Master Thesis

August 27, 2010

Merlin

Role-based Access Control and Workflow System for the Information Management System Merlin

Jaroslav Habr

of Hradec Králové, Czech Republic (02-727-873)

supervised by

Prof. Dr. Harald Gall

Dr. Gerald Reif

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Master Thesis

Merlin

Role-based Access Control and Workflow System for the Information Management System Merlin

Jaroslav Habr

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Jaroslav Habr, jaro_habr@access.uzh.ch

Project period: 09. Mars 2010 - 27. August 2010

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank all people involved in this project for their support. In the first place, I want to thank Professor Harald Gall for giving me the opportunity to write this thesis at the Software Evolution and Architecture Lab and for providing me with the infrastructure. Then, I want to thank Dr. Gerald Reif for giving me the idea for this project and for supervising my daily work. His big engagement, critical thoughts and precise work helped to bring this project to a higher level. In our meetings, he inspired me with new ideas and encouraged me to come up with even better solutions. I also want to thank him for proofreading this thesis.

Second, I want to thank Matthias Hert for testing our application and for expressing some constructive criticism. I also want to thank him for proofreading my thesis and for sharing his experience in writing. Then, I thank Maria Olivares who supported us during the development by sharing her domain knowledge and inspiring us with innovative ideas. Her detailed test feedback was of great value to us.

A big thanks goes to my friends Mark and Thomas. I want to thank Mark for being a part of this project and for his active cooperation. It was very motivating to work on his side, to discuss and develop solutions to problems we were facing during this project. Without Mark, the project would not have come that far. I want to thank Thomas for sharing his experience in Web application development, providing me with some great ideas concerning implementation of security and for helping me out with the tricky Shibboleth setup.

Last but not least, I want to thank my parents for their great support during my whole studies and for giving me such wide opportunities. They often had to suffer under my inattentiveness during this project, especially when things did not go that well. I highly appreciate the distractions they provided for cheering me up.

Abstract

The Faculty of Economics, Business Administration and IT of the University of Zurich is being certified by the European EQUIS and the American AACSB accreditation authorities. In order to apply for an accreditation, the faculty must provide self-assessment reports that contain comprehensive information about publications, scientific staff, research and teaching as well as service issues on a yearly basis. Currently, this information is gathered by compiling data into Excel sheets. Because of the faculty's great number of research groups, this results in an almost unmanageable amount of files that must be consolidated and formatted by hand. To improve this process, the idea for a Web-based information management system called Merlin came up. Its main purpose is to be a centralized repository for employee and publication data.

The focus of this master thesis lies in the conception and implementation of a role-based security system and the development of user and organizational unit related management functionalities using Java's next generation Web application framework Grails. The authentication service of the security system was developed to integrate with the University of Zurich's Authentication and Authorization Infrastructure (AAI), which is based on the Shibboleth open source project. The authorization service implements the concepts of a role-based access control (RBAC) model by supporting role hierarchies, multiple role inheritance, role delegations and instance level access control. Merlin was developed to integrate into the existing application landscape and to make use of employee and organizational unit data provided by the University of Zurich's enterprise resource planning software SAP. Organizational unit data is imported initially by parsing a XML file whereas employee data is integrated based on nightly CSV file imports triggered by a cron job. The employee centric functionality covers customizable profile page containing all necessary information such as personal data, education or appointment. Merlin's organizational unit centric functionalities aim to provide an editable organizational unit profile, a clear organizational structure of the faculty with a correct affiliation of the its members. To achieve this, an interactive setup wizard with automatic employee membership requests has been implemented. Finally, functionality for CSV employee data export is provided to support the generation of self-assessment reports.

For quality assurance purposes, an evaluation based on user tests has been carried out. The users were assigned to groups based on the function they have in the organization. Each group had to perform a defined set of task. The tests showed that the functionality concerning authentication and authorization worked as expected. However, there were some issues with comprehensibility of the workflow related to tasks. The users wish a comprehensive documentation in form of a FAQ and more help texts. With these tests, we were able to fix some issues and thus improve the quality of the application.

Zusammenfassung

Die Wirtschaftswissenschaftliche Fakultät der Universität Zürich lässt sich durch zwei unabhängige Akkreditierungsstellen, namentlich die europäische EQUIS und die amerikanische AACSB, zertifizieren. Um die Akkreditierungen zu erhalten, müssen von der Fakultät jährlich Daten über wissenschaftliche Mitarbeiter, veröffentlichte Publikationen und Forschungs- und Lehrtätigkeiten der Lehrstühle konsolidiert und in Form von diversen Berichten eingereicht werden. Gegenwärtig werden diese Daten manuell gesammelt. Die Berichte werden anschliessend aus einer Vielzahl von einzelnen Excel-Dateien in einem mühsamen Prozess angefertigt. Um die Berichterstellung zu erleichtern, hat sich die Fakultät entschlossen, ein Informationsmanagementsystem namens Merlin zu entwickeln. Es soll als zentrales Archiv für Mitarbeiter- und Publikationsdaten dienen und somit die Grundlage für die Reporterstellung bilden.

