

Universidade do Minho  
Escola de Engenharia

António Manuel de Almeida Campos

Framework para aplicações de  
monitorização de Sistemas Embebidos,  
Web based e Open-Source

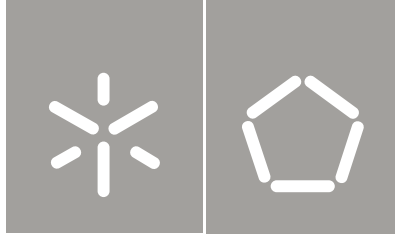
Framework para aplicações de monitorização de  
Sistemas Embebidos, Web based e Open-Source

António Manuel de Almeida Campos

UMinho | 2013

outubro de 2013





Universidade do Minho  
Escola de Engenharia

António Manuel de Almeida Campos

Framework para aplicações de  
monitorização de Sistemas Embebidos,  
Web based e Open-Source

Tese de Mestrado  
Ciclo de Estudos Integrados Conducentes ao Grau de  
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do  
Professor Doutor Paulo Cardoso

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE AUTORIZAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, \_\_\_\_/\_\_\_\_/\_\_\_\_

Assinatura: \_\_\_\_\_

*To my mother and father*

*Labor Omnia Vincit*



FRAMEWORK FOR EMBEDDED SYSTEMS MONITORING APPLICATIONS,  
WEB-BASED AND OPEN-SOURCE

by

António Campos

DISSERTATION

*Presented to the Faculty of the School of Engineering of the University of Minho  
in Partial Fulfillment of the Requirements for the Degree of Master of Science,  
under the supervision of Professor Paulo Cardoso.*

Department of Industrial Electronics

University of Minho

Guimarães, 2013





# Acknowledgements

First of all, I'd like to thank professor Paulo Cardoso for being my advisor, for giving me the independence to work and, most of all, for questioning the whys and the why nots of my work. *"From now on our relationship is between co-workers! I will not demand anything from you, this is your work. I will discuss it with you!"*.

I'd also like to thank to the professors and technicians of the Industrial Electronics Department. Even though you imposed many challenges that gave me a lot of headaches and even made me sweat, I am now ready to face the real world. Also, to all the teachers that came in my path since 1994, you all taught me something and established the base for me to be here now.

I would also like to thank my parents, Celso and Natália. You both gave me the strength to continue, to always accomplish more, to not be afraid of whatever may come my way. Without you I wouldn't be here today! Thank you mom! Thank you dad! (One down, one to go!)

To Liliana, my sister, for the nagging. Now that you are where I was a few years ago, enjoy it to the fullest, you can go wherever you want to!

To Marta, my girlfriend. You appeared in the beginning and in the end, in a time where everything was going downhill and somehow you managed to help me getting back on track. *"Get out of bed, you have got to work!"*

To Susana, for making me get on track five years ago and for the sharing.

To all my brothers from other mothers, for being with me all these years. For the adventures we had together, for the long nights studying, for the jokes, for the brotherhood and most of all for being my friends.

To all of you, that helped me to be where I am now, thank you!

# Resumo

Monitorizar consiste em observar, supervisionar ou controlar um sistema. Como tal, todas as aplicações de monitorização desempenham uma função semelhante em sistemas diferentes. Sendo assim, pode dizer-se que todas as aplicações de monitorização são desenvolvidas sobre uma base comum, uma vez que a tarefa a desempenhar é semelhante, sendo os dados diferentes entre aplicações. Pode assumir-se então, que desenvolver uma aplicação deste tipo de raiz consome mais recursos que o necessário.

Existem muitos sistemas embebidos que necessitam de monitorização, alguns têm até o acesso físico restringido, criando a necessidade de monitorização remota. Este projecto tem como objectivo responder a esta necessidade através da criação de uma ferramenta capaz de implementar aplicações de monitorização remota que utilizem a Web como meio de comunicação, fornecendo uma arquitectura para que sistemas embebidos possam comunicar com o utilizador através da Web.

O Desenvolvimento de Software Orientado a Características (FOSD) é utilizado como método de desenvolvimento nesta ferramenta, permitindo o desenvolvimento de uma Linha de Produção de Software (SPL) para o domínio de monitorização remota de sistemas embebidos.

O resultado deste trabalho é uma SPL capaz de implementar aplicações de monitorização remota constituídas por um interface do utilizador, um interface com a plataforma, uma base de dados e um conjunto de serviços Web que estabelecem um daemon de comunicação, de forma simples através da utilização do plug-in para o Eclipse, Feature IDE.

Esta solução facilita o processo de desenvolvimento e reduz drasticamente o time-to-market de uma aplicação de monitorização específica. Utiliza também os conceitos FOSD e SPL, que podem ser úteis em outros domínios.

# Abstract

Monitoring consists on observing, supervising or controlling a system. Therefore, all monitoring applications carry a similar task, only in different systems. Knowing this, one can easily say that monitoring applications are built upon the same base, since the task is similar, only the data changes. Thus, developing a monitoring application from scratch usually consumes more resources than needed.

There are many embedded systems that need to be monitored, some of them are even hard to reach, creating the need for remote monitoring. This project aims to answer this need by creating a tool that is able to implement remote monitoring applications that use the Web as communication infrastructure providing an architecture for embedded systems to communicate with a user via the Web.

Feature-Oriented Software Development (FOSD) is used as the fundamental development methodology to develop this tool, allowing the development of a Software-Product Line (SPL) for the remote monitoring of embedded systems domain.

The result is a SPL, able to implement remote monitoring applications comprising a user interface, a platform interface, a database and a set of Web services, that establish a communication daemon, in a fast and easy way by using the Feature IDE plug-in for Eclipse.

This solution greatly eases the development process and reduces drastically the time-to-market of a specific remote monitoring application. It also uses the FOSD and SPL concepts, that can be useful in many other domains.

**Keywords:** Feature-Oriented Software Development, Software-Product Lines, monitoring, Feature-Oriented Domain Analysis (FODA), Feature-Oriented Software Implementation (FODI), Feature IDE, Web services, REST, AJAX, HTML5

# List of Abbreviations

**AJAX** Asynchronous JavaScript and XML

**API** Application Programming Interface

**CGI** Common Gateway Interface

**CRUD** Create, Read, Update and Delete

**CSS** Cascading Style Sheets

**DAO** Database Access Object

**DEI** Industrial Electronics Department

**DOP** Delta-Oriented Programming

**FIFO** First In, First Out

**FODA** Feature-Oriented Domain Analysis

**FODI** Feature-Oriented Software Implementation

**FOP** Feature-Oriented Programming

**FOSD** Feature-Oriented Software Development

**FST** Feature Structure Tree

**GCC** GNU C Compiler

**GUI** Graphical User Interface

**HTML** Hypertext Mark-up Language

**HTTP** Hypertext Transfer Protocol

**IBM** International Business Machines

**IDE** Integrated Development Environment

**IPC** Inter-Process Communication

**IP** Internet Protocol

**JSON** JavaScript Object Notation

**JVM** Java Virtual Machine

**MIME** Multi-purpose Internet Mail Extensions

**ORM** Object-Relational Mapping

**OS** Operating System

**PDF** Portable Document Format

**POJO** Plain Old Java Object

**RIA** Rich Internet Application

**RAM** Random Access Memory

**REST** Representational State Transfer

**RDBMS** Relational Database Management System

**RPC** Remote Procedure Call

**SOAP** Simple Object Access Protocol

**SPL** Software-Product Line

**SOAP** Simple Object Access Protocol

**SQL** Structured Query Language

**TCP/IP** Transmission Control Protocol/Internet Protocol

**UM** University of Minho

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**URN** Uniform Resource Name

**W3C** World Wide Web Consortium

**XML** eXtensible Mark-up Language

# Table of Contents

	Page
Acknowledgements . . . . .	v
Resumo . . . . .	vi
Abstract . . . . .	vii
List of Acronyms . . . . .	viii
Table of Contents . . . . .	xi
List of Figures . . . . .	xiv
<b>Chapter</b>	
1. Introduction . . . . .	1
1.1. Motivation . . . . .	1
1.2. Objectives . . . . .	2
1.3. Contents and Organization . . . . .	3
2. Domain Analysis . . . . .	5
2.1. State Of the art . . . . .	6
2.1.1. Software Product Lines . . . . .	6
2.1.2. Web Technologies . . . . .	11
2.2. Problem Space Specification . . . . .	14
2.2.1. Domain problem specification . . . . .	15
2.2.2. User Interface . . . . .	16
2.2.3. Server . . . . .	20
2.2.4. Monitored Platform . . . . .	26
2.2.5. Feature Diagram . . . . .	30

2.2.6. Cross-Tree Constraints . . . . .	34
3. Domain Implementation . . . . .	37
3.1. Tools and techniques . . . . .	37
3.1.1. Eclipse . . . . .	38
3.1.2. Feature IDE . . . . .	38
3.1.3. Jersey and JAX-RS . . . . .	44
3.2. SPL Implementation . . . . .	45
3.2.1. Server . . . . .	49
3.2.2. User Interface . . . . .	59
3.2.3. Platform Interface . . . . .	63
4. Instantiation . . . . .	66
4.1. Requirement Analysis . . . . .	67
4.2. Software Generation . . . . .	69
4.2.1. Web service deployment . . . . .	70
4.2.2. User Interface and database deployment . . . . .	72
4.2.3. Platform Interface deployment . . . . .	74
5. Results . . . . .	77
5.1. Test Systems Specifications . . . . .	78
5.1.1. Servers . . . . .	78
5.1.2. Client . . . . .	78
5.1.3. Platforms . . . . .	78
5.2. Basic Application . . . . .	79
5.3. Full Application . . . . .	82
5.4. Test conclusions discussion . . . . .	86
6. Conclusion and Future Work . . . . .	88
6.1. Conclusion . . . . .	88
6.2. Future Work . . . . .	90
References . . . . .	92
<b>Appendix</b>	
A. . . . .	95



A.1. Feature Diagram . . . . .	96
A.2. Feature description . . . . .	97
A.2.1. Platform features . . . . .	97
A.2.2. Server features . . . . .	97
A.2.3. User interface features . . . . .	101
A.3. Feature constraints . . . . .	102
B.Domain Impementation . . . . .	105
B.1. Flowcharts . . . . .	105
B.2. User interface site map . . . . .	122
B.3. Web service path hierarchy . . . . .	123
B.4. Web services specification . . . . .	124
B.4.1. Action service . . . . .	124
B.4.2. Alert service . . . . .	125
B.4.3. Authentication service . . . . .	126
B.4.4. Management service . . . . .	127
B.4.5. History service . . . . .	130
B.4.6. Miscellaneous service . . . . .	133
C.Instantiation . . . . .	134
C.1. Editing the Database connection properties . . . . .	134
C.2. Example of a web.xml deployment descriptor . . . . .	135

# List of Figures

2.1. Evolution of software reuse . . . . .	6
2.2. Functionality types of a functionality diagram . . . . .	9
2.3. Top view of the framework architecture . . . . .	16
2.4. Web application block diagram . . . . .	17
2.5. Web service block diagram . . . . .	25
2.6. Monitored platform block diagram . . . . .	27
2.7. User interface feature diagram . . . . .	31
2.8. Web services feature diagram . . . . .	32
2.9. Database feature diagram . . . . .	33
2.10. Monitored platform feature diagram . . . . .	34
2.11. Constraints definition flow . . . . .	35
3.1. Java code and FST of artifact BaseDB, taken from the Berkeley DB case study[1]	43
3.2. Composition of two Java methods in FeatureHouse[3] . . . . .	43
3.3. Framework architecture divided into levels . . . . .	47
3.4. SPL main branches . . . . .	47
3.5. Web services data flow . . . . .	50
3.6. Web services feature diagram . . . . .	52
3.7. Database feature diagram . . . . .	56
3.8. Database relational diagram . . . . .	57
3.9. User interface feature diagram . . . . .	60
3.10. Platform feature diagram . . . . .	64

4.1. Configuration for the case at hand. . . . .	69
4.2. Dynamic Web project creation . . . . .	70
4.3. New Connection Profile Wizard . . . . .	73
4.4. SQL execution result . . . . .	74
4.5. Platform interface compiling and running . . . . .	75
4.6. The details file before and after being edited to reflect the platform communication ports and the Web service Uniform Resource Identifier (URI) for new alerts . . . .	76
5.1. Test case connection diagram . . . . .	79
5.2. Login page . . . . .	80
5.3. Home page for the basic application . . . . .	81
5.4. Unanswered alerts warning . . . . .	81
5.5. Troubleshooting page - Received alerts . . . . .	82
5.6. Troubleshooting page - Received alerts . . . . .	82
5.7. Platform command line output . . . . .	83
5.8. Detail file used for the basic configuration . . . . .	83
5.9. Home page for the full application . . . . .	84
5.10. No privilege page . . . . .	84
5.11. Actuator Bay management page . . . . .	85
5.12. User management page . . . . .	85
5.13. User History page . . . . .	86
A.1. Feature diagram . . . . .	96
B.1. Login Web service flowchart . . . . .	106
B.2. Check privilege Web service flowchart . . . . .	107
B.3. Logout Web service flowchart . . . . .	108
B.4. Add Web services flowchart . . . . .	109
B.5. Edit Web services flowchart . . . . .	110
B.6. Delete Web services flowchart . . . . .	111
B.7. List history Web services flowchart . . . . .	112

B.8. Find history Web services flowchart . . . . .	113
B.9. Delete entry from history Web services flowchart . . . . .	114
B.10. Clear history Web services flowchart . . . . .	115
B.11. Current alert Web service flowchart . . . . .	116
B.12. Current action Web service flowchart . . . . .	117
B.13. List Web services flowchart . . . . .	118
B.14. GET request flowchart . . . . .	119
B.15. POST request flowchart . . . . .	120
B.16. Platform interface flowchart . . . . .	121
B.17. Web service URI hierarchy . . . . .	123
C.1. <i>DBconnection.properties</i> before and after editing . . . . .	134
C.2. <i>web.xml</i> definition example . . . . .	135

# Chapter 1

## Introduction

### 1.1 Motivation

Embedded systems are often used as control systems to larger systems, electronic or not. When this is the case there is the need of monitoring the system operation by receiving feedback from the system sensors and taking actions, if that is the case, using the system actuators. Some of these systems may be hard to reach or even physically unreachable, thus creating the need of remote monitoring.

The development of remote monitoring applications is a time consuming task, taking considerable amounts of time, making the development process longer and increasing the time-to-market of a given system. The monitoring of a system is just a part of the system as a whole and all the monitoring applications perform similar tasks, namely user authentication, data acquisition and presentation, alarm report and actuation, being the the biggest difference between on the data values and their meaning. Thus, all these applications share a common base where the task at hand is the same. This makes it possible for the development of a tool capable of implementing this common base, easing the design process of a specific system and reducing its time-to-market.

Since monitoring is done remotely, there is the need to transfer data between two terminals that can be apart for thousands of kilometres. To do so, there is no better communication infrastructure than the Internet, which implements the broadest communication platform ever developed by the human being, allowing a rapid transfer of large amounts of data between two

terminals, even if one of the terminals is in a remote area.

## 1.2 Objectives

The main objective of this work is to implement a framework for remote monitoring applications that use the Web as a communication platform, by using a SPL approach.

By using a SPL it is possible to develop a framework that is easy to use and intuitive for the application developer, allowing the rapid implementation of monitoring applications that need minor tweaking to fit a specific system/problem. Therefore the main goal is to study the remote monitoring domain and implement SPL capable of implementing Web based remote monitoring applications, following a FOSD approach, allowing the characterization of the domain and the implementation of a framework for the domain.

The implemented framework should be able to generate applications comprising a communication architecture able to establish communication between the user interface and the remote systems to be monitored, a database server for data storage and a user interface supported by a Web browser, using the latest Asynchronous JavaScript and XML (AJAX) and HTML5 techniques. The framework must implement a basic user interface, since this is the part of the system that can vary greatly from application to application.

The implemented architecture must keep a list of monitored systems as well as an history for all the data transactions performed between the system elements, all accessible by the user. The user must also be able to perform administrative tasks on the system, being able to add, edit or remove the details of the system components.

Between all the tasks to be carried, the following main tasks can be identified:

1. Define data sources to be monitored;
2. Define the database from the above characterization;
3. Identification of the messages to be traded;
4. Implementation of a set of Web services to allow the interaction between users, monitored systems and the database;

5. The specification and implementation of a SPL for the specified domain;
6. Testing of the framework by instantiating it;

Finally, the framework must also be open-source, meaning that its code is freely available for anyone who wants to use it and to improve the tool. This makes way for the creation of a community that can use and maintain the framework in the future, potentiating the improvement and evolution of it.

### 1.3 Contents and Organization

This work aims to specify how to design and implement a SPL by using FOSD. Therefore, this document is structured according to the FOSD development phases, providing a *Domain Analysis* on *chapter 2*, comprising a study of the State of the Art of the technologies used to develop this project and an analysis to the remote monitoring of embedded systems domain, specifying the domain components and how they communicate. This chapter also divides the domain into features presenting a feature diagram that results from this process, relating all the domain features between them.

On *chapter 3*, *Domain Implementation* is presented, specifying the tools and techniques used to implement the framework and explaining how the code is structured, as a way to help the understanding of the SPL.

*Chapter 4* ends the FOSD development phases, by specifying the last two phases, *Requirement Analysis* and *Software Generation*, stating how to *instantiate* the SPL to a specific problem within the remote monitoring of embedded systems domain. This chapter, which has a structure similar to a *tutorial*, specifies the necessary work flow to implement and deploy a monitoring application for a specific problem by using the SPL.

On *chapter 5*, instantiation *results* will be exposed, showing the ability of the framework to automatically generate fully functional systems that need little customization to address specific problems within the domain.

At last, in *chapter 6*, *conclusions* of the development process of this project will be discussed, stating if the work was well performed and what can be improved. *Future work* identified thus far is also presented.

After these chapters *appendix A* will show and explain the schematics resulting from the *Domain Analysis* phase of the project, *appendix B* will demonstrate schematics resulting from the *Domain Implementation* phase, namely the flowcharts of the implemented code, the site map of the user interface and how to connect to the Web services. *appendix C* will expose useful advices on the instantiation process.



## Chapter 2

# Domain Analysis

The proposed project consists on the design and implementation of a framework for the development of remote monitoring applications for embedded systems. However, embedded systems are computer systems and monitoring an embedded system remotely imposes almost the same constraints as monitoring a regular computer system, thus making this project also suitable for the remote monitoring of a computer system that is not an embedded system, as long as it respects all the constraints imposed by the framework.

All the applications developed with this framework have the same structure, therefore all of them will have a user interface, provided by a web application running on a Web browser, a set of Web services responsible for storage and data redirection, and the monitored systems, referred as platforms. Since embedded systems have a vast nature, the framework must be prepared for the implementation of small monitoring applications, such as house security monitoring, but must be as well prepared for large applications, such as monitoring offshore wave power generation platforms in high seas. Because all applications within the remote monitoring domain address different problems, being different from each other in their specific function, they all share a common base, having a great number of functionalities in common, thus creating the need for a mechanism that is able to handle these common aspects while leaving space for the development of the functionalities that makes each application unique. This mechanism, which has a rapidly growing user base, is the Software-Product Line development process and will be explained further ahead.

## 2.1 State Of the art

This section presents a study on the State of the Art of all the concepts present on the project. Since SPL development is one of the core concepts, it will be exposed first. Later an insight on Web Technologies, with special attention given to Web Services will also be given, since this is also one of the core concepts.

### 2.1.1 Software Product Lines

According to Bosch, the notion of constructing a software system by composing components has been discussed since the late 1960s, where reuse was accomplished by the systematic usage of subroutines. During the following decades several module-based approaches were proposed, most remarkably the development of object-oriented frameworks, composed of abstract classes through which concrete classes can be implemented by inheritance. This effort to achieve software reuse led to the acceptance of two very important lessons, being the first that opportunistic reuse, the act of salvaging code for use in other applications, is not effective because a successful reuse programme must be a planned and proactive effort. The second lesson is that bottom-up reuse does not function in practice[4]. Successful reuse, like most developing techniques, is accomplished by employing a top-down approach. However, if there are existing components, such as communication protocols that need to be used, the design process may also have some bottom-up decisions. During the 1990s and the following decade, the combination of software architectures with component and service based software development led to the notion of SPL.

Figure 2.1 shows the evolution of software reuse techniques, from the usage of subroutines in the 1960s to SPLs in the 2000s.

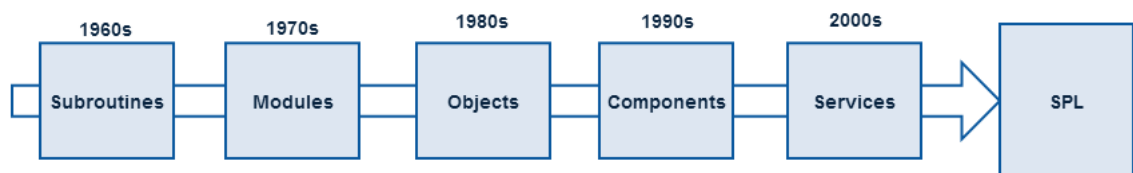


Figure 2.1: Evolution of software reuse

A Software-Product Line can be described as a software engineering methodology that allows

the exploring of common functionalities within a product family, in a way that is possible to create a new product from a common base [5]. The focus of SPL is not to develop a single solution for a given problem, but to develop different products through the same software framework, in this particular case for the product family within the embedded system remote monitoring domain.

The SPL development effort is incremental and makes it possible to develop the SPL core, as well as the architecture and some features first, then develop one or more products/applications, and then develop more features in a cyclic way, where SPL usage will create more features that can be used to develop other applications. This means that the usage of the SPL will make it evolve to a more solid solution for the achieving of a greater level of automation.

The way a SPL is planned, implemented and instantiated is a process known as FOSD [30]. This process is divided in four distinct phases, which have different aims:

1. *Domain analysis* Intends to specify where a software domain varies and where it does not vary, resulting in a feature model. This process is also known as Feature-Oriented Domain Analysis (FODA).
2. *Domain implementation* Implementation of all software systems simultaneously. Features are mapped into code. This process is also known as Feature-Oriented Domain Implementation (FODI).
3. *Requirement analysis* Mapping a problem requirements to features and selection of the features needed to a specific software system. The result is a configuration.
4. *Software generation* Automatic building of a software system, using software composition, based on a given configuration.

A closer look to the previous list shows that the first two phases are specific to the domain engineering, where a domain is analysed and its features specified and implemented, leading to the creation of the SPL. The other two phases are specific of the application domain, where a specific problem is analysed and the features that can answer that problem are selected from the SPL, allowing the implementation of an application for the specific problem.

## Domain Analysis

The first step for successfully implementing a SPL is the domain analysis. Domain analysis consists on the analysis of a given domain, determining its requirements and where the domain can vary or not.

Feature modelling, proposed as a part of FODA [14], is used to specify these variations. Feature modelling is represented using feature diagrams, which provide graphic representation of domain features and establish an easy way to determine the variations of the domain.

These diagrams represent system features in a hierarchical way, and at the same time specify their relations. The top node is the base node, representing the concept and the descending nodes represent the functionalities of the domain. The selection of the needed features to answer a specific problem results in a configuration. This configuration does not have all the features of the diagram and is unique, since each configuration specifies a specific application type.

Therefore, one can easily conclude that the reason to use such diagrams is to generate several different configurations, which generate specific applications, answering different problems within a given domain.

Since not all configurations are valid and specify useful software systems, constraints specified during the domain implementation will later specify which configurations are valid within a domain. Also, features can be mandatory or optional, cumulative or alternative as shown on figure 2.2.

Figure 2.2 specifies four types of functionalities:

- a) *Cumulative Functionalities* Functionalities that are cumulative. An analogy can be made with the logical OR operation.
- b) *Alternative Functionalities* The use of one functionality excludes all others.
- c) *Mandatory Functionalities* Functionalities that are mandatory to every configuration, usually these are application specific core functionalities.
- d) *Optional Functionalities* Functionalities that are optional and are only used when needed

In a feature diagram, features can also be abstract, meaning that they are not mapped to implementation artifacts, or concrete, meaning that they are mapped to implementation artifacts.

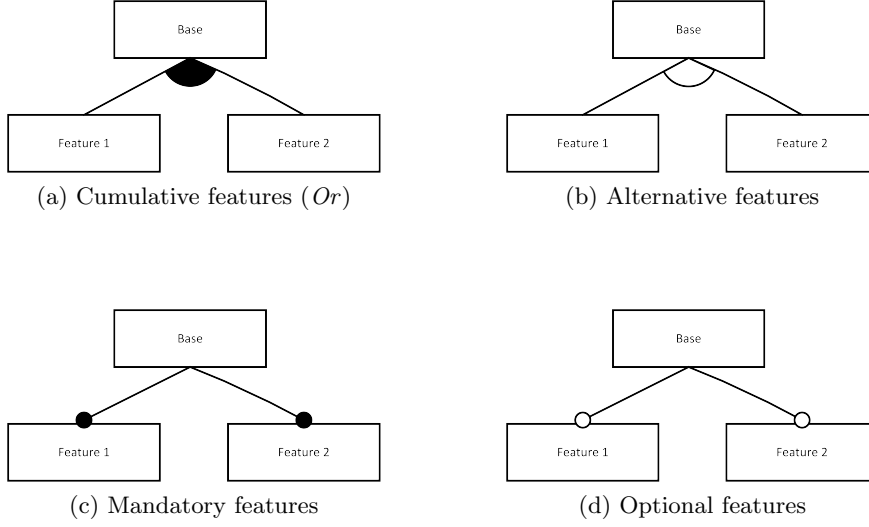


Figure 2.2: Functionality types of a functionality diagram

Abstract features help the analysis and reasoning of the set of program variants which can be generated from a domain implementation and feature model [30] by completing the SPL feature model with a visual component that helps the understanding of the SPL structure, where a concrete feature also implements code. A model can also have composition rules, that are basically cross tree constraints which are propositional formulas to further aid in the configuration. For example, *feature1 implies feature2* means that if *feature1* is selected, then *feature2* must also be selected.

Feature models specify variability within a domain and have a strict set of semantics, as stated by figure 2.2. When configuring a feature model to a specific problem, the selection of a feature implies the selection of the parent feature. If a feature is selected, then all mandatory sub-features are selected as well. In an OR group at least one feature must be selected and in an alternative group only one feature can be selected. Finally all the composition rules must be respected. If all this is respected, then a configuration is valid and can be used, otherwise it is invalid.

## Domain Implementation

The use of feature models allows to specify the variations of a domain without knowing which variation technique will be used during implementation. The main goal of implementing a SPL

is to map features to source code, enabling the automatic generation of a software system for a specific configuration. There are several techniques to implement SPL. However, these techniques are usually based on a specific program language, referred to as *host language*.

- *Feature-oriented programming* Proposed by Christian Prehofer [19] as an extension to object-oriented programming, this technique decomposes Classes into feature modules, where each implements a certain feature. A feature module can contain methods and fields from several classes and can be composed to a program based on a given configuration and a certain order of features.
- *Delta-oriented programming* This variation technique, proposed by Schaefer et al. [21], proposes a core module, written in the host language (C++, Java, etc.) and a set of delta modules. Each delta module can add or remove methods, fields and classes, and even change the super class of a given class.

In the configuration phase of SPL development, a composer identifies the delta modules that fulfil an application condition and applies them to the core module, resulting in the generation of a software system.

- *Aspect-oriented programming* According to Kiczales et al. [13], aspect-oriented programming provides a meta language to transform existing object-oriented programs. A specific position in the control flow of an object-oriented program is called a join point, and pointcuts specify a set of join points. A piece of code injected at a pointcut is called an advice. An aspect defines pointcuts and advices to inject in the code of the host language. One generates a new software system when implements a subset from a set of implemented aspects.
- *Preprocessor (Conditional Compiling)* Other technique for SPL implementation are preprocessors [11]. Preprocessor directives<sup>1</sup> are inserted as comments in a given host language. Upon a certain configuration, the preprocessor comments out the annotated parts with a false application condition.

---

<sup>1</sup> #if, #endif

## Requirement Analysis

After the SPL is implemented, it is ready to be instantiated by the domain problems. To do so, the SPL user must analyse the problem at hand, identifying which are the specific requirements and constraints, relating them with the SPL features. After doing so, a configuration is generated with the selected features. As stated previously every configuration is unique and addresses a specific problem.

## Software Generation

This is the last fase of the FOSD and consists on the automatic generation of the software system, based on the previously defined configuration, by using a domain implementation technique. Since a configuration is unique and the software generation depends directly on the configuration, it can be said that every generated software system is also unique within the domain.

### 2.1.2 Web Technologies

The Internet represents the greatest communications platform ever, either in size and in scope, connecting almost 6 billion people around the world. However, the Internet is just an interlinked network of computers that communicate using the TCP/IP protocol that provides an address for every computer connected within the network. To make the Internet useful there is a large number of resources and protocols that can be used, being one of them the Hypertext Transfer Protocol (HTTP) provided the basis for the World Wide Web, referred from now on as the Web.

The Web is a system of interlinked hypertext documents that are accessed via the Internet using a web browser and was proposed in 1989 by Sir Tim Berners-Lee [31], who is currently the director of the World Wide Web Consortium (W3C)<sup>2</sup>. This system uses a client-server architecture, where a server receives and processes request sent from a client, usually a web browser. The communication does not rely on any operating system and it is done using the HTTP methods<sup>3</sup> from the HTTP protocol.

Since the first implementation in 1991, the Web as been rapidly evolving from a static Web,

---

<sup>2</sup>International community responsible for maintaining the World Wide Web and whose mission is to lead the Web to its full potential

<sup>3</sup>GET, PUT, POST and DELETE are the most common ones

where an user only went to get information, to a dynamic Web, where the user has an active role and where the information will adjust in the way the user wants. To make this possible there is a number of technologies running both in servers and clients to provide the aspects described previously.

## **Client Side Technologies**

Usually a Web client is a Web browser that renders web pages to the user. In the last few years websites have become more attractive and dynamic, and currently there are powerful web applications that resemble native applications running on a local machine. Although this applications communicate with the server and a part of the processing is done in the server side, most notably data access, the client side programming also plays a crucial role. For example, with HTML5, the candidate<sup>4</sup> standart for the Hypertext Mark-up Language (HTML) language specified by the W3C in 2011, the development and rendering of dynamic web pages is now easier, replacing older technologies like Apache Flex<sup>5</sup>, that are harder to program and impose a great load on the client side as well as bearing the need of external plug-ins to render the page elements such as video or animations that react to the user. Popular Web pages like Youtube and Grooveshark are currently porting to HTML5.

Javascript is also a programming language executed on the client's side to provide even more advanced capabilities such as user interaction, browser control, document alteration after display and asynchronous communication. Javascript is a scripting language that can be used with HTML5 and that has a syntax influenced by the C language and copies some names and conventions from Java, hence Javascript. This language is also a multi-paradigm language that supports object-oriented, imperative and functional programming. Even tough its major use is in web pages, there are traditional applications, such as PDF documents or desktop widgets that use Javascript.

By merging HTML5, JavaScript and Cascading Style Sheets (CSS) one can create powerful Rich Internet Applications (RIAs) that communicate asynchronously with the server by using AJAX techniques that allow a Web application to carry on its operation while waiting for the

---

<sup>4</sup>HTML5 has been submitted as candidate for the next HTML standart on August 6, 2013[]

<sup>5</sup>Flex is a Flash based framework that has been passed from Adobe to the Apache Foundation, to be maintained as an independent project, enabling Adobe to focus on HTML5 development.



server response, thus resembling to a native application running on the computer. Currently, every time there is a reference to HTML5, what is referenced is the junction of the previously referred technologies that provides the resources for the creation of powerful dynamic Web applications.

## **Server Side Technologies**

In the first days of the Web servers only answered requests sent from clients. However, the evolution of the Web created the need for processing information sent by the client. This is accomplished by the server when it allows that the processing is done by an application running in the server machine. The server pre-processes a request and sends it to the application using CGI or its alternatives, the request is then processed by the application and sent back to the server that is responsible to answer the client's request. These applications can be programmed using several programming languages:

- Java
- ASP/ASP.NET
- Python
- PHP
- Perl
- ColdFusion
- LUA
- Ruby
- C/C++

CGI was the first method that made advanced processing available at the server side and, although it has suffered many upgrades over the years, such as FastCGI or SCGI, it is being replaced by other architectures, such as Java Enterprise Edition, that runs Java code in a Java servlet container in order to provide dynamic content to the client. These servlet containers often run Web services that communicate with the client using HTTP methods. According to the W3C a Web Service is a software system designed to support the interoperability between software applications running in a variety of platforms [9]. This communication is done using XML or JSON messages encapsulated into communication methods like Remote Procedure Call (RPC) or Simple Object Access Protocol (SOAP)<sup>6</sup> [6] that encode the XML or JSON messages with the data into

---

<sup>6</sup>SOAP is the successor of RPC and has the same transport and interaction neutrality as well as the same envelope, header and body

envelopes and send it using the HTTP methods. In 2000, Roy Fielding proposed, in his Ph.D. thesis, a new and simpler approach called Representational State Transfer (REST) [8] that defines that all the data sent between server and client encoded on the Uniform Resource Locator (URL) of the request instead of a data envelope. This approach is currently experiencing a rapid usage growth because, unlike RPC is stateless, and therefore easier to design and to integrate with different software parts. The data from each call being stored in the request URL makes the interaction with the web service Application Programming Interface (API) much simpler, due to the fact that POST data is not stuffed in the client, and the debugging of client code becomes simpler. Also, since REST is fully compliant with the HTTP protocol, this approach makes it possible to reuse HTTP libraries, that exist for almost every programming language, and to use HTTP reverse proxies that can be used to implement advanced features such as caching, authentication, compression, logging or redirection.

## 2.2 Problem Space Specification

When developing a framework one can't think in answering a specific problem, but must think how to answer all the problems within a given domain, the problem space. To do so commonalities and variabilities in the domain must be identified, through the use of feature modelling. This technique allows the capturing and managing of the common and variable aspects in a product line throughout all the phases of product-line engineering[7].By using SPL engineering new products within a domain can be created rapidly, based on a common set of reusable assets, such as system architecture, models, components, development processes, among many others.

As stated previously, there are four phases when developing a SPL using FOSD, being the first the domain analysis. In this phase the domain is analysed and verified. From it results a feature model specifying the variable and common aspects within a domain, as well as the constraints imposed by the domain.

The following sections will give an insight on this development phase by using the proposed framework as an example.

### 2.2.1 Domain problem specification

Many embedded systems are designed to operate without direct feedback to the user. There are also some embedded systems that operate in remote locations where user access is difficult and there are systems that are distributed, being composed by a number of subsystems, meaning that there may be a need to monitor a number of different devices at the same time. Developing applications for these situations usually takes a lot of time and consumes a great number of resources. Therefore it is desirable to reduce the time taken and the resources consumed.

It also intended to make possible the development of remote monitoring applications for embedded systems using the Web as the communication infrastructure. Applications developed using the proposed framework must be capable of, at least, the following:

- User profile management.
- Event registry and alert windows.
- Basic GUI generation (HTML5 application in browser).
- Remote system communication Daemon.
- Database.

