, 6.2 and 6.3. Each approach has had its own notion and drawbacks. Based on these, one concept has been chosen that forms the basis of the prototypical implementation. Furthermore the difference and handling of synchronous replies and asynchronous callbacks has been addressed in chapter 6.4. An overview of the prototypical implementation was given in chapter 7.

Currently services are deprovisioned as soon as they return their result. This may cause unnecessary re-provisioning and deprovisioning if a service is called multiple times throughout a workflow. In the future it may be possible to analyze the entire workflow first and then pass all requirements to the ESB. The ESB can then analyze after which service call a service may be deprovisioned since it will be no longer needed in the current workflow.

There are multiple other possible subjects that make for interesting future work. In particular how the architectural concept introduced in chapter 6.3 “Concept III: Leveraging the power of the NMR” may be further refined and prototypically implemented. As this would loosen the current coupling to the HTTP protocol on which all service interactions are based. It would provide a more protocol independent approach and subsequent higher powerfulness of the system. Additionally it may be researched how the introduction of messaging queues may help with providing persistency and stability to the overall system, e.g. via Apache ActiveMQ[32] which is bundled with ServiceMix.

The concepts developed in this thesis only focus on a simple service discovery via functional properties. Yet, service discovery is only one part of a full service selection process. In such a process non-functional properties have also to be taken into account. Non-functional properties may, for example, be specified via the use of WS-Policy [WSP07]. The Service Registry, which was only treated as a mockup in this thesis since it

[32] http://activemq.apache.org/

was a necessity to the overall system, is also part of future works. In [VHK+14] a possible future metamodel is described that is crucial for a proper service selection process. An extended architecture for service selection is also introduced. Additionally [VHK+14] talks about a service package selection process which follows the previous service selection to match provisioning requirements with provisioning capabilities.

The metamodel of the Service Package Repository, which was introduced in chapter 5.4, has a Requirements entity which holds additional technical information about a service package. It has to be evaluated if further metainformation about a service package has to be stored in the Service Package Repository.

In chapter 5.1 "Provisioning Manager as an Internal/External Component" it was mentioned that the Provisioning Manager and the Bootware Remote might be consolidated into one component. This idea was discarded for this thesis and the parallel thesis of Reinfurt [Rei14]. It would have introduced additional complexity to both works and be out of scope of the theses. Nevertheless it has to be evaluated if this idea is viable. Both components are based on the OSGi framework and as such provide modularity and loose coupling. They are even both using the same implementation of an OSGi framework, Apache Felix. This should ease the integration of both components into a joint component. Possible advantages are the reduction of the overall complexity of the system and a more convenient routing logic.

Another idea is to introduce a Plugin Repository to the architecture. This would be especially useful if Provisioning Manager and Bootware Remote were forged into one component. A plugin repository would store different plugins, which for example enable the provisioning of services via different Provisioning Engines. In contrast to the current design where plugins for the Provisioning Manager have to be run on the ESB's OSGi framework, they could be hosted remotely and be loaded dynamically if required. This would decouple the creation and development of Provisioning Engine Plugins from the core SWfMS system and allow third parties to create plugins for the Provisioning Manager and Bootware Remote. The Service Package Repository follows the same concept, where service packages can be created and uploaded independently from the rest of the SWfMS. Thus, a Plugin Repository would be a reasonable step towards increased modularity and loose coupling. As the Provisioning Manager, the Bootware Remote and their respective plugins run on the Apache Felix OSGi framework, it may be a good start to look at the Apache Felix OSGi Bundle Repository[33] which also supports the OSGi Repository Specification defined in [OSG12].

[33] http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

9 List of Figures

10 List of Listings

11 References

[AAG12] The Apache Software Foundation: *Apache Axis2 Architecture Guide.* Available: http://axis.apache.org/axis2/java/core/docs/Axis2ArchitectureGuide.html

[Bar09] N. Bartlett: *OSGi in Practice.* Available: http://njbartlett.name/files/osgibook_preview_20091217.pdf

[BBH⁺13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner: *OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications.* In Lecture Notes in Computer Science, *Service-Oriented Computing*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 692–695.

[CBC⁺05] C. H. Crawford, G. P. Bate, L. Cherbakov, K. Holley, and C. Tsocanos: *Toward an on demand service-oriented architecture, IBM Syst. J,* vol. 44, no. 1, pp. 81–107, 2005.

[Cha04] D. A. Chappell: *Enterprise Service Bus*. Beijing, Cambridge, Farnham, Köln, Sebastopol, Taipei, Tokyo: O'Reilly, 2004.

[CVM14] The Apache Software Foundation: *Apache Camel: VM Component.* Available: http://camel.apache.org/vm.html

[EIP12] G. Hohpe: *Enterprise Integration Patterns.* Available: http://www.eaipatterns.com

[GSK⁺11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter: *Conventional Workflow Technology for Scientific Simulation.* In Computer Communications and Networks, *Guide to e-Science*, X. Yang, L. Wang, and W. Jie, Eds, London: Springer London, 2011, pp. 323–352.