Der Fokus dieser Masterarbeit liegt in der Konzeption und Implementierung eines rollenbasierten Sicherheitssystems und der Entwicklung von benutzer- und organisationseinheitspezifischen Verwaltungsfunktionalitäten unter Verwendung des Webapplikations-Framework Grails. Der Authentifizierungsservice des Sicherheitssystems basiert auf der Authentifizierungs- und Autorisierungsinfrastruktur (AAI) der Universität Zürich. Der Autorisierungsservice implementiert die Konzepte einer rollenbasierten Zugriffskontrolle (RBAC) und unterstützt Rollenhierarchien, mehrfache Rollenvererbung, Rollendelegation und instanzbasierte Zugriffsregelung. Merlin wurde mit dem Ziel entwickelt, sich nahtlos in die bestehende Applikationslandschaft der Universität Zürich einzufügen und Daten aus der universitätsinternen Enterprise Resource Planning Software SAP zu integrieren. Während die Daten der Organisationseinheiten initial durch das Parsen einer XML-Datei importiert werden, werden die Mitarbeiterdaten mithilfe eines Cronjobs jede Nacht aus einer CSV-Datei geladen. Merlin bietet jedem Benutzer ein individuell anpassbares Profil, welches alle relevanten Informationen zur Person, Ausbildung oder zum Anstellungsverhältnis übersichtlich darstellt. Die Struktur der Fakultät wird durch einen Baum abgebildet, von welchem aus die Profile der einzelnen Lehrstühle besucht werden können. Um die Zuordnung von Mitarbeitern zu Lehrstühlen zu pflegen, muss die Zugehörigkeit zu einem Lehrstuhl vom jeweiligen Leiter bestätigt werden. Um die Erstellung der für die Akkreditierung nötigen Berichte zu unterstützen, können Mitarbeiterdaten aus Merlin in eine CSV-Datei exportiert werden.

Die Evaluation dieser Arbeit wurde in Form von Benutzertests durchgeführt. Die Tester wurden in Gruppen mit verschiedenen Zugriffsrechten eingeteilt und mit Aufgaben beauftragt. Die Auswertung zeigt, dass das Sicherheitssystems korrekt funktionierte. Der Delegationsarbeitsfluss wurde jedoch nicht in allen Fällen verstanden. Viele Tester äusserten den Wunsch nach einer detaillierteren Beschreibung der Funktionalität in Form einer FAQ und grössere Unterstützung durch Hilfetexte. Basierend auf diesen Tests konnten wir einige Probleme beheben und somit die Qualität von Merlin erhöhen.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Application Scope . . . 2
- 1.3 Focus of this Thesis . . . 2
- 1.4 Structure of the Thesis . . . 2

2 Related Work **5**
- 2.1 Access Control Policies . . . 5
- 2.2 Role-Based Access Control (RBAC) . . . 6
  - 2.2.1 Basic Elements . . . 6
  - 2.2.2 The Security Principles . . . 6
- 2.3 Roles and Role Hierarchies . . . 7
- 2.4 Delegation within RBAC . . . 9

3 Requirements **11**
- 3.1 General Requirements . . . 11
  - 3.1.1 System Context . . . 11
  - 3.1.2 Roles . . . 13
- 3.2 Functional Requirements: Use Cases . . . 14
  - 3.2.1 Visitor . . . 14
  - 3.2.2 General Use Cases for Authenticated Users . . . 15
  - 3.2.3 Employee . . . 16
  - 3.2.4 Researcher . . . 17
  - 3.2.5 Chair . . . 18
  - 3.2.6 Admin . . . 19
- 3.3 Non-Functional Requirements . . . 21
  - 3.3.1 Core Technology: The Web Application Framework Grails . . . 21
  - 3.3.2 Maintainability . . . 21
  - 3.3.3 Intuitive User Interface using modern Technologies . . . 21
  - 3.3.4 Data Quality . . . 22
  - 3.3.5 Corporate Design . . . 22

4 Implementation **23**
- 4.1 Agile Web Development with Grails . . . 23
  - 4.1.1 The Principles . . . 23
  - 4.1.2 The programming Language: Groovy . . . 25
  - 4.1.3 The Architecture . . . 25