By having a top-down approach to the problem domain, one can determine that all the applications developed using the framework will have the architecture shown on figure 2.3 in the top level. The architectures of the framework components will be shown further ahead.

Even though applications developed using the framework consist of three main components<sup>7</sup>, the framework only focuses on the monitoring part of the application. This means that applications developed using the framework will only address the user interface, the server components<sup>8</sup> and the interface between the platform and the server, leaving the platform software that will control and retrieve data from the platform to be implemented by the developer. The interface between the server and the platform implemented by the framework will impose constraints on the application developer, creating the need of respecting the communication constraints imposed by the rest of the system. These constraints include the communication protocols between the platform interface and the "language" the system defines for communication.

---

<sup>7</sup>User interface, Server and monitored platform

<sup>8</sup>Web services and database

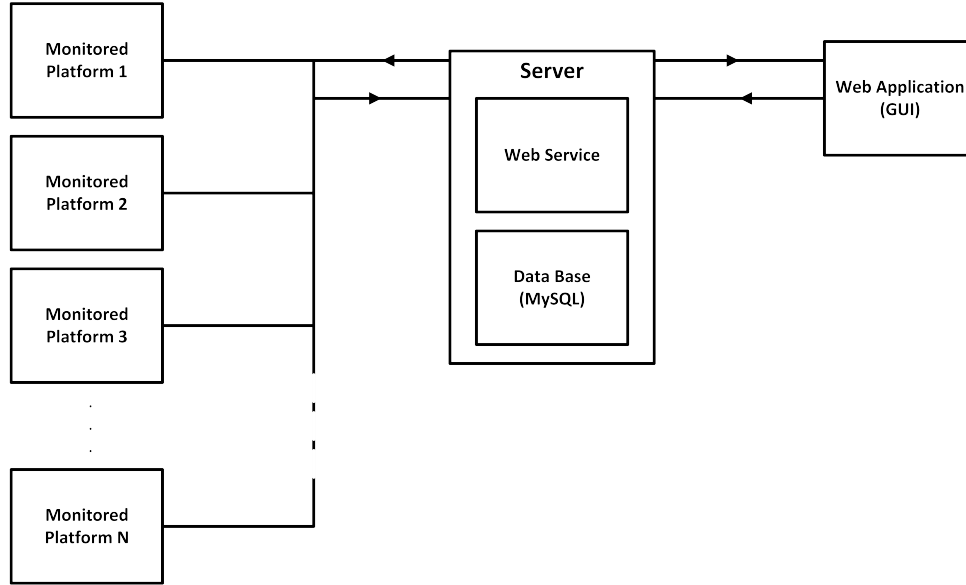


Figure 2.3: Top view of the framework architecture

Since the framework will be composed by three distinct components, to ease the development process as well as the usage process of the framework, every component represents a specialization level of the SPL that will be designed and implemented separately, meaning that the SPL is composed by three distinct feature models, one for the server, one for the user interface and another for the platform interface. The existence of three distinct feature models creates the problem of lack of consistency between them when instantiating the SPL, features chosen for the server may not correspond to the features chosen for the user interface, creating an impossible configuration. To avoid this, a set of feature model constraints must be defined, assuring that if a feature is selected in the user interface feature diagram, then the corresponding feature will also be selected in the server feature diagram. By doing this, consistency between subsystems is assured and all the generated applications will function as intended.

### 2.2.2 User Interface

The user will interact with the monitored platforms by using a Web application through a Web browser. This Web application will be developed in HTML5 and Javascript, and will use AJAX techniques for dynamic communication with the server. These are implementation constraints that must be followed.

AJAX implements a set of asynchronous communication techniques that allow the creation of very powerful Web applications with improved functionality of what has been seen until this point. Web sites are no longer static and require a large amount of data transaction between the client and the server and if these transactions are not made asynchronously there will be a huge performance loss, because the site will always wait for an answer from the server to continue its operations. The usage of asynchronous techniques allows the site to carry other tasks while it waits for an answer, improving performance and usability. Figure 2.4 shows the block diagram of the Web application. As it can be noted, the Web application is divided into two distinct subcomponents that transact data between them, the communication back-end, responsible for Web communication with the Server, and the User Interface front-end that has direct interaction with the user.

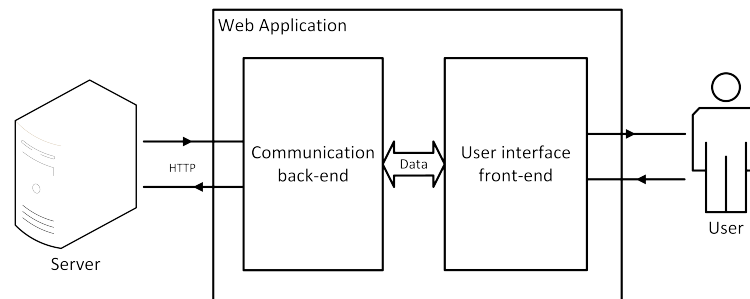


Figure 2.4: Web application block diagram

Since all the communication with the server is done using the Web, the communication back-end does not have a great degree of variation. In this part of the system the greatest variation is on the data that is transmitted, but that is architecture independent because there is no data analysis on this stage.

The User Interface front-end interacts directly with the user. Different applications mean different interfaces, the degree of variation in the implemented pages is not huge but applications may seem completely different from each other visually after completed, due to the usage of different visual components. The User Interface has a structure composed by many different windows with different purposes, such as a management window, a history window or a troubleshoot window. Every window has visual elements such as images, text, buttons or even multimedia content,

these elements differ both in type and format as in number. One window may have three images and a text box and another may have only one image but have a video and a list.

This makes the task of dividing the user interface into features simple, since each window can be represented as a feature from the user interface. Each window also has different sub features, selected depending on the tasks that must be carried by a window. The following list defines the windows that can be selected and their function:

1. *Login window* User authentication prompt
2. *Not Allowed window* The user is redirected to it when it has insufficient access privileges for a given window
3. *Home window* This window is the user interface lobby, where the user can select which window he wants to use
4. *Troubleshoot window* Where the user can receive and analyse alerts and send actions to a platform
5. *Management window* Used to manage the several components of a system (e.g. users, platforms)
6. *History window* Used to show system logs. This window shows what happened in the system and when it happened

To access the application the user must input his access credentials. This means that the access will be limited to authorized users, and it also means that users can have distinct access privileges, which means that specific actions are only available to those who have access to it. In other words, one can only access an User Interface resource if he has permission to do so. For example, a technician can access error windows and do troubleshooting but the system administrator can access these windows but can also access the management window and manage users and platforms. By imposing user privilege levels, and since the only way to affect platform operation is through the User Interface, the system guarantees that only authorized users can operate specific resources, the chance of human error by under-qualified staff.

Even though the SPL implements a user interface based on the web, the application developer is free to implement a user interface as a native application running on a computer system. This is possible because the interface between user interface and server is made using REST which uses HTTP methods to establish communication. Therefore, if there are libraries that make HTTP communication possible for a given programming language (e.g. libCurl for C or cpp-netlib for C++), as long as the constraints imposed by the server are respected<sup>9</sup>, the developer can implement a user interface for a given computer system and use it instead of using the one generated by the framework.

Since each application will have a distinct user interface, there are variation spots and commonalities that have to be identified in order to establish what are the common and variable aspects between the user interfaces that are able to be generated by the framework.

The base of each user interface is common and is composed by a login page, a home page and a troubleshooting page. These pages will be a part of every user interface and match the basic requisites of the domain, providing the most basic way a user has to monitor platforms, consisting only on accessing alerts sent by a platform and the taking of actions in case of need.

Along with these common, and mandatory, aspects there are also other aspects that can be added to implement a more complete interface that depend mostly on the specific problem requirements. These aspects vary from system to system and depend on a specific system requirements. These variable aspects focus on management and logging needs that a specific system can have.

There may be systems where there is the need to manage system components, such as platforms, sensor or users. This creates the need for a management page that can vary its structure depending on what are the systems components. While a system may need to manage platforms and users, another system may need to manage users, platform groups, platforms and the platforms sensors, creating variations points inside variation points.

Among with management, there are systems that can have as a requirement the logging of troubleshooting or management actions, thus creating another variation spot in the user interface. Inside the history page there can also be variations because what a system may need to log may not be what other system needs to log, creating variation points inside this page as well.

---

<sup>9</sup>namely the Web services URIs and their path parameters, the HTTP methods used by each Web service and the return types for the Web services response

Web pages have a visual aspect, implemented in CSS. Even though the goal of this project is no to implement the final user interface, a basic interface that is usable has to be implemented, thus there are also styles to be applied to a specific user interface. These styles are different between them and are exclusive, meaning that only one can be used in a specific user interface and along with the other features chosen for it, it will distinguish the user interface from all the others in the system.

### **2.2.3 Server**

The architecture exposed on figure 2.3 states that the server establishes the frontier between the user interface and the platform. Therefore all the data transmitted between these two sub-systems passes through the server, where it is processed and stored. Basically the server acts as a data repository from where the user interface gets data from the various components of the system and redirects actions from the user interface, where they are defined and sent, to the correct platform.

Using a server establishing this frontier is not strictly crucial, since each embedded system, being a computer system, can implement a web server where it can host a web application, that could be used as the user interface for the platform and also host a database server, where data could be stored. However, this would impose too much load on the embedded systems and could affect their operation, and there would be a separate user interface for each platform, which would create the need to check each user interface at the time for problems in the platforms, imposing a usability problem that could affect user monitoring of the platforms. Also, in case there is limited connectivity with a platform, sending an action to it would be impossible, creating the need for the user to check manually if the connectivity to the platform is restored in order to send and action to it, which is not desirable since the user may end up by not addressing all the alerts sent by a platform.

By having a server establishing a proxy between the user interface and the platforms, the load imposed by the user interface, as well as all the web services running to support it, and the database, is taken off the embedded system and put in another computer system with higher resources available. Thus the embedded system only needs to monitor its data and send it, if needed to the server, saving the embedded system resources for the monitoring of the platform, which is the intended task for the embedded system. By having a single data repository, where



the data from all the platforms is stored, there is only the need to have a single user interface where all this data can be accessed, improving usability and making the monitoring performed by the user more efficient. Finally, in case the user sends an action to a platform and the connectivity is limited, the Web service responsible for action handling can check periodically if connectivity is established with the platform, and send the action when this happens.

When a platform issues an alert the server stores it in the database, established a ordered list of alerts that need addressing. The user interface polls the database periodically for new alerts and presents them to the user who takes action on the chosen alert. When taking action the user interface sends the action to the server, that redirects it to the correct platform and removes the alert from the database. Since all the systems in this domain will operate over the Web, there can be times when there is no Web connection between the server and the platform, in this case the action is put in a queue that periodically tries to send the action to the platform, thus assuring that an alert is answered.

The server may also provide other services to the user besides troubleshooting. Management does not affect platforms directly but allows a user to manage all the aspects related to a platform namely the sensors and actuators of a platform, if a platform is inserted in a group and even the platform details, such as ID and location. User management is also provided by the server, enabling the addition and removal of authorized users from the system and the editing of user details.

Other main service that can be provided by the server is history, that keeps a log of the selected operations performed on the system<sup>10</sup>, if selected from in the feature model. History is stored in the database with all the information relative to the system action and a time stamp from when it happened.

## Database

The database plays a crucial role in the system. As it was exposed, almost all the actions taken upon a given system are reflected on the database. User authentication for example, allows access to only the users present in the database. Alerts are also stored in the database before being

---

<sup>10</sup>These operations comprise not only actions taken on platforms but also alerts issued by platforms, management and authentication

answered and actions that are being taken are also stored in the database if there is no connection to the platform. Information about platforms and their components are also stored in the database and in case there is the need to keep a log, it is kept in the database as well. Therefore, it can be said that the database describes a monitoring system and the relation between the system components.

By analysing the problem domain, there are some tables that can be identified as common to all the systems and others that may not be present if not needed. Every system has users, therefore, there must be a users table specifying a user details. Since every user must be authenticated to have access to the user interface, there must also be a session table, where a session ID and the corresponding user are kept.

Like users, every system also has platforms, therefore there must also be a table describing these platforms. Every platform is connected to the Web, thus having web details, stored in another table. A platform has sensors and actuators, thus there is the need for platforms also describing both sensors and actuators. These sensors and actuators may be grouped in sensor and actuator bays, depending on function of the platform, thus there must also be tables describing these bays. At last, platforms may also be grouped and thus, if this is the case, there may also be a table describing the platform groups.

Platforms issue alerts, receive actions and must also "speak the same language" as the user, therefore there must be tables defining the alerts and the actions that can be transacted between the user and the platform, as well as a table that implements the alert queue and another that implements the action queue, in case there is no connection to the platform.

If there is the need to keep an history, there must also be tables to store this information. These tables describe the history of alerts emitted by the platforms, actions taken, user authentication and on the management of platforms and their components.

The previous paragraphs state which tables can compose the database of any given system developed using the framework. However there are some systems that are more complex than others, and therefore the tables may not always be all necessary. The most basic system must have the users, session, platform, platform web, action, current action, alert and current alerts tables in order to function properly, these tables are common to all the systems. Other tables may be added depending on the needs of a more complex system.

At this point, all the possible tables in the database are defined, but a specific system may not implement a full database as it was described. All these systems have platforms, users, alerts, actions, sensors, and actuators. But not all implement platform groups sensor and actuator bays, and may even not need to keep an history, varying with what a specific system needs. When this is the case, the tables present in the database will correspond to the system components, thus making the database of a specific system different from the database of other specific system created using this framework.

The chosen server was the MySQL community server because it is open-source and free, it is enough for what this domain needs and has a large user base that embodies a large community that gives support to the server. Also, since it is developed by Oracle, that is one of the biggest database providers in the world<sup>11</sup>, it provides a solid solution for this framework needs. The MySQL community server is also supported by Java, providing an easy API for accessing the database, and establishing the connection between Java code and the database via a connector that abstracts the programmer from the complexities of the database connection.

## Web Service

Data will be transmitted between all the components, but the server establishes a frontier between the platform and the user interface. Since the framework only implements the platform interface with the web service, the platform application is now a black box with a set of sensors and actuators and a defined communication method.

When communicating with a Web service, a client (platform or user interface) sends a request to a specific URI<sup>12</sup>, thus addressing a specific web service, with a specific function. After the Web service action is performed, a response is sent back to the client with the operation result.

Data sent from platforms and user interface to the server depends on its origin. A platform sends alerts with details about the origin of the alert, that can be an error or simple information regarding the platform, to the Web service. The user interface, however, can send different types of requests to the Web services. It can be a request on a management action, where it specifies the

---

<sup>11</sup>The Oracle RDBMS is widely used in key market segments such as banking and healthcare with proven reliability since 1978.

<sup>12</sup>A URI is different from a URL. The URI only defines the address where to send requests, the URL is composed by the URI and the arguments sent to the server. (e.g *www.myuri.com* = URI; *www.myuri.com?myarguments* = URL)

the action to be taken and the data that corresponds to it, it can be an authentication request to grant access to a user or it can be an action sending request so that the Web service can redirect it to the platform, among other requests. The Web services responds to all the requests in XML format, with each response depending on what was the request. Responses can vary from *OK*, stating that the request associated actions were carried out successfully, to a list of unanswered alerts or users registered in the system. In case there is an error while processing a request, the Web services answer with the exception code.

In order to be able to send an action, received from the user interface, to a platform without a request from the platform, the web service establishes socket communication with the platform and sends the action code to it. Upon receiving the code, the platform applies it in order to solve whatever issue it is destined for.

Every system in the domain must have at least a set of Web services that establish communication between a platform and the user, a Web service to receive alerts and another to send actions to the platform, establishing a communication channel with the user.

Authentication is also mandatory in every system, therefore there must be a Web service responsible for logging in a user, another for logging out and lastly, since the access to user interface pages is made based on privileges, there must also be a service to retrieve a given user privilege from the database and pass it to the user interface.

The previous Web services are common to all the applications in the domain providing user authentication in the system and a communication channel between the platforms and the user interface. However, more complex applications may also have the need to implement management options and even to keep an history of what has happened in the system. Therefore, a system may also have a set of Web services responsible for managing system components and another set of Web services that keep an history from all the actions taken upon the system.

By now, it is noticeable that there is a close relation between the Web services and the features selected for the user interface. This happens because the Web services process all the data displayed by the user interface and the user interface addresses a given Web service directly when requesting data from the server.

Web service architecture is based on a 3 tier architecture, shown on figure 2.5, composed of three distinct layers, each with a specific function. This architecture is used to make code

editing easier, since it logically separates the service layer, responsible for the interface with the user interface, from the business layer, where data is processed, which is also separated from the data layer, which makes the interface with the database. This favours code edition and reuse by allowing localized changes in specific parts without affecting the rest of the code and by defining the code for the different layers in different libraries, which makes it possible to port it to other projects.

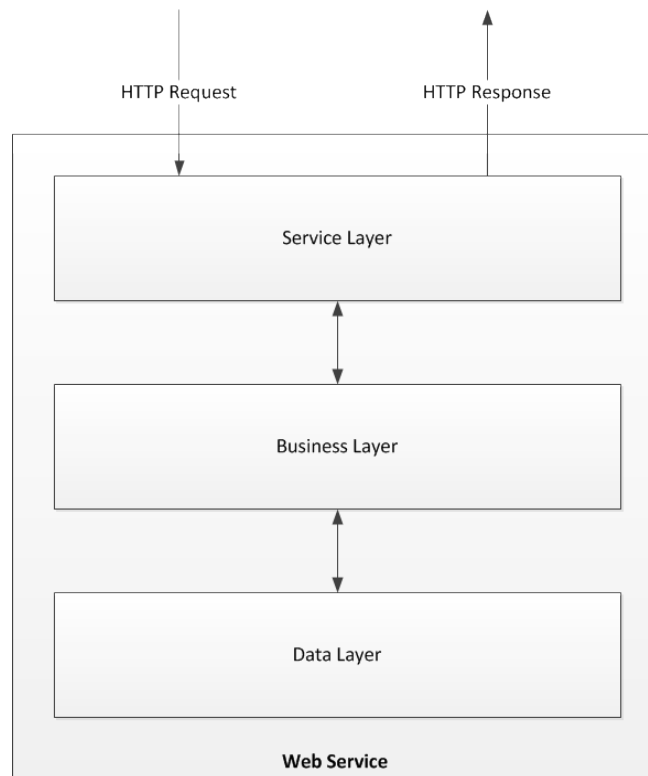


Figure 2.5: Web service block diagram

The set of Web services varies from application to application. The basic set assures the basic functions carried out by the user in the user interface, therefore each application will have an authentication service to assure user authentication, and alert and action dealing web services to provide the ability to troubleshoot problems.

To establish Web service communication with the database there is a DAO layer. This layer implements features that are common to all the systems, such as access to user, platform, action and alert tables, that provide troubleshooting and authentication access to the database. However,

depending on what are a specific system needs this layer can also implement other features, such as sensor and actuator access to the database, varying from system to system depending on what a specific system needs.

Model features exist to map database tables content in the Web services so that it can be used. These models depend on the DAO layer and its features have the same DAO layer features variation and commonalities, varying where the DAO features vary, depending on system requirements.

In case there is the need to manage system components, there are also management features that implement management web services and vary from system to system. These features are related to the management features in the user interface and vary in the same way. These are the features that grant data processing and database access for management in the user interface, providing the data that will be shown to the user in the user interface.

Like management features, there can also be logging features that also correspond to the history features in the user interface. These features have the same variation points as the correspondent features in the user interface and grant data processing and database access for logging related tasks in the user interface.

Finally there are features that can be useful for a system to have, depending on its needs. These features are not strictly necessary for system functioning but can improve the way some tasks are carried in a system.

#### **2.2.4 Monitored Platform**

The final sub-system of the system is the monitored platform. A specific application might be composed by several platforms, however all the platforms will run the same code to perform monitoring actions and can be described as the same. A platform has a unique identification that is done at least by the IP address, but it can also be identified by an ID code assigned by the system.

Platform monitoring consists on sending sensor data to the Web service via HTTP, that will send it to the proper recipient and on receiving actions, through a socket, from the user to implement via the platforms actuators. Figure 2.6 gives a better insight on a possible architecture of the monitoring program running on a platform.

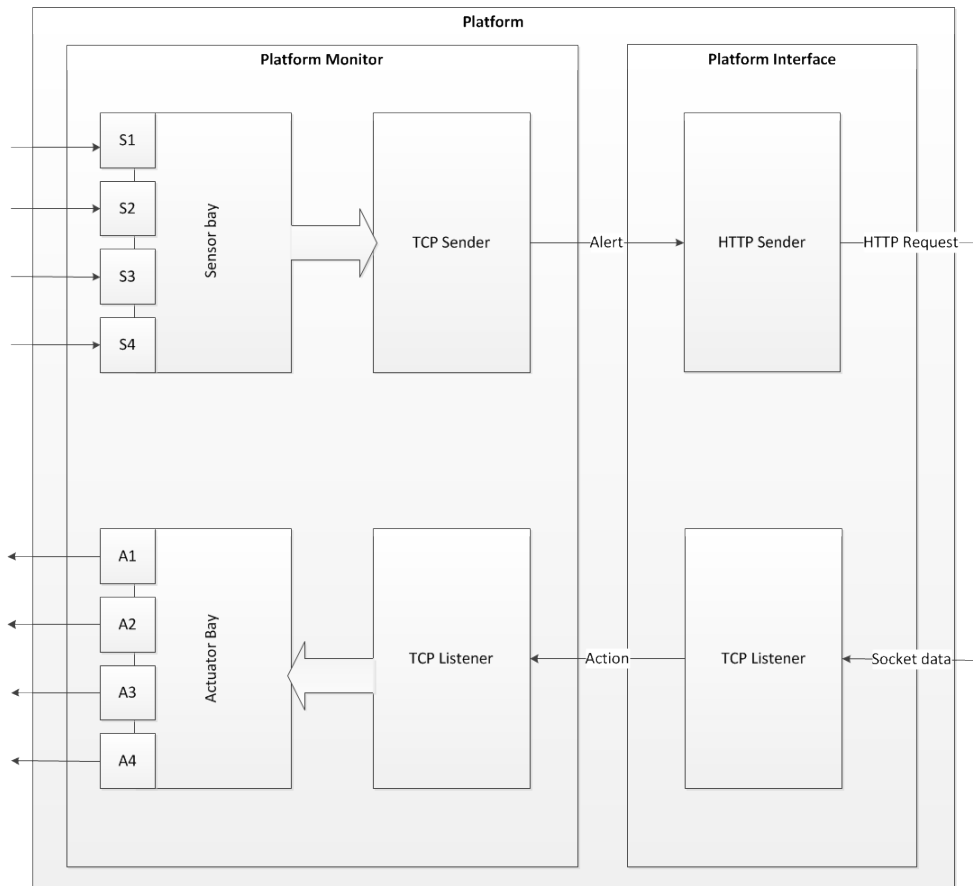


Figure 2.6: Monitored platform block diagram

Each platform has sensors and actuators that carry out troubleshooting actions at the platform level.

A sensor is the device that converts a physical parameter related to the system (e.g. temperature, proximity, acceleration, humidity, etc) and converts it into an electrical signal that can be measured by the embedded system performing the monitoring. Sensors have a set of parameters that determine their functioning and that assure their correct functioning under pre determined conditions.

- *Accuracy* Determining how accurate a read value is and how much it can deviate from the real value.

- *Environmental conditions* Limiting the usage in certain physical conditions, such as temperature and humidity, than can influence the measured values.
- *Range* Measured limit of the sensor, outside of which it can't read.
- *Calibration* In order to adapt the sensor readings to the environment.
- *Resolution* Smallest increment in the read value detected by the sensor.
- *Repeatability* Repeated measurement of a reading under a given environment.

On the other hand, while a sensor reads values related to the physical environment, an actuator acts upon the surrounding environment by transforming an electric signal into motion (e.g. activating a fan when temperature is too high). Actuator action depends on the values read by the sensor in order to establish control of the system. The parameters that define actuators are performance metrics and define how an actuator will operate under certain circumstances.

- *Force* Force applied by the actuator on a load. It is measured in dynamic and static loads, where the dynamic load is the actuator force capability while in motion and the static load is the force capability while the actuator is stopped.
- *Speed* Maximum speed at which the actuator can create motion. Its maximum value is when there is no load attached to the actuator and it decreases invariably as the load increases.
- *Operating Conditions* Operation conditions under which an actuator can operate.
- *Durability* Amount of time that an actuator can function properly under certain usage and actuator quality conditions.
- *Energy Efficiency* Amount of energy spent by the actuator to carry on a task correctly in a certain time interval. A lower energy spending level in a fixed amount of time means that the actuator is more efficient.

Each sensor and actuator have an ID code and a type associated. This allows the user to monitor not only the platform as a whole, but to monitor at a sensor and actuator level, increasing the granularity, making it possible to detect and correct specific problems, since failure



is not always general, but can be in a specific part of the device. The sensor bay is responsible for reading sensor data periodically and to send it, a suitable approach would be to read the sensors sequentially after a given time interval and send their data sequentially. Analogically, the actuator bay receives actions that must be carried by a given actuator and applies commands to the actuator to carry it. These are the main sensor actions, but there are other operations that can be performed on both sensors and actuators, like requesting the current status of the sensor or actuator.

Even though an explanation on a platform architecture is presented in this subsection, the SPL will only implement the interface between the platform and the server. The application developer is responsible for developing the monitoring application running in the platform, having the freedom to implement the application in the most suitable language for the problem and with the features he desires. However, the system treats a platform as a black box able to receive and send data, this imposes a problem in case of the occurrence of a critical error or if there is no connection between the platform and the server. Thus, in case of critical error the application must be able to take preliminary action (e.g. Halt, Wait) on the platform while waiting from user input. Also, after issuing an alert on a smaller error a platform must enable a timer and wait for an action, if the action is not received in that amount of time, then the platform must halt, resuming operation when the action is received, this will prevent that a small error creates a bigger error due to the lack of correction.

By following these recommendations the developer will assure the correct functioning of the system.

Even though the platform is a black box, the SPL must define it to aid in the understanding of the SPL architecture, since the features for the rest of the system depend on the platform components. Therefore a platform interacts with the server by sending alerts and receiving actions, this is common to all the possible platforms within this domain. Then a platform has elements that are used to specify its structure. Thus each platform has an ID in the system, and can have other aspects such as sensors and actuators, but also bay of sensors and actuators, these features vary from system to system.

The last commonality between platforms is that each platform runs on an embedded system that can runs an operating system. This operating system is not the same to all the platforms

and varies from platform to platform. As in all computer systems, two operating systems cant run concurrently on a platform, thus a platform only runs an operating system at any given time.

### 2.2.5 Feature Diagram

The main product of domain analysis, besides all the architecture definition, is the feature diagram.

Analysing the proposed architecture, the domain can be divided in three main components, the platform, the server and the user interface, each with a set of sub features that specify a part of the component function. The following subsections will demonstrate each of the three main branches of the feature diagram

#### User Interface

The user interface is constituted by Web pages, where each carries a specific task, therefore each web page is a distinct feature in this user interface.

1. *Login* Specifies the authentication mechanism visually, providing the user with a login prompt and other authentication mechanisms.
2. *Home* Specifies the home page for the user interface. It has a hTop and a hBottom features, implementing the top and the bottom of the home page. This page is extended by other features and this guarantees code correction and a correct interpretation by the web browser.
3. *Troubleshoot* This page is where the user interacts with the platform directly, providing a visual feedback from alerts and a prompt to take actions
4. *Management* Where system components are managed. Providing a visual interface for adding, editing and deleting system components.
5. *History* Provides the interface where the user can access the logs from every aspect of the system.

Since this interface is implemented using HTML5, there are also styles for it, defining the basic colors and the placement of buttons in the web page.

The generator feature is used to generate the HTML, JavaScript and CSS files that compose the user interface.

Figure 2.7 shows a feature diagram with all the features that compose the user interface, providing a visual representation of their relations and hierarchy.

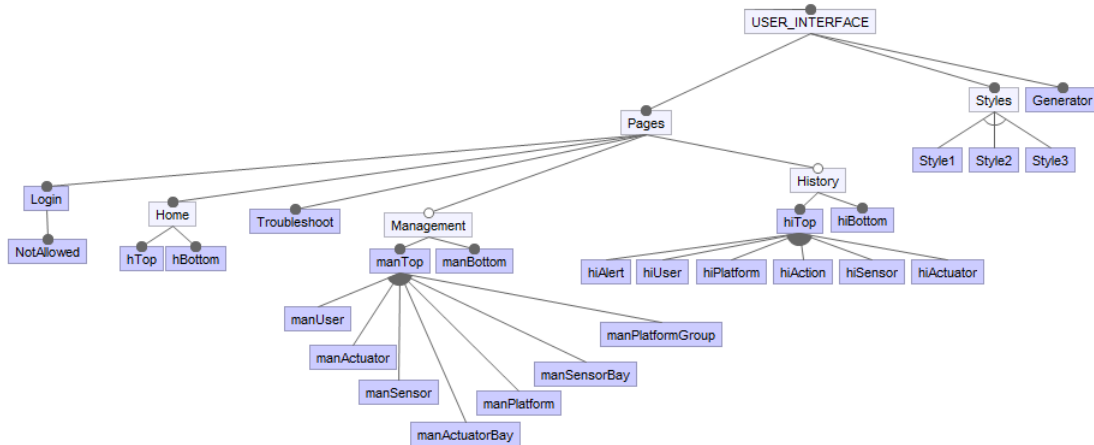


Figure 2.7: User interface feature diagram

There is a close relation between the Web services and the user interface. The user interface implements most of the features of the Web services on the client side.

## Server

In this framework scope the server is composed of two distinct systems, the Web services and the database server. Both systems are also divided into features and it makes sense to expose them separately, providing a better understanding of each system.

As it was referred previously, the platforms need to communicate with the user. To achieve this goal there is a set of Web services that receive data from the platforms and send it to the user, and receive data from the user and send it to a specific platform.

These Web services reflect almost all the system requirements. Basically they process in the server side the tasks carried by the user interface and also implement data redirection between platforms and user interface.

There are six main sub features:

1. *Authentication* All systems have privilege based access where a user has login credentials

(user name and password) and a privilege associated.

2. *Alerts* Defining the web services responsible for the handling of the alerts sent by platforms.
3. *Actions* Where the web services that send actions from the user to the platform are defined.
4. *DAO* Corresponding to the Web services interface with the database server. Each sub feature of this feature corresponds to a specific database table, implementing the interface with that table, making it possible to retrieve data correctly from the database.
5. *Model* This feature specifies the data models, mapping the database tables to code, so that data can be manipulated easily by the Web services code.
6. *Manage* Where management operations are defined. Providing a management daemon for the user to administrate the system.
7. *Hist* Providing logging ability to the system. Specifies features that implement this ability.
8. *Utilities* Intended to be extended in the future, currently only provides features to list system components. Implements features that are not specified in the objectives but may be useful.

The relation between these features is shown in the feature diagram depicted on figure 2.8.

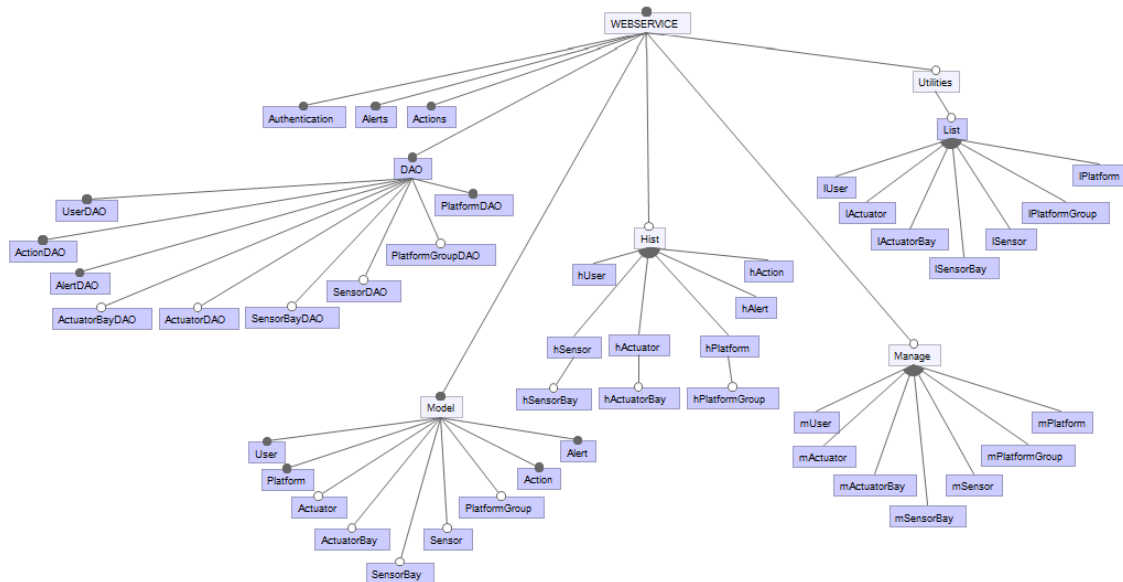


Figure 2.8: Web services feature diagram

The database will provide a persistent data storage mechanism. This feature diagram, shown on figure 2.9, is simple and each feature corresponds to a database table.

Each table describes a part of the system, such as platform data or the user details. There are platforms that depend on others to work correctly.

The features *dbTop* and *dbBottom* specify the top and the bottom of the database script generated by these features and are therefore mandatory, otherwise the script may not run in the database server.

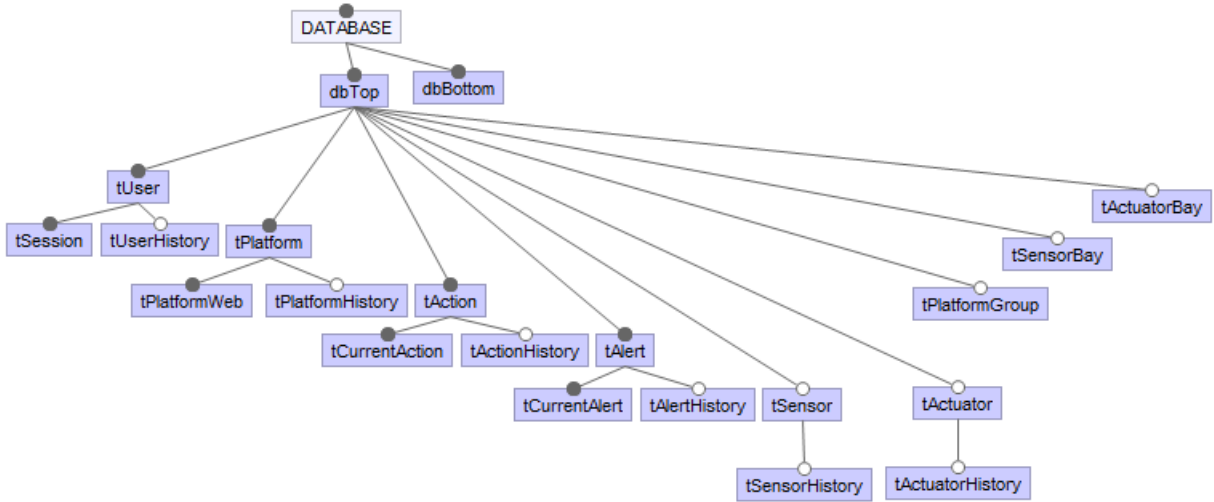


Figure 2.9: Database feature diagram

## Platform

A deeper analysis in the platform system helps to determine that each platform has a set of elements that work together to carry on the platform task. The elements are the sensors, actuators and their bays, as well as platform related details, such as the platform ID and the group where a platform is inserted. Since there is a large number of different platforms, not all these elements may be always needed, therefore only the platform ID is mandatory, since every platform must have an ID.

