Diploma Thesis

Modeling of Dynamic Network-based Services

David Hausheer, ETH Zurich, D-ELEK

TIK No. DA-2001.05
Winter Term 2000/2001

Supervisor: Prof. Dr. B. Stiller, ETH Zurich, TIK
Tutor: Dr. M. Feridun, IBM Zurich Research Laboratory

February 28, 2001
Abstract

Application Service Providers (ASP) host services such as software applications for clients, i.e., individual consumers and enterprises. The clients should be able to dynamically create and configure a service for use in their current projects, without much knowledge about the underlying infrastructure and the applications providing the service. Installing such customized services on the ASP infrastructure is a difficult and time consuming process when done manually. Automating this process has obvious advantages for a large number of clients. This, however, requires that the dependencies of a service and its components need to be resolved and provisioned automatically. Hence, a common service composition model is needed.

In this thesis a model for defining services is proposed. This model is based on emerging XML technologies. To validate this model, a software prototype was implemented in Java. Through a simple user interface based on the Java servlet technology a client is able to configure, install and modify a service.
Acknowledgements

This thesis could not have been developed without the generous help of several people. The author wishes to acknowledge his great debt to all those who have contributed to this thesis by discussing problems, explaining things and reading through presentation slides or chapters of this paper: Dr. Metin Feridun as my tutor and Prof. Dr. Burkhard Stiller as my supervisor; Jens Krause, Stefan Pleisch, Dr. Christian Hörtnagl, Anthony Bussani and Dr. Sean Rooney in the Distributed Systems and Network Management group of IBM Research; Christian Rohner, Raimund Brandt and Enikő Fey, particularly for all the discussions on the promenades. Thanks a lot also to the people in the Basketball Team of the IBM Hobby Club for the refreshing Wednesday evenings.
# Contents

Abstract iii

Acknowledgements v

1 Introduction 1
   1.1 Goal of the Thesis ........................................... 2
   1.2 Outline of the Thesis ........................................ 3

2 Service Model Concepts 5
   2.1 Evaluation .................................................. 5
   2.2 Terminology ................................................ 6
   2.3 Service Model ............................................... 7
      2.3.1 Service Element ....................................... 8
      2.3.2 Subservice Element .................................... 9
      2.3.3 Configure Element ..................................... 10
   2.4 Index Model ................................................ 10
      2.4.1 Service Element ....................................... 10
      2.4.2 Instance Element ...................................... 10
   2.5 Package Model .............................................. 11
      2.5.1 Package Element ....................................... 11
      2.5.2 Requires Element ...................................... 11
   2.6 Configuration Model ....................................... 11
      2.6.1 Configure Element ..................................... 12
      2.6.2 Controlscript Element ................................ 12
      2.6.3 Exec Element ........................................ 12

3 Architecture 13
   3.1 ASP Infrastructure .......................................... 13
   3.2 Configuration Server ....................................... 14
      3.2.1 Presentation Layer .................................... 14
      3.2.2 Integration Layer ..................................... 14
      3.2.3 Installation Layer .................................... 15
   3.3 HTTP Client ................................................ 15
CONTENTS

4 Prototype Implementation
   4.1 Login ............................................................ 17
   4.2 Add a Service .................................................. 18
   4.3 Configure a Service .......................................... 19
      4.3.1 Enable or Disable a Subservice ....................... 20
      4.3.2 Configure a Subservice ................................ 21
   4.4 Install the Service .......................................... 21
      4.4.1 Install Method .......................................... 22
   4.5 Various Java Classes ...................................... 26
      4.5.1 Config Class ............................................ 26
      4.5.2 Logger Class ............................................ 27

5 User Manual
   5.1 Installation of the Prototype Application ................ 29
   5.2 Service Integration .......................................... 30
   5.3 Add a Client .................................................. 30

6 Conclusion
   6.1 Future Work .................................................. 31

A Examples
   A.1 Service Model ............................................... 33
   A.2 Index Model .................................................. 34
   A.3 Package Model ............................................... 34
   A.4 Configuration Model ....................................... 35

B Sourcecode
   B.1 Service Class Methods .................................... 37

Bibliography ......................................................... 39

List of Figures ...................................................... 41

List of Tables ....................................................... 43
Chapter 1

Introduction

Today, with the fast evolution of the Internet, there is an emergence of Application Service Providers (ASP) that provide network and server infrastructures to support application hosting. An ASP offers individuals and enterprises access to software applications (which are referred to by the generic term "Services") over the Internet or other network technologies. As network connections become fast enough for dealing with large datasets and programs over the Internet, ASP provided services are expected to become an important alternative to self hosting applications, not only for individuals and smaller companies with low budgets for information technology but also for larger companies as outsourced services.

A simple schema of the ASP environment is shown in Figure 1.1. The clients of an ASP are usually referred to as the Content Providers, e.g. real media providers, web application providers or game providers. Hence, the terms "content provider" and "client" will be used as synonyms. ASPs enable a client to publish content using standard applications, such as web servers or video servers, and run client supplied applications. As an example of a client, we can use a small flower shop that would like to sell flowers over the Internet. In order to do this, it needs a web-based system to publish a flower webpage and run a web application to support commercial transactions. For such a client, it is probably too expensive to buy and
maintain his own server infrastructure to make his services available on the web. Therefore he may contact an ASP which will set up a web server and install the client supplied content and applications on the underlying ASP infrastructure for a fee (usually monthly or yearly). Users, i.e., the end-consumers of the service, can now access this service over the Internet to buy flowers offered by the flower shop. In the following, we do not discuss the interaction between a user and the application service, e.g., the flower shop, in more detail. We focus on the core and interesting aspects of how to enable such an application service, i.e., on the interaction between the client and the ASP.

1.1 Goal of the Thesis

A client should be able to dynamically create new services or modify and configure an already existing service. When done manually, the installation of such customizable services is a difficult and time consuming process for the ASP. As the number of clients of an ASP increases, automated service installation becomes more advantageous. However, the client looses the ability to configure the service. In other words the ASP is no longer flexible enough for different requirements of the clients. Hence an application service configuration interface as shown in Figure 1.2 needs to be developed, that enables a client to create and configure an application service.