4.2 Security . . . 29
4.2.1 Requirements . . . 29
4.2.2 A Word on Security Plugins . . . 30
4.2.3 Architecture . . . 32
4.2.4 Authentication . . . 32
4.2.5 Basic Elements of RBAC . . . 34
4.2.6 Role Hierarchy . . . 35
4.2.7 Authorization . . . 37
4.2.8 Pulling it all together . . . 41
4.2.9 Delegation . . . 41
4.3 Domain Model and Basic Functionality . . . 44
4.3.1 Overview . . . 44
4.3.2 Employee . . . 46
4.3.3 Organizational Unit . . . 49
4.3.4 Assigning Employees to Organizational Units . . . 50
4.4 Data Source: SAP . . . 51
4.4.1 Receiving Data from SAP . . . 52
4.4.2 Cron Job for Nightly Imports . . . 53
4.5 CSV Export of Employee Records . . . 53

5 Deployment Environment **55**
5.1 Server Setup . . . 55
5.1.1 Apache HTTP-Server . . . 55
5.1.2 Apache Tomcat . . . 57
5.1.3 Connecting Apache to Tomcat . . . 57
5.2 Shibboleth Setup for Authentication . . . 58

6 Evaluation **61**
6.1 Test Setup . . . 61
6.1.1 Test Environment . . . 61
6.1.2 User Base and Time Schedule . . . 61
6.1.3 Approach . . . 62
6.2 User Feedback and Improvement Suggestions . . . 63
6.2.1 Task related Feedback . . . 63
6.2.2 General Feedback . . . 65
6.3 Conclusion . . . 66

7 Conclusion and Future Work **67**
7.1 Developing with Grails . . . 67
7.1.1 Agility . . . 67
7.1.2 Grails Plugins . . . 69
7.1.3 Groovy is cool . . . 70
7.2 Other Aspects and Project Experience . . . 70
7.2.1 External Resources are Barriers . . . 70
7.2.2 Changing Requirements . . . 71
7.3 Reflection on Contribution . . . 72
7.3.1 Security . . . 72
7.3.2 Functionality . . . 72
7.3.3 User Interface and Design . . . 73
7.3.4 Conclusion . . . 73
7.4 Future Work . . . 73

A A List of used Tools **75**
A.1 Source Code and Project Management 75
A.2 Grails Plugins 75
A.3 Javascript 76

B Content of the CD-ROM **77**
B.1 Thesis 77
B.2 Framework and Application 77
B.3 SAP Data Files used for Import 77

List of Figures

2.1 Basic elements of role-based access control (RBAC) . . . 7
2.2 Role hierarchy . . . 8

3.1 System Context . . . 12
3.2 Use Cases . . . 15

4.1 The Grails Stack . . . 26
4.2 Architectural Pattern: MVC . . . 27
4.3 Security Architecture Overview . . . 33
4.4 Basic Elements of RBAC . . . 35
4.5 Role Hierarchy . . . 36
4.6 Authorization Sequence Diagram . . . 42
4.7 Delegation in Merlin . . . 43
4.8 Task States . . . 44
4.9 User Interface for Task Handling . . . 45
4.10 Domain Model Overview (Simplified) . . . 46
4.11 Employee Class . . . 47
4.12 Setup Wizard . . . 48
4.13 Employee Profile . . . 49
4.14 Organizational Unit Model . . . 50
4.15 Employee-Organizational Unit Relationships . . . 51
4.16 Anatomy of SAPImporter . . . 52
4.17 CSV Exporter . . . 54

5.1 Communication between Apache HTTP-Server and Apache Tomcat . . . 58

List of Tables

4.1 Mapping of Permissions onto Roles . . . 37
4.2 Public Interface of Authorization Service . . . 39
4.3 Delegation Role Assignment . . . 44

List of Listings

4.1 Simple Closure . . . 25
4.2 Dynamic Finders . . . 27
4.3 Injecting a Service Instance into a Controller . . . 28
4.4 View Helpers in Grails: Tag Libs . . . 29
4.5 Rendering the Publications Template . . . 29
4.6 Shibboleth Action in the SessionController . . . 33
4.7 Authenticate Filter . . . 37
4.8 Public Section in Security Filters . . . 38
4.9 Declaration of Privileges . . . 38
4.10 Edit Employees Filter . . . 38
4.11 HasPermission() Method of the Authorization Service . . . 39
4.12 Employees() Method in the Authorization Helper . . . 40

4.13 Security Tag Examples . . . 41
4.14 ImportEmployees() Method in SAPImporter . . . 52
4.15 Example of an Organizational Unit in XML . . . 53
4.16 Cron Job to trigger nightly Employee Imports . . . 53
5.1 Apache2.conf File . . . 56
5.2 Apache Config for Merlin (Excerpt) . . . 56
5.3 Apache SSL Config for Merlin (Excerpt) . . . 56
5.4 Tomcat Config . . . 57
5.5 Activation of AJP Protocol in Apache Tomcat Config . . . 57
5.6 Worker File Properties . . . 58
5.7 Declaration of Worker File in Apache Config . . . 58
5.8 Shibboleth Registration of new Site . . . 59
5.9 Shibboleth Request Map for Merlin . . . 59
5.10 Shibboleth Application Override for Identification of a second Application . . . 59
7.1 Groovy's XMLSlurper . . . 70

Chapter 1

Introduction

"The best way to predict the future is to implement it."