Platforms must also communicate with the rest of the system, therefore it emits alerts and receives actions from the user. By doing so a platform can be controlled by a user, who takes actions, and can also alert the user if there is a problem.

To establish this communication the platform must have a communication interface with the rest of the system. This interface depends on the operating system of the platform because it uses system resources and these are different depending on the operating system architecture. Since a platform only runs a determined operating system at the time, these features are alternative.

All these aspects that define a platform are related, and that relation is shown on 2.10.

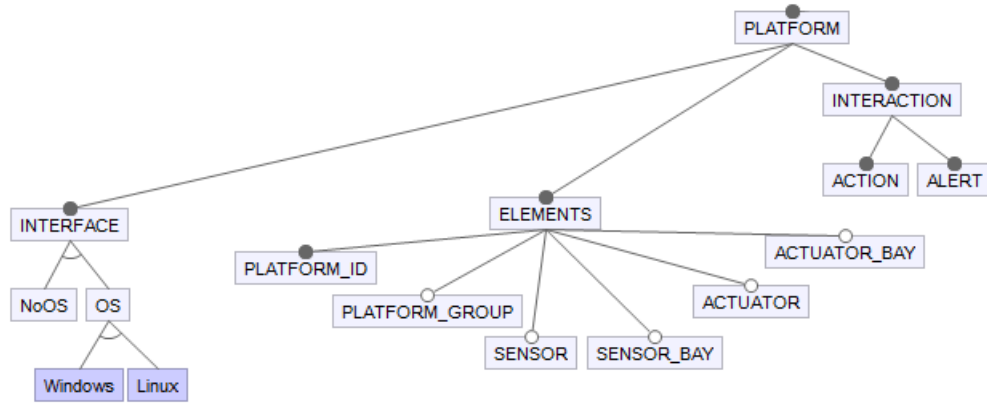


Figure 2.10: Monitored platform feature diagram

## 2.2.6 Cross-Tree Constraints

The definition of cross-tree constraints in the feature diagram is crucial for the functioning of the code generated using this SPL. Constraints guarantee that there is consistency between features, because there are features that may need other features to work correctly, or there even may be conflicts between features that are not on the same branch. In this SPL the three specialization levels need to establish communication channels between them and need to be consistent between them, for example, if a platform has sensor bays, these bays need to have a representation in the server and in the user interface. Thus creating the need to establish constraints between the branches of the feature diagram.

In this case constraints are helpful for the configuration process as well, easing it. They are defined in a way that the server is the center of the system, and even though it has features that can be selected by the user, constraints select the server features automatically based on the selections on the platform and the user interface levels. Figure 2.11 shows the constraint definition flow, stating that features selected on the two top components are reflected on the lower

one.

In other words, since the platform is the base of the system, when making a new configuration, one must start by selecting the features in the platform level. These features correspond to the platform characteristics. Then one must proceed to the user interface level where the operations to be performed in the system, such as management or history keeping, are selected. By selecting features in this two levels, the corresponding features on the server level will be selected automatically, since there were constraints defined for this, guaranteeing that the developed system is consistent throughout all the levels and that it will work as expected.

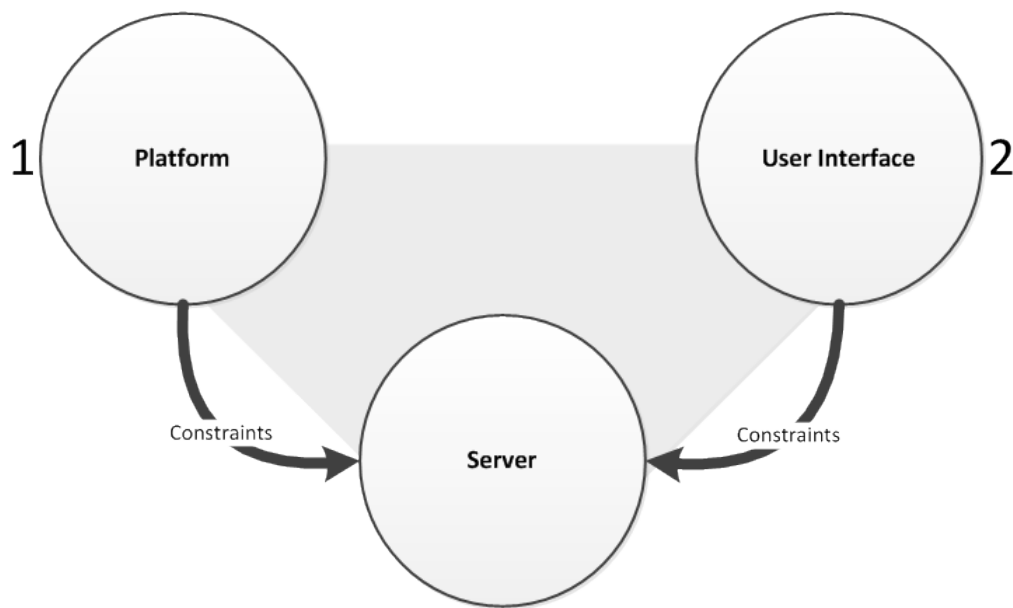


Figure 2.11: Constraints definition flow

Constraints have not only been defined as cross level constraints, but also been defined within a level, for example in the server, the *ActuatorDAO* feature, that implements the methods for database operations regarding actuator information, needs the *Actuator* feature that implements the actuator model and, of course, it also needs the *tActuator* feature which implements the database table that describes the actuators.

There are also redundant constraints, meaning that they aren't strictly needed in the feature diagram, because the diagram definition implements them automatically (e.g mandatory or alternative features). This happens because these constraints refer to mandatory features. However, these constraints were defined because these features may not be mandatory in the future, thus

assuring the correct functioning of the SPL in case that happens.



## Chapter 3

# Domain Implementation

In the previous chapter a domain analysis was presented. It showed the system components and their relations while specifying the domain variabilities. In this chapter the tools and techniques used to implement the framework will be demonstrated. Furthermore, the SPL implementation will be shown level by level, giving a deeper insight on how the framework code is structured.

### 3.1 Tools and techniques

One of the constraints of this project is that the developed framework has to be open-source. This also imposes that the tools and programming languages used in the implementation phase are open-source as well, otherwise, there would be no point in developing an open-source framework if the end user<sup>1</sup> would need to acquire proprietary tools to use this framework.

It is also desirable that the used tools have a large user base or a developing team that is able to provide solid support to the usage of the tools. The end user may encounter tool problems and because the framework will most certainly evolve in the future and its developers may not always be the same.

This section specifies used tools, what they are able to do and how they were used. It also specifies which programming languages were used and why.

---

<sup>1</sup>The end user of the framework is the application developer

### 3.1.1 Eclipse

One of the tools that provides support to a vast array of programming languages and methodologies, being used widely around the world in many different fields is Eclipse.

Eclipse has grown since its inception in 2001. What was originally developed by a team at IBM, in 2001, and supported by only a group of software vendors, evolved into an independent foundation to maintain the project and act as a host to the giant eclipse community, constituted by individuals, but also by organizations from a cross section of the software industry.

Eclipse<sup>2</sup> is an IDE that provides tools to manage workspaces, build, launch and debug applications and allowing an easy customization of the programming experience. Eclipse provides a platform with an architecture that allows indefinite extension with more sophisticated tools. Therefore, one can say that Eclipse provides a basic workspace built on a mechanism for discovering, integrating and running plug-in modules[20]. Eclipse also provides a marketplace where plug-ins can be downloaded and easily added to the workspace, a user can also add plug-ins by specifying the on-line repository where the plug in is available.

Currently Eclipse has a vast range of community developed plug-ins that allow the customization to suit the software developer needs. In the scope of this project, Eclipse uses Java development plug-ins, Java EE development plug-ins, database development plug-ins and the Feature IDE plug-in. The first three plug-ins provide development support for Java and Java EE as well as database development, Feature IDE is a specific plug-in that makes Eclipse support FOSD and will be explained later on this chapter.

By using these plug-ins, Eclipse supports all the phases of the FOSD and allows the implementation of the framework at hand. On top of this, it also allows the implementation of a specific application after instantiation, making it the ideal tool for the implementation process. The used Eclipse version for implementing the framework was Eclipse Juno (4.2).

### 3.1.2 Feature IDE

Feature IDE is an open-source Eclipse plug-in for FOSD development maintained by the Database Workgroup from the Faculty of Computer Science from the *Otto Von Guericke University Magde-*

---

<sup>2</sup>Detailed information on Eclipse can be found on the Eclipse project page: <http://www.eclipse.org>

burg. Currently it has a growing user base stemming from all over the world and the latest version is 2.6.5, which is based on Eclipse 4.2.

It provides a coherent user interface and task automation for what previously required the use of complex tool chains, while supporting several software composition techniques, which have very identical user interfaces, making it qualified for usage in teaching and comparison of SPL implementation techniques. Since it provides a solid tool, with support from a team of developers, that abstracts many of the aspects of FOSD development, Feature IDE is also suited to be used in business environments for the generation of frameworks<sup>3</sup> for a vast array of different domains.

Since 2004, when it was first presented, Feature IDE has greatly evolved and currently supports the following FOSD tools and languages:

- *Feature House* - Software composition framework supported by a corresponding toolchain[1]<sup>4</sup>;
- *Feature C++* - C++ extension to support FOP[2];
- *AspectJ* - Simple and practical aspect-oriented extension to Java[12];
- *Delta J* - Java instantiation of DOP[21];
- *Antenna* - Set of Ant tasks for the development of wireless Java applications[18];
- *Munge* - Purposely-simple Java preprocessor[27];

As stated, Feature IDE supports all four phases of FOSD.

The result of the first phase, *domain analysis*, is a feature model. Feature IDE allows the graphical construction of a feature model by providing a graphical editor, which allows the easy addition and removal of features. Feature models can change a lot over time, therefore this editor provides a way to move features, with its sub-features, to a new parent feature. Feature models are stored in eXtensible Mark-up Language (XML) format and can be edited both graphically or textually simultaneously because Feature IDE provides a function that allows the change between both views easily. These models can also be exported to external tools and stored in several graphic formats as well as PDF.

Feature models can also contain cross-tree constraints, which serve to relate features between them by using boolean relations, for example if feature  $x$  is selected, feature  $y$  must also be

---

<sup>3</sup>Successful SPLs have been implemented with up to 10 000 features and cross tree constraints.

<sup>4</sup>Supports: Java, C, C#, Haskell, Alloy, JavaCC, XHTML, XMI/UML and Ant

selected, or, if feature  $w$  is selected, then features  $r$  and  $z$  can't be selected. For this purpose, Feature IDE provides a constraint editor with content assist and syntactic and semantic checking.

In Feature IDE, features mapped to implementation artifacts (e.g: classes, interfaces, methods, etc.) are called *concrete features*. However, Feature IDE also supports *abstract features* which aren't mapped to any artifact. This distinction is necessary for the automated analysis and reasoning on the set of application variants that can be generated from a feature model and its corresponding *domain implementation*. Basically, abstract features can be described as features which are used to structure a feature model but don't have any impact at implementation level, because they aren't mapped to any code[29].

*Domain implementation* in Feature IDE consists on the mapping of concrete features to the different features by using one of the different SPL implementation techniques described above. The mapping depends on the chosen SPL implementation technique.

The third stage of FOSD is *requirement analysis*. Feature IDE provides a configuration editor to meet this end. This editor gets the feature model as the input and provides the several configuration choices, based on the feature model disposition.

The user selects the desired features and saves the selections into a configuration file. There can be many configurations at the same time, but only one can be marked as current configuration, for which source code is composed and compiled. During configuration, the editor marks it as valid or invalid. Invalid configurations don't respect the feature model constraints and don't result in composed application code. Configuration choices are only given according to the feature model, so that its constructs are respected. Therefore mandatory features cannot be eliminated and two alternative features cannot be selected at the same time.

The last phase to be covered by Feature IDE is software generation. This is accomplished automatically by using the feature model, the implementation artifacts and a valid configuration. The build is made in the standard Eclipse way and, if the automatic build option is enabled, a new build is performed every time a source file changes. Feature IDE composes files to a specific folder which is then used as the input folder for the compiler of the host-language. Therefore all build options (e.g. build path and build parameters) can be configured as usual for the host-language

in its development tools (e.g. JDT<sup>5</sup>, CDT<sup>6</sup>). Feature IDE provides a way to define feature order. This defines the order on how software artifacts are composed, assuring that composed code has a logically correct sequence.

While generating new applications, programming errors may arise in the generated code, detected by the Eclipse compilation plug-ins. However, this isn't where errors must be corrected, errors must be corrected at their source, in the feature code where they first appear. Feature IDE provides error propagation, which is the ability to locate the exact position of an error in a feature when only the error position in generated code is known.

Feature IDE provides integration for the four phases of FOSD by supporting dependencies between particular phases. This means that when the feature model is changed, all configurations are checked for validity, which may result in valid configurations turning into invalid. The renaming of features is also covered by propagating the changes to configurations and domain implementation. Feature IDE also synchronizes changes by checking the existence of features in the domain implementation.

## Feature House

Feature IDE generates code by using a valid configuration that defines which features are to be used to generate a new application. Since code is directly mapped to features, there must be a way to join their code in a way that it can be usable as a whole defining an application code. This process is named software composition.

Software composition can be defined as *the construction of software applications from components that implement abstractions pertaining to a particular problem domain*[16]. By doing this software composition methods raise the abstraction level for the application developer, improving the way to deal with complexity. Even though there are some benefits on using software composition, the biggest one is in the increased flexibility of the composable software systems. Thus, a system built from components should be easily recomposable as a way to address new requirements for different problems within a domain[17].

Feature House is a software composition tool chain, used by Feature IDE, based on three main

---

<sup>5</sup>Eclipse Java Development Tooling

<sup>6</sup>Eclipse C/C++ Development Tooling

ideas:

1. Language independent model of software artifacts;
2. Superimposition as a language independent software composition paradigm;
3. Artifact-language specification based on attribute grammars;

This tool chain supports a range of programming languages<sup>7</sup>, including Java, and uses the basis of superimposition into which new languages can be plugged on demand, by merging software artifacts corresponding substructures[1].

Feature House results from previous work, integrating a previous software composition tool, the FSTComposer[3], which uses Feature Structure Trees (FSTs) as a model of the software artifacts structure that represents the essential structure of a software artifact while abstracting it from language specific details. Each node of a FST has a name that corresponds to a structural element and a type that represents the syntactic category of a structural element. A FST is, basically, a stripped-down syntax tree containing the necessary information to specify the modular structure of an artifact and for its composition with other artifacts.

FSTs are composed by inner nodes, called *nonterminals*, and leaves, called *terminals*. Nodes denote the artifact modules<sup>8</sup>, while leaves carry the modules content<sup>9</sup>. Figure 3.1 demonstrates the relation between the code, in this case Java, of an artifact and the graphic representation of the FST where the code is contained. As it can be noted, the first two nodes represent the packages *com* and *sleepycat* and the third represents the *Database* class, these are all *nonterminal* nodes, the terminal nodes that represent the contents of the *Database* class, *state* and *triggerList* are private variables of the class (fields), and *notifyTriggers* is a protected method of the same class.

Software composition is then accomplished by the superimposition, denoted by '●' of corresponding FSTs, made by merging their nodes, identified by their names, types and relative positions, starting from the root and descending recursively. This merging is done by overriding one method with the other.

---

<sup>7</sup>Java, C#, C, Haskell, JavaCC, Alloy and UML

<sup>8</sup>Classes and packages

<sup>9</sup>Method bodies and field initializers

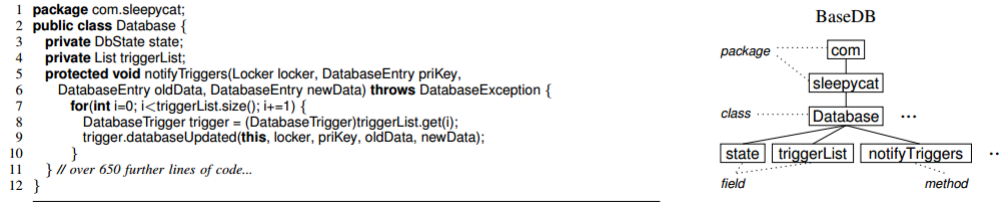


Figure 3.1: Java code and FST of artifact BaseDB, taken from the Berkeley DB case study[1]

FeatureHouse uses the *original* keyword to define the method to be overridden, hence defining how merging is accomplished. A code parser is used to classify *original* as a method name, locating every occurrence of the *original* keyword while composing two method bodies and substituting each occurrence with the original method body, wrapping it. This makes the *original* keyword unable to be used as a method name or as any other kind of identifier. Figure 3.2 shows an example of how code composition is achieved. It shows the original method, the method that will override the original method and the result of the merging of both methods.

If the programming language used to implement the code has *original* in its grammar, then FeatureHouse restricts its semantics, limiting the usage of the *original* keyword, so that it is able to support composition of method bodies. The result of this process, which is completely transparent to the user, is syntactically correct code.

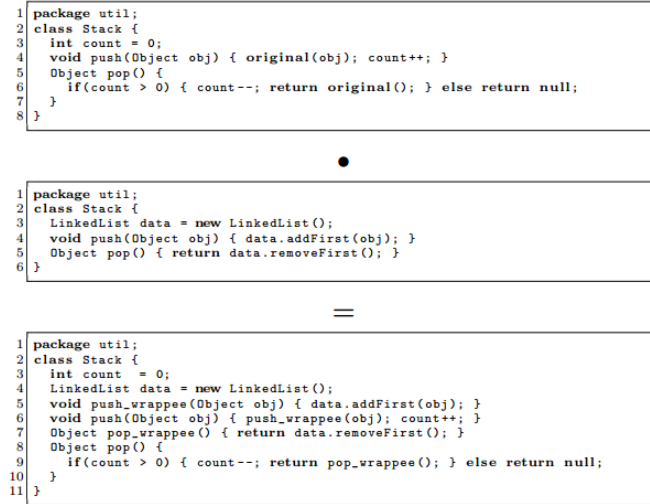


Figure 3.2: Composition of two Java methods in FeatureHouse[3]

Feature IDE uses feature diagrams as FSTs defining the code structure. A top feature implements the original method that can be overridden by the methods defined in features located in a lower position of the feature diagram. FeatureHouse is then used to parse the features code and to merge it, based solely on the features chosen by the SPL user and their relations, creating syntactically correct code that is fit to be used to implement an application.

### 3.1.3 Jersey and JAX-RS

Java was chosen for implementation because it offers wide support for web service development, and its usage as SPL development language makes sense since the Web service part of the system represents the most of the SPL implementation.

Although there are a few Web service implementation protocols, being SOAP the most used, REST was chosen as the implementation protocol because it simplifies the interaction between client and web service, and since it is fully compliant with HTTP, makes it possible for the use of HTTP libraries existing for Java. To implement RESTful Web services Java provides the JAX-RS API that uses Java SE 5 annotation to simplify the development and deployment process of Web service clients and endpoints. Furthermore, JAX-RS is extended by several implementations, from which are included Apache CXF, RESTeasy, Restlet, Apache Wink, WebSphere Application Server, WebLogic Application Server and Jersey, where Jersey is the reference implementation of JAX-RS, and thus more suitable for this project, since it does not depend on any specific application server.

Jersey extends JAX-RS by providing its own API which implements additional features and utilities that make RESTful Web services and client development easier[25].

JAX-RS concepts and resources include:

- **Root Resource Classes:** Plain Old Java Objects (POJOs) annotated with *@Path* and have at least one method annotated with *@Path* or a resource designator<sup>1011</sup>;
- **Parameter Annotations:** used to annotate resource parameters so that information can be extracted from a request. These annotations include: *@PathParam*, *@QueryParam*,

---

<sup>10</sup> *@GET*, *@PUT*, *@POST*, *@DELETE*, the HTTP methods

<sup>11</sup> *@Produces* and *@Consumes* are also Root Resource Classes and specify the MIME media types of the data exchanged by the service and the client



*@DefaultValue, @QueryParam, @MatrixParam, @CookieParam and @FormParam;*

- **Sub-resources:** methods of Root Resource Classes where the *@Path* annotation is also used, enabling common functionality for a number of resources to be grouped together a potentially reused;

Every HTTP request has an associated life cycle, REST requests are essentially HTTP requests and therefore have a life cycle too. Since a new instance of a root resource class is created every time the request URI path matches the root resource, its life cycle matches the request life cycle. This makes way for a simple programming model where constructors and fields can be used without the concern for multiple concurrent requests on the same resource. This does not create performance issues, since, due to the optimization of the Java Virtual Machine (JVM), many objects will be created and discarded to serve the HTTP request and response.

## 3.2 SPL Implementation

To implement the SPL that will give body to the framework, defining a generic set of software artifacts that can be used to compose a specific software application[17], the domain must be divided into pieces, called features, specified in the feature diagram. These features are mapped directly to code, in this case Java code.

Since the Web services represent the main and biggest part of all the architecture and Java provides a useful API and other resource for their implementation, the SPL is implemented using Java. Thus, SPL features are directly mapped into Java code, even though there are subsystems that are programmed in other languages, their features are also implemented using Java. Java is then used to generate the code in other programming languages.

By selecting different features<sup>12</sup>, to suit different problems, one can create a range of different applications to answer specific problems that are different between them. Chapter 4 will show how to do this, by specifying how to instantiate the SPL to a specific problem. The final code is generated by using the software composition technique described in section 3.1.2.

Taking a first look at the architecture defined in chapter 2, one can easily divide it in three

---

<sup>12</sup>Configuration

distinct levels, one for each part of the architecture<sup>13</sup>. Thus, it makes sense to also divide the SPL into three levels (specialization levels), corresponding to the architecture subsystems defined in figure 2.3.

Comparing figure 2.3 with figure 3.3 it is easily noticeable the relation between the levels defined for the SPL and the subsystems defined by the architecture. There is a *platform level*, corresponding to the platform interface running on the platform subsystem, a *user interface level*, that corresponds to the user interface subsystem that does the interface with the user, and there is a *server level* matching the server subsystem. In the layered architecture of the server, the three outer layers (service, business and DAO) correspond to the Web services part of the server and the inner layer (core) is the database server, where data is stored.

Figure 3.3 also shows the interface between all the levels. It can be noticed that the user interface communicates with the service layer, to where it sends HTTP requests that are processed and then answered in the form of a HTTP response. There is also an interface between the platform and the server. Platforms send alerts as HTTP to the service layer but notice that the server sends actions to the platforms via a socket, and not as HTTP requests, thus sending data from the business layer and not from the service layer. The usage of sockets, instead of HTTP, is done so that the server can send data to the platforms without a request, otherwise this would not be possible, this also reduces the message overhead because unlike HTTP it only the data is sent.

Even though it is easier and more simple to implement a feature diagram for each level, feature IDE doesn't yet support cross diagram constraints that are used to assure consistency between the levels. Therefore, all the architecture must be described in a single feature diagram composed by three main branches, where each main branch represents a level of the SPL. Feature IDE supports cross-tree constraints, assuring consistency between levels, since it allows the automatic selection of one or more features depending on a previous logical condition (*e.g. if one is selected then two is also selected*).

Currently the SPL is comprised of 120 distinct features, that describe the domain components. From these 120 features, 92 are *concrete* and 28 are *abstract*. This means that 92 of these features are mapped to code, while the remaining 28 are not and exist to help describe and understand the domain. The 92 concrete features are coded in Java following the composition rules specified

---

<sup>13</sup>User Interface, Server and Platform Interface

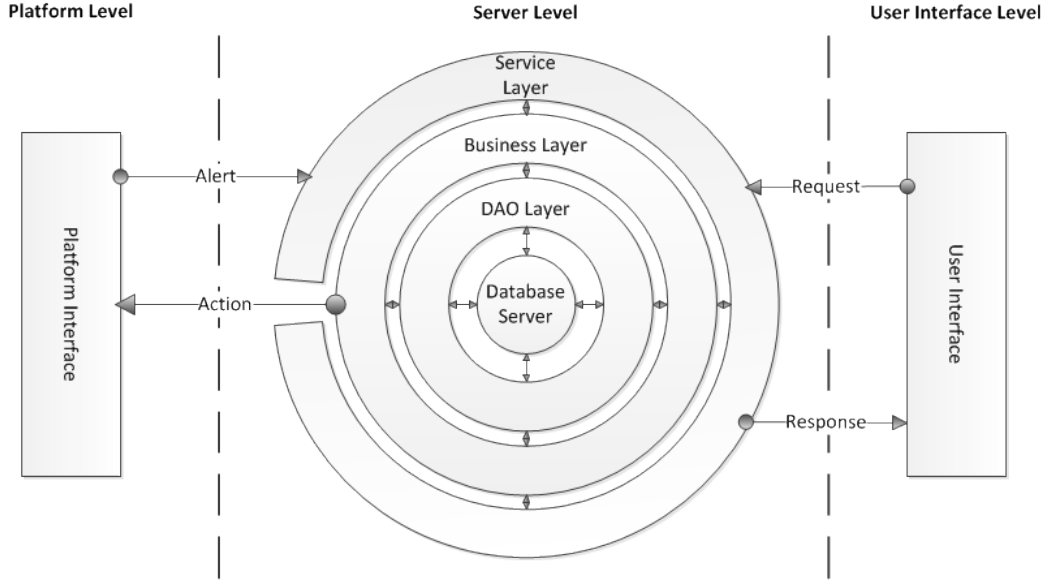


Figure 3.3: Framework architecture divided into levels

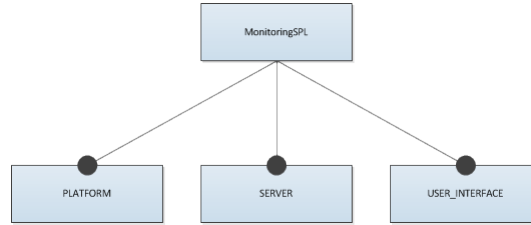


Figure 3.4: SPL main branches

by FeatureHouse in sub section 3.1.2. This means that each feature implements one or more Java methods and some of these methods are defined in more than one feature, in this case one feature implements the basic code, and is overridden by another feature that uses the *original* keyword where the basic code is needed. By doing this the second feature defines the entry point to the method and when it needs the code of the original method invokes it by using the original keyword. Whenever the software composer finds the original keyword it looks for the original implementation of the method, defines a wrapper for that method and replaces the original keyword with the wrapper, thus calling the original method and combining both features.

The following code listing shows the *Login* method from the business layer overridden when the user selected the feature that defines the logging of user actions. The software composer changed the name of the *Login* method to *Login\_wrappee\_Authentication* and the second method, that

defines the entry point for the authentication, calls it. So the first method logs in a user in the system and the second registers it in the user history. If the history feature had not been selected, the Login method would have not been overridden and would be the entry point for the login.

```
private String Login__wrappee__Authentication (String username, String password)throws Exception
{
    String status = null;
    DAOFactory monitor = DAOFactory.getInstance("monitor.jdbc");
    UserDao userDao = monitor.getUserDAO();
    User dbUser = new User();
    dbUser = userDao.FindByUsername(username,password);
    if(dbUser.getUsername().equals(username))
    {
        status = userDao.CreateSID(dbUser);
    }
    else
    {
        status = "NOK";
    }
    return status;
}

public String Login(String username, String password)throws Exception
{
    //Override the original method declared in the authentication feature
    //don't forget to add function arguments in generated code [Login(username,password)]
    String status = Login__wrappee__Authentication(username, password);
    DAOFactory monitor = DAOFactory.getInstance("monitor.jdbc");
    UserHistory uHistory = new UserHistory();
    UserHistoryDAO uHistoryDAO = monitor.getUserHistoryDAO();
    if(status == "NOK")
    {
        uHistory.setUserID(username);
        uHistory.createTimestamp();
        uHistory.setDescription("User with username: "+uHistory.getUserID()+" tried to log in at "+uHistory.
            getTimestamp());
        uHistoryDAO.Create(uHistory);
    }
    else
    {
        uHistory.setUserID(username);
        uHistory.createTimestamp();
    }
}
```

```

        uHistory.setDescription(uHistory.getUserID()+" successfully logged in at "+uHistory.getTimestamp()+"
            with sID: "+status);
        uHistoryDAO.Create(uHistory);
    }
    return status;
}

```

Now that the way code is composed is explained, there is only one thing left to understand, the feature diagrams of each level of the SPL. Even though the feature diagram is a product of the Domain Analysis, since features are mapped to code it makes sense to expose the feature diagrams at the same time that their operation is also explained.

### 3.2.1 Server

The server receives data as HTTP requests and carries on a number of operations that depend on what Web service is addressed through the URI. Requests are received in the service layer, processed in the business layer and stored in the database, which communicates with the Web services via the DAO layer. The following subsections expose the way data flows between all the layers and also define the database and how it describes the system.

#### Web services

As it was explained earlier, the Web services have a layered architecture. When a request is received it is propagated through all the layers and its data is processed, after processing is done the server sends a response back to the client, thus satisfying the client request. When a client sends a request to a specified Web service, what really happens is that the client is addressing a Java method that is mapped to a Web address by using the *@Path* annotation that defines the method access path. *@GET* and *@POST* annotations define what HTTP method is allowed to be used with the Web service, *@Consumes* defines what type of data the Web service consumes (receives). In the implemented code all the Web services consume data encoded in the Web service URL. *@Produces* defines the encoding of the data sent from the Web service to the client as a response to a request, in the Web services implemented all these responses are encoded in the XML format.

For example, the server sends a GET request to the `/user/list` Web service, requesting the list of users and their details. Upon reception of the request, the service layer invokes the appropriate method from the business layer, that invokes the DAO method responsible for fetching the user list from the database. The data received from the database is then sent back up in the reverse way, the DAO layer method returns to the business layer that returns to the service layer that sends the requested list as a response in XML format.

This data flow is shown of figure 3.5.

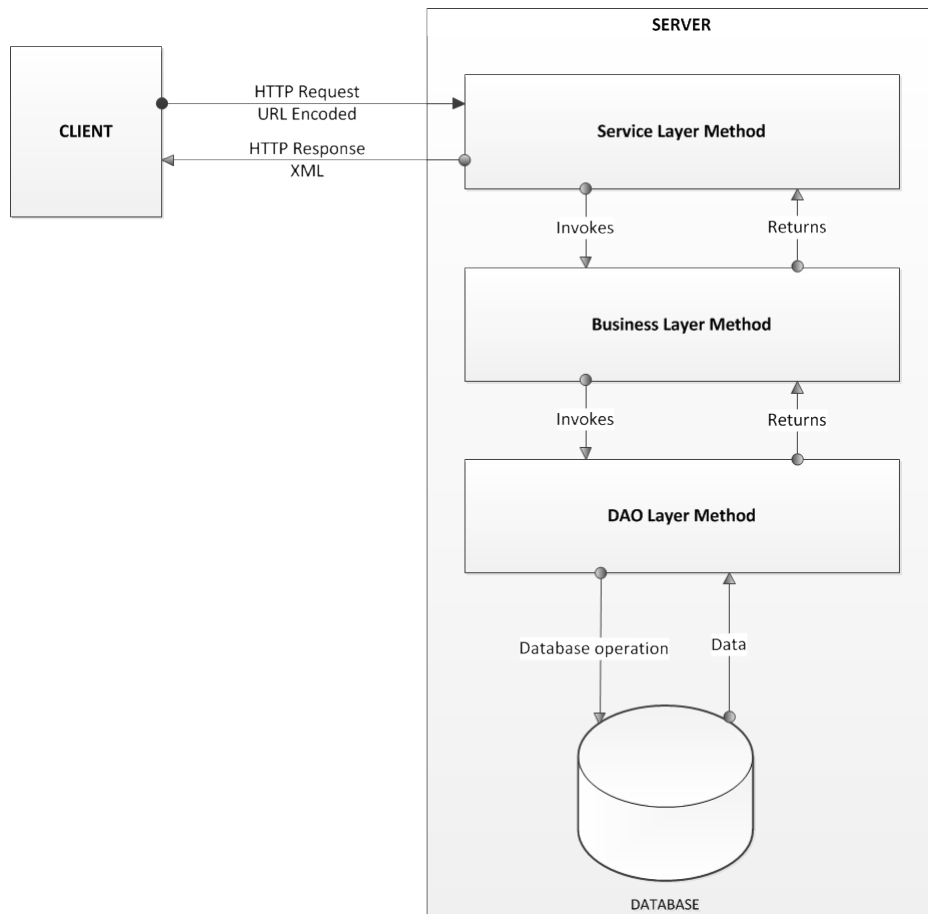


Figure 3.5: Web services data flow

In order to simplify the way the layered architecture of the Web services is implemented, promoting a modular implementation so that a possible changing of code is easier, the layers are not only implemented in different Java classes, but also in different Java packages. This makes it possible to change a complete layer by just changing the package or to change part of it by

changing a class, as long as the method interfaces are respected. The implemented packages are:

- *mon.spl.srv* for the service layer;
- *mon.spl.biz* for the business layer;
- *mon.spl.dao* for the Database Access Object (DAO) layer operations;
- *mon.spl.model* defining Java beans<sup>14</sup> that are used to provide a database table model in Java code where data can be temporarily stored for processing and transmission;

While defining the feature diagram, shown on figure 3.6, one of the goals was to abstract these layers and to define the features of the Web services as the tasks the Web services can carry. Therefore features are divided in platform interaction features, *Actions* and *Alerts*, *Management* features, *History* features, *Authentication* and *Utilities*, that implement Web services that may be useful, such as lists. The only layer that is explicitly defined in the features is the DAO layer and also the data *Model*. The DAO layer is explicitly defined because by doing so it helps the SPL evolution on supporting other database systems. Currently the framework only supports MySQL databases, however the need to use another Relational Database Management System (RDBMS) (e.g. Oracle database, Microsoft SQL server, Sybase, IBM DB2) may arise. If this happens, and since the DAO layer is different for each RDBMS, the DAO layer will represent a variation point in the system where the framework user can select a DAO layer for a specific database system from an array of available DAO layers, each specific to a database system.

In this diagram it can be noticed the usage of abstract features, in light blue, to group feature groups. For example the *Model* feature does not implement any code, but groups all the models as derived from the Model branch.