![Application Service Configuration Interface](image)

Figure 1.2: Application Service Configuration Interface

The development of such a configuration interface poses two important challenges for an ASP:

- the delegation of the service integration and configuration to the client
- the automated installation of the service

To meet the above challenges an abstract model for application services is developed that describes the structure of the service components used to provide the various functionalities of a service. The challenge of this thesis is to collect and analyze the different aspects, components and interactions of application services and create a service model based on this
When installing application services, it is important to check carefully the dependencies among the service components that may arise. A model that covers these characteristics should define semantics for describing the components that are available, the functionalities they provide and the requirements that need to be fulfilled prior to installation.

For a more comprehensive view of the problem, the following issues need to be addressed. Different services may provide the same functionality. Among them, resource conflicts arise, when the services use the same resources at the same time. In other cases several services with similar behaviour can exist without raising any conflicts. Thus the following problems need to be solved:

- the service components and its functionalities need to be classified
- the dependencies, configuration requirements and conflicts of services need to be modeled
- the configuration parameters of a service need to be localized and described in the service model

Common package management software, such as Red Hat’s Package Manager (RPM) [10] or SuSE’s Yet another Setup Tool (YaST) [16] already handles several of the above problems. However, they can only be used to install services on a single machine. The model and the prototype implementation proposed in this thesis are not intended to replace any of these tools. They use and extend them (particularly RPM) to solve the problems that were described above, which includes a potentially distributed infrastructure.

1.2 Outline of the Thesis

The contents of the thesis are organized into five chapters and two appendixes. The appendixes include some examples of models and code fragments that are referenced within this paper. In the current chapter, the ASP environment was introduced and its characteristics and problems were discussed. Given this environment, the goal for the thesis was described. It was shown how this contribution fits into the current environment. In the next chapter a short technology overview of XML based models is given. After that, the various basic concepts used in this thesis and the models describing a service environment are presented. In Chapter 3 a more detailed description of the architecture is given on which the prototype is implemented. Chapter 4 covers the prototype implementation with its various Java classes and stylesheets that are used for the data transformations. Using a set of examples, the functions performed by this prototype are shown, based on screen shots of the client interface. The user manual in Chapter 5 describes how the prototype can be installed on the system and how new services can be integrated. Finally, in Chapter 6 a short summary is given on
how the work reported in this thesis contributes to the challenges for an ASP. It presents also possible future extensions of the current prototype.
Chapter 2

Service Model Concepts

In this chapter the service model and the underlying concepts are discussed in detail. The prototype described later in Chapter 3 and 4 uses this model as a representation of the data. The model, on the one hand, describes the structure of data, i.e., the dependencies of service components or packages. Defining the service on an abstract level makes the complex problem of service configuration more transparent, e.g., the system dependency. On the other hand, the model serves as a common language or communication protocol between the ASP and the client and it enables the ASP to visualize the service to the client. Finally, the actual state of the installation process can be represented in the model, e.g., the components that are currently installed on the underlying infrastructure. Thus what we get is a formal description of a standardized service model.

2.1 Evaluation

Various emerging technologies have been evaluated for describing the service model. All these technologies are XML based, i.e., its syntax follows the standard of the Extensible Markup Language (XML) [3]. XML is a universal format for structured documents and data on the Web. The reason why only XML based technologies were evaluated for the modeling part is the wide acceptance of XML, its good readability and the possibility to visualize it in a web browser. The following list gives an short overview on these service description technologies and their usage.

**WSDL** is the Web Services Description Language [2]. It is an XML format for describing network services as a set of endpoints operating on messages. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint.

**UDDI** is the Universal Description, Discovery and Integration specification [1]. It defines a way to publish and discover information about Web services. A UDDI-compliant registry provides an information framework for describing services exposed by an entity or business.

**SOAP** is the Simple Object Access Protocol [17]. It is a lightweight XML based protocol for exchange of information in a decentralized, distributed environment.
ADS is the Advertisement and Discovery of Services [8] protocol for Web services. It is
simplifying the announcement of available Web services to inquiring software agents.

The evaluation of these technologies has shown that there is a variety of semantics for
describing how to publish, discover and access a service, but not how to configure a service.
Therefore we decided to develop a new service description model, that uses plain XML syntax
and is not correlated to one of the above technologies. For describing the service model the
Document Type Definition (DTD) syntax is used. DTD is a simple way to describe XML
document classes and can be used to validate the document instances. We could also use
XML schema to describe the model. XML schema [18] is another XML based technology
that provides a means for defining the structure, content and semantics of XML documents.
The advantage of XML schema is a better validation of datatypes. However, to keep the
model description as simple as possible, DTDs are used instead of XML schemas.

2.2 Terminology

At first the core concept is explained by defining the basic terms that are used in the model.

Subservice is a classification for service components that provide a specific sub-function for
a service.

Service is a collection of various hierarchically structured subservices, that together provide
a common functionality.

Service Offering is an instance of a service with a certain service scope and a default service
configuration. This serves as a starting point for the client specific configuration and is
a means of service differentiation.

Provider is a service component, e.g, an application or a module, that provides a specific
subservice. This term should not be mixed up with an Application Service Provider
that was described in Chapter 1.

Service Index is an ordered list or a catalog of services and service offerings.

Package is an archive that contains all the installation files of a specific provider. A package
can have dependencies to other required packages.

Configuration is a set of parameters that describe the client-customized behaviour of a
subservice.

An example of a service is shown in Figure 2.1. For this service called "Webserver" the
service offering level "Economy" is selected. There are four different subservices: an HTTP
Server, provided by Apache, an encryption module (SSL), a scripting language processor
module (PHP) and a database (mySQL) that can be connected to the webserver using the
scripting module. Each of these subservices could be replaced by other providers, therefore
it is not necessary to know these specific technologies. There are dependencies between the
subservices, which are denoted in the connections among the puzzles. For instance, the
database requires the scripting module as an interface in order to be connected to the HTTP
server.
2.3._service_model

Four overlapping submodels have been developed to implement the above concept. In Figure 2.2 the relationship between these submodels is shown. The core service model, defines a service; an generic instance of the service model, i.e., a service offering, can be referenced in the index model, that lists all the provided services and offerings; each subservice in the service model is mapped on a specific provider. The package dependencies of these providers are described in the package model; finally, the configuration model covers the default configuration of a subservice and the respective provider.

2.3 Service Model

This model is the core of the service description. From the above terminology it covers the following elements: service (including a service offering), subservice (including a provider) and configuration. Figure 2.3 shows an example of the service model structure. The depth
of the model is unbounded, therefore only a few elements are shown to illustrate the model hierarchy. In XML, elements are used for a 1:n-relationship, i.e., an element can have several child elements of the same type. In contrast, an attribute describes a 1:1-relationship between the parent element and itself. In Appendix A.1 an example of the service model is shown.