[GTK⁺10] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio: *Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services, IEEE Trans. Serv. Comput,* vol. 3, no. 3, pp. 223–235, 2010.

[GTS+09] D. Guinard, V. Trifa, P. Spiess, B. Dober, and S. Karnouskos: *Discovery and On-demand Provisioning of Real-World Web Services, 2009 IEEE International Conference on Web Services*, pp. 583–590, 2009.

[HoW10] G. Hohpe and B. Woolf: *Enterprise Integration Patterns: Designing, building, and deploying messaging solutions,* 14th ed. Boston, Mass, Munich: Addison-Wesley, 2010.

[IbA11] C. Ibsen and J. Anstey: *Camel in Action*. Greenwich, Conn: Manning, 2011.

[IBM05] IBM: *Develop Web services with Axis2, Part 1: Deploy and consume simple Web services using the Axis2 runtime.* Available: http://www.ibm.com/developerworks/library/ws-webaxis1/

[JBI05] R. Ten-Hove and P. Walker: *Java Business Integration (JBI).* Available: https://jcp.org/en/jsr/detail?id=208

[JMX00] *Java Management Extensions.* Sun Microsystems Inc, 2000, http://download.oracle.com/otndocs/jcp/7770-jmx-1.0-fr-spec-oth-JSpec/.

[NIS11] P. Mell and T. Grance: *The NIST definition of cloud computing*. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

[NMR08] The Apache Software Foundation: *5. JBI - Apache ServiceMix - Apache Software Foundation.* Available: https://cwiki.apache.org/confluence/display/SM/5.+JBI#id-5.JBI-NormalizedMessageRouter

[ODA13] The Apache Software Foundation: *Apache ODE - Architectural Overview.* Available: http://ode.apache.org/developerguide/architectural-overview.html

[ODE13] The Apache Software Foundation: *Apache ODE - Endpoint Configuration.* Available: http://ode.apache.org/endpoint-configuration.html

[OSE10] The OSGi Alliance: *OSGi Service Platform Enterprise Specification: Release 4, Version 4.2.* Available: http://www.osgi.org/download/r4v42/r4.enterprise.pdf

[OSG09] The OSGi Alliance: *OSGi Service Platform Core Specification: Release 4, Version 4.2.* Available: http://www.osgi.org/download/r4v42/r4.core.pdf

[OSG12] The OSGi Alliance: *OSGi Core Release 5.* Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf

[Pap08] M. Papazoglou: *Web Services: Principles and Technology*: Pearson Education, 2008.

[Rei14] L. Reinfurt: *Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments*. Diploma Thesis 3616, 2014.

[SAG+13] S. Strauch, V. Andrikopoulos, S. Gómez Sáez, and F. Leymann: *Implementation and Evaluation of a Multi-tenant Open-Source ESB.* In Lecture Notes in Computer Science, *Service-Oriented and Cloud Computing*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, K.-K. Lau, W. Lamersdorf, and E. Pimentel, Eds, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 79–93.

[SAL+12] S. Strauch, V. Andrikopoulos, F. Leymann, and D. Muhler: *ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses.* In 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 456–463.

[Sch13] V. Schneider: *Dynamic Provisioning of Web Services for Simulation Workflows*. Diploma Thesis 3473, 2013.

[TOS13] *Topology and Orchestration Specification for Cloud Applications Version 1.0.* OASIS Standard, 2013, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html.

[VCB09] C. Vecchiola, X. Chu, and R. Buyya: *Aneka: A Software Platform for .NET-based Cloud Computing, High Speed and Larce Scale Scientific Computing*, pp. 267–295, 2009.

[VCK+12] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya: *Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka, Future Generation Computer Systems*, vol. 28, no. 1, pp. 58–65, http://www.sciencedirect.com/science/article/pii/S0167739X11000896, 2012.

[VHK+14] K. Vukojevic-Haupt, F. Haupt, D. Karastoyanova, and F. Leymann: *Service Selection for On-Demand Provisioned Services.* In Proceedings of the 18th IEEE International EDOC Conference (EDOC 2014). (Accepted for publication)

[VKL13] K. Vukojevic-Haupt, D. Karastoyanova, and F. Leymann: *On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows, Service-Oriented Computing and Applications (SOCA), IEEE 6th International Conference*, pp. 91–98, 2013.

[WAC06] *Web Services Addressing 1.0 - Core.* World Wide Web Consortium, 2006, http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/.

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

[WSA07] World Wide Web Consortium: *Web Services Addressing Working Group.* Available: http://www.w3.org/2002/ws/addr/

[WSB07] *Web Services Business Process Execution Language Version 2.0.* OASIS Standard, 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[WSP07] *Web Services Policy 1.5 - Framework.* World Wide Web Consortium, 2007, http://www.w3.org/TR/2007/REC-ws-policy-20070904/.

[WSS06] *Web Services Security: SOAP Message Security 1.1.* OASIS Standard, 2006, https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[XPA99] *XML Path Language (XPath) Version 1.0.* World Wide Web Consortium, 1999, http://www.w3.org/TR/1999/REC-xpath-19991116/.

All links were last followed on July 9, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. 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