Starting by the *Authentication* feature, this feature implements three Web services, *Login*, *Logout* and *Check Privilege*. It is intended to provide the user interface with login ability and since the access to the user interface windows is based on privilege, it provides a way to get the access privilege of a user using the session ID. When a user logs in the system, a random session ID is created, stored in the database and sent to the user interface as a response to the login request. While the user navigates through the user interface, every time he accesses a new window, the

---

<sup>14</sup>A Java bean is a class with only data fields and their getters and setters that is, among other things, serializable

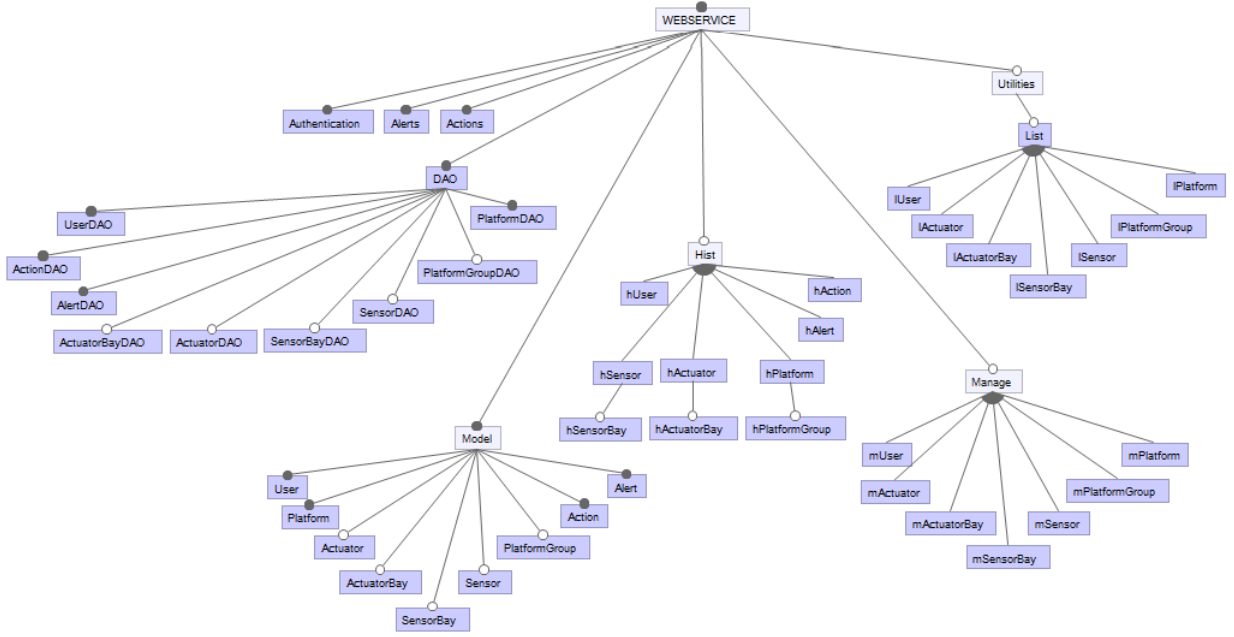


Figure 3.6: Web services feature diagram

user interface requests the user access privilege to the server, to where it sends the session ID. This session ID is used as a *select* argument to the database, because it is related to the user name, to get the user privilege. After the session ID is retrieved from the database, it is sent to the user interface as a response to the client. When the user logs out, the user interface sends the session ID to the server that uses it to remove the session ID from the database, thus logging the user out.

The *Alerts* feature implements all the services related to alerts. Both management Web services as well as Web services to store current alerts<sup>15</sup> in the database. This feature provides a way to manage the available alerts, defining which are the possible alerts to be sent by a platform, by implementing Web services to add, edit and delete alerts from the database. These are basic Create, Read, Update and Delete (CRUD) operations on the database, and therefore when addressed they just access the database to perform the desired database operation (*insert*, *select*, *update* or *delete*), sending the feedback of the operation back to the user interface. Besides these services, this feature also provides Web services to deal with alerts sent by the platform

<sup>15</sup>Current alerts are defined as all the alerts sent by the platform to the server that have not been addressed by the user



to the server. The platform is also a client of the server and sends alerts by accessing the *alert/current/new* Web service. This service stores the alert received from the platform in the database, on a table specific for that end. The user interface then polls the database for alerts sent by platforms in a periodic basis and the Web service responds the user interface by sending a list of all the alerts that have not been attended that are present in the database. This list is then shown the user that selects the an action to address each alert. To avoid confusion, note that the response the server sends to the platform upon receiving a new current alert is generic ("*OK*"), stating only that the alert was received by the server, and not the action to address that alert.

*Actions* are closely related to alerts in the way that they can be used to address the alerts sent by the platform, however actions can be sent without an alert being issued. Like the *Alerts* feature, the *Actions* feature, provides Web services to perform CRUD operations in the table that defines the possible actions to be taken upon a platform. But it also provides a Web service to send actions to a platform. This Web service, receives the code of the desired action to be sent to the platform with the code of the alert associated to it and the user session ID. It then proceeds to remove the alert code from the current alert table in the database and send the code of the selected action to the platform via a socket. This action is sent directly from the business layer and not from the service layer as it is the result of business operations. When the action is sent successfully the Web service sends the response to the user interface proving that the action was sent successfully. In case of a socket connection error, a socket exception is thrown and the action is not sent, in this case the alert stays in the current alert table so that it can be addressed later.

The *Manage* feature is abstract and groups all the different management features on the server side. These features perform CRUD operations on the database, providing a way to manage system components such as platforms, sensors and actuators, as well as users. Even though there are seven distinct management features, their operation is the same, only performed on different database tables. For example, the *mUser* platform provides Web services to add a new user to the system, to edit a given user details and to delete a user from the system. The same is valid for all the remaining six management features, *mActuator*, for actuator management, *mActuatorBay* for actuator bay management, *mSensor*, for sensor management, *mSensorBay*, for sensor bay management, *mPlatform*, for platform management and *mPlatformGroup* for platform

group management. These features are completely independent from each other.

As a way to provide the ability of a system to record all the actions taken upon it, the *Hist* feature was implemented. This abstract feature groups a set of features that override methods that belong to other features as a way to give them logging ability. For example, the basic operation of the Login Web service is to generate a session ID and to store it in the database every time an user logs into the system. The *hUser* feature overrides the Login method in the business layer and after the generation of the session ID and its registry in the database, the new Login method invokes the DAO layer method that will register in the user history table the user login with the corresponding time stamp, thus creating an history of user actions. The same is done by *hAction*, that creates the history of all the actions taken, *hAlert*, that creates the history of all the alerts received and *hSensor*, *hActuator* and *hPlatform* that create histories of all the management actions in the respective components.

The *DAO* feature is mandatory, since it implements the DAO layer of the Web services. Its children can be mandatory or not depending to what system component they relate. The *DAO* feature, besides grouping the other DAO features, implements a DAO library that allows the connection to the database, this library implements classes and methods to access the database with exceptions control, in case there are any errors in the database connection or in the transferred data between the DAO layer and the database. The remaining DAO features implement the lower level code to perform CRUD operations on the database tables. Each DAO feature implements database operations for its corresponding system component, thus *UserDAO* relates to user management and authentication operations, *ActionDAO* relates to action management and action handling operations, *AlertDAO* relates to alert management and alert handling operations, *ActuatorDAO* and *ActuatorBayDAO* relate with actuator and their respective bays management, *SensorDAO* and *SensorBayDAO* relate with sensor and sensor bay management and *PlatformDAO* and *PlatformGroupDAO* relate with platform and platform group management respectively.

*Model* features, grouped by the abstract feature *Model*, implement a class each that defines a Java bean to store the data retrieved from the database. These models are simple classes only with data fields and their getters and setters that are serializable and can be easily encoded to XML by using the *@XmlRootElement* and *@XmlElement* annotations. This makes it possible to transform a model, coded in Java, directly into XML and send it to the user interface as a response

to a request. These models can be mandatory or optional and are directly related to the *DAO* features, that in turn can also be mandatory or optional and are directly related to the mandatory and optional components of the system.

The last set of features, is the *Utilities* group of features that only implements lists for the components of the system. The *List* features provide methods, for troubleshooting and management classes, that select all the data from a given database table, maps each entry of the database to an instance of a Java bean defined by a Model feature and adds that instance to a list. Since models are easily encoded in XML the full list is also encodable in XML and it can be sent as a response to the user interface.

## Database

To store all the details and records associated with this system and to provide ease of access when data is needed, a database was implemented. This database was implemented in MySQL and it was also divided into features, where each table represents a feature from the feature diagram, as shown on 3.7.

Unlike the Web services the database is not implemented in Java code, being implemented as a SQL script that can be run in the MySQL server to generate the database. The SQL code is written in a file using Java file writing methods. When a feature is selected it adds the table code to the file. In the end the file is generated by a process that must be run after configuration.

*dbTop* and *dbBottom* implement the top and the bottom of the SQL file, corresponding to functional code related to the database that is going to be created instead of code that implements database tables.

In order to understand all possible database configurations one must understand how the full database, depicted on 3.8, works. Therefore the database will be exposed as a whole specifying tables and their relations. The following list makes a match between the features of the database feature diagram and the table that each feature implements:

- *dbTop* - Implements the top of the database script
- *dbBottom* - Implements the bottom of the database script
- *tUser* - Implements the User table

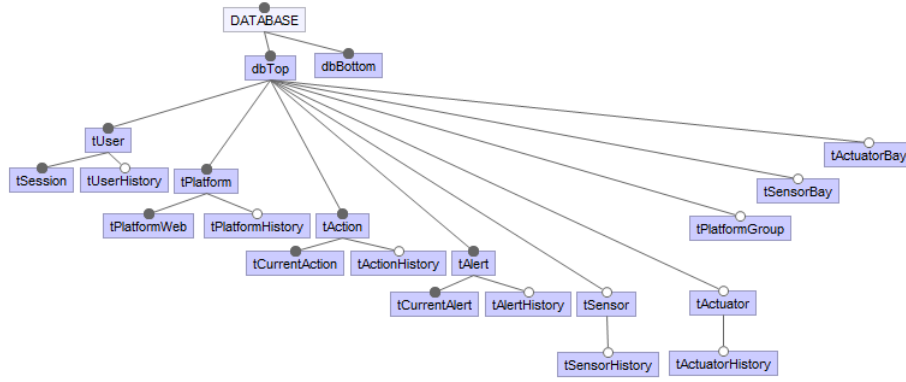


Figure 3.7: Database feature diagram

- *tSession* - Implements the Session table
- *tUserHistory* - Implements the UserHistory table
- *tPlatform* - Implements the Platform table
- *tPlatformWeb* - Implements the PlatformWeb table
- *tPlatformHistory* - Implements the PlatformHistory table
- *tAction* - Implements the Action table
- *tCurrentAction* - Implements the CurrentAction table
- *tActionHistory* - Implements the ActionHistory table
- *tAlert* - Implements the Alert table
- *tCurrentAlert* - Implements the CurrentAlert table
- *tAlertHistory* - Implements the AlertHistory table
- *tSensor* - Implements the Sensor table
- *tSensorHistory* - Implements the SensorHistory table
- *tActuator* - Implements the Actuator table
- *tActuatorHistory* - Implements the ActuatorHistory table

- *tPlatformGroup* - Implements the PlatformGroup table
- *tSensorBay* - Implements the SensorBay table
- *tActuatorBay* - Implements the ActuatorBay table

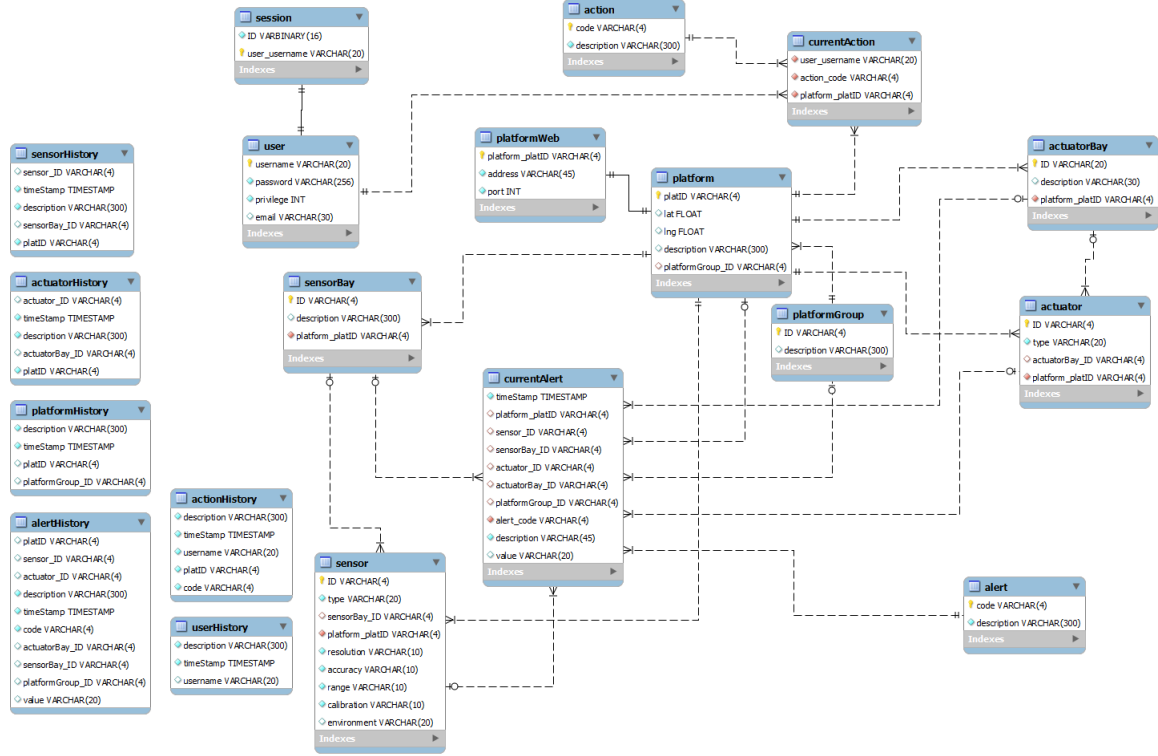


Figure 3.8: Database relational diagram

The central part of the system is the Platform, a platform is characterized by an ID and has a location defined by latitude and longitude values. It also has a description. Directly associated to a platform, in a 1:1 identifying relationship are the Web details of a platform, these details specify the IP address of a platform and the port used for communication. The platform ID relates both tables.

Platforms can be grouped, a group is constituted by one or more platforms and is identified by an ID. Each group also has a description.

A platform can have one or more sensors that are identified by an ID, have a type and are characterized other aspects, such as range, calibration, resolution, accuracy, resolution and the

environmental conditions they can work in. These sensors can be grouped into sensor bays that have one or more sensors and are also identified by an ID. Each platform has one or more sensor bays.

Like sensors a platform can also have one or more actuators, these actuators are identified by an ID and have a type. Actuators can be grouped into actuator bays that have an ID and also belong to, one or more, to a platform.

Alerts are emitted by platforms every time data is sent to the server, therefore every outgoing message from a platform is an alert. Upon reception of a request from the platform, in this case a *POST* request with all the alert related details, the Web service creates a *current alert* entry in the database, specifying that the received alert has not yet been addressed by the user.

Current alerts are related to system components since they are originated by them and emitted by the platform. Thus a sensor, or any other platform element, can be the source of an alert which will have an associated code as well as all the other aspects related to it, such as a description of the alert, a time stamp and the value registered by the source of the alert<sup>16</sup>.

To address an alert, the user interacts with the user interface to send actions to the server that redirects it to the platform. Available actions are defined by a code and also have a description. When the user issues an action to a platform, the action becomes a *current action* that is applied in a platform by a user, thus it is characterized by the action code, the platform where is going to be applied and by user name of the user who sent it. A user can be responsible for sending one or more actions to a platform.

Users are also characterized in a database table by their user name, their password, privilege level and e-mail. These fields are used for user authentication that produces a session ID. This session ID is stored in the session table that has a 1:1 identifying relationship with the user table, directly relating a give user name to a session ID.

In order to provide logging ability to the system there are tables that are responsible for holding the data related of the operations carried out in or by the system.

Sensor history is defined by the sensor ID or the sensor bay ID, it also holds the platform ID and adds a time stamp to this information, defining when the operation took place. A description defines what happened in a given sensor at a given time.

---

<sup>16</sup>This can be used when a sensor is the source of the alert to send the sensor registered value to the server

The actuator history has the same structure as the sensor history, but holds data related to operations carried upon actuators. Therefore it has the actuator ID, the actuator bay ID and the platform ID. A time stamp defines when the operation took place and a description specifies which operation was carried out.

Platform history is also present and is characterized by the platform ID and the platform group ID, a time stamp and a description of which operation was carried on a platform at a given time.

The logging of alerts allows the user to keep track of what alerts were issued by a platform. Therefore this table bears all the same fields and the current alert table, plus a time stamp and a description of the emitted alert. Since logged alerts have a value field, this is where registered values can be stored for further consulting.

Actions, that can also be logged, are identified by the action code, the platform ID of the platform where the action was taken, a description of the taken action, a time stamp that specifies when it was taken and the user name of the user who took the action.

Finally, the user history table specifies all the user related operations, except the actions, that were carried in the system. This table is characterized by three fields specifying the user name, a description of what operation the user did and a time stamp, specifying when the user carried out the operation.

### **3.2.2 User Interface**

The user interface provides a way for the user to interact with the system. By accessing it, depending on what features are selected, the user can log into the system, troubleshoot the system platforms by sending actions and receiving alerts, manage system components by adding, editing or remove their details from the database, and even consult what events have occurred in the system in the past.

Earlier in this chapter it was specified that features in the SPL are mapped to Java code. However, the user interface is implemented using HTML5, JavaScript and CSS, that work together to implement a Web application that sends request and receives responses asynchronously from the server. Since all three languages are scripting languages that are interpreted by the Web browser, the used strategy was to use Java to generate files with the respective language code

written. By doing this HTML5, JavaScript and CSS code is basically mapped to Java code which is also mapped directly to features, thus creating a way to map these Web languages to features.

Features in the user interface feature diagram, shown on figure 3.9, are grouped into three main branches, that are mandatory. The first branch specifies the *Pages* that compose the user interface, the second branch specifies their visual *Styles* that provide a more appealing visual feedback to the user and the third implements the *Generator*. This *Generator* feature is the only concrete feature from the three main features and implements the *Main* class.

The *Main* class is implemented not to be run in a server, like all the other implemented classes, but to be compiled natively on the development machine. This class is a part of a standalone package that is not related to the other implemented packages, and is responsible for the generation of the script files that compose the user interface and the database, depending on the features chosen at instantiation. The process of compiling this class and running the generated executable file is simple and makes it possible for the easy generation of the files that compose the user interface on demand.

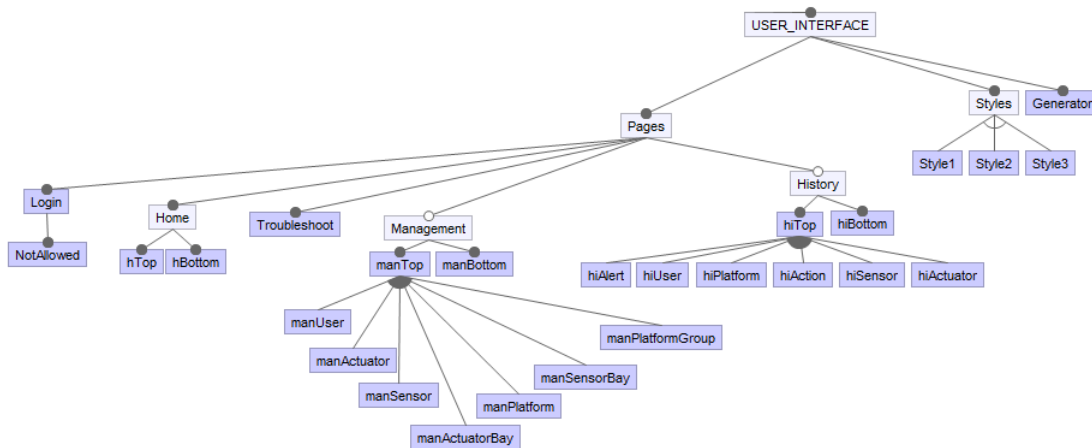


Figure 3.9: User interface feature diagram

The *Styles* are a mandatory part of this interface as well. HTML5 itself does not implement rich visual components, thus there is the need to have CSS scripts to accomplish that. There are three different styles, *Style1*, *Style2* and *Style3*, that can be used on the user interface. These styles can only be used one at the time and provide basic usability by defining where HTML components are placed, how they are shown, their colors and other visual aspects that help to improve the user



interface usability. Creating a new styles in the SPL is accomplished by creating a new feature for it, cloning the class of the other styles, and editing the Java file writer string in order to create the desired CSS file.

The last group of features defines the *Pages* that compose the user interface. These features are closely related to the features of the Web service because they provide a way for the user to interact with the Web services, thus they match almost all the features that compose the Web services feature diagram.

*Authentication* is the part of the system that gives the user a way to log in the system, it is also the feature that sends the user access privilege to the user interface. Therefore the *Login* feature, that presents the user with a login page sends the user credentials to the Login Web service and receives the session ID, as a response, in case the user is allowed to access the system, is directly related to the Authentication feature in the Web services feature diagram.

The Login page presents the user with a regular login prompt. Upon sending the user credentials to the server, in case the input credentials match the ones on the database, the user interface receives a randomly generated session ID as the server response. This session ID is stored in a persistent variable provided by the browser and is passed to other pages when the user accesses them. Every time a user accesses a page the page access level is compared with the stored user privilege level, and if the user privilege is bigger than the page access privilege then the user is allowed to access the page, if not it is redirected to a page that presents a "not allowed" message. This page only presents a text message and is implemented by the *NotAllowed* feature.

When the user is granted access to the interface it is redirected to the home page. This home page is very simple, only presenting the user with a text message and buttons that act as links to the other user interface pages, and is implemented by two concrete features, *hTop* and *hBottom*. The first feature implements the top of the home page, the head and the top part of the body, and the second the bottom. This happens because the HTML format has a clear syntax on how Web pages are organized, a web page starts with a head section (`< head > ... < /head >`) that is followed by a body (`< body > ... < /body >`). Since there are features from other branches that extend the home page, by adding buttons that link to other pages, the *hTop* feature implements the top of the page, then features from other branches implement their code in the home page file and to finish *hBottom* adds the tag `< /body >` to the home page file, marking its end.

Since Feature IDE provides a way to define the order on which feature code is composed, if the features that implement the bottom of the Web pages are after the features that implement the rest of the Web pages, correct composition is assured resulting in a correctly composed Web page.

The most crucial feature of the user interface is the *Troubleshoot* feature. This feature implements a way to let the user receive alerts from the platforms and to send actions to address those alerts. It is a mandatory feature that starts by adding a button for the troubleshoot page to the home page. After that it implements a HTML page that displays a table where alerts are shown and a return button, and a JavaScript file that implements the communication back-end with the server. When the troubleshoot page loads it requests the list of current alerts to the server, upon receiving it the page creates a table where it displays all the current alerts and their details. A select button on each line of the table provides a way for the user to choose an alert to be addressed, when an alert is chosen the page substitutes the alerts table by another table that displays the available actions to be sent to the platform. When the user selects an action, the action code and all the details of the corresponding alert are sent to the server that is responsible for sending the action code to the correct platform. This page also implements a way to send actions to a platform without receiving an alert. When accessing the troubleshoot page there is a button that displays the available actions table to the user and a drop-down list where the user can select the platform where the alert is to be sent.

As a way to know when there are alerts that need to be addressed, and since the server can't send data to the user interface without a request from it, all the pages from the user interface send a request for the existence of alerts to the server, every twenty seconds, or any other period that can be defined by editing the code of the HTML files. If the response is not null then there are alerts that need to be addressed and the user is prompted to access the troubleshoot page. Every time there is an alert request, the server responds with the full current alert list, sending all unanswered alerts to the user interface. Upon reception of the list the troubleshoot page is refreshed to match the unanswered alerts in the database.

Although alerts are related to actions, this relation is only meant for logging purposes, the platform does not receive the alert that triggered an action. The user interface makes it possible to send actions without alerts by specifying the alert code *0 - No Alert* and to dismiss alerts without

taking real action in the platform also by defining the action code *0 - No Action*.

Matching the *Manage* feature from the Web services feature diagram there is the *Management* feature. This feature implements the management page, with *manTop* and *manBottom* carrying the same tasks as *hTop* and *hBottom*. This page is extended by seven other features, *manUser*, *manActuator*, *manActuatorBay*, *manSensor*, *manSensorBay*, *manPlatform* and *manPlatformGroup*, that add the ability to manage their corresponding system components. The display for the management of all these components is the same, consisting on a table that shows the components present in the system and a set of controls that allow adding, editing and the removal of a given component. These controls implement a set of text boxes that the user fills out and which data is sent to the server as a request to the matching Web service.

The last group of features is *History*, matching the *Hist* feature in the Web services feature diagram. Like the *Home* and *Management* features it also implements the top, *hiTop*, and bottom, *hiBottom*, independently. Like the *Management* feature there are six features, *hiAlert*, *hiAction*, *hiUser*, *hiPlatform*, *hiSensor* and *hiActuator*, that extend the history page by adding tables that display the history of the components related to the system. This page also adds a few controls that create the ability to use a time stamp to search for a given entry, to delete a specify entry from the history and to clear the entire history of a system component, in a way that is similar to the history of a web browser.

Navigation through all these pages is very simple, there are only six distinct HTML files, in a situation where all the features are selected, and JavaScript is used to generate much of the elements that are seen, such as tables and buttons, by using AJAX. This gives the interface dynamic, providing more usability than it would have if the page was static.

### 3.2.3 Platform Interface

The goal of the subsystems described earlier is to provide a way to monitor and manage platforms. These platforms are, in what concerns this project, computer systems that monitor and control the action of other systems. Since there is a broad range of systems to be monitored the systems can vary widely between them and thus it is better to let the application developer implement the monitoring application, or platform application, autonomously and just provide a way for this application to communicate with the rest of the system. The best way to do so is to provide the

application developer with a platform interface that provides a fixed way of communication.

The platform interface is implemented as a *.zip* file that, when unzipped, deploys the code to be run in the platform to make the interface between the platform and the server.

Even though this code does not vary much, being the only difference the operating system, a feature diagram, depicted on figure 3.10, was designed for the platform. This feature diagram describes the interface as the only concrete feature and as a part of a bigger system, the platform.

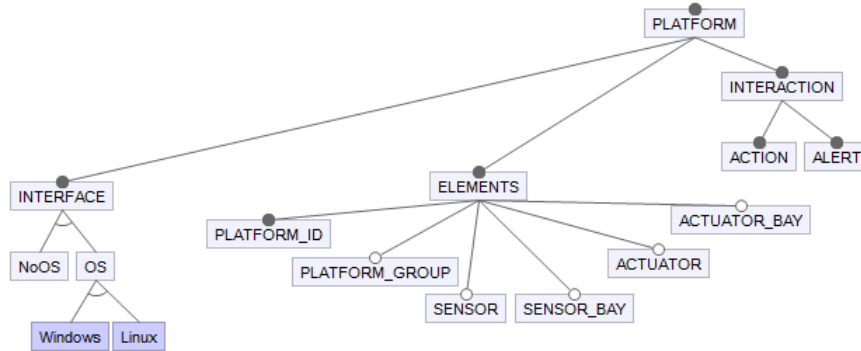


Figure 3.10: Platform feature diagram

In this diagram the importance of abstract features is heavily noticed, as it uses abstract features to characterize a platform and describe where it can and cannot vary. So a platform has an *INTERFACE* feature, that defines the way the platform communicates with the server, it has an *ELEMENTS* feature that defines the elements that a platform can have. Each platform has an *ID*, can be inserted in a group, can have sensors and/or actuators that can also be grouped into bays, and a it also has an *INTERACTION* feature, that defines that the way the platform interacts with the rest of the system is done by using actions and alerts.

The only concrete part of this feature diagram specifies the operating system that is running in the platform and implements the interface process that the platform will run in order to be able to communicate with the server. This interface process communicates with the monitoring process running in the platform and makes the interface between the monitoring process and the server. The interface process communicates with the server by using a socket to receive data from the server, corresponding to actions, and a HTTP client to send requests to the server, corresponding to alerts.

The interface implements two distinct threads, one that receives data from the platform ap-

plication and sends it to the server and another that receives data from the server and sends it to the platform application. Their operation consists on having a socket server running on each thread, one listening to requests from the server, redirecting the data via a socket client to the platform application upon receiving a connection, and the other listening to requests from the platform application and sending the data to the server by sending a HTTP request to the server with the alert details. Thus establishing a bridge between both subsystems.

Details concerning the ports of these sockets and the URL the alert web service, to which the system will send alerts, are defined in a text file, providing the ability for the developer to edit it, that is read by the platform interface that creates all these mechanisms matching the developer preferences.

Even though there is a platform interface between the server and the platform, to guarantee the correct communication and understanding of alert and actions between both components, the developer must respect the constraints imposed by the rest of the system when implementing the platform application. The first constraint is that communication with the platform interface is done using sockets. The second is that in order for the alerts emitted by the platform are understood by the user and the actions taken by the user are understood by the platform, alert codes emitted by the platform must match the ones in the alert table on the database, and actions received the by the platform must also have the same meaning to the platform than the meaning they have for the user. The third constraint is that platform element identification must match the database, or there will be a server error when an alert is emitted on an element that does not exist in the database. If all these constraints are respected, the system shall function as desired.

## Chapter 4

# Instantiation

Chapters 2 and 3 expose the way the domain is analysed and implemented as a way to develop the SPL that embodies this framework. In this chapter, the work flow used to implement the SPL will be presented, presenting the user with all the necessary steps to implement a specific application.

SPL instantiation is also a part of FOSD, consisting on the two last phases of this development process:

1. *Requirement Analysis*, where a specific problem is analysed and where its constraints and requirements are identified, resulting in the selection of the features, from the SPL that better suit a solution for the problem;
2. *Software Generation*, where a software composition tool joins the selected features, previously mapped to code developed to suit the software composition tool, and automatically generates the application code;

Even though the generality of the generated code can be considered final, most of the cases require a fine tuning of some parts of the code so that it completely suits the problem at hand. This code editing will mostly be done in the user interface code, since it is the part that can vary the most.

In this chapter both Requirement Analysis and Software Generation are exemplified by giving an example on a possible problem statement:

*An application must be developed to monitor the operation of a set of remote offshore power generation platforms. The platforms are placed in high seas and have a Web connection, via GSM, for communication. These platforms have a set of sensors that will send information to the user, who must also be able to perform actions on the platforms, such as change the state of the platform. Logs must be kept from actions performed as well as received alerts and user session logging is also required. Since there may be an increase in the number of platforms in the future, and there is the possibility of removing platforms due to malfunction, the user must also be able to add, remove and edit platforms in the system, as well as their groups. All the platforms run the Linux operating system.*

## **4.1 Requirement Analysis**

Before starting the development process of the specific application the developer must have Eclipse properly configured with Java and Java EE development tools, as well as the Feature IDE plug-in. A connection between Eclipse and MySQL server is advisable, but not strictly necessary, since the database can be easily deployed directly in the MySQL server.

To begin the development process, the Feature IDE project containing the SPL must be imported to the Eclipse Workspace from the file system. After the project is correctly imported, the development process can be started.

By analysing the problem statement the following elements can be identified on the system as a whole:

1. Platform Groups
2. Platforms
3. Sensors
4. Actuators

Furthermore, the platforms run Linux and communicate remotely via the Web, meaning that a way must be created to communicate with the platforms, resulting on the need of a Web service architecture to communicate with the remote platforms.

There are also other features that must be present for management and data analysis, these features will be implemented and accessible via an interface to the user:

1. Platform Groups management
2. Platforms management
3. Alert reception and logging
4. Action taking and logging
5. User session log

After all the system characteristics are identified, the application developer must create a *configuration* on the Feature IDE project of the SPL and select the desired features, as shown on figure 4.1. To ease the feature selection and to make sure that there is consistency between user interface, server and platforms, cross-tree constraints were defined between the three specialization levels of the SPL.

These constraints were defined in a way that makes the feature selection easy. The developer must only select the desired features in the platform and user interface levels of the SPL. Server features are selected automatically depending on the features selected on the user interface and platform, assuring consistency between all the subsystems. Constraints were defined this way because the server is the central part of the system and establishes a proxy between user interface and the platform. Therefore, server features will match features selected in both the platform and the user interface, creating a connection between them. It makes no sense to specify server features alone when they depend on other selected features, this would be a source for inconsistency in the system and it could create communication problems between platforms and the user interface.

Therefore, in this case, the developer starts by selecting platform features for this specific system (*Linux, Platform, Platform Group, Sensor, Actuator*), and then proceeds to select the intended user interface features, stated in the problem statement (*Platform and Platform Group management, Action and Alert logging, User session logging*).

Even though this process is simple, it is not mandatory that is done in this order, the developer may select server features directly, but by doing this he will need to select platform and user



interface features as well. Since the constraints are only defined to be applied from the user interface and platform levels to the the server level and not the other way around, by doing this the developer will take more time and need much more concentration, because he needs to ensure consistency between levels, being highly prone to errors, that is why this method is not advised when a full system is being developed.

This process ends when the created configuration is selected as current and the code for the application is generated.

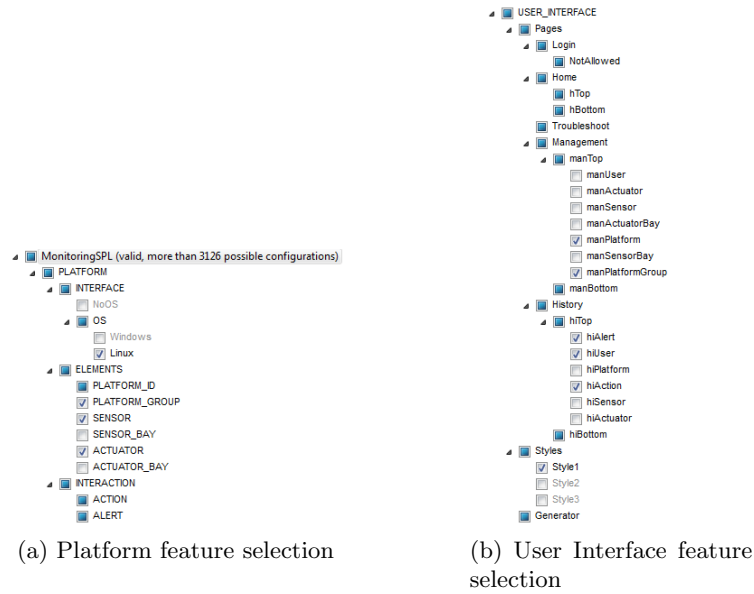


Figure 4.1: Configuration for the case at hand.

## 4.2 Software Generation

Since Feature IDE is a very automated tool, the software generation process will be transparent to the developer<sup>1</sup>. This process which results in a set of different Java packages that are intended for different ends. Even though the final code is generated after requirement analysis, the instantiation process is not finalized, there are a few tasks that must be performed so that the system is usable.