2.3.1 Service Element

The service element is the root element of a service. In a XML document only one root element can exist. This element groups the subservices together and integrates them to one single service. The attributes of the service element are explained beneath. The following description is taken from the Document Type Definition (DTD) of the service:

```xml
<!ELEMENT service (subservices+)>  
<!ATTLIST service name CDATA #REQUIRED>  
<!ATTLIST service class (Economy | Business | First | none) #REQUIRED>  
<!ATTLIST service location CDATA #REQUIRED>  
<!ATTLIST service status (enabled | disabled) #REQUIRED>  
<!ATTLIST service active (true | false) #REQUIRED>  
<!ATTLIST service controlscript CDATA #IMPLIED>
```

As shown above the service element must contain at least one subservice element. Usually, the first subservice element contains the core functionality of the service and is able to integrate the additional subservices. The above attributes have the following meaning:
2.3. SERVICE MODEL

**Name** contains the unique name for the service.

**Class** describes the current service class or, by other words, the service offering selected by the client. By now, one of three different offerings can be selected, i.e., "economy", "business" and "first", or none if there is no service differentiation.

**Location** contains the actual service file location (URL). Usually it is given as a filename in the form `<name>-<class>.xml` (e.g. `webserver-economy.xml`). The path to this file is indicated by the `path-location` attribute of the respective service index (see below).

**Status** indicates whether this service is enabled to be installed, or disabled so that it will be uninstalled during the next installation process. Service uninstallation includes the removing of all involved subservices.

**Active** can either be true or false. It is true when a service control script was found in a subservice configuration and the service with the current scope and configuration could be started successfully after installation on the system.

**Controlscript** indicates the location, i.e., the absolute path of the service control script, if any were found during the installation process. With this control script the later described prototype is able to start and stop the service.

### 2.3.2 Subservice Element

The subservice element serves as a classification for similar providers and can be described as follows:

```xml
<!ELEMENT subservice (subservice*,configure*)>
<!ATTLIST subservice name CDATA #REQUIRED>
<!ATTLIST subservice provider CDATA #REQUIRED>
<!ATTLIST subservice type (required | optional) #REQUIRED>
<!ATTLIST subservice status (enabled | disabled) #REQUIRED>
<!ATTLIST subservice active (true | false) #REQUIRED>
<!ATTLIST subservice address CDATA #IMPLIED>
```

The mapping between subservices and providers is done by the ASP. In a service offering the provider of a subservice is fixed. However, an ASP is free to modify this attribute anytime. As shown above, subservices can contain other subservices as suppliers of additional features.

**Name** contains the unique name for the subservice. This is a category of providers.

**Provider** is the name of the currently deployed provider for this subservice.

**Type** indicates whether this subservice is required by the parent element, or optional.

**Status** has the same meaning as in the service element. This attribute can be "toggled" by the client, if the type is optional.

**Active** can either be true or false. It is true when the installation and configuration of this subservice was successful.

**Address** is a link address (URL) to the currently installed subservice. It is used by the client for testing the subservice after installation.
2.3.3 Configure Element

The configure elements contain parameters that can be used to add client specific configurations to the subservices. These elements can be modified by the client by means of web forms as described later in Chapter 4. The structure of a configure element is as follows:

```xml
<!ELEMENT configure (EMPTY)>
<!ATTLIST configure name CDATA #REQUIRED>
<!ATTLIST configure value CDATA #REQUIRED>
```

2.4 Index Model

The service index is used in two places: First in listing the service offerings by the ASP. The client is able to choose a service instance out of this list. Second, each client has its own service index that lists the services that were already selected by this client. Both indexes use the same underlying model. The elements of the index model are similar to those used in the service model. In Appendix A.2 an example illustrates this model. The following root element defines a service group:

```xml
<!ELEMENT services (service+)>
<!ATTLIST services path-location CDATA #REQUIRED>
```

Path-location is the path to the various service documents of the service index list.

2.4.1 Service Element

This element describes a service item in the service index. At least one instance element, i.e., a specific service offering, needs to exist.

```xml
<!ELEMENT service (instance+)>
<!ATTLIST service name CDATA #REQUIRED>
```

2.4.2 Instance Element

An instance element describes a service offering.

```xml
<!ELEMENT instance (EMPTY)>
<!ATTLIST instance class (Economy | Business | First | none) #REQUIRED>
<!ATTLIST instance location CDATA #REQUIRED>
```

This element can only exist once per class and service, because the the location attribute is used as key and must therefore be unique. Alternatively, additional offering classes could be named.
2.5 Package Model

In this model the dependencies of the various providers and among the packages are described. In Appendix A.3 an example illustrates the package model. The structure is as follows:

```xml
<!ELEMENT packages (package+)>
<!ATTLIST packages base-location CDATA #REQUIRED>
```

**Base-location** is the path to the package file `package.<system>.xml`. This can either be a system path, or an internet address (e.g. ftp address).

### 2.5.1 Package Element

This element describes a package, i.e., an archive file containing the installation files of a provider.

```xml
<!ELEMENT package (requires*)>
<!ATTLIST package name CDATA #REQUIRED>
<!ATTLIST package version CDATA #REQUIRED>
<!ATTLIST package release CDATA #REQUIRED>
<!ATTLIST package location CDATA #REQUIRED>
```

**Name** is the name of the package. This corresponds to the provider name. It does not need to be unique, since several versions can exist for the same provider.

**Location** is the filename of the package, e.g., `apache-1.3.14-1.i386.rpm`.

### 2.5.2 Requires Element

This element indicates package requirements (dependencies) for the current package.

```xml
<!ELEMENT requires (EMPTY)>
<!ATTLIST requires name CDATA #REQUIRED>
<!ATTLIST requires version CDATA #IMPLIED>
<!ATTLIST requires release CDATA #IMPLIED>
```

If the **version** attribute is omitted, the first package that matches is taken, when resolving the package dependencies.

Similar to the **requires** element, there could be an element **conflicts**. This can be added in future work and the prototype implementation could be extended to support this.

2.6 Configuration Model

Finally the configuration model is presented. In Appendix A.4 an example illustrates this model. Three different elements are described here: **configure** (the client specific parameters), **controlscript** (the service control script) and **exec** (various binaries or scripts that need to be executed for configuration after the installation). The structure is as follows:
CHAPTER 2. SERVICE MODEL CONCEPTS

<!ELEMENT config (configure*,controlscript?,exec*)>
<!ATTLIST config name CDATA #REQUIRED>
<!ATTLIST config location CDATA #REQUIRED>

Name corresponds to the respective provider.

Location is the filename of this document, usually given in the form <provider>.conf.xml.

2.6.1 Configure Element

This element contains a parameter that can be specified by the client. This parameter is used in a subservice to add client specific configuration.

<!ELEMENT configure (item*)>
<!ATTLIST configure type (text | menu | hidden) #REQUIRED>
<!ATTLIST configure name CDATA #REQUIRED>
<!ATTLIST configure default CDATA #REQUIRED>

<!ELEMENT item (EMPTY)>
<!ATTLIST item value CDATA #REQUIRED>

Type indicates how this configuration parameter is visualized in a web browser. A text type is represented as a text input field. The menu type enables the client selecting the parameter value out of a list. Finally, a hidden parameter cannot be modified. It can be shown though.

Name corresponds to the respective configure name in the service model.

Default is a default value for this parameter.

Item contains an option that can be used as a menu item. This is only effective in the menu mode, and will be ignored in other modes.

2.6.2 Controlscript Element

This element corresponds to the controlscript attribute in the service model. In the installation process the location attribute is copied to the service document.

<!ELEMENT controlscript (EMPTY)>
<!ATTLIST controlscript location CDATA #REQUIRED>

2.6.3 Exec Element

This element is used to execute some scripts or binaries after installation of the package. It should be used carefully, since the process is executed with the rights of root. In future work a security manager can be added to the prototype, that only allows the execution of a set of predefined configuration scripts.

<!ELEMENT exec (EMPTY)>
<!ATTLIST exec cmd CDATA #REQUIRED>
Chapter 3

Architecture

The architecture as shown in Figure 3.1 is composed of three parts: the ASP infrastructure, given as a set of servers and an internal network connecting them; a configuration server, used to integrate, install and configure services and finally the HTTP client, i.e., a simple web browser, that enables the content provider (client) to access the Java application that runs inside the configuration server.

![Figure 3.1: Architecture](image)

3.1 ASP Infrastructure

The infrastructure of an ASP consists of various kinds of servers, proxy servers, routers and a network connecting them. These physical resources can be managed by a management infrastructure, e.g., the ICorpMaker [15]. ICorpMaker is an application that is currently being developed at IBM Research. Its main goal is to support an Application Service Provider (ASP) creating an instant corporation (ICorp) with virtual servers and virtual networks. These virtual resources are allocated on the underlying physical infrastructure of the ASP and can be modified dynamically. This enables an ASP to allocate enough resources to run client services with an appropriate quality of service (QoS). The ICorpMaker delegates the intelligence for resource allocation to the clients themselves by allowing them to increase or decrease the current allocation. Thus the clients can focus on their core business, the
development of content and applications, rather than having to care about the technical infrastructure and the problems that arise with it.

Within this thesis the ASP infrastructure is simplified to one single server, on which the client can create a service. This behaviour can be extended in future work and the current prototype application can be modified accordingly.

3.2 Configuration Server

The configuration server provides the application service configuration interface, introduced in Chapter 1. The aim of this interface is to: (1) automate the service installation process and make it more transparent and (2) delegate the configuration of the services to the client. It enables a client to install and configure services on the underlying infrastructure of the ASP. Inside the configuration server, a Java application, which is responsible for the service installation, configuration and control on the ASP infrastructure is running. Through Java servlets, the client is able to access this application. The configuration server was implemented using the Tomcat Java application server [14]. Tomcat is a reference implementation for the Java servlet technology [11] and has a built-in HTTP server.

The architecture of the configuration server is divided into three different layers as shown in Figure 3.2: the presentation layer, which is an interface to the client and serves both as a representation of the service and as an entry point for the client; the integration layer, which is the core of the application as it controls the complete configuration and installation process; finally the installation layer, which is an interface to the underlying system and therefore has parts that are quite system dependent. In the following subsections these layers are explained in detail.

3.2.1 Presentation Layer

In this layer the representation and control interface of a service is implemented. The service model is presented to the client using HTML technology. Actions on the service configuration can be performed and the installation process can be started using buttons, forms and links. The Java classes in the presentation layer implement the Java servlet API through which the client can access the service engine.

To translate the internal XML models into HTML documents that can be read by a web browser, XSL [4] stylesheets are used. These transformations are performed by the application using an XSLT [7] processor, which is provided by the Xalan [12] libraries.

3.2.2 Integration Layer

This layer manages the integration and configuration of the services. The "service engine" serves as an internal representation of the service document, e.g., webserver-economy.xml, and the client specific index document index.xml, containing the services selected by the client. Moreover, it controls the service installation and configuration process and is able to start and stop the service.
3.3. **HTTP CLIENT**

![Diagram of Configuration Server Architecture](image)

**Figure 3.2: Configuration Server Architecture**

To read the XML documents which are internally used as a substructure for the Java classes, an XML parser is used, which is provided by the Xerces [13] libraries.

### 3.2.3 Installation Layer

This layer serves as an interface between the configuration server and the underlying infrastructure. There are three important classes in this layer: the **installer**, which installs and removes packages on the underlying infrastructure; the **configurator**, which supports the configuration of providers; finally, the **controller**, which enables the service engine to start and stop the service. Up to the integration layer, the prototype application can run on any system that is able to run a Java virtual machine (JVM). However, in the classes of this layer, system dependent commands and applications are used to perform the functions, e.g., the `rpm` command, that is used in the installer. To run this application on another system, for example on Windows NT (the current application was tested on Red Hat Linux), these classes, particularly the installer and the controller, need to be adjusted with appropriate methods. The package dependencies described in the document `packages.<system>.xml` and the provider configurations in `<provider>.conf.xml` are used in the installation layer, to perform the installation.

### 3.3 HTTP Client

To connect to the configuration server, i.e., to access the Java servlets, the client can use a personal computer running a simple web browser capable of HTML 4.01 [5], e.g., Netscape Communicator 4.7 or Internet Explorer 5.0. Through this simple interface, the client is able
to dynamically install and configure powerful applications from almost anywhere on the world and without the need of his own powerful infrastructure.
Chapter 4

Prototype Implementation

This chapter describes the implementation of the configuration server introduced in Chapter 3. An overview on the implemented Java classes and the involved XML documents and XSL stylesheets is given in Table 4.1 - 4.3 ordered by the different layers. In the following sections the processes performed by these classes are explained, based on the various functions of the prototype and illustrated by various screen shots of the client interface.