<sup>1</sup>Refer to chapter 3 for automatic software generation details

### 4.2.1 Web service deployment

The Web service code is contained in the *mon.spl.srv*, *mon.spl.biz*, *mon.spl.dao* and *mon.spl.model* packages. The developer must change Eclipse to the *Java EE* perspective and create a new *Dynamic Web Project*, generating the *web.xml* deployment descriptor, as depicted on figure 4.2. This project will be used to create the Web services deploy, resulting in a *.war* file that will then be moved to the server. The *web.xml* file is the descriptor where Web services related aspects, such as URL, Servlet name and the Web services entry package are defined.

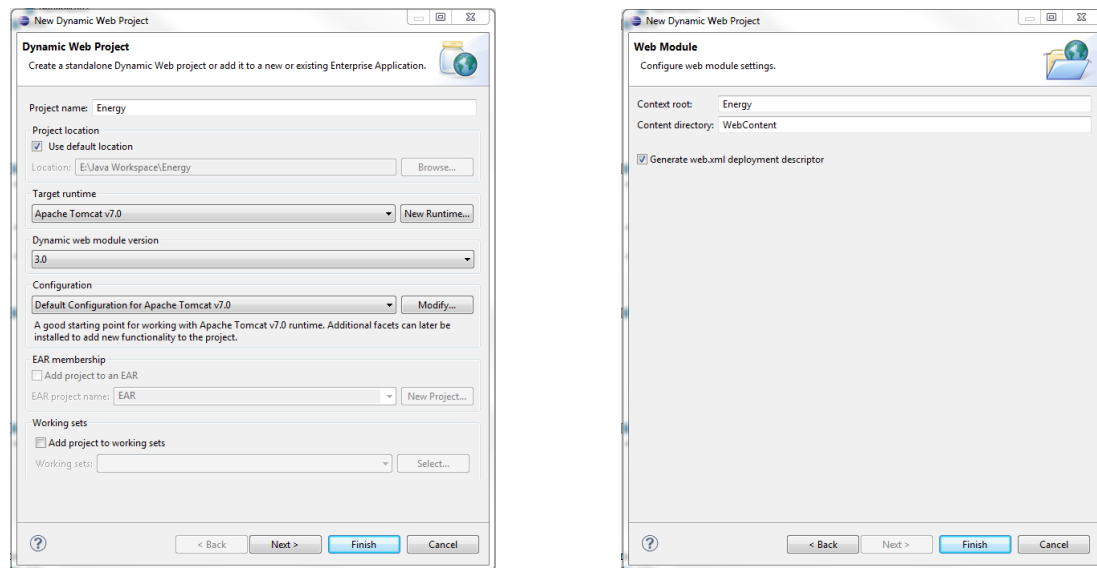


Figure 4.2: Dynamic Web project creation

After this is done, there are Java libraries that must be added to the project build path<sup>2</sup>, these libraries are the *JAX-RS* libraries and the *MySQL Java Connector*. Without these libraries the Web service will not run when deployed in the server.

After this the Web service project is created, the user must now copy the Web service packages from the *src/CONFIGNAME* folder in the Feature IDE project to the *src* folder in the Web service project.

Right now, the web service is created, the only thing left to do is to register Jersey as the

---

<sup>2</sup>Copy the JARs to the *WEB-INF/lib* folder

servlet dispatcher for REST requests. Doing so is fairly easy, it only requires the editing of the *web.xml* deployment descriptor to reflect the project specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>webservice.displayname</display-name>
  <servlet>
    <servlet-name>Servlet Name</servlet-name>
    <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
      <param-name>
        com.sun.jersey.config.property.packages
      </param-name>
      <param-value>mon.spl.srv</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet Name</servlet-name>
    <url-pattern>/baseURL/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The details contained in this deployment descriptor reflect the Web service parameters and its configuration. The package *com.sun.jersey.config.property.packages* defines where Jersey will search for the Web service classes, the *mon.spl.srv* specifies the entry point for the web services, any other package would render the deployed Web services useless, since it does not define any URI. In the servlet mapping, there is the definition of a URL pattern. This pattern defines part of the base URL where the application will be placed (e.g. *http://localhost:8080/webservice/rest/\**).

Inside the *mon.spl.dao* package there is a file called *DBconnection.properties*. This file specifies the connection properties to the MySQL server and must be edited so that the Web server *knows* where to connect, when it needs a database connection, and which user is connecting. Since

a MySQL server connection is based on user access and therefore the Web service must be an authorized user in the database server. After editing this file must be moved from the *mon.spl.dao* package to the *src*<sup>3</sup> directory of the project, since that is the place where the Web service will look for the database address as well as the user credentials.

After all these steps, the Web service is configured and ready to be deployed. To deploy the Web service the developer must export the Web service project as a *.war* file, and after the *.war* file is created move it to the server folder where it will be run from, in Apache Tomcat the folder is *TOMCATINSTALLFOLDER/webapps*<sup>4</sup>.

Even though the Web service is configured and ready to run, any request sent to it before the database is deployed will result in *HTTP error 500*. The developer must first deploy the database before testing the web service operation.

#### 4.2.2 User Interface and database deployment

The user interface and database deployment are specified in the same subsection because both result in the generation of scripts by an executable file. Since the generator is the same, generating both Web pages and MySQL scripts, it makes sense to explain both elements at the same time.

Generating HTML and MySQL code is a simple process, the user must only run the *mon.spl.client*<sup>5</sup> package as a Java application, and a set of HTML, JavaScript and CSS pages will be created, as well as a MySQL script containing the definition of all the database. As it is expected, these scripts reflect the features selected during the requirement analysis phase.

##### Database deployment

There are two ways to deploy the database, one is to deploy it directly into the server by running the generated MySQL script on MySQL administrative console or by pasting the code in the MySQL workbench and then commit it to the database server. The second way is by executing the generated Structured Query Language (SQL) file in Eclipse, which requires that eclipse is connected to the database server. Both methods require the configuration of server connection

---

<sup>3</sup>This is the root directory of the project

<sup>4</sup>The process should be just as easy when using Glassfish

<sup>5</sup>Right-click on the package and select *run as Java application*

details, however MySQL workbench does not need a MySQL connector to connect to the database server, while Eclipse does.

The user should start by creating the *monitor* database in the MySQL server. After that, on Eclipse, the user must change the Eclipse view to Database Development, then in the solution explorer create a new database connection choosing MySQL as the *Connection Profile Type* and defining a name for the new database connection. After this, there is the need to define the database to connect to, its address and the user credentials for accessing the database server, as well as the MySQL JDBC Driver<sup>6</sup>, as shown on figure 4.3. To add the driver to Eclipse, click on the *New Driver Definition* button and define the driver path and the connection parameters.

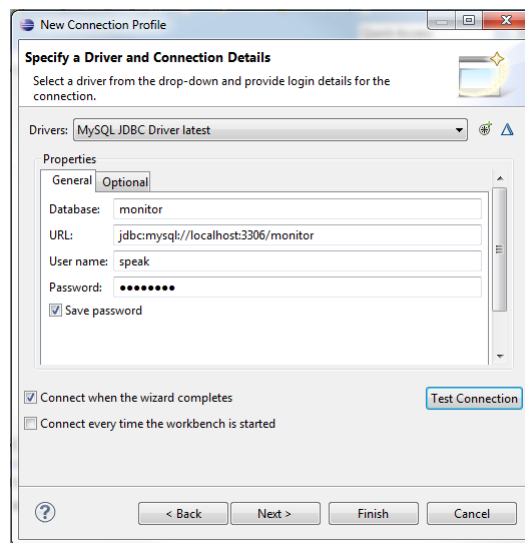


Figure 4.3: New Connection Profile Wizard

Now that the Database connection between Eclipse and the MySQL server is defined, return to the Java EE perspective and execute SQL files by right clicking on the SQL file and choosing that option. A wizard will appear to select the profile for the file. Specify the database server type, the connection profile name and the database name and click *OK*. This will start the execution which will execute the SQL statements in the file, which define the database, in the MySQL server, deploying the database.

Figure 4.4 provides an example of the result of the execution of a SQL script, implementing the *monitor* database in the database server

---

<sup>6</sup>The latest version of this driver can be downloaded from the Oracle website

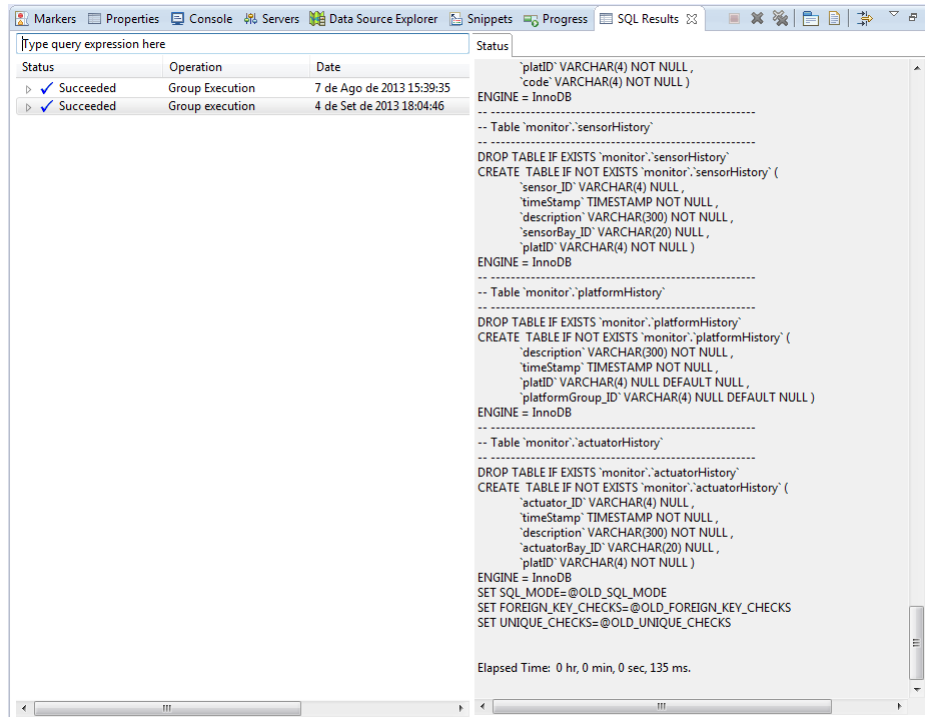


Figure 4.4: SQL execution result

## Web page deployment

After the Web pages have been created there must be some editing done to them. The developer may want to keep the look and the pages as they are generated, but must at least change the URLs used to communicate with the Web services. Otherwise, such communication will be impossible. *Appendix C* defines which lines, in the generated files, must be changed to achieve this goal.

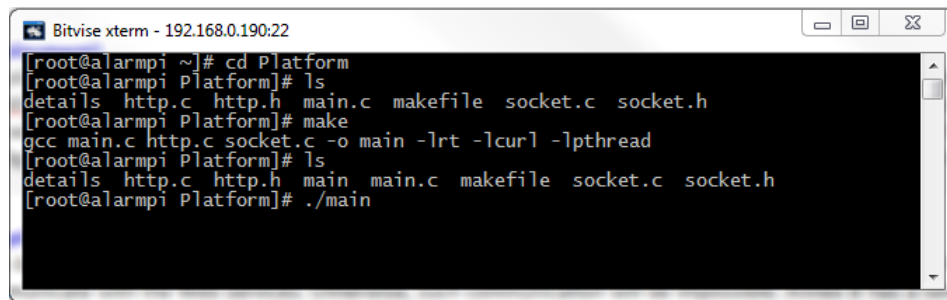
After the pages are edited, they can be deployed directly into a Web server by just putting them into a directory and moving it to the directory where the server runs Web pages from.

### 4.2.3 Platform Interface deployment

As it has been stated previously, different systems, mean different platform types, with different sensors and architectures. Therefore, to give the developer more flexibility when developing a new monitoring application only an interface with the server is implemented and there are a few constraints, imposed by the framework, that the developer must follow.

This interface between the server and the platform is deployed as a *.zip* file, in the *mon.spl.platform*

package. This file must be decompressed and compiled for the right operating system and architecture with a C compiler. For example if the used platforms are controlled by an Atmel AVR32 embedded system, the files must be compiled using the GNU C Compiler (GCC) compiler for the AVR32 architecture and linking the compilation with the *pThread* and *libCurl* libraries<sup>7</sup>. After compiling the developer will get the executable file for the interface and a text file called details, figure 4.5 demonstrates this process. This text file contains four fields, one per line that must be edited in order to allow communication between server and platform. The platform interface reads this file to define which ports and addresses to use.



```

[root@alarmpi ~]# cd Platform
[root@alarmpi Platform]# ls
details http.c http.h main.c makefile socket.c socket.h
[root@alarmpi Platform]# make
gcc main.c http.c socket.c -o main -lrt -lcurl -lpthread
[root@alarmpi Platform]# ls
details http.c http.h main main.c makefile socket.c socket.h
[root@alarmpi Platform]# ./main

```

Figure 4.5: Platform interface compiling and running

The first line is the address of the Web service that receives and processes alerts<sup>8</sup> and the second line defines the platform port that receives connections from the server (Actions). The third line defines the port used to receive alerts from the monitoring process in the platform and the fourth line defines the port used to pass data received from the interface to the controlling process.

Figure 4.6 provides an example of the contents of the details file, before and after editing.

<sup>7</sup>A makefile is provided to perform this task

<sup>8</sup>e.g. <http://192.168.0.198:8080/full.cfg/rest/alert/current/new>

```

1 Line1 - Web Service Address
2 Line2 - Server Port 1
3 Line3 - Server Port 2
4 Line4 - Client Port
5
6
7 EDIT THIS FILE (ERASE THIS LINES)
8 EXAMPLE:
9 http://192.168.0.198:8080/de.vogella.jersev.first/rest/alert/current/new
10 8080
11 3000
12 4000
13

```

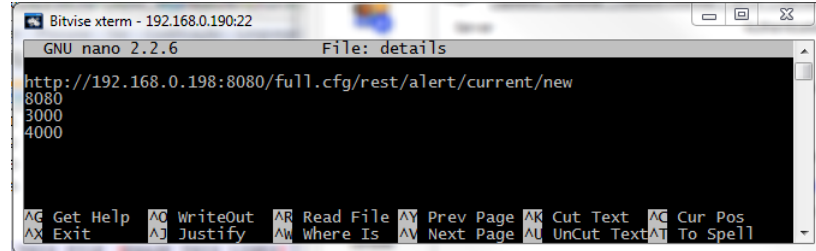


Figure 4.6: The details file before and after being edited to reflect the platform communication ports and the Web service URI for new alerts

The deployed process just implements an interface between the Web services and the platform. The developer must implement his own controller application for the platform which will pass data to this interface via sockets. There must also be a local copy of the alert and action tables from the database in each platform<sup>9</sup> so that both the server and the platform may *understand* each other, since communication is made through alert and action codes, where each code has a different meaning.

---

<sup>9</sup>The developer may choose which format to use for this



## Chapter 5

# Results

The best way to prove the correct operation of the framework, is to instantiate it and to carry on some tests on the resulting applications.

Two distinct test cases were planned:

1. A basic test case, where an application comprising only the mandatory features, representing the less complex application, is implemented;
2. A full test case, comprising all the features, except mutually exclusive features

By testing the most complex application after testing the most simple application, both implemented by instances of the framework, the ability of the framework to generate applications from the simplest case to the most complex one. Therefore, if both applications function correctly, it can be expected that the framework can be used to implement every application with a complexity between the two test cases, thus being suitable for all the applications of the domain.

These test cases will only simulate applications, since there were no specific systems on which this framework could be tested. However, this does not impose a problem since if generic information can be managed and transmitted from point to point using generic test cases, there is no reason why it should fail in a specific system when instantiated to it and respected all the constraints imposed by the framework.

## 5.1 Test Systems Specifications

The code for both applications will run in the same hardware. Since there was not a specific case to test on, platforms were simulated by out of the shelf computer systems.

### 5.1.1 Servers

To prove that the system can run even if the Web server and the database server are in different machines, two distinct machines were used.

- *Web server* - Quad core Intel Core-i7 3610QM Processor @ 2.3 GHz, 6GB DDR3 1600 MHz SDRAM, 500 GB 7200 Rpm hard disk, 10/100/1000 Base T Ethernet Adapter. 64 bit Windows 7 Professional and Apache Tomcat v7.0 as Web server and Java servlet container.
- *Database server* - AMD Sempron 3400+ Processor @ 1.6 GHz, 2GB DDR2 400 MHz SDRAM, 100 GB 7200 Rpm hard disk, 10/1000/1000 Ethernet Adapter. 64 bit Ubuntu Server Linux with Linux kernel 3.2.0-51 and MySQL Server version 5.5.

### 5.1.2 Client

The client machine, where the user interface was tested, was the same as the Web server machine. Even though the machine is the same, the system will also work if all the systems are distributed, since the address to where the client directs its requests can be other than *localhost*.

Since there are Web browsers known to not yet support fully HTML5 and JavaScript, the aim of this test is to check browser compatibility with the user interface. Even though the user interface is basic without many visual elements, there can be some compatibility issues. To achieve this the system was tested in the three most used Web browsers *Google Chrome 29.0.1547.76 m*, *Mozilla Firefox 23.0.1* and *Microsoft Internet Explorer 10.0.9200.16686*, corresponding to the latest versions available of these Web browsers as of testing date.

### 5.1.3 Platforms

Two different platforms were used, one simulating a low resource system, even tough not the lower resource system available, and another simulating a system with higher resources.

- *Platform 1* - Raspberry Pi Model B: Broadcom BCM2835 SoC (700 MHZ ARM11 with ARMv6 Instruction Set), 512 MB of RAM, 2 GB Class 4 SD Card non volatile storage, 10/100 Ethernet Adapter. Arch Linux v2011.11 for ARMv6, Linux kernel 3.6.11-12-ARCH+.
- *Platform 2* - Dual core Intel Core 2 Duo T2330 Processor @ 1,8 GHz, 2GB DDR2-667 SDRAM, 80 GB SATA 5400 Rpm, 10/100/1000 Ethernet Adapter. Linux Mint 15 "Olivia", Linux kernel 3.8.

The way all these elements are connected between them is the same for both test cases and is defined by the framework architecture and is demonstrated visually on figure 5.1:

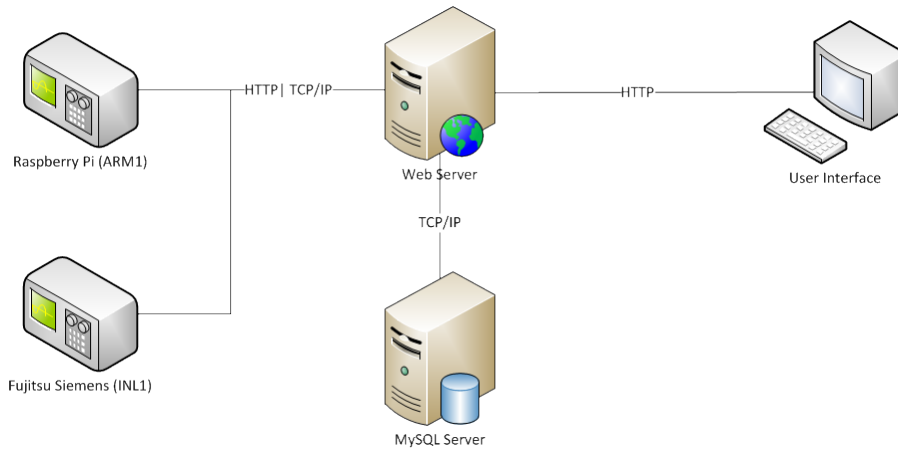


Figure 5.1: Test case connection diagram

## 5.2 Basic Application

The first test to be performed was to instantiate the framework to the most simple configuration possible. Only mandatory features and exclusive features that depend on a mandatory upper feature (e.g. the operating system of the platform) are selected in the SPL, thus simulating the simplest application that can be developed.

This application only has user authentication ability and troubleshooting ability, thus what will be tested is the user authentication and the ability to send data correctly from the user interface to the platforms. Since both test platforms only simulate real platforms, the data sent will also be simulated, not corresponding to any specific case.

There are ten distinct alerts, ranging from *al1* to *al10* and ten distinct actions that can be performed to answer the alerts or that can just be sent to a platform.

The test results shown for this application will demonstrate the basic user interface and the reception of actions by the platform. These tests were performed using the Mozilla Firefox Web browser with the Firebug plug-in, which allows an efficient debugging of Web pages, allowing the discovery of potential errors that may appear.



Figure 5.2: Login page

Figures 5.2 and 5.3 show the login and the basic homepage generated. As it can be seen, the interface is simple, because it is meant to be changed by the application developer, but usable.

Figure 5.4 demonstrates the warning shown when there are alerts that need the user attention, redirecting the user to the troubleshooting page.

The troubleshooting page with the received alerts can be seen on figure 5.5 and the action sending can be seen on figure 5.6.

To conclude this test results, the console prompt of the Raspberry Pi (ARM1) is shown. It demonstrates a server exception code, received as a response, due to sending of an alert from the platform corresponding to a sensor that does not exist in the database, it also shows the received actions. As it can be seen, the platforms only receive the action code.

For test purposes, each platform ran the platform interface and two processes that implement

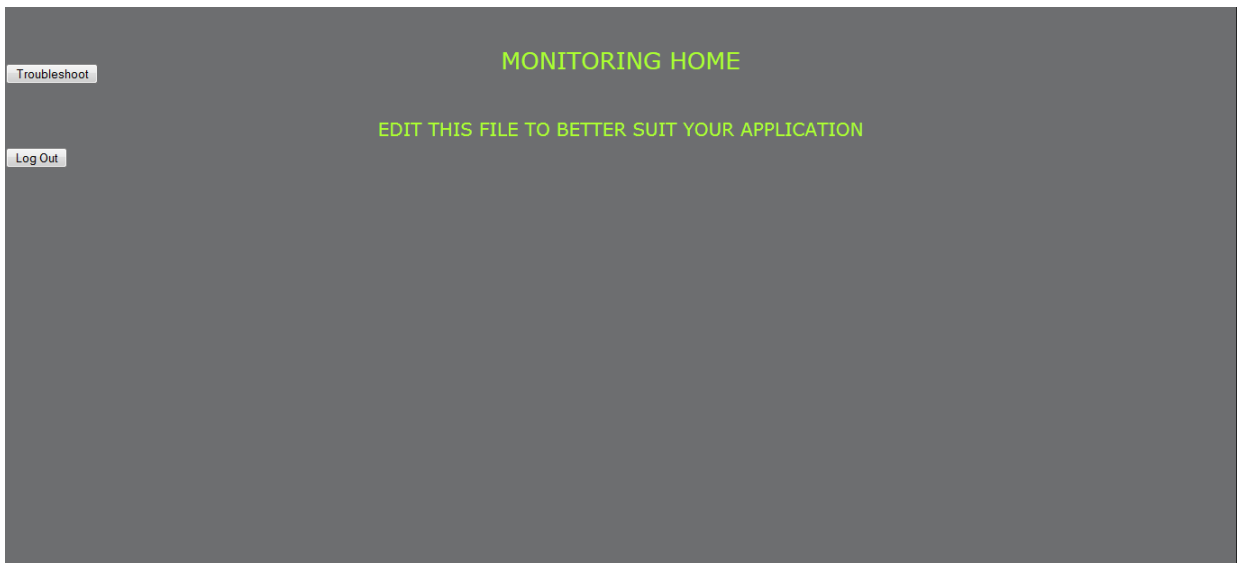


Figure 5.3: Home page for the basic application

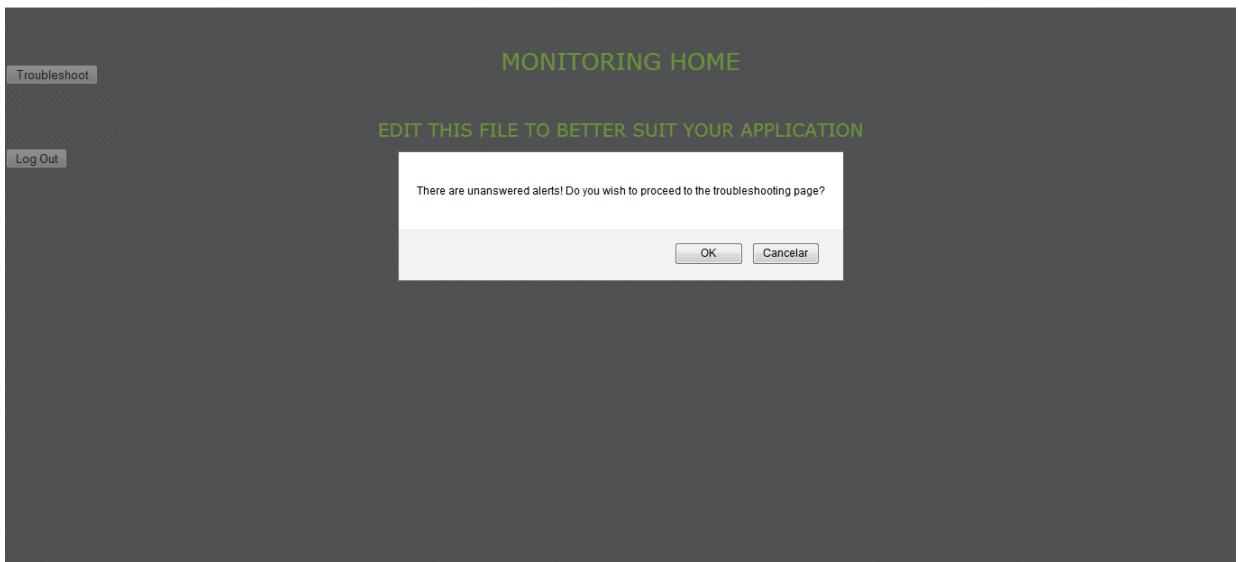


Figure 5.4: Unanswered alerts warning

a Web socket server to receive actions and a Web socket client to send actions to the server. Both processes communicate with the platform interface, thus proving its correct functioning. Figure 5.8 shows the details file used in this test. This file is used by the platform interface to define the web service address and the ports to be used.

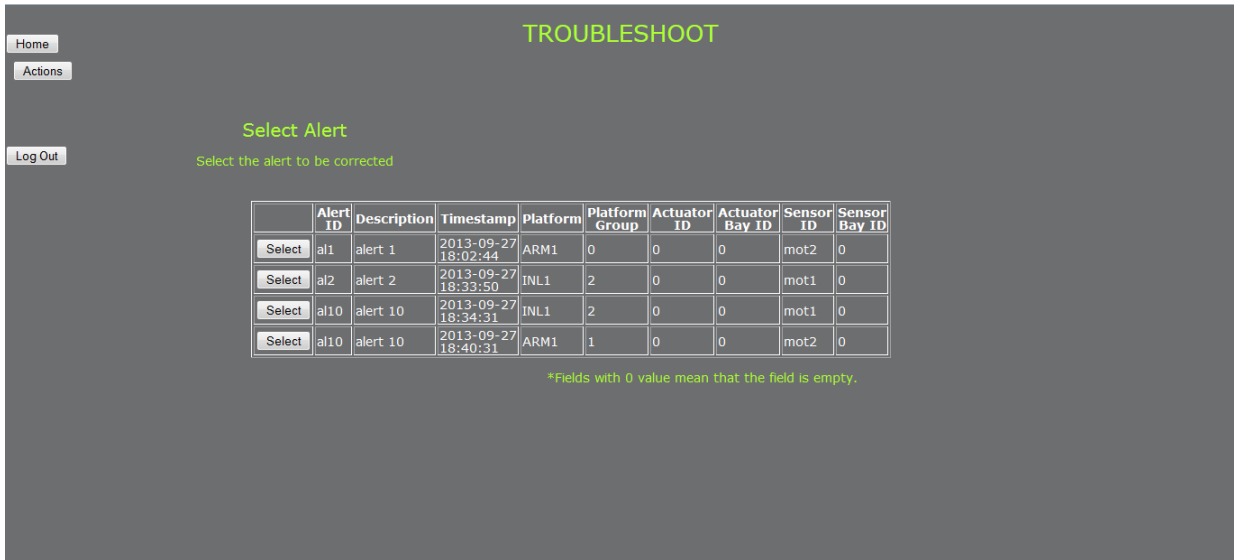


Figure 5.5: Troubleshooting page - Received alerts

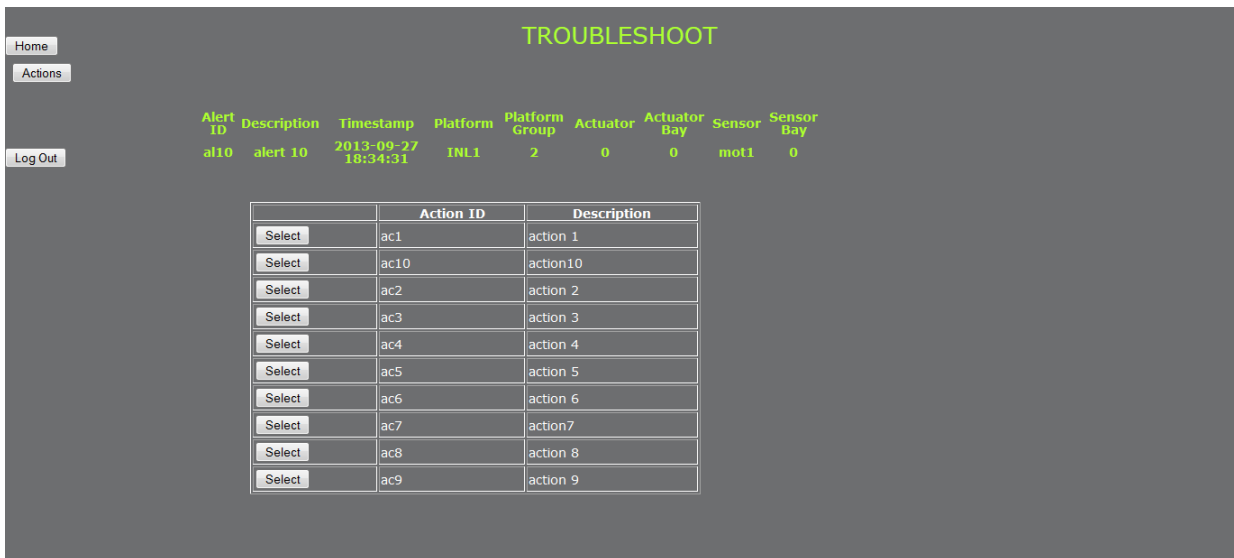


Figure 5.6: Troubleshooting page - Received alerts

## 5.3 Full Application

This test was performed by instantiating the SPL to the most complex application possible. All features are selected, thus simulating the most complex application that can be implemented using the SPL.



the same.



Figure 5.9: Home page for the full application

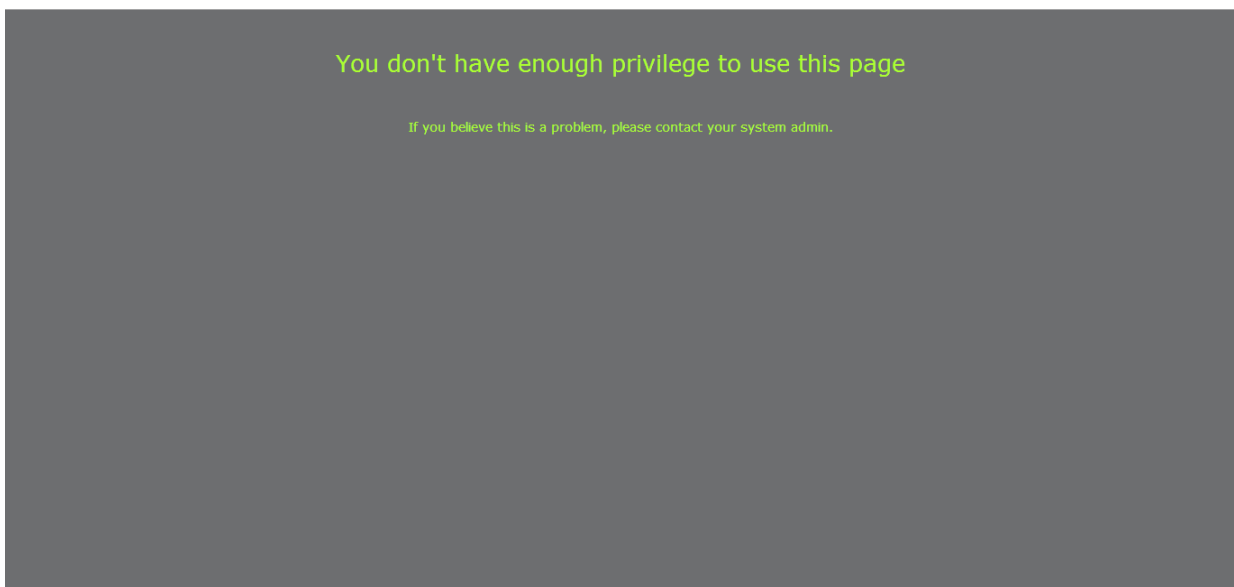


Figure 5.10: No privilege page

Figure 5.9 shows the home page of the user interface in the full application. Comparing it with the home page for the basic application it can be noticed that there are two new buttons, one to the history page and another to the management page.



Privilege based access was also tested, the no privilege page that appears when a user doesn't have enough access privilege is shown on figure 5.10. The management page requires a privilege of 4 or higher to grant access, a user with privilege level 2 tried to access it and it was barred.



Figure 5.11: Actuator Bay management page

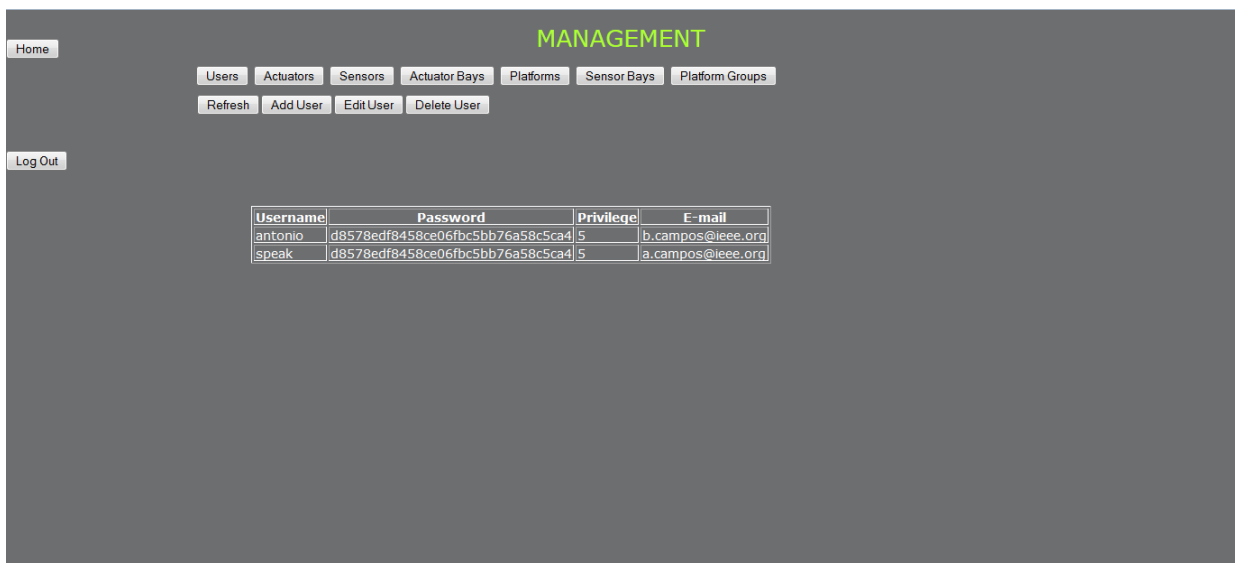


Figure 5.12: User management page

Management was tested by adding the several system components, editing them and removing it. MySQL workbench was used to check if the changed were correctly committed to the database.