<table>
<thead>
<tr>
<th>IndexServlet.class</th>
<th>ServiceServlet.class</th>
</tr>
</thead>
<tbody>
<tr>
<td>index.xsl</td>
<td>service.xsl</td>
</tr>
<tr>
<td>provided-services.xsl</td>
<td>configure.xsl</td>
</tr>
<tr>
<td></td>
<td>install.xsl</td>
</tr>
</tbody>
</table>

Table 4.1: Java Classes and Stylesheets in the Presentation Layer

<table>
<thead>
<tr>
<th>Index.class</th>
<th>Service.class</th>
</tr>
</thead>
<tbody>
<tr>
<td>index.xml</td>
<td>&lt;name&gt;-&lt;class&gt;.xml</td>
</tr>
</tbody>
</table>

Table 4.2: Java Classes and XML documents in the Integration Layer

<table>
<thead>
<tr>
<th>Controller.class</th>
<th>Installer.class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurator.class</td>
<td>packages.&lt;system&gt;.xml</td>
</tr>
<tr>
<td>&lt;provider&gt;.conf.xml</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Java Classes and XML documents in the Installation Layer

Typically, a client first logs into the configuration service, as described in Section 4.1. Then he selects a service offering, as explained in Section 4.2 and configures it, as described in Section 4.3. Finally, the ASP installs the service. This is explained in Section 4.4.

4.1 Login

In order to access the service application through Java servlets, the client needs to login using his username and password. If access is granted, the username is used to look for the respective client directory. This directory contains the client’s service index file index.xml with all
selected and configured services, e.g., `webserver-economy.xml`. When a client accesses the servlet for the first time, a new session is created that serves as a storage for objects used later in the same session, e.g., the index object or the service object. Thus the parsing of the respective XML documents in order to create these objects, i.e., the internal representation of the XML documents, is done only once per session.

4.2 Add a Service

After a successful login the client is able to add a new service. Therefore the option "add Service" is selected as shown in Figure 4.1.

![Figure 4.1: Service Index](image)

In Figure 4.2 the procedure for adding a new service is shown in detail. The HTTP client sends an HTTP GET request with the query string `?addService=true` to the IndexServlet. From there the method `getProvidedServices` of the index class is invoked. The index class translates the XML document `index.xml`, which contains a list of the services provided by the ASP, to HTML using the stylesheet `provided-services.xsl`. An XSLT processor is used for this transformation.

The client is now able to choose a service out of the provided list. Usually a service is available with a set of service classes, e.g., "economy", "business" or "first". This is shown in Figure 4.3.

Once the client has chosen a service by selecting a service offering, the HTTP client sends an HTTP GET request to the IndexServlet with the query string
4.3 Configure a Service

After adding a new service, the client is able to configure it by selecting the "configure" option shown in Figure 4.1. The current session is handed over from the index servlet to the service servlet. The servlet returns an overview of the selected service as shown in Figure 4.4.
To configure a service, the client can perform two actions:

- enable or disable a specific subservice
- configure a subservice by sending client specific parameters

These two functions are explained below.

4.3.1 Enable or Disable a Subservice

To enable or disable a subservice the client selects the "toggle" option of this subservice as shown in Figure 4.4. The procedure of this is shown in Figure 4.5.

The HTTP Client sends an HTTP GET request to the service servlet with the query string \(?toggle=<subservice>\) \(^1\). The servlet invokes the method \(\text{toggleSubserviceStatus}(<\text{subservice}>)\) in the service class. There, the \(\text{status}\) attribute of the respective subservice is changed according to its prior value. If the \(\text{type}\) attribute has the value \(\text{required}\), the subservice cannot be disabled \(^2\). After that, the servlet invokes the method \(\text{getServiceTransformed}\), which transforms the current service to HTML, using the stylesheet \(\text{service.xsl}\) \(^3\). Now, the status of the according subservice is changed and the subservice will be installed or removed on the system during the following installation process.
4.3.2 Configure a Subservice

To configure a subservice the client selects the "configure" option as shown in Figure 4.4. The following procedure is shown in detail in Figure 4.6.

The HTTP client sends an HTTP GET request with the query string ?configure=<subservice> to the service servlet. The servlet invokes the method getServiceConfig(<subservice>) in the service class, which transforms the according subservice element to HTML using the stylesheet configure.xsl. The respective output is shown in Figure 4.7.

Using an HTML form the client can edit the subservice parameters. The HTTP client submits these parameters to the servlet sending an HTTP POST request with the according name/value-pairs. For each parameter the servlet invokes the method setConfigureValue(<subservice>,<name>,<value>) in the service class. If the according element already exists it is modified. Otherwise a new configure element is created and appended to the respective subservice element.

4.4 Install the Service

To install a service the client selects the option "install service" as shown in Figure 4.4. The procedure for the service installation is shown in Figure 4.8.

The HTTP client sends an HTTP GET request to the service servlet with the query string ?install=false. The value false is used to get a confirmation page before really installing the service. The servlet invokes the method getInstallTransformed(false) in the service class, in order to transform the current service document to HTML using the stylesheet
install.xsl. This shows the actual service scope with the client supplied subservice parameters. The service is now ready for installation. By selecting "ok" the client can start the installation. The installation process is performed by the method install in the service class, which is described in the next section. The installation of the service can be observed by looking at the log output as shown in Figure 4.9.

4.4.1 Install Method

To install the service on the underlying system, the method install() in the service class is invoked, as already described above. The according procedure is shown in Figure 4.10.

At first the dependencies of each subservice provider must be resolved. This can be done using the method getInstallPackages in the service class. This method returns an ordered list of packages that need either to be installed or removed in the following installation process. Two stylesheets are used to perform this, install-packages.xsl and filter-packages.xsl. In the XML document packages.xml the provider dependencies are described by the requires elements, as explained in Chapter 2. These dependencies are resolved using the stylesheet install-packages.xsl. For each enabled subservice a list of packages is created, that need to be installed. For the disabled subservices, the opposite happens. Therefore the XML structures of the service document and the package document are performed recursively. The produced XML output is of the following structure:

```
<!ELEMENT packages (package*|package-not-found)>
<!ELEMENT package (EMPTY)>
<!ATTLIST package action (add | remove) #REQUIRED>
<!ATTLIST package name CDATA #REQUIRED>
<!ATTLIST package version CDATA #REQUIRED>
```
4.4. INSTALL THE SERVICE

Figure 4.6: Subservice Parameter Configuration

<!ATTLIST package release CDATA #REQUIRED>
<!ATTLIST package location CDATA #IMPLIED>

<!ELEMENT package-not-found (EMPTY)>  
<!ATTLIST package name CDATA #REQUIRED>  
<!ATTLIST package version CDATA #REQUIRED>  
<!ATTLIST package release CDATA #REQUIRED>

**Action** indicates whether the package should be installed (add) or uninstalled (remove) during the installation process.

**Package-not-found** is generated, if a package requirement didn’t match with any of the available packages. If this happens, an exception is thrown, and the installation is stopped.

This output is directly forwarded to the stylesheet `filter-packages.xsl`\(^2\). In the above package list a package can occur several times, because there can be multiple references to the same package. The filter stylesheet makes sure, that a package is not listed more than once in the result. If the package was on the add list, only the first one can pass the filter. If the package was on the remove list, only the last one. Otherwise, dependencies will break during installation. It could also happen, that a package is on the add list and on the remove list. In that case it must not be removed, since it is required by another package. Therefore the package is deleted on the remove list. The following example shows the condition for a package on the add list to remain on the list:

```xml
<xsl:if test="not(//package[position()] &lt; $position]
```
Controller Before installing any packages on the underlying system, the current service must be stopped using the method `shutdownService`, which invokes the `controller` class \(^3\). This class is used as a system interface for the prototype application to control the service process. In Table 4.4 the system dependent methods of the controller are shown. A control script (e.g. `/etc/rc.d/init.d/httpd`) must be provided by one of the subservices. If no such script is found during the installation, the service cannot be started or stopped.

### Table 4.4: Controller Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startService</code></td>
<td><code>/etc/rc.d/init.d/httpd start</code></td>
</tr>
<tr>
<td><code>stopService</code></td>
<td><code>/etc/rc.d/init.d/httpd stop</code></td>
</tr>
<tr>
<td><code>restartService</code></td>
<td><code>/etc/rc.d/init.d/httpd restart</code></td>
</tr>
</tbody>
</table>

The resulting package list is now prepared for installation.
Installer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>System command</th>
</tr>
</thead>
<tbody>
<tr>
<td>installRpmPackage</td>
<td>rpm -i</td>
</tr>
<tr>
<td>upgradeRpmPackage</td>
<td>rpm -U</td>
</tr>
<tr>
<td>uninstallRpmPackage</td>
<td>rpm -e</td>
</tr>
<tr>
<td>isInstalled</td>
<td>rpm -q</td>
</tr>
</tbody>
</table>

Table 4.5: Installer Methods

**Installer** After stopping the service, packages can be added or removed on the system using the method `addPackage` of the `installer` class. This class is used to install packages on the underlying infrastructure. At the moment the same machine is used for installing as the prototype application is running on. In future work this function can be scaled, so that packages can be installed on any machine of a distributed infrastructure. Only single packages can be installed. So the installation order must already have been determined. This has been done in the method `getInstallPackages`, which was described above. The installer class is highly system dependent. In Table 4.5 the methods and the according system commands are shown.

The packages that are not used anymore are removed if possible. This can happen if, for example, a subservice is disabled that was active before.
CHAPTER 4. PROTOTYPE IMPLEMENTATION

Figure 4.9: Service Installation

Configurator After a successful installation of a subservice, the configurator class is used to configure the subservice. This class is invoked in the method `configure` of the service class. The configurator class is basically used to filter configuration files for tokens that contain client specific parameters. To perform this, the method `filter` is used. Internally the filter problem is divided into smaller pieces that are solved by the methods `filterStringForKey` (to filter one key in a string) and `filterString` (to filter all keys) respectively. After filtering the configuration files, they are copied to the target location and possible configuration scripts are executed as indicated in the provider configuration file `<provider>.conf.xml`.

Finally the service will be restarted. When the service could be installed and configured successfully, the client can return to the service overview and check the installation selecting the option “refresh”. Now all the enabled subservices have an `active` attribute that is set to `true` and a `location` attribute is provided to test the subservice. This is shown in Figure 4.11. The client is able to modify the service scope and the client specific configurations anytime he wants.

4.5 Various Java Classes

4.5.1 Config Class

The config class is used as an internal representation for the global configuration document `conf.xml` of the prototype application. In there a couple of paths are set, that can be queried using this API. All paths are relative to the application home. Table 4.6 indicates the meanings of the various methods.
4.5. VARIOUS JAVA CLASSES

4.5.2 Logger Class

The logger is used to write the output and the errors of the various functions to the log files. Beside the log string, the name of the invoking class and the actual time are written to the log. Two instances of loggers are created in the service class, an output and an error log. These two objects are used by all other classes in the application. To write a log string the method \texttt{write} is used. There are another two important methods: \texttt{connect} and \texttt{disconnect}. These are used to write the log output not only to the log files, but also to the user interface. So to "listen" to the output of one of the loggers, the servlet can be connected to the logger using these methods. This function is for instance used during the installation process, as shown in Figure 4.8 at step ④.

Figure 4.10: Install Method
CHAPTER 4. PROTOTYPE IMPLEMENTATION

Figure 4.11: Service Overview after Installation

<table>
<thead>
<tr>
<th>Method</th>
<th>Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLogger</td>
<td>logger</td>
<td>/WEB-INF/logs/error.log</td>
</tr>
<tr>
<td>getConfDir</td>
<td>conf</td>
<td>/WEB-INF/conf</td>
</tr>
<tr>
<td>getPackageRepository</td>
<td>packages</td>
<td>/WEB-INF/conf/packages.redhat-6.2.xml</td>
</tr>
<tr>
<td>getStylesheetPath</td>
<td>stylesheets</td>
<td>/WEB-INF/stylesheets</td>
</tr>
<tr>
<td>getServicePath</td>
<td>services</td>
<td>/WEB-INF/services</td>
</tr>
<tr>
<td>getServiceIndex</td>
<td>index</td>
<td>/WEB-INF/services/index.xml</td>
</tr>
<tr>
<td>getAppHome</td>
<td>home</td>
<td>/var/tomcat/webapps/asp</td>
</tr>
</tbody>
</table>