The system behaved as it was expected in this part.

Figures 5.11 and 5.12 show the actuator bay and the user management pages, demonstrating the management actions that are possible to be taken on system components.

	Username	Time Stamp	Description
<input type="checkbox"/>	antonio	2013-08-26 16:25:14	User antonio successfully asked for privilege at 2013-08-26 16:25:14.431
<input type="checkbox"/>	antonio	2013-08-26 16:07:11	User antonio successfully asked for privilege at 2013-08-26 16:07:11.82
<input type="checkbox"/>	antonio	2013-08-26 16:07:12	User antonio successfully asked for privilege at 2013-08-26 16:07:12.705
<input type="checkbox"/>	antonio	2013-08-26 16:25:08	User antonio successfully asked for privilege at 2013-08-26 16:25:08.243
<input type="checkbox"/>	antonio	2013-08-26 16:25:09	User antonio successfully asked for privilege at 2013-08-26 16:25:09.047
<input type="checkbox"/>	antonio	2013-08-26 16:25:49	User antonio successfully asked for privilege at 2013-08-26 16:25:49.132
<input type="checkbox"/>	antonio	2013-08-26 16:26:41	User antonio successfully asked for privilege at 2013-08-26 16:26:41.922
<input type="checkbox"/>	antonio	2013-08-26 16:26:51	User antonio successfully asked for privilege at 2013-08-26 16:26:51.688

Figure 5.13: User History page

History was also checked, the system registers every event regarding user authentication, system management, alert and actions. It adds a correct time stamp to give a temporal measurement of the event, as it can be seen on figure 5.13. This part of the system also behaved as expected.

## 5.4 Test conclusions discussion

After all the tests were performed, some bugs were identified and corrected. The main difference on both test cases is the number of Web services and user interface features, the basic application only allows troubleshooting and the full application allows troubleshooting, system management and history enquiring. For both applications the platform interface is the same, since it is a basic interface with a very specific task that has no variation and both systems ran Linux.

The implemented applications perform as expected, redirecting the data from the platforms to the user interface and sending actions from the user interface with the correct platform.

Platform and user administration perform as expected making it possible for the manipulation of information in the database from the user interface. There are a few small bugs while using

Internet Explorer, since it does not yet implement full JavaScript support. However, this is not a major concern since older versions of Internet Explorer have known compatibility issues when dealing with JavaScript, Internet Explorer 10 represents only a small market share in Web Browser usage and Microsoft is working towards full HTML5 and JavaScript support on oncoming versions of Internet Explorer. Even though there were issues while running the user interface on Internet Explorer, the user interface ran as expected in both Google Chrome and Mozilla Firefox.

Platforms send and receive correct data without any issue, both in the less resourceful system and in the more powerful system. However, the transaction of large quantities of data, from a range of different sources, was not tested, but since it depends not only on the implemented software but in the hardware infrastructure as well the system should work well with large data transactions if the hardware infrastructure is capable of handle such transactions.

When instantiating this platform to real systems such as a offshore platform or a domotics system in a house, the SPL is expected to perform well in both cases, with data transmitted correctly from the monitored system to the user. Depending on the management and history features selected both cases can be both as complex or as simple. However, the offshore generation platform is expected to have less customization performed after instantiation than the domotics system. This will happen because the offshore monitoring application is intended to be used by a technician and an offshore platform has less sensors than a house equipped with a domotics system. The domotics system is expected to have a larger part of its code edited by the application developer because there can be a great number sensors and actuators in this type of systems and, since the application is intended to be used by a regular user, without any skills in electronics, the user interface needs to be more intuitive. The SPL is fit to be used in both systems, being able to implement a communications daemon between user interface and monitored platforms, and also implement a user interface with privilege based access and ability to manage all the aspects related to the systems as well as keeping a log from all the actions taken upon the systems.

In what this framework is concerned, both systems can be similar. Both impose the need for communication between user and monitored platform as well as user authentication. If the management requirements and history requirements are alike, then what will differentiate both these application is the data transactions from platform to user interface, the amount of data transacted and the editing done after the instantiation process.

## Chapter 6

# Conclusion and Future Work

### 6.1 Conclusion

Monitoring is a fundamental aspect in many systems that need control. However the development of monitoring applications can be a resource intensive task, consuming many resources that can be applied in other projects or that may lead to a the loss the competitive edge in the market.

The development of a framework to aid in the development monitoring applications can reduce the resources used while developing monitoring applications, improving productivity and final product quality while decreasing the costs associated with development and the time-to-market of a product.

The SPL provides the application developer with a good level of abstraction, where the developer only has the need to know which are the features needed for each subsystem, hiding code complexities and the technologies used by each subsystem.

Using a SPL approach to implement a framework, is simple. SPLs provides the application developer with a good level of abstraction, hiding code complexities and requiring that the developer only defines which features are needed to answer a specific problem.

By analysing a domain, one can easily identify the main features within it, and by employing a top-down approach sub features can also be identified. This process provides an easy way to analyse and define which features are common between the systems of a given domain and where the systems of a domain vary. The usage of feature diagrams to describe a domain and its features

greatly improves the understanding of the variations within a domain. A SPL provides a good level of abstraction to an application developer, since he only needs to identify the features that a specific system must have from a set of previously defined features related to each other.

Feature IDE provides a tool for the development of SPLs that allows the mapping of code, in several different languages, to features. This simplifies the SPL development process, since one can think in features as blocks of code that can relate or extend other blocks, creating a usable final product with a great level of abstraction and automatic code generation.

In the domain of this project, the usage of Feature IDE to design and implement a SPL that represents the remote monitoring domain made possible the development of a solution that implements all the subsystems identified during the domain analysis phase, even though they are implemented in different programming languages.

The implemented SPL provides a great amount of automatic code generation, implementing code for all the subsystems based on the selected features. It implements a fully usable monitoring infrastructure, based on the architecture defined in chapter 2, composed by a user interface, that provides visual feedback to the user, a server, composed by a database and a set of web services that embody a communication daemon. A platform interface assures correct communication between a platform and the server, and also gives the developer the freedom to implement the platform application in the way that is best for the problem at hand.

Even though the SPL can generate fully usable systems, when instantiated each application has the need to implement custom code to address its most crucial aspects. Also, after code generation there is the need to configure the system components<sup>1</sup> in order to assure correct communication between all the subsystems and to assure that data is transacted correctly.

One can say that the usage of SPLs as a methodology for the management of the commonalities and variabilities of a framework simplifies the framework development process. Feature IDE is an adequate tool for this end providing a way to map features into blocks of code, easing the development process even more and providing a way to easily instantiate the SPL to a specific system, by selecting a few check boxes, abstracting the user from automatic code generation. By joining both these main factors, the implemented framework fulfils the main project objectives and is easy to use, with the greatest amount of time during instantiation being the system configuration

---

<sup>1</sup>Web server, database server, platform interface and user interface

instead of the code generation, proving that it greatly reduces the amount of resources used to implement a monitoring application. However, this framework still has much to evolve and the best way to do it is to use it in several different problems within the monitoring domain, this will identify the main problems and create the need to improve code and create new features in order to minimize user intervention and create a more solid solution that can be used in a broader domain.

## 6.2 Future Work

Future work should focus on the evolution of the framework to be able to be used as a solid development tool that can be used reliably by development teams. Since this is the earliest usable version of this framework, there is still a lot to do to make it a more complete tool that can be used as a development asset or product. Even though a structure is created in an autonomous way there were certain aspects, that due to the lack of time, were not implemented or need a better implementation. These aspects will improve the framework and the resulting applications usability by implementing new features.

The following list shows the identified future work as of the finishing of the first version, other missing features may be identified in the future as the framework will evolve through usage.

- *HTTPS connection support* - Encryption of the data transmitted between clients and server.
- *Improvement of the Web service access restrictions* - To only allow authorized users to access the Web services. This can be achieved by sending the session ID in every request and querying the database for its existence. This can also improve the privilege based access.
- *Creation of a deployment wizard* - The goal is to provide the framework user with an interface that allows the rapid configuration and deployment of a new application.
- *Scalability support* - Allowing the development of any application and not caring about the amount of data transferred.
- *Business Intelligence* - Currently, data manipulation in the business layer of the Web services is simple. However, data can be manipulated in a way to better improve monitoring by,

for example, analysing alert data and assigning a severity level to it or even by discovering patterns in the received data. Data Mining concepts may help to meet this end.

- *Improve instantiation automation* - Create scripts in order to make the instantiation process more transparent to the user, reducing the need for source code editing after the final code is generated.
- *Improve the generated user interface* - There is much to improve in the user interface. It can become more intuitive to use and even more appealing visually, with transition effects and tabs instead of windows.

# References

- [1] APEL, S., KASTNER, C., AND LENGAUER, C. Language-independent and automated software composition: The featurehouse experience. *Software Engineering, IEEE Transactions on* 39, 1 (2013), 63–79.
- [2] APEL, S., LEICH, T., AND ROSENMÜLLER, M. Featurec++ feature-oriented and aspect-oriented programming in c++. Tech. rep., Otto-von-Guericke University Magdeburg, 2005.
- [3] APEL, S., AND LENGAUER, C. Superimposition: A language-independent approach to software composition. In *Software Composition* (2008), C. Pautasso and r. Tanter, Eds., vol. 4954 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 20–35.
- [4] BOSCH, J. *Design and Use of Software Architectures*. ACM Press Books. Addison Welsey, 2000.
- [5] CARDOSO, N. *Middleware and Tools to Develop a Video Surveillance System for Security, Control and Comfort (SVSC - Toolkit)*. PhD thesis, University of Minho, 2013.
- [6] CERAMI, E. *Web Services Essentials*, 1st ed. O'Reilly, February 2002.
- [7] CZARNECKI, K., HELSEN, S., AND EISENECKER, U. W. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.
- [8] FIELDING, R. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [9] GROUP, W. W. Web services architecture. Tech. rep., W3C, February 2004. 11.



- [10] HORNBY, A. S. *Oxford Advanced Learner's Dictionary of Current English*, 5th edition ed. Oxford university Press, 1995.
- [11] KÄSTNER, C. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, May 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9.
- [12] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of aspectj. Springer-Verlag, pp. 327–353.
- [13] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP* (1997), SpringerVerlag.
- [14] KYO C. KANG, SHOLOM G. COHEN, J. A. H. W. E. N. A. S. P. Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute, 1990.
- [15] NORTHROP, L. Software product line essentials. Tech. rep., Software Engineering Institute, Carnegie Mellon University, 2008.
- [16] OSCAR NIERSTRASZ, T. D. M. Research directions in software composition. *ACM Computer Surveys* 27, 2 (June 1995), 262–264.
- [17] OSCAR NIERSTRASZ, L. D. *Object-oriented Software Composition*. Prentice Hall, 1995, ch. 1, pp. 3 – 28.
- [18] PLEUMANN, J., YADA, O., AND WETTERBERG, E. Antenna: An ant-to-end solution for wireless java, 2013.
- [19] PREHOFER, C. Feature-oriented programming: A fresh look at objects. In *ECOOP* (1997), pp. 419–443.
- [20] ROY, D. Faq what is eclipse, 2013.
- [21] SCHAEFER, I., BETTINI, L., DAMIANI, F., AND TANZARELLA, N. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond* (Berlin, Heidelberg, 2010), SPLC'10, Springer-Verlag, pp. 77–91.

- [22] SCHOLTZ, B. Dao tutorial - the data layer, 2013.
- [23] SOMMERVILLE, I. *Software Engineering*, 9th ed. Addison-Wesley, 2011.
- [24] TEAM, J. D. Jax-rs application, resources and sub-resources, 2013.
- [25] TEAM, J. D. Restful wev services in java, 2013.
- [26] TEAM, J. S. D. Secure random (java platform se 7), 2013.
- [27] TEAM, T. M. D. Munge: A pruposley-simple java preprocessor, 2013.
- [28] THÜM, T., KÄSTNER, C., ERDWEG, S., AND SIEGMUND, N. Abstract features in feature modeling, 2006.
- [29] THÜM, T., KÄSTNER, C., ERDWEG, S., AND SIEGMUND, N. Abstract features in feature modeling. In *Proceedings of the International Software Product Line Conference (SPLC)* (August 2011), IEEE Computer Society, pp. 191–200.
- [30] THOMAS THÜM, CHRISTIAN KÄSTNER, F. B. J. M. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming* (May 2012).
- [31] TIM BERNERS-LEE, R. C. World wide web: Proposal for a hypertext project, November 1989.

# Appendix A

The result of domain analysis is the feature diagram of the SPL. This appendix starts by demonstrating the feature diagram for the implemented SPL. This diagram shows features hierarchically going from top features, that perform bigger tasks, descending to lower features, that perform more specific tasks. A feature description is also provided, describing what is the role of each feature in the SPL.

In the end of this appendix, the constraints imposed by the feature model are specified. The constraints assure the correct functioning of the system after instantiation.

## A.1 Feature Diagram

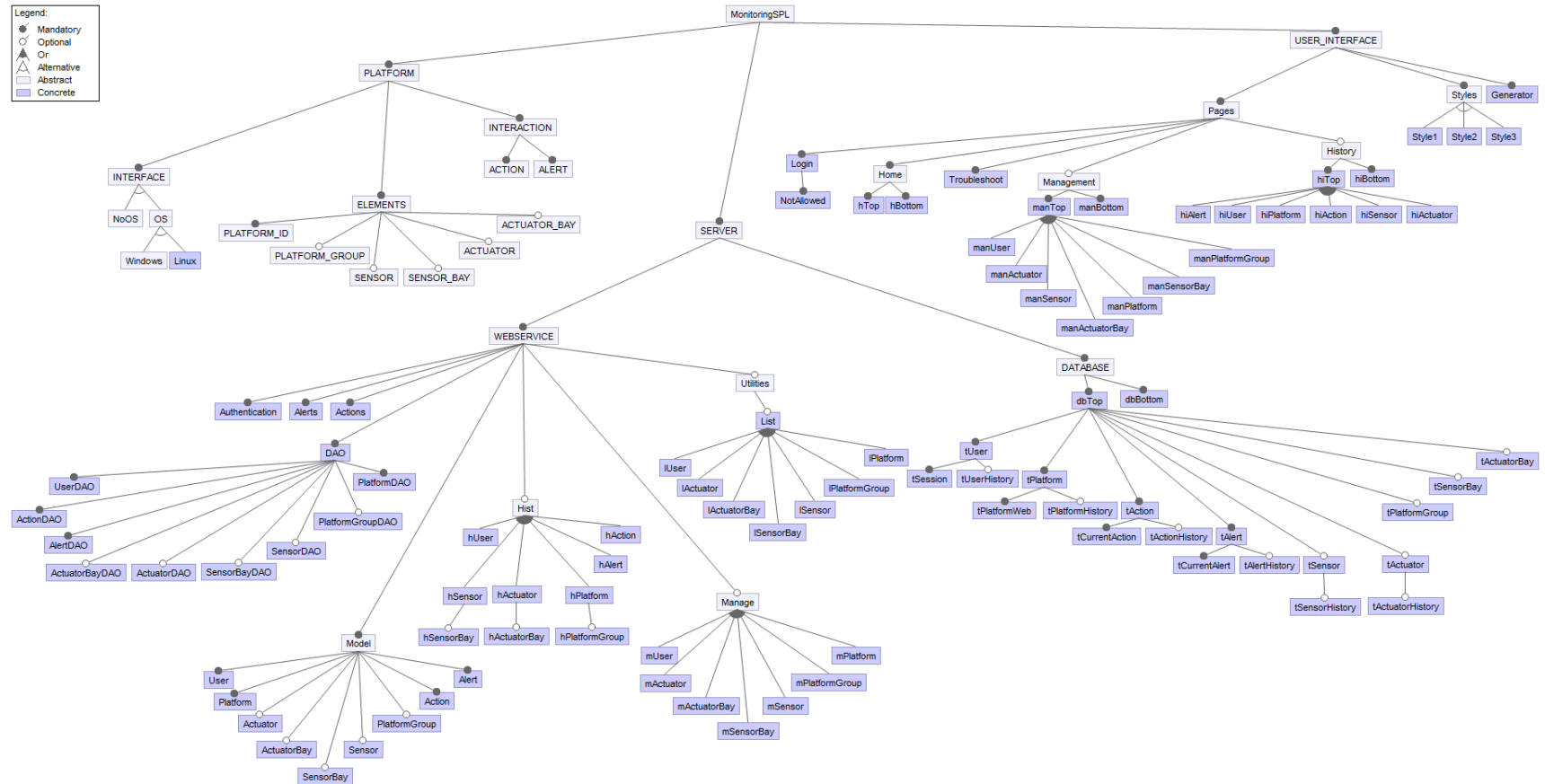


Figure A.1: Feature diagram

## A.2 Feature description

### A.2.1 Platform features

Feature Name	Description	Concrete
<i>INTERFACE</i>	Describes the interface with the rest of the system	No
NoOS	Represents a platform running on a single purpose embedded system	No
<i>OS</i>	Represents a platform running an embedded system with an operating system	No
Windows	Platform running the Windows	Yes
Linux	Platform running Linux	Yes
<i>ELEMENTS</i>	Specifies the elements that compose a platform	No
PLATFORM_ID	Specifies the platform identity	No
PLATFORM_GROUP	Specifies if the platform is within a group	No
SENSOR	Specifies if the platform has sensors	No
SENSOR_BAY	Specifies if the platform sensors are grouped	No
ACTUATOR	Specifies if the platform has actuators	No
ACTUATOR_BAY	Specifies if the platform actuators are grouped	No
<i>INTERACTION</i>	Specifies the interaction between the platform and the rest of the system	No
ACTION	Platform inbound data	No
ALERT	Platform outbound data	No

### A.2.2 Server features

Server features are divided in two distinct sets of features. One for the Web services and another for the database

## Web service features

Feature Name	Description	Concrete
<i>Authentication</i>	Authentication ability in the server side	Yes
<i>DAO</i>	DAO layer base	Yes
UserDAO	Add ability for the DAO layer to access to the user table	Yes
ActionDAO	Add ability for the DAO layer to access the action table	Yes
AlertDAO	Add ability for the DAO layer to access the alert table	Yes
ActuatorBayDAO	Add ability for the DAO layer to access the actuator bay table	Yes
ActuatorDAO	Add ability for the DAO layer to access the actuator table	Yes
SensorBayDAO	Add ability for the DAO layer to access the sensor bay table	Yes
SensorDAO	Add ability for the DAO layer to access the sensor table	Yes
PlatformGroupDAO	Add ability for the DAO layer to access the platform group table	Yes
PlatformDAO	Add ability for the DAO layer to access the platform table	Yes
<i>Model</i>	Describe the database tables in the Web services	No
User	Describe the user table	Yes
Platform	Describe the platform table	Yes
Actuator	Describe the actuator table	Yes
ActuatorBay	Describe the actuator bay table	Yes
SensorBay	Describe the sensor bay table	Yes
Sensor	Describe the sensor table	Yes

PlatformGroup	Describe the platform group table	Yes
Action	Describe the user action	Yes
Alert	Describe the alert table	Yes
<i>Hist</i>	Add logging ability to the Web services	No
hUser	User actions logging	Yes
hSensor	Sensor management logging	Yes
hSensorBay	Sensor bay management logging	Yes
hActuator	Actuator management logging	Yes
hActuatorBay	Actuator bay management logging	Yes
hPlatform	Platform management logging	Yes
hPlatformGroup	Platform group management logging	Yes
hAlert	Received alerts log	Yes
hAction	Sent actions log	Yes
<i>Manage</i>	Management ability on the server side	No
mUser	User details management	Yes
mActuator	Actuator details management	Yes
mActuatorBay	Actuator bay details management	Yes
mSensorBay	Sensor bay details management	Yes
mSensor	Sensor details management	Yes
mPlatformGroup	Platform group details management	Yes
mPlatform	Platform details management	Yes
<i>Utilities</i>	Features that are not strictly necessary but may be helpful	No
<i>List</i>	Element listings	Yes
lUser	List all the entries in the user table	Yes
lActuator	List all the entries in the actuator table	Yes
lActuatorBay	List all the entries in the actuator bay table	Yes
lSensorBay	List all the entries in the sensor bay table	Yes

ISensor	List all the entries in the sensor table	Yes
IPlatformGroup	List all the entries in the platform group table	Yes
IPlatform	List all the entries in the user platform	Yes

### Database features

the database is implemented as a SQLscript, table definitions are inserted between the head and the tail of the script.

Feature Name	Description	Concrete
<i>DATABASE</i>	Defines the database	No
<i>dbTop</i>	Implements the top of the database script	Yes
<i>tUser</i>	Implements the user table	Yes
tSession	Implements the session table	Yes
tUserHistory	Implements the user history table	Yes
<i>tPlatform</i>	Implements the platform table	Yes
tPlatformWeb	Implements the platform web table	Yes
tPlatformHistory	Implements the platform history table	Yes
<i>tAction</i>	Implements the action table	Yes
tCurrentAction	Implements the current action table	Yes
tActionHistory	Implements the action history table	Yes
<i>tAlert</i>	Implements the alert table	Yes
tCurrentAlert	Implements the current alert table	Yes
tAlertHistory	Implements the alert history table	Yes
<i>tSensor</i>	Implements the sensor table	Yes
tSensorHistory	Implements the sensor history table	Yes
<i>tActuator</i>	Implements the actuator table	Yes
tActuatorHistory	Implements the actuator history table	Yes



tPlatformGroup	Implements the platform group table	Yes
tSensorBay	Implements the sensor bay table	Yes
tActuatorBay	Implements the actuator bay table	Yes
dbBottom	Implements the bottom of the database script	Yes

### A.2.3 User interface features

Feature Name	Description	Concrete
Generator	Generate the user interface files and the SQL script file	Yes
<i>Pages</i>	Describes the web pages that compose the user interface	No
<i>Login</i>	Implements the Login page	Yes
NotAllowed	This page appears when a user is not allowed to access a part of the interface	Yes
<i>Home</i>	Home page implementation	No
hTop	Implements the head of the home page	Yes
hBottom	Implements the bottom of the home page	Yes
Troubleshoot	Implements the troubleshoot page	Yes
<i>Management</i>	Management page implementation	No
<i>manTop</i>	Management page top	Yes
manUser	Interface for user management	Yes
manActuator	Interface for actuator management	Yes
manSensor	Interface for sensor management	Yes
manActuatorBay	Interface for actuator bay management	Yes
manPlatform	Interface for platform management	Yes
manSensorBay	Interface for sensor bay management	Yes
manPlatformGroup	Interface for platform group management	Yes

manBottom	Management page bottom	Yes
<i>History</i>	User interface for log view	No
<i>hiTop</i>	History page top	Yes
hiAlert	Visual feedback of the alert logs	Yes
hiUser	Visual feedback of the user management logs	Yes
hiPlatform	Visual feedback of the platform management logs	Yes
hiAction	Visual feedback of the action logs	Yes
hiSensor	Visual feedback of the sensor management logs	Yes
hiActuator	Visual feedback of the actuator management logs	Yes
<i>Styles</i>	CSS styles for the user interface	No
Style1	CSS style 1	Yes
Style2	CSS style 2	Yes
Style3	CSS style 3	Yes

### A.3 Feature constraints

Constraints assure the correct functioning of a specific application by establishing boolean relations between features.

1. PLATFORM\_ ID *implies* PlatformDAO
2. PlatformDAO *implies* Platform *and* tPlatform
3. Authentication *impleis* UserDAO
4. UserDAO *implies* User *and* tUser *and* tSession
5. Authentication *implies* Login *and* NotAllowed
6. PLATFORM\_ GROUP *implies* PlatfromGroupDAO
7. PlatfromGroupDAO *implies* PlatformGroup *and* tPlatformGroup
8. SENSOR *implies* SensorDAO
9. SensorDAO *implies* Sensor *and* tSensor
10. tSensor *implies* tSensorBay

11. ACTUATOR *implies* ActuatorDAO
12. ActuatorDAO *implies* Actuator and tActuator
13. tActuator *and* tActuatorBay
14. SENSOR\_ BAY *implies* SensorBayDAO
15. SensorBayDAO *implies* SensorBay and tSensorBay
16. ACTUATOR\_ BAY *implies* ActuatorBayDAO
17. ActuatorBayDAO *implies* ActuatorBay and tActuatorBay
18. ACTION *implies* ActionDAO
19. ActionDAO *implies* Action and tAction
20. ALERT *implies* AlertDAO
21. AlertDAO *implies* Alert and tAlert
22. DATABASE *and* USER\_ INTERFACE *implies* Generator
23. mUser *implies* UserDAO
24. manUser *implies* mUser and lUser
25. manPlatform *implies* mPlatfrom *and* lPlatform
26. mPlatfromGroup *implies* PlatfromGroupDAO
27. mPlatfrom *implies* PlatformDAO
28. manActuator *implies* mActuator *and* lActuator
29. manPlatformGroup *implies* mPlatformGroup *and* lPlatformGroup
30. manActuatorBay *implies* mActuatorBay *and* lActuatorBay
31. manSensor *implies* mSensor *and* lSensor
32. mSensor *implies* SensorDAO
33. manSensorBay *implies* mSensorBay *and* lSensorBay
34. mSensorBay *implies* SensorBayDAO
35. mActuator *implies* ActuatorDAO
36. mActuatorBay *implies* ActuatorBayDAO
37. hiAlert *implies* hAlert
38. hAlert *implies* tAlertHistory
39. hiAction *implies* hAction
40. hAction *implies* tActionHistory
41. hiUser *implies* hUser
42. hUser *implies* tUserHistory
43. hiPlatform *implies* hPlatform
44. hPlatform *implies* tPlatformHistory

45.  $\text{hiSensor}$  *implies*  $\text{hSensor}$

46.  $\text{hSensor}$  *implies*  $\text{tSensorHistory}$

47.  $\text{hiActuator}$  *implies*  $\text{hActuator}$

48.  $\text{hActuator}$  *implies*  $\text{tActuatorHistory}$

49.  $\text{tCurrentAction}$  *implies*  $\text{tActuator}$  *and*  $\text{tSensor}$  *and*  $\text{tSensorBay}$  *and*  $\text{tActuatorBay}$  *and*  $\text{tPlatform}$  *and*  $\text{tPlatformGroup}$

## Appendix B

# Domain Impementation

### B.1 Flowcharts

This section specifies flowcharts of the three subsystems. These flowcharts specify the operation of the the several components of each subsystem. Each flowchart states its function and the group of features that it is aimed for.

The User interface flowcharts specify how a GET and a POST request are performed in this project. GET is used to ask the server for lists of alerts, actions, components and history and POST is used when a operation is performed that involves sending data to the server, like management actions, troubleshooting or history management. These requests are received by the server that processes them depending to which Web service they are aimed to.

Specifying these Web services are also flowcharts that specify Web services operation by group, since operations in the same group are similar, only differing on the data contents and the database table addressed, it makes sense to expose flow charts this way.

Ending, there is a flowchart for the platform interface. It specifies the what are the operations performed by the platform interface to transmit data from the server to the platform monitoring process.