Table 4.6: Config Methods
Chapter 5

User Manual

5.1 Installation of the Prototype Application

To install and run the prototype application, Tomcat 3.2.1 needs to be on the system. Assuming Linux is used, the binary is provided in a rpm file, that can be found either on the enclosed CD or under http://jakarta.apache.org/tomcat/. The rpm file can be installed using the command

```
 rpm -i tomcat-3.2.1-1.noarch.rpm
```

After Tomcat is configured appropriately and running, the prototype application can be installed, using the tar file `asp.tar.gz` which is on the CD. This file can be uncompressed to any directory, e.g., `/home/<username>/asp`, using the commands

```
 cd /home/<username>
tar xzf asp.tar.gz
```

Now the application can be compiled and installed, using the build mechanism of the "Ant" project:

```
 cd asp
./build.sh
```

This will install the application and overwrite the Tomcat configuration files `server.xml` and `tomcat-users.xml`. After this installation, Tomcat needs to be restarted, using the command

```
 /etc/rc.d/init.d/tomcat restart
```

To test the application, a web browser can be pointed to the following location:

```
 http://localhost:8080/asp
```

As default login name the username "hau" and the password "123456" can be used.
5.2 Service Integration

By now, the service integration, i.e., composing the subservices, providing the default configurations and describing the package dependencies, is done manually by editing the XML documents. These were described in Chapter 2. To integrate a new service the following steps are fulfilled:

- A new service document (<name>-<class>.xml) is created and stored in the main service directory. The service index index.xml is updated accordingly.
- If a new subservice provider is added, the package repository file packages.xml is updated with the new packages that are required by this provider.
- In the configuration directory, a new provider configuration file (<provider>.conf.xml) is added and a new context (directory) for each new subservice provider is created. Inside these directories as many configuration files as necessary can be placed. These files can have configuration parameters, i.e., tokens that are overwritten with the according values at installation time and copied to the location as indicated in the provider configuration file. Finally the default values for the tokens and the locations of any scripts or binaries that need to be executed for the provider configuration are supplied.

In future work, the integration process can be supported with a separate user interface that automates the various steps and checks that everything required by the prototype application is done.

5.3 Add a Client

To add a new client with username and password, the file tomcat-users.xml in the configuration directory can be updated accordingly. At the moment, the client is not able to change this password. After an update of this file, the Tomcat application server needs to be restarted. In future, the client information can be stored in a database, e.g., mySQL, that can be changed without restarting the server. The server configuration file server.xml needs to be changed to load the client information from the database. There is already an example in the Tomcat distribution that can be uncommented. This process can also be automated in the future. Then the client is able to change these parameters himself. In addition, for each client a directory with the client’s username needs to exist and a default service index document needs to be generated and placed in that directory.
Chapter 6

Conclusion

Hosting services for clients poses a number of important challenges for an ASP. One such problem, which we address in this paper, is the installation of new services. Services may require complex configurations and depend on various subservices. Therefore service installation is often a difficult and time consuming process. As the number of clients increases, the ASP greatly benefits from an automated service installation process. Moreover, it allows to delegate the configuration of the service to the client.

In this paper we propose a model as a step to formalize and standardize the definition of services. To validate our model, we implemented a Java based prototype, i.e., the configuration server, to automate the service configuration and installation process. The client interacts with the configuration server using a Java servlet based web interface. Through this interface the client is able to modify the service scope and to submit client specific configurations for the behaviour of a service. The functionality of the model and the prototype implementation was tested using the Apache webserver and some extension modules as example services. However, the proposed model gives a general description of a service and could therefore also be used for other applications, such as videoservers or gameservers.

6.1 Future Work

Within the limited amount of time in this thesis, only a few aspects of services modeling could be addressed.

In the current prototype, provided services and service offerings are given as a list, from which the client can choose a service. When the amount of provided services by an ASP is large, the client should be able to search for a service offering that matches his needs, using, e.g., a service query language. Therefore, a service registry is needed that could be accessed by the client. The Universal Description Discovery and Integration (UDDI) specification [1] and the Web Service Description Language (WSDL) [2] could be used for registering services. Through the Service Object Access Protocol (SOAP) [17] clients and users would be able to search for and access the registered services.

Finding an appropriate pricing model for the provided services is an important market factor for an ASP. One possibility would be to set up basic fees for each service offering.
Optional features and configurations of a service would incur additional fees. Further pricing models need to be discussed.

The current prototype is scaled for one server only. However, the infrastructure of an ASP typically contains many servers and the components of a service may be distributed among them. Constraints can be added to the service description model for defining which parts of a service need to be placed on the same server. The prototype needs to be extended to support distributed services.

For the integration of new services and service offerings, i.e., the creation of new instances of the service model, which is usually done by the ASP, a text or XML editor is currently used. This process can also be automated and a similar interface as that used for the clients could be implemented.
Appendix A

Examples

A.1 Service Model

webserver-economy.xml

<?xml version="1.0"?>

<service name="Webserver"
    class="Economy"
    location="webserver-economy.xml"
    status="enabled"
    active="true"
    controlscript="/etc/rc.d/init.d/httpd">

    <subservice name="HTTP Server"
        provider="apache"
        type="required"
        status="enabled"
        active="true"
        address="http://localhost:80"/>

    <configure name="Host" value="localhost"/>

    <configure name="Logging" value="on"/>

    <subservice name="Encryption Module"
        provider="mod_ssl"
        type="optional"
        status="enabled"
        active="true"
        address="https://localhost:443"/>

    <subservice name="Scripting Module"
        provider="mod_php"
        type="optional"
A.2 Index Model

index.xml

```xml
<?xml version="1.0"?>
<services path-location="/asp/service">
  <service name="Webserver">
    <instance class="Economy" location="webserver-economy.xml"/>
    <instance class="Business" location="webserver-business.xml"/>
    <instance class="First" location="webserver-first.xml"/>
  </service>
</services>
```

A.3 Package Model

packages.redhat-6.2.xml

```xml
<?xml version="1.0"?>
<packages base-location="/usr/src/redhat/RPMS/i386">

  <package name="apache"
    version="1.3.14"
    release="3"
    location="apache-1.3.14-3.i386.rpm"/>

  <package name="mod_ssl"
    version="2.7.1"
    release="3"
    location="mod_ssl-2.7.1-3.i386.rpm"/>

  <requires name="apache" version="1.3.14" release="3"/>
  <requires name="openssl"/>

</package>
```