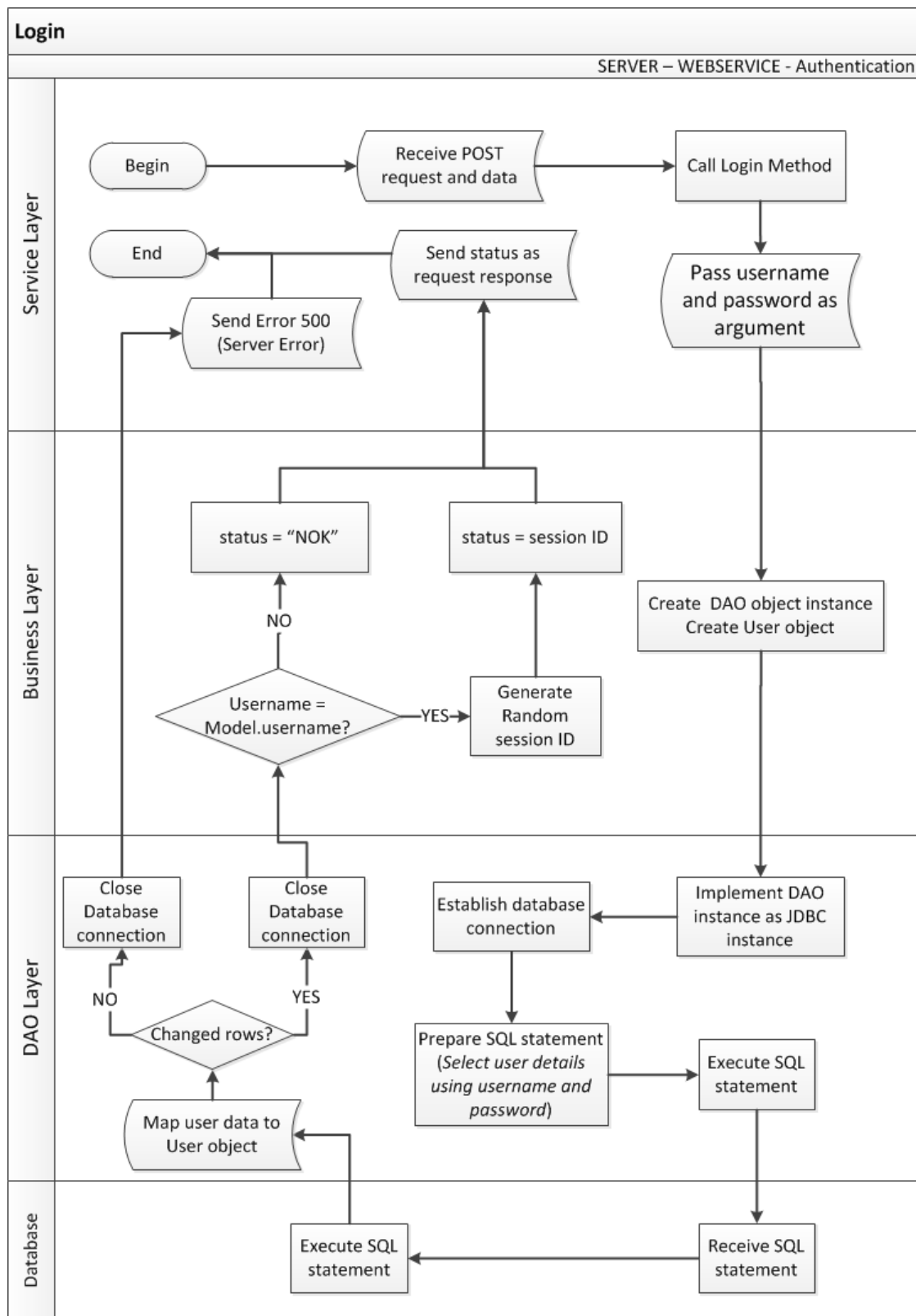


Figure B.1: Login Web service flowchart

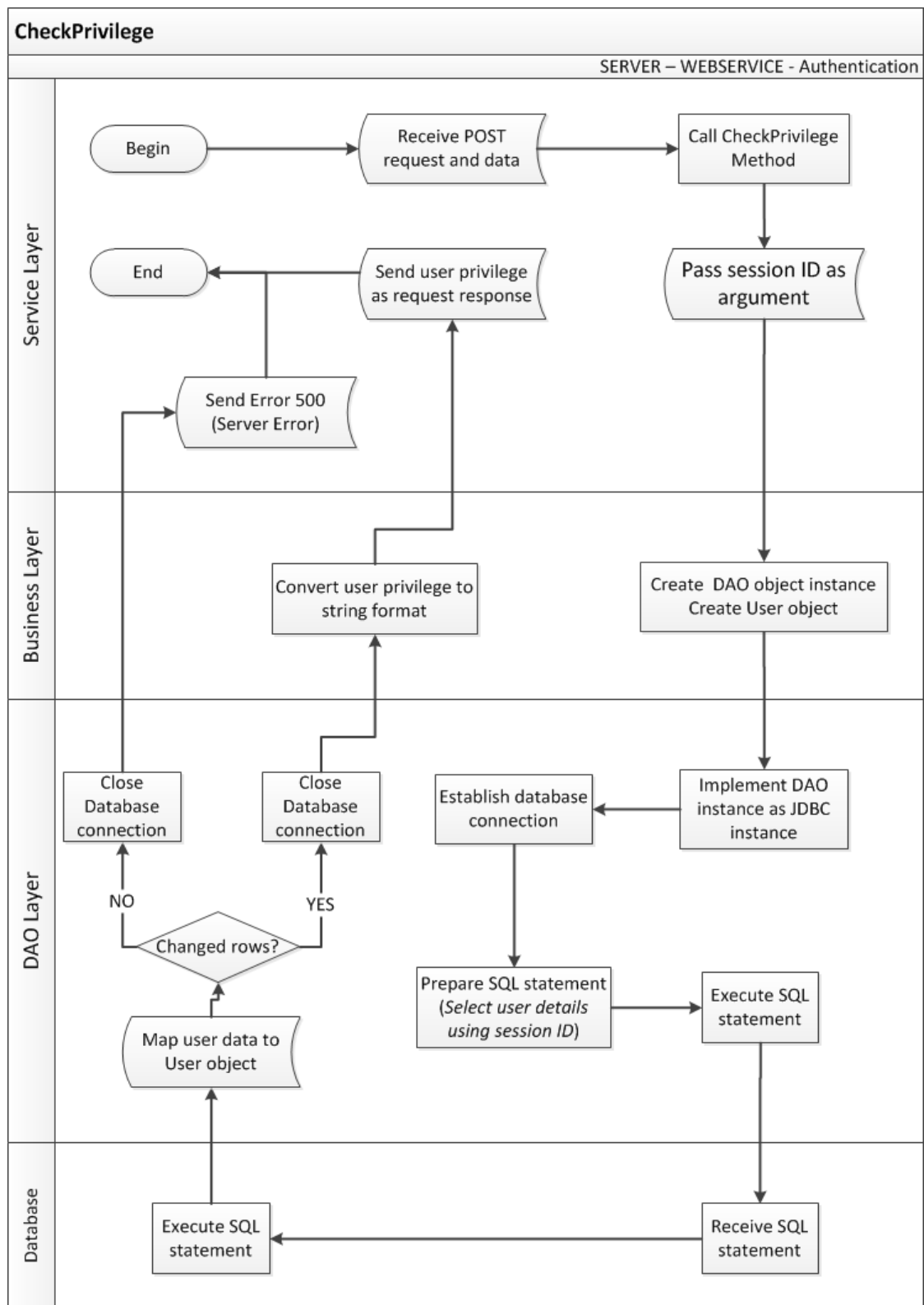


Figure B.2: Check privilege Web service flowchart

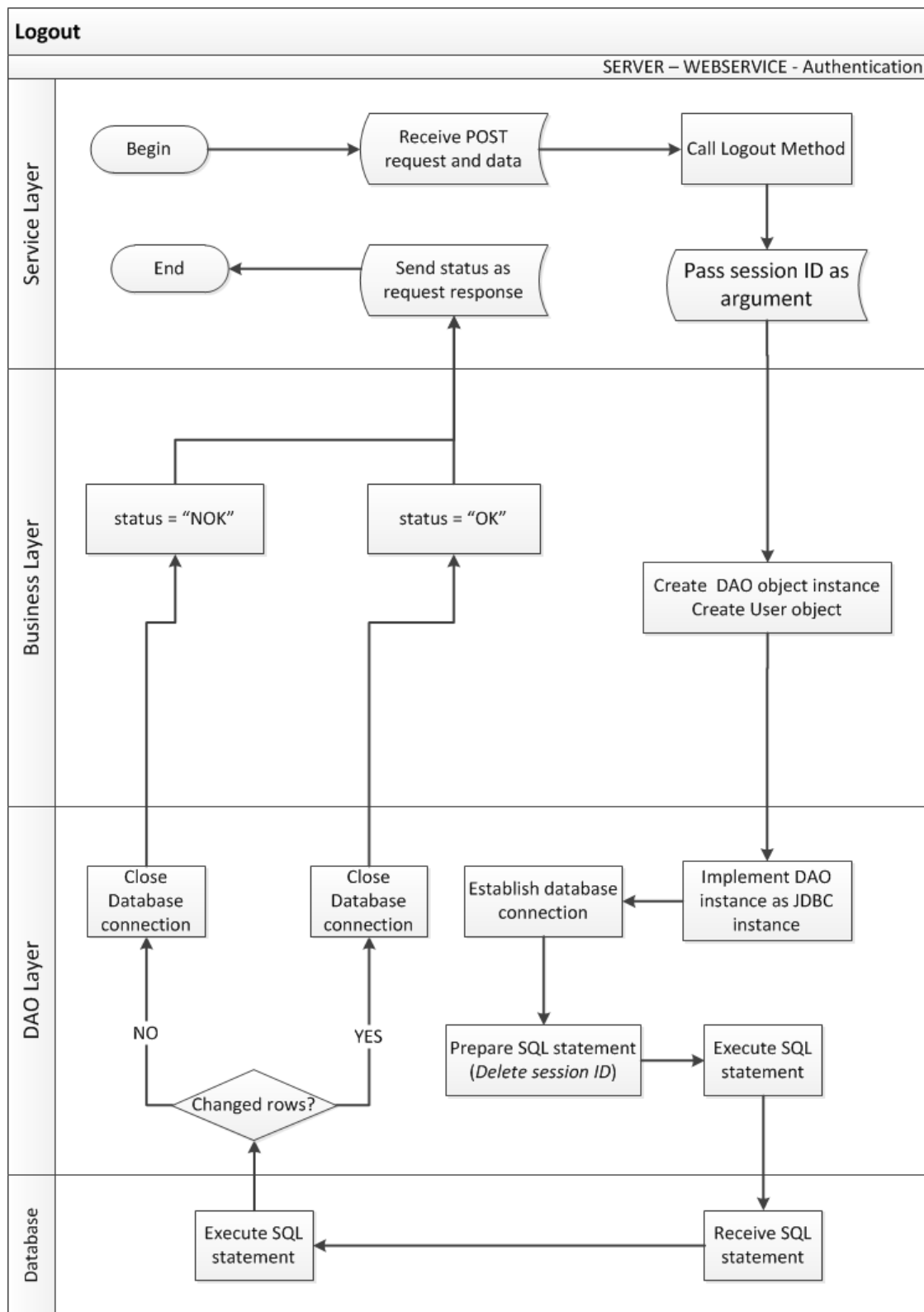


Figure B.3: Logout Web service flowchart



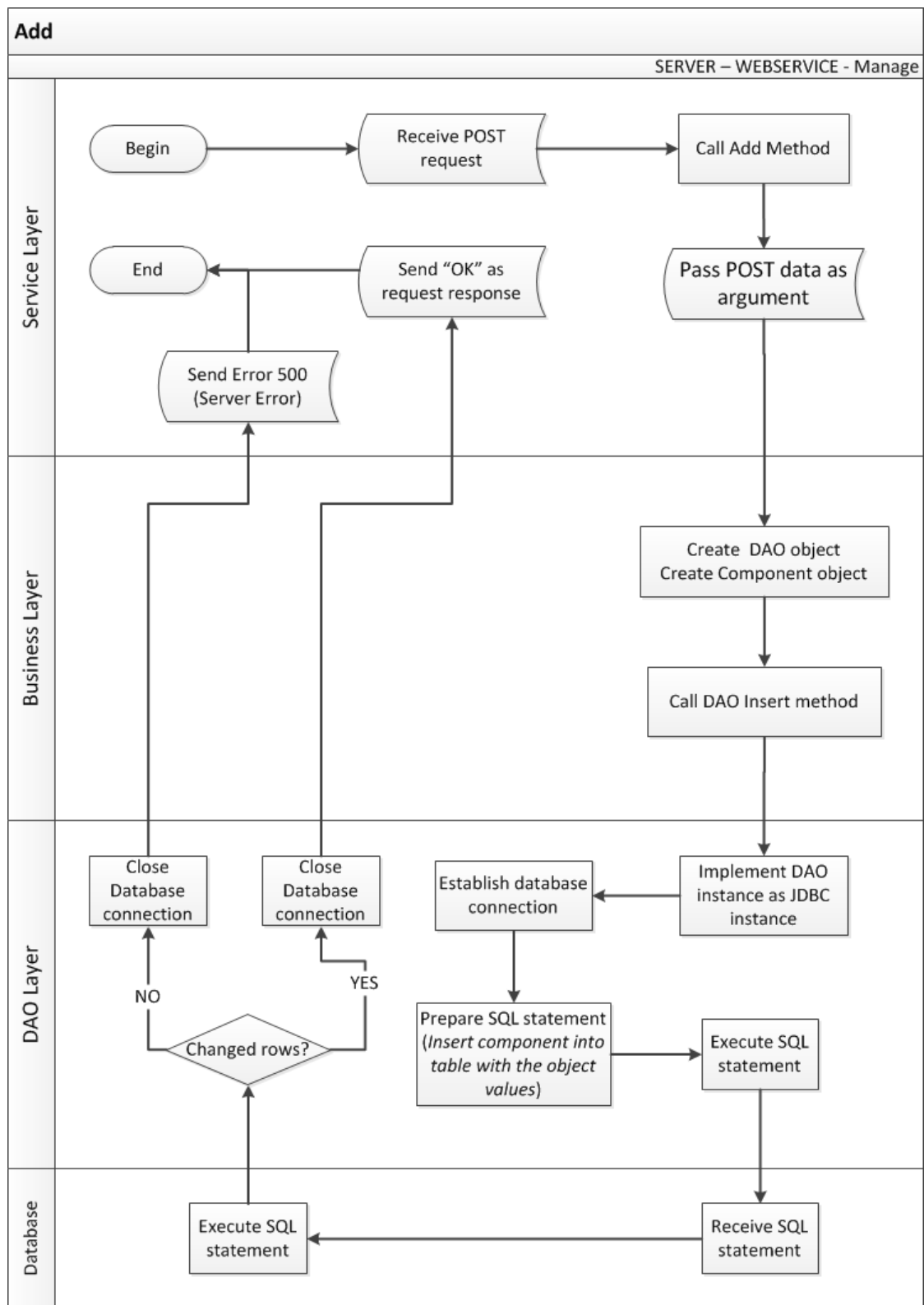


Figure B.4: Add Web services flowchart

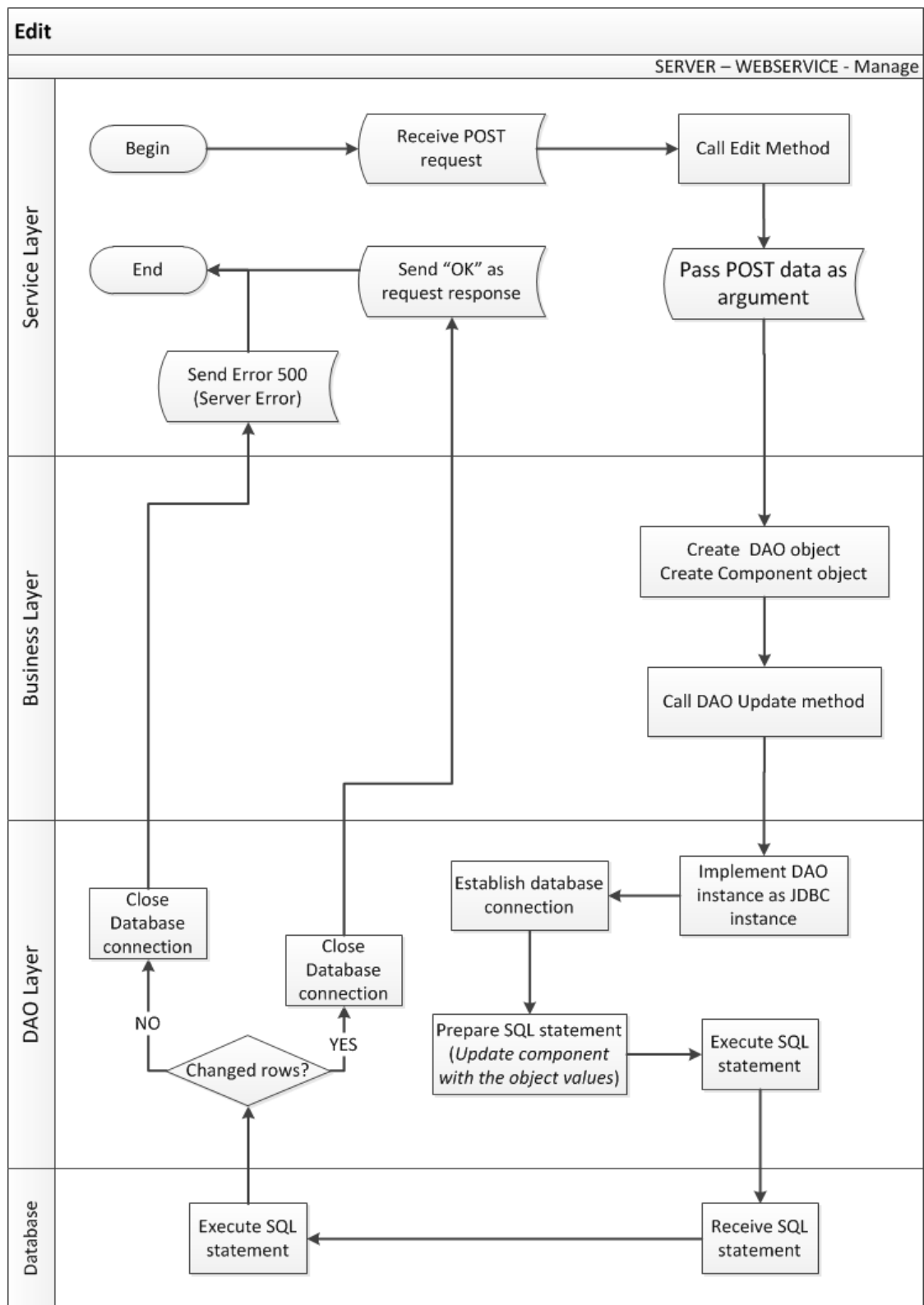


Figure B.5: Edit Web services flowchart

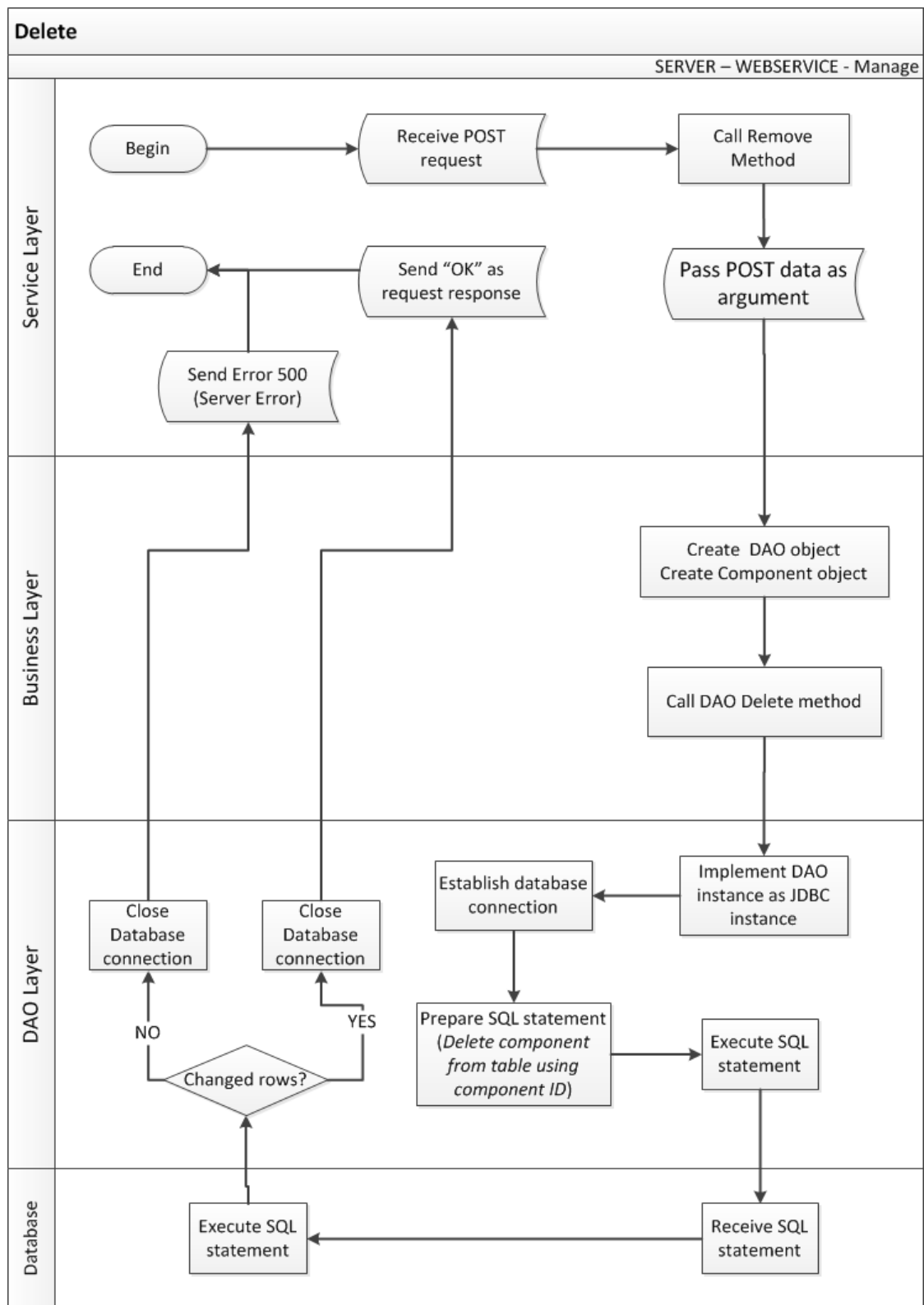


Figure B.6: Delete Web services flowchart

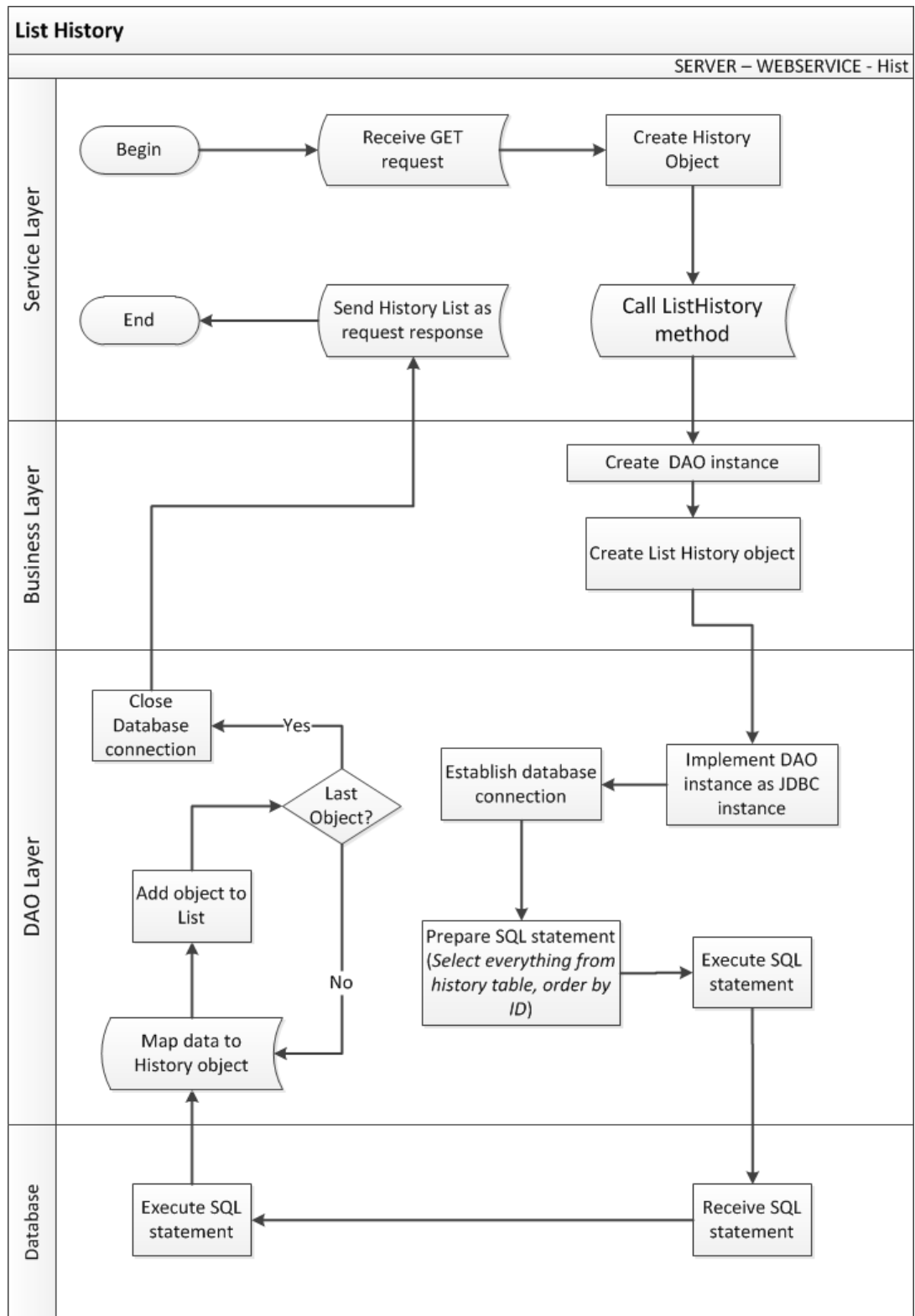


Figure B.7: List history Web services flowchart

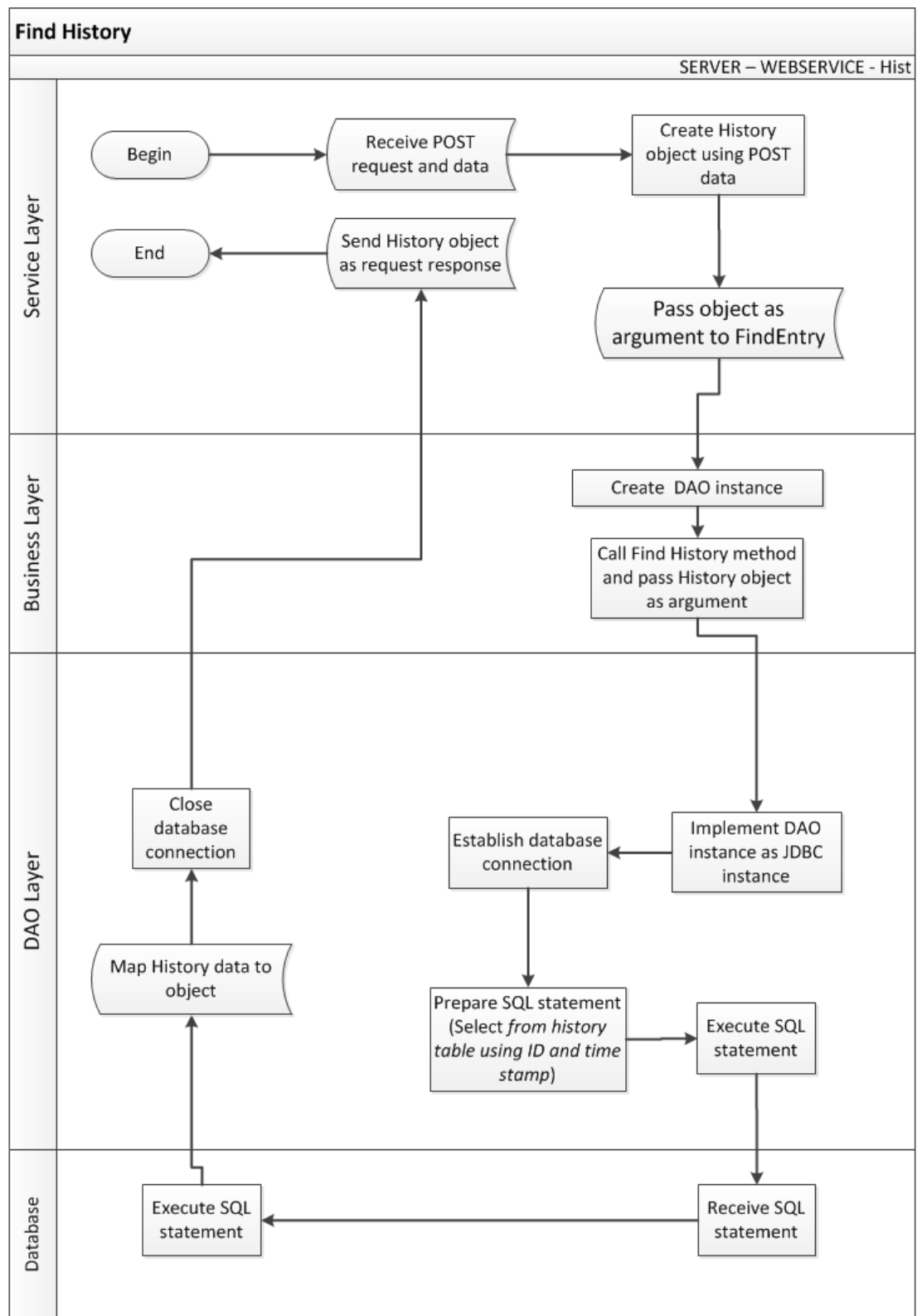


Figure B.8: Find history Web services flowchart

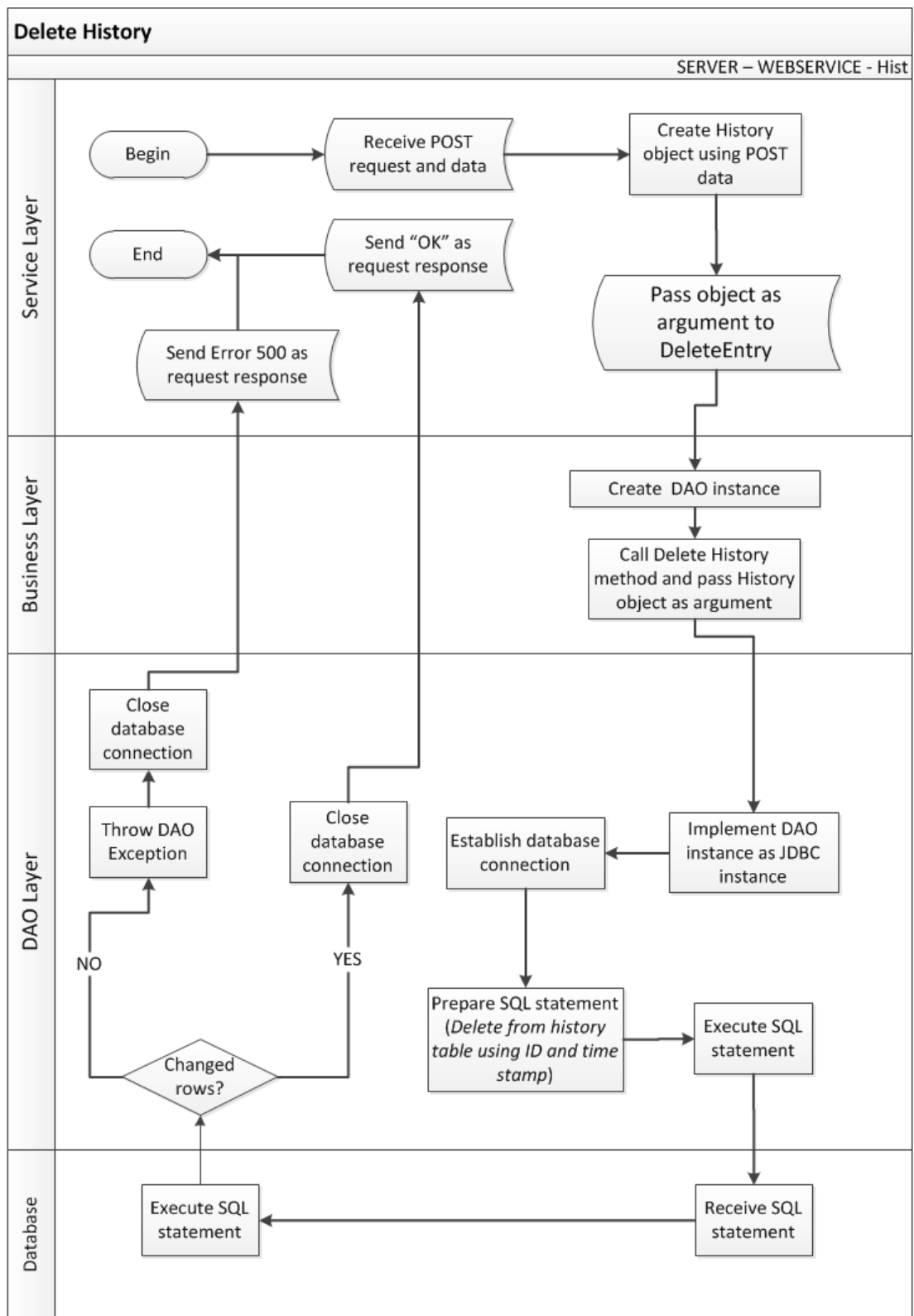


Figure B.9: Delete entry from history Web services flowchart

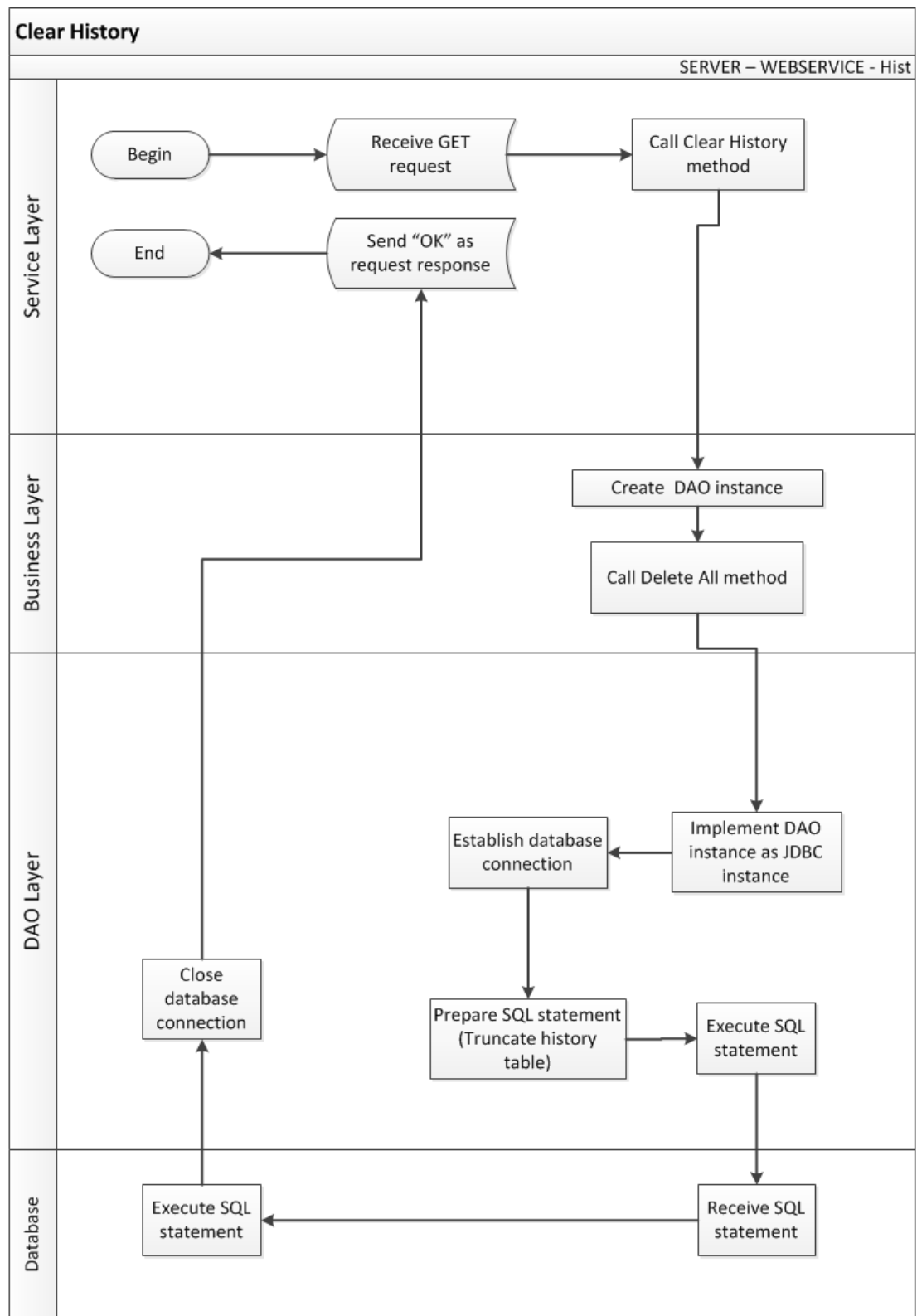


Figure B.10: Clear history Web services flowchart

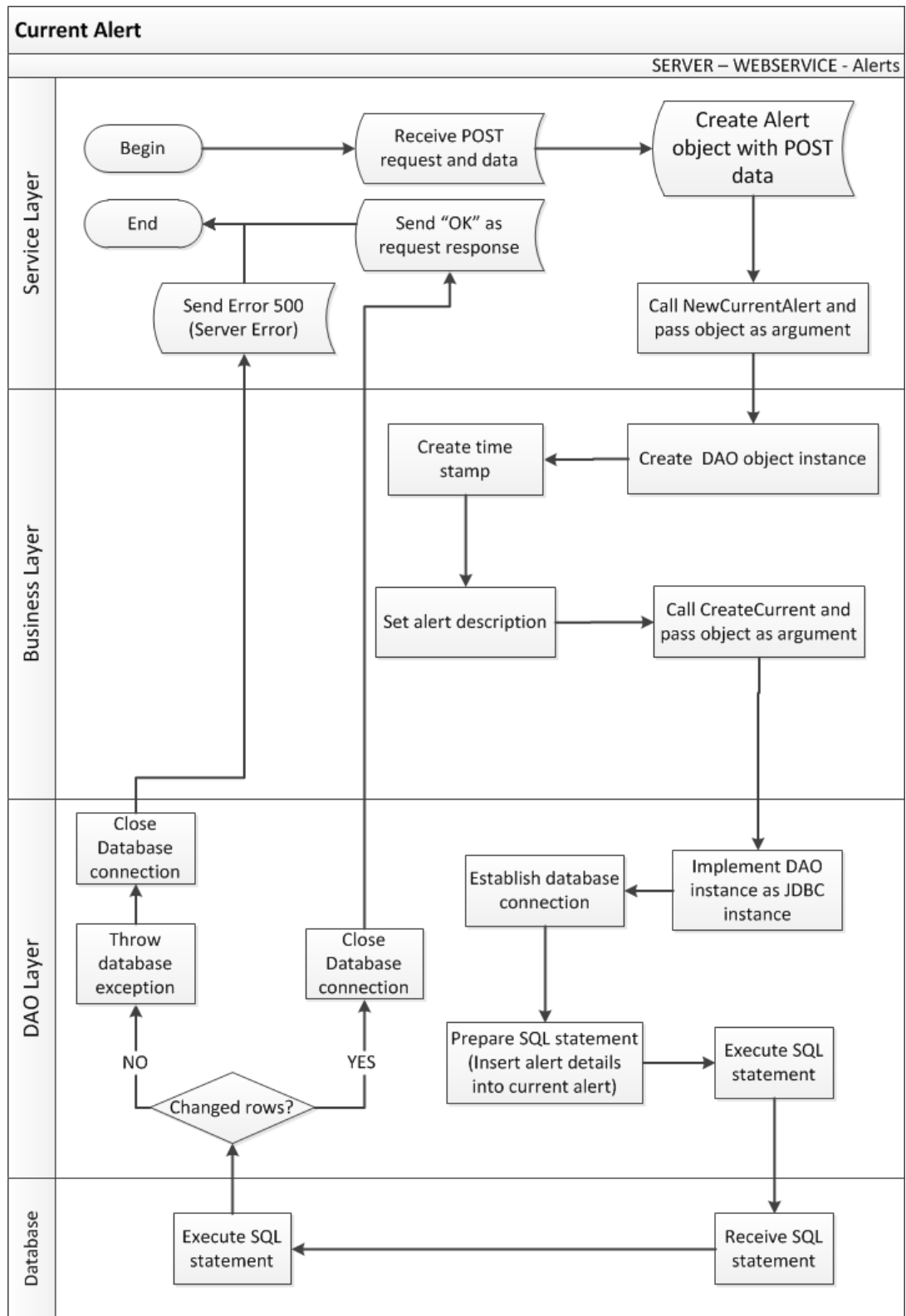


Figure B.11: Current alert Web service flowchart



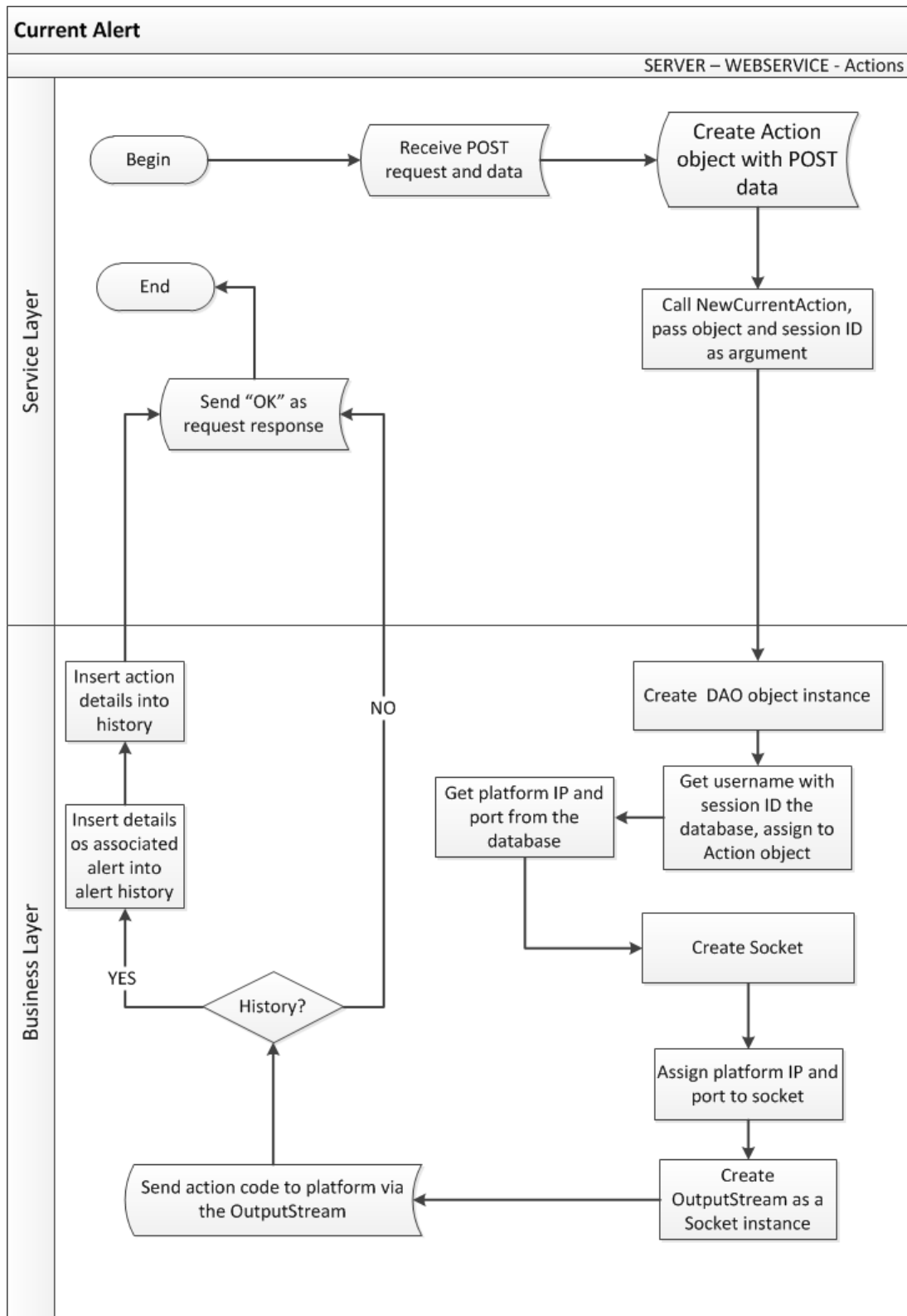


Figure B.12: Current action Web service flowchart

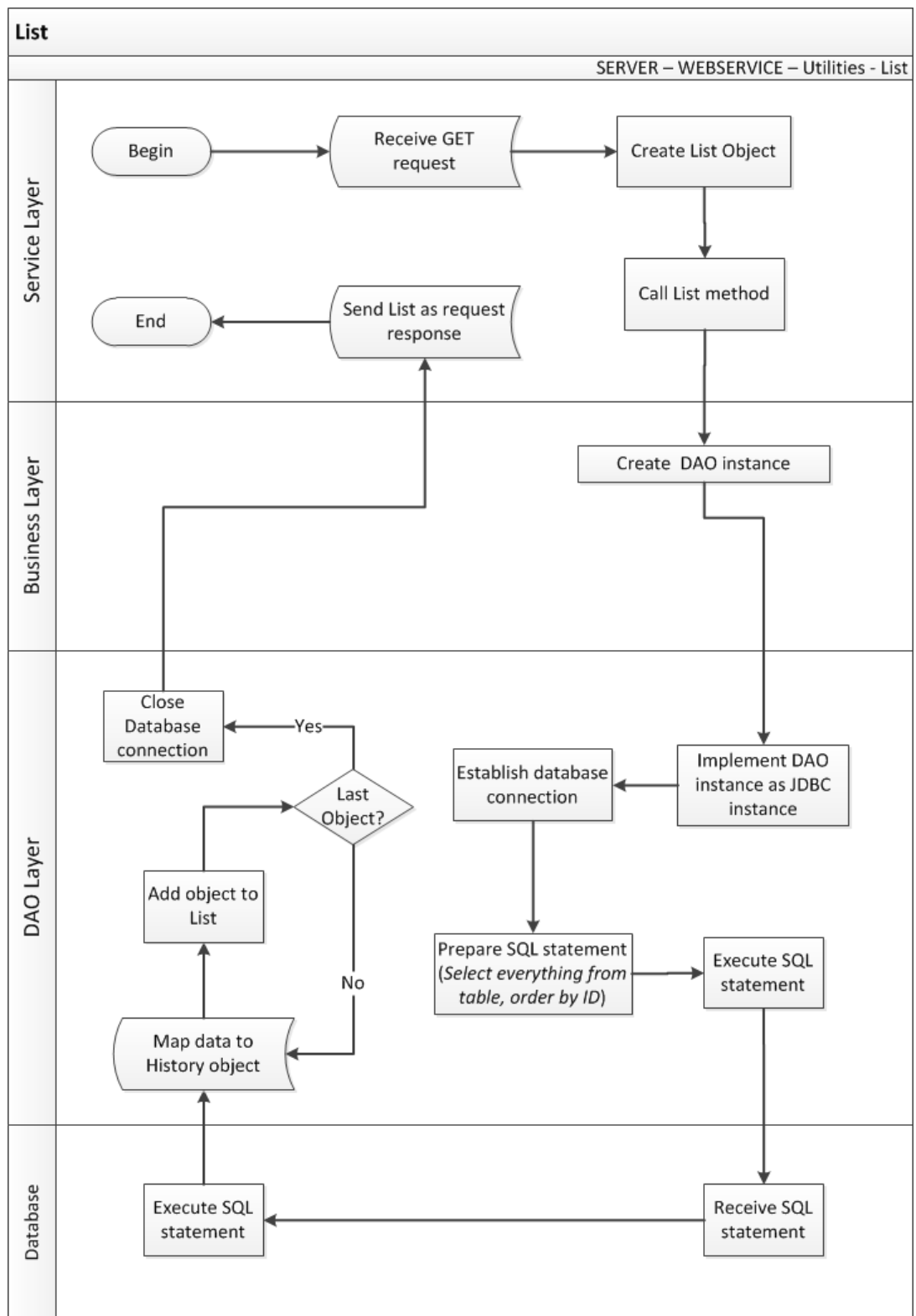


Figure B.13: List Web services flowchart

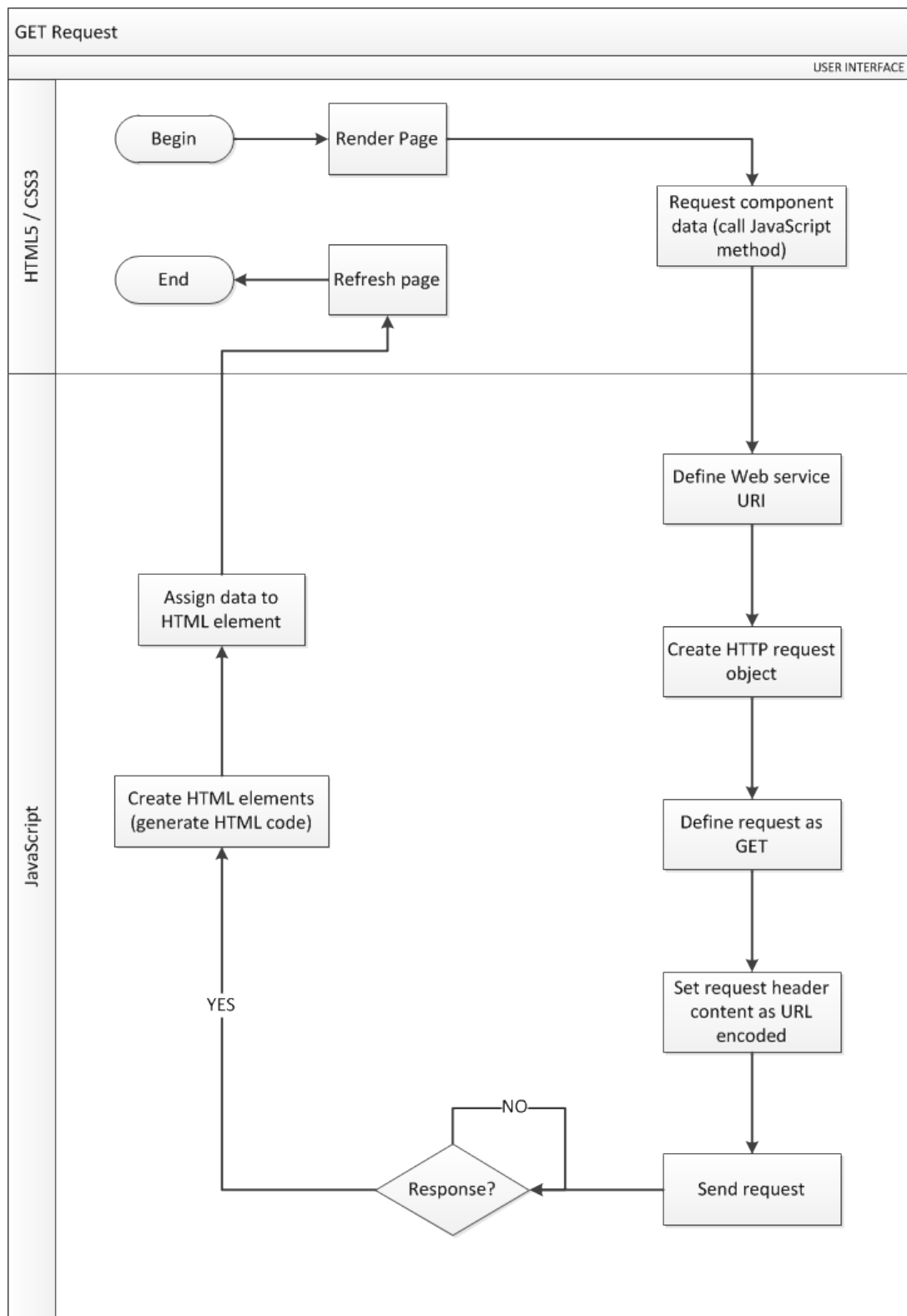


Figure B.14: GET request flowchart

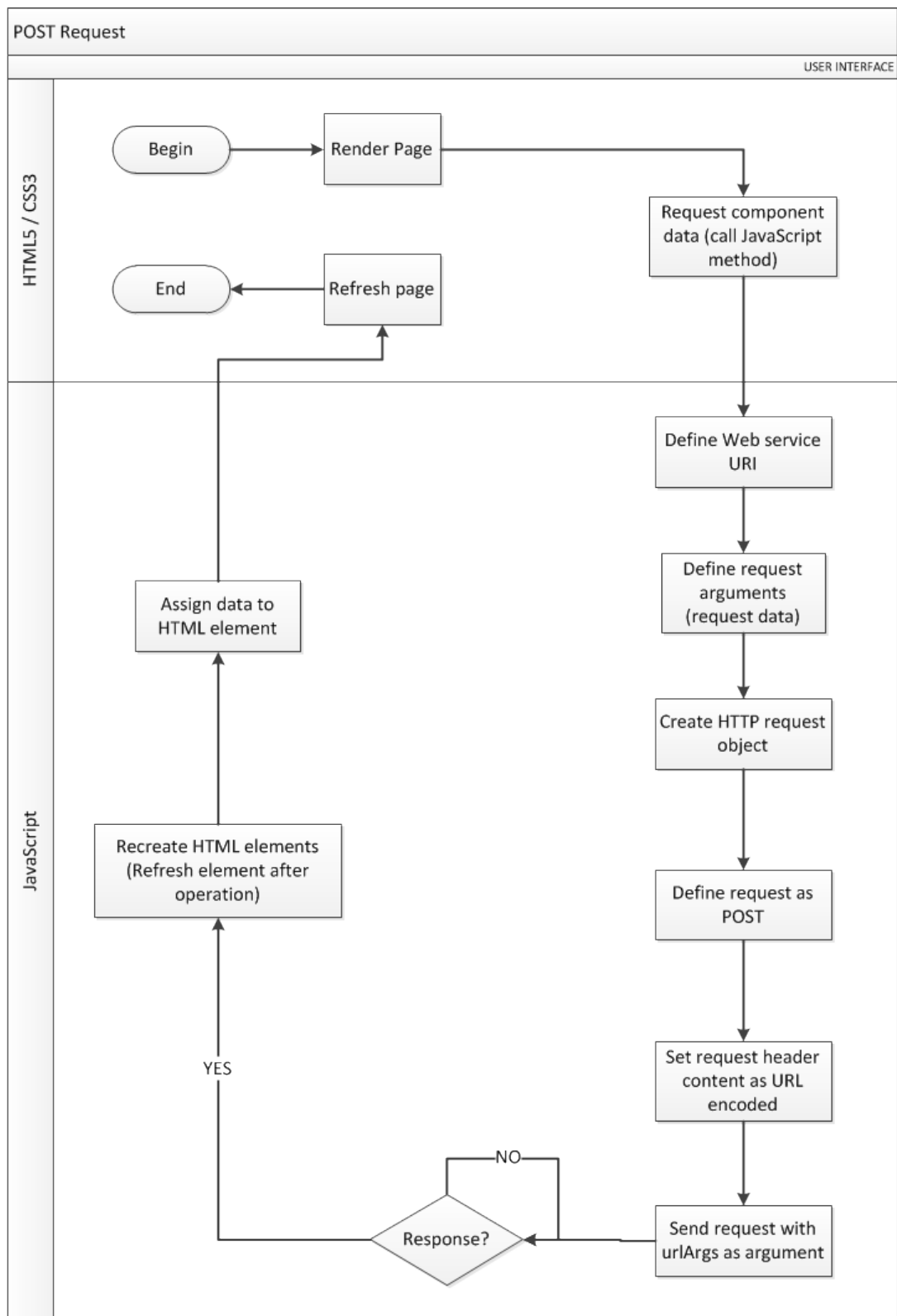


Figure B.15: POST request flowchart

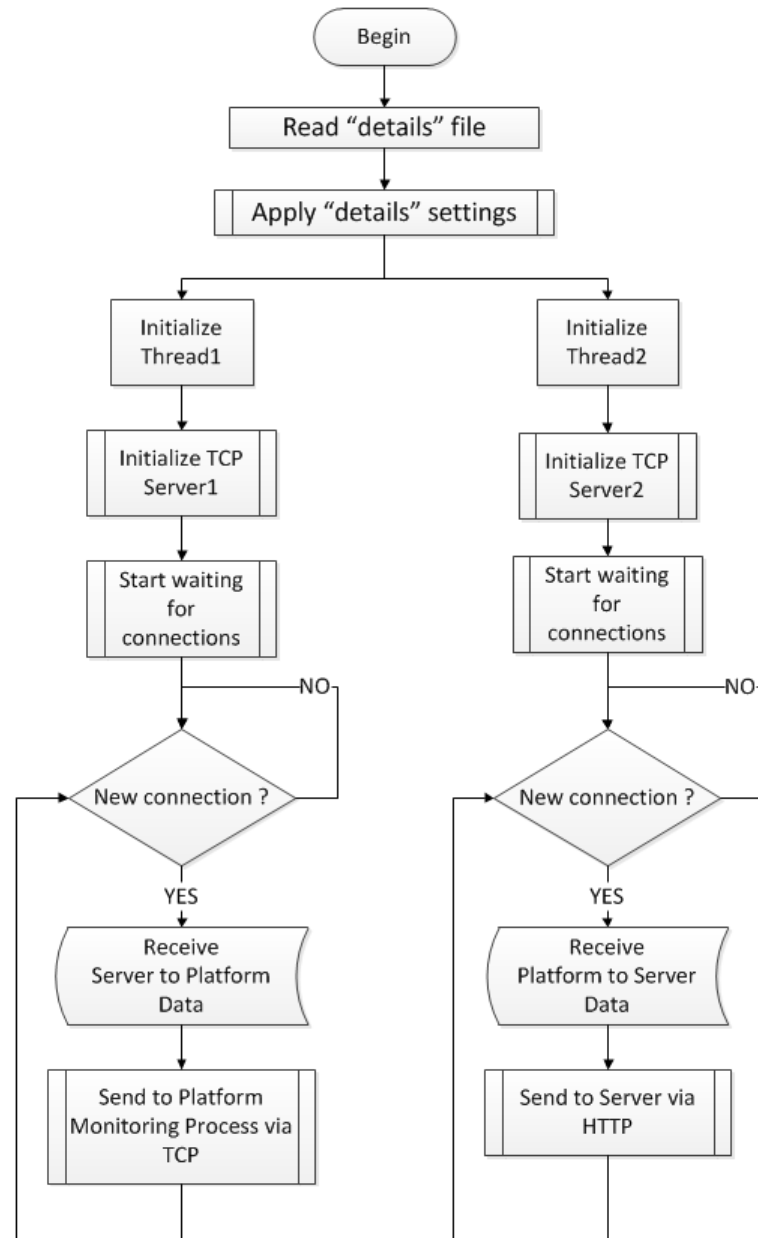
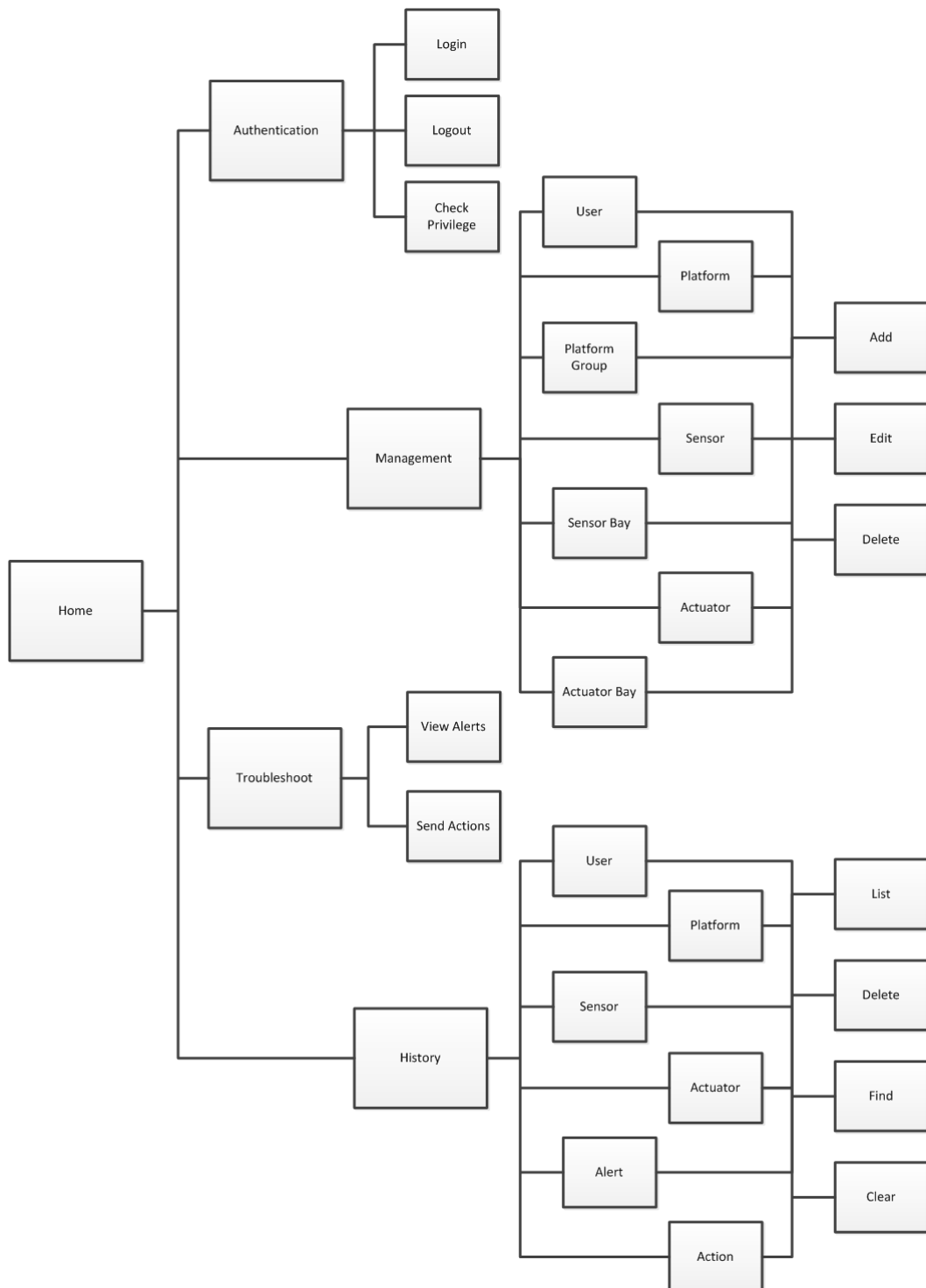


Figure B.16: Platform interface flowchart

## B.2 User interface site map



### B.3 Web service path hierarchy

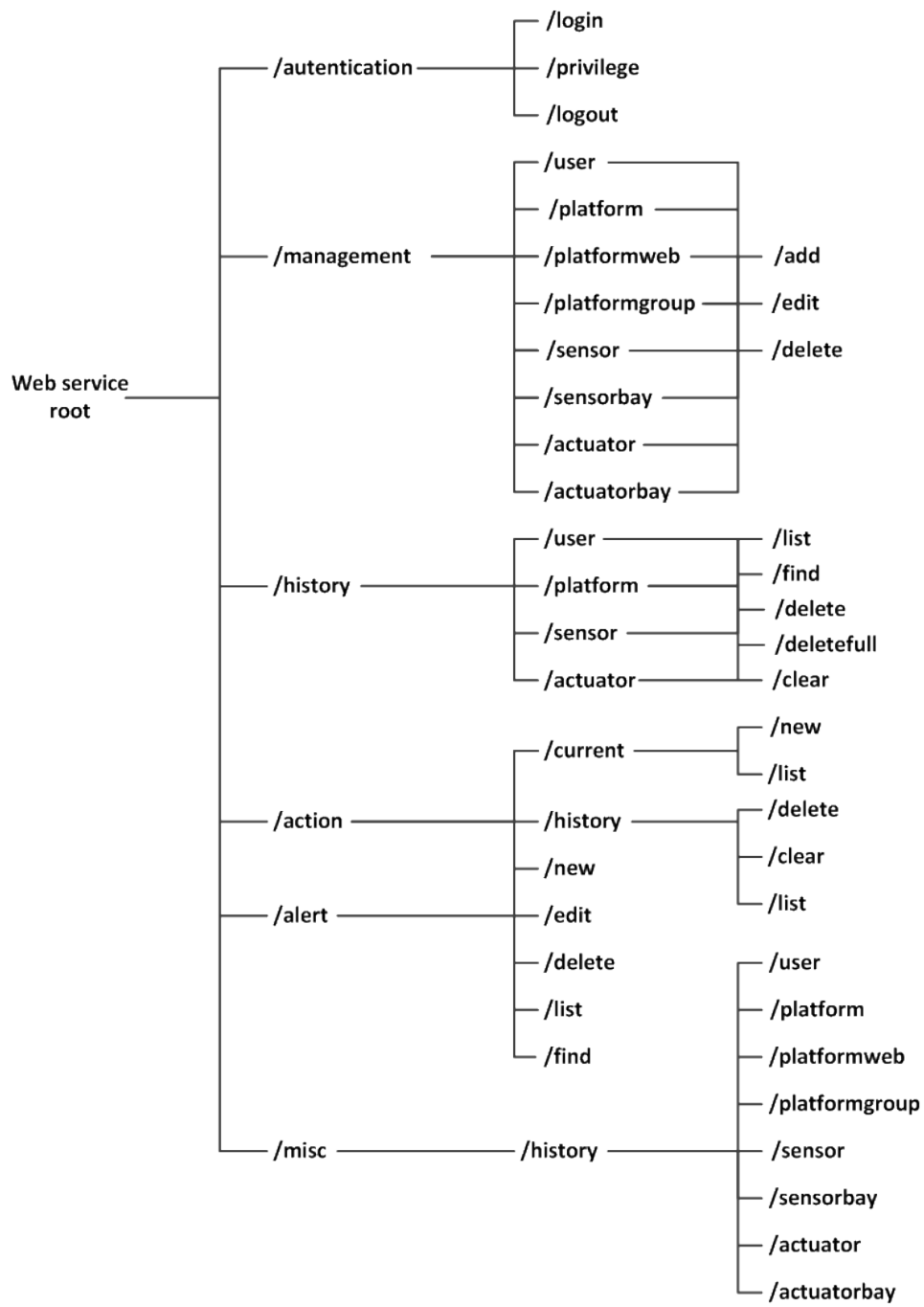


Figure B.17: Web service URI hierarchy

## B.4 Web services specification

This section specifies the Web services URI, their HTTP methods and their path parameters.

### B.4.1 Action service

The action service can be found at the */action* path.

1. */new* - Insert new action details in the database
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code, desc*
2. */edit* - Edit the details of an action
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code, desc*
3. */delete* - Delete an action
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code*
4. */list* - List all the action in the database
  - (a) **HTTP Method:** *GET*
5. */find* - Find an action in the database
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code*
6. */current/new* - Insert a new current action in the database
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *alertID, actionID, sID, platID*



7. */current/list* - List all the current actions

(a) **HTTP Method:** *POST*

8. */history/list* -List all the entries in the action log

(a) **HTTP Method:** *GET*

9. */history/delete* - Delete a specific entry from the action log

(a) **HTTP Method:** *POST*

(b) **Path Parameters:** *code, timeStamp*

10. */history/clear* - Clear all the action log

(a) **HTTP Method:** *GET*

#### B.4.2 Alert service

The alert service can be found at the */alert* path.

1. */new* - Insert new alert details in the database

(a) **HTTP Method:** *POST*

(b) **Path Parameters:** *code, desc*

2. */edit* - Edit the details of an alert

(a) **HTTP Method:** *POST*

(b) **Path Parameters:** *code, desc*

3. */delete* - Delete an alert from the database

(a) **HTTP Method:** *POST*

(b) **Path Parameters:** *code*

4. */find* - Find an alert in the database

(a) **HTTP Method:** *POST*

- (b) **Path Parameters:** *code*
- 5. */list* - List all the alert in the database
  - (a) **HTTP Method:** *Get*
- 6. */current/new* - Insert new action details in the database
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code, platID, groupID, sensorID, sensorBayID, actuatorID, aBayID*
- 7. */current/list* - List all the current alerts
  - (a) **HTTP Method:** *GET*
- 8. */history/list* - List all the alerts in the alerts log
  - (a) **HTTP Method:** *GET*
- 9. */history/delete* - Delete an entry from the alerts log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *code, timeStamp*
- 10. */history/clear* - Clear the alerts log
  - (a) **HTTP Method:** *GET*

### B.4.3 Authentication service

The authentication service can be found at the */authentication* path.

- 1. */Login* - User login service
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *username, password*
- 2. */privilege* - Ask the server for a user privilege based on the session ID

- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *sID*
3. */logout* - User logout service
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *sID*

#### B.4.4 Management service

The management service can be found at the */management* path.

- 1. */user/add* - Add a new user the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *username, password, priv, email*
- 2. */user/edit* - Edit the details of an user
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *username, password, priv, email*
- 3. */user/delete* - Delete an user from the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *delete*
- 4. */actuator/add* - Add a new actuator to a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid, type*
- 5. */actuator/edit* - Edit the details of an actuator from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid, type*

6. */actuator/delete* - Delete an actuator from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid*
7. */actuatorbay/add* - Add a new actuator bay to a platform in the the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid, desc*
8. */actuatorbay/edit* - Edit the details of an actuator bay from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid, desc*
9. */actuatorbay/delete* - Delete an actuator bay from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid*
10. */sensorbay/add* - Add a new sensor bay to a platform in the the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid, desc*
11. */sensorbay/edit* - Edit the details of a sensor bay from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid, desc*
12. */sensorbay/delete* - Delete a sensor bay from a platform in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *bayid, platid*
13. */sensor/add* - Add a new sensor to a platform in the system

- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid, type*
14. */sensor/edit* - Edit the details of a sensor from a platform in the system