</package>

```xml
<package name="openssl" />
```
A.4. CONFIGURATION MODEL

```
<packages/>

A.4 Configuration Model

apache.conf.xml

<?xml version="1.0" encoding="ISO-8859-1"?>

<config name="apache" location="apache.conf.xml">

  <configure type="text" name="Host" default="localhost"/>

  <configure type="menu" name="Logging" default="on">
    <item value="on"/>
    <item value="off"/>
  </configure>

  <controlscript location="/etc/rc.d/init.d/httpd"/>

  <exec cmd="/sbin/chkconfig --add httpd"/>

</config>
```
Appendix B

Sourcecode

The data contained on the enclosed CD is shown in Table B.1.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/asp/conf.......</td>
<td>Configuration files used by the prototype, e.g., packages.xml,</td>
</tr>
<tr>
<td></td>
<td>&lt;provider&gt;.conf.xml, conf.xml</td>
</tr>
<tr>
<td>/asp/etc.........</td>
<td>Configuration files used by Tomcat, e.g., web.xml, server.xml,</td>
</tr>
<tr>
<td></td>
<td>tomcat-users.xml</td>
</tr>
<tr>
<td>/asp/lib.........</td>
<td>Java libraries used by the prototype, e.g., xalan.jar, xerces.jar</td>
</tr>
<tr>
<td>/asp/services....</td>
<td>Service documents, e.g., webserver-econommy.xml, index.xml</td>
</tr>
<tr>
<td>/asp/src.........</td>
<td>Source code of the prototype implementation (Java files)</td>
</tr>
<tr>
<td>/asp/stylesheets.</td>
<td>Stylesheets used by the prototype</td>
</tr>
<tr>
<td>/asp/web.........</td>
<td>HTML files</td>
</tr>
<tr>
<td>/download........</td>
<td>Various downloaded applications and libraries, e.g., Tomcat, Xalan and Xerces</td>
</tr>
<tr>
<td>/manuals.........</td>
<td>Manuals and specifications of standards, e.g., servlets, XML schema, UDDI, etc.</td>
</tr>
<tr>
<td>/packages........</td>
<td>RPM packages used to install on the ASP infrastructure (built for Red Hat Linux 6.2)</td>
</tr>
<tr>
<td>/varia...........</td>
<td>Some scripts and documents used in tests</td>
</tr>
<tr>
<td>/asp.old.........</td>
<td>Older versions of the prototype</td>
</tr>
<tr>
<td>/wstk-2.0.........</td>
<td>Web service development kit, used in the evaluation</td>
</tr>
<tr>
<td>/thesis...........</td>
<td>Tex, PDF and and EPS files used for this paper</td>
</tr>
<tr>
<td>/presentation....</td>
<td>Powerpoint slides and screen shots used for the presentation</td>
</tr>
</tbody>
</table>

Table B.1: Data on the CD

B.1 Service Class Methods

An overview of the methods in the service class is given in Table B.2.
<table>
<thead>
<tr>
<th>Method</th>
<th>Invoking / Invoked Class</th>
<th>Action performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNode</td>
<td>-/-</td>
<td>applies the XPathAPI</td>
</tr>
<tr>
<td>getServiceTransformed</td>
<td>ServiceServlet/-</td>
<td>transforms the service to HTML</td>
</tr>
<tr>
<td>setDefaultConfig</td>
<td>ServiceServlet/-</td>
<td>sets the subservice configuration to default</td>
</tr>
<tr>
<td>getServiceConfig</td>
<td>ServiceServlet/-</td>
<td>transforms the subservice configuration to HTML</td>
</tr>
<tr>
<td>getInstallTransformed</td>
<td>ServiceServlet/-</td>
<td>transforms the install confirmation to HTML or starts the installation process</td>
</tr>
<tr>
<td>setConfigureValue</td>
<td>ServiceServlet/-</td>
<td>changes a subservice configuration value</td>
</tr>
<tr>
<td>toggleSubserviceStatus</td>
<td>ServiceServlet/-</td>
<td>modifies the status attribute of a subservice</td>
</tr>
<tr>
<td>store</td>
<td>-/-</td>
<td>serializes the internal data to an XML document</td>
</tr>
<tr>
<td>getInstallPackages</td>
<td>-/-</td>
<td>resolves and filters the package dependencies</td>
</tr>
<tr>
<td>install</td>
<td>-/Installer</td>
<td>performs the service installation</td>
</tr>
<tr>
<td>findServiceScript</td>
<td>-/-</td>
<td>looks for a service script for use in the controller</td>
</tr>
<tr>
<td>configure</td>
<td>-/Configurator</td>
<td>performs the service configuration</td>
</tr>
<tr>
<td>getServiceScript</td>
<td>-/-</td>
<td>returns the control script if existing</td>
</tr>
<tr>
<td>isServiceActive</td>
<td>-/-</td>
<td>indicates if the service could be started</td>
</tr>
<tr>
<td>isSubserviceActive</td>
<td>-/-</td>
<td>indicates if the subservice could be installed</td>
</tr>
<tr>
<td>startupService</td>
<td>-/Controller</td>
<td>starts the service with the controller</td>
</tr>
<tr>
<td>shutdownService</td>
<td>-/Controller</td>
<td>stops the service with the controller</td>
</tr>
</tbody>
</table>

Table B.2: Service Class Methods
Bibliography


List of Figures

1.1 ASP Environment .................................................. 1
1.2 Application Service Configuration Interface .................... 2

2.1 Service Example .................................................... 7
2.2 Model Relationship ............................................... 7
2.3 Service Model Structure Example ............................. 8

3.1 Architecture ...................................................... 13
3.2 Configuration Server Architecture ........................... 15

4.1 Service Index ..................................................... 18
4.2 Add a Service .................................................... 19
4.3 Provided Services ............................................... 20
4.4 Service Overview before Installation ........................ 21
4.5 Service Scope Configuration .................................. 22
4.6 Subservice Parameter Configuration .......................... 23
4.7 Subservice Configuration ....................................... 24
4.8 Install the Service .............................................. 25
4.9 Service Installation ............................................. 26
4.10 Install Method .................................................. 27
4.11 Service Overview after Installation ........................ 28
List of Tables

4.1 Java Classes and Stylesheets in the Presentation Layer . . . . . . . . . . . . 17
4.2 Java Classes and XML documents in the Integration Layer . . . . . . . . . . 17
4.3 Java Classes and XML documents in the Installation Layer . . . . . . . . . . 17
4.4 Controller Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.5 Installer Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.6 Config Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

B.1 Data on the CD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
B.2 Service Class Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38