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid, type*
15. */sensor/delete* - Delete a sensor from a platform in the system
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, bayid, platid*
16. */platformgroup/add* - Add a new platform group the system
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, dsc*
17. */platfromgroup/edit* - Edit the details of a platform group
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, dsc*
18. */platformgroup/delete* - Delete a platform group from the system
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id*
19. */platform/add* - Add a new platform in the system
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, lat, lng, dsc, gID*
20. */platform/edit* - Edit the details of a platform
- (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, lat, lng, dsc, gID*

21. */platform/delete* - Delete a platform from the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id*
22. */platformweb/add* - Add the web details of a platform already in the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, ip, port*
23. */platformweb/edit* - Edit the web details of a platform
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id, ip, port*
24. */platform/delete* - Delete a platform's web details from the system
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *id*

#### B.4.5 History service

The history service can be found at the */history* path.

1. */user/list* - List the log entries from all the users
  - (a) **HTTP Method:** *GET*
2. */user/find* - Find a specific entry in the user log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *username, timestamp*
3. */user/delete* - Delete an entry from the user log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *username, timestamp*

4. */user/clear* - Clear the user log
  - (a) **HTTP Method:** *POST*
5. */sensor/list* - List the log from all the sensors
  - (a) **HTTP Method:** *GET*
6. */sensor/find* - Find a specific entry in the sensor log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *sensorid, timestamp*
7. */sensor/delete* - Delete an entry from the sensor log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *sensorid, timestamp*
8. */sensor/deletefull* - Delete all the log entries for a given sensor
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *sensorid*
9. */sensor/clear* - Clear all the sensor log
  - (a) **HTTP Method:** *GET*
10. */actuator/list* - List the log from all the actuators
  - (a) **HTTP Method:** *GET*
11. */actuator/find* - Find a specific entry in the actuator log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *actuatorid, timestamp*
12. */actuator/delete* - Delete an entry from the actuator log
  - (a) **HTTP Method:** *POST*

- (b) **Path Parameters:** *actuatorid, timestamp*
- 13. */actuator/deletefull* - Delete all the log entries for a given actuator
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *actuatorid*
- 14. */actuator/clear* - Clear the actuator log
  - (a) **HTTP Method:** *GET*
- 15. */platform/list* - List the log from all the platforms
  - (a) **HTTP Method:** *GET*
- 16. */platform/find* - Find a specific entry in the platform log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *platid, timestamp*
- 17. */platform/delete* - Delete an entry from the platform log
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *platformid, timestamp*
- 18. */platform/deletefull* - Delete all the log entries for a given platform
  - (a) **HTTP Method:** *POST*
  - (b) **Path Parameters:** *actuatorid*
- 19. */platform/clear* - Clear the platform log
  - (a) **HTTP Method:** *GET*



#### B.4.6 Miscellaneous service

The miscellaneous service can be found at the */misc* path.

1. */list/user* - List all the users registered in the system

(a) **HTTP Method:** *GET*

2. */list/actuator* - List all the actuators

(a) **HTTP Method:** *GET*

3. */list/actuatorbay* - List all the actuator bays

(a) **HTTP Method:** *GET*

4. */list/sensor* - List all the sensors

(a) **HTTP Method:** *GET*

5. */list/sensorbay* - List all sensor bays

(a) **HTTP Method:** *GET*

6. */list/platformgroup* - List all the platform groups

(a) **HTTP Method:** *GET*

7. */list/platform* - List all the platforms

(a) **HTTP Method:** *GET*

8. */list/platformweb* - List all the platform web details

(a) **HTTP Method:** *GET*

## Appendix C

# Instantiation

### C.1 Editing the Database connection properties

For an application to be able to connect to the database there is a file specifying the database user details, the database address and the database Java connector. This file is called *DBconnection.properties* and must be moved from the *mon.spl.dao* package to the root file of the application code, so that it can be read by the application.

The user must edit the file as shown on figure C.1, where the top file is the file immediately after the application generation and the bottom file is the one after edition, for use with a specific database.

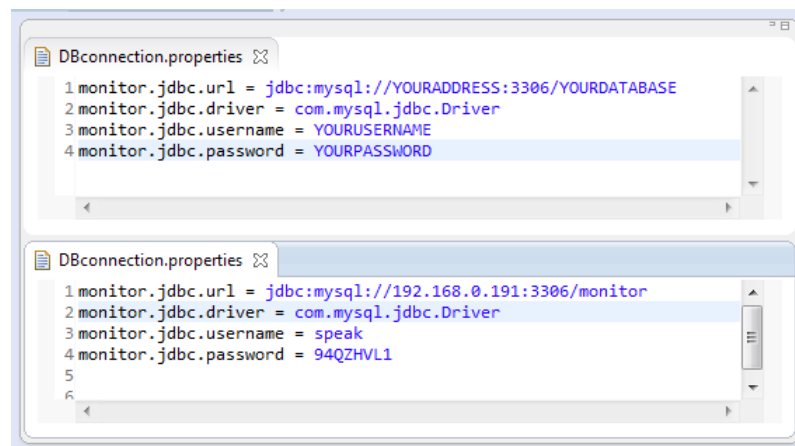


Figure C.1: *DBconnection.properties* before and after editing

## C.2 Example of a web.xml deployment descriptor

Figure C.2 shows how a *web.xml* deployment descriptor can be defined so that the Web services can be accessible from the Web.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xmlns="http://java.sun.com/xml/ns/javaee"
4         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6         id="WebApp_ID" version="2.5">
7   <display-name>FullConfig</display-name>
8   <servlet>
9     <servlet-name>Monitoring REST Service</servlet-name>
10    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
11    <init-param>
12      <param-name>com.sun.jersey.config.property.packages</param-name>
13      <param-value>mon.spl.srv</param-value>
14    </init-param>
15    <load-on-startup>1</load-on-startup>
16  </servlet>
17  <servlet-mapping>
18    <servlet-name>Monitoring REST Service</servlet-name>
19    <url-pattern>/rest/*</url-pattern>
20  </servlet-mapping>
21  <!-- Context Listener -->
22  <listener>
23    <listener-class>mon.spl.biz.BackgroundThread</listener-class>
24  </listener>
25 </web-app>
```

Figure C.2: *web.xml* definition example