David Heinemeier Hansson

The Faculty of Economics, Business Administration and IT of the University of Zurich is being certified by two independent accreditation authorities – the European EQUIS[1] and the American AACSB[2]. The application for the accreditation requires self-assessment reports, which contain comprehensive information about the faculty. More precisely, consolidated data about scientific publications, personal information and teaching activities must be provided on a yearly basis.

Currently, this information is gathered by exchanging Excel sheet among the faculty members, who have to fill in their data manually. Because the faculty comprises of a vast number of research groups, this procedure results in an unmanageable amount of files. In order to generate a report, the data must further be consolidated and formatted by hand. This is a very tedious and highly exhausting task for the people in charge.

1.1 Motivation

Inspired by the situation described above, the idea of a Web based information management system was born. The main purpose of this system is to provide a centralized platform, which takes care of the data management and supplies the reporters with precalculated data. This idea is driven by the fact that most of the information required for the self assessment reports is already available in the University of Zurich's data warehouse, but is distributed over several systems such as the publication repository ZORA (Zurich Open Repository and Archive)[3] or the human resource management system SAP. Therefore, it would be natural to build a platform that makes use of the available data sources in order to improve the current process of information gathering and processing. For the people involved in the report generation, such a revolutionary system seems magical – this is why we called it *Merlin*.

[1] http://www.efmd.org/index.php/accreditation-/equis
[2] http://www.aacsb.edu/
[3] http://www.oai.uzh.ch/index.php

1.2 Application Scope

Merlin is supposed to be used by all members of the Faculty of Economics, Business Administration and IT for personal data and publication management purposes, which forms a considerably large user basis. One of the biggest concerns of Merlin is to integrate neatly into the existing application landscape. Basically, there are two main areas that Merlin is concerned with: an *employee and organizational unit centric* area and a *publication centric* area.

As with every modern Web application, Merlin should offer its users their own profile page, where they should be able to manage their personal data such as contact details, education information or appointment details. Furthermore, Merlin's aim is to provide a clear organizational structure of the faculty with a correct affiliation of the faculty members to their organizational units. For the most part, this shall be achieved my means of data imports from the University of Zurich's enterprise resource planning system SAP. Because people in organization such as the Faculty of Economics, Business Administration and IT do have different functions, Merlin shall provide a corresponding security system in order to protect the information from unauthorized access.

Every active research member has a number of publications that he must be able to manage. Merlin's requirement is to support all functionalities needed for successful publication entry and editing. Since there is already a great number of publication data available in separate systems such as PAX[4] or ZORA, Merlin must import this data in order to fulfill the requirement of being a complete publication repository and to support the generation of yearly reports.

1.3 Focus of this Thesis

With respect to all these requirements, Merlin was implemented in cooperation of two students. The first master thesis covers the import and implementation of the publication management [Ode10]. The focus of this thesis lies in the conception and implementation of a sophisticated, role-based security system that provides authentication through the University of Zurich's authentication service *AAI*[5] and a tailored access for each user to protected Merlin's resources. Further, this thesis focuses on the import of employee and organizational data from the SAP system and provides the implementation of functions related to user and organizational data management. Finally, the support for the accreditation reports is implemented by providing employee related data in form of a CSV export.

1.4 Structure of the Thesis

The thesis is divided into seven chapters. *Chapter 2* discusses related work on role-based access control (RBAC) and provides the theoretical background used to develop our security concept. It explains, how roles can be structured into a hierarchy and how delegation in RBAC works. *Chapter 3* presents the system context of Merlin, how it fits into the University of Zurich's application landscape and describes the functional and non-functional requirements in detail. The first part of *Chapter 4* gives an overview of the used technology and shows how it influences the architecture of Merlin. The second part reveals the concept and implementation details of the security architecture. The third part focuses on the domain model used to implement the required functionality. The last part shows how external data is integrated into Merlin. *Chapter 5* presents the

[4] http://www.ifi.uzh.ch/pax/
[5] http://www.id.uzh.ch/dl/mobil/AAI.html

deployment environment, especially the interaction between Apache and Tomcat, and explains, how authentication with the Shibboleth Service Provider works. *Chapter 6* describes the setup and tasks we prepared for the user tests and summarizes the results and desired improvements. *Chapter 7* retrospects the work by outlining some thoughts about its progress and discusses problems that occurred during the project.

Chapter 2

Related Work

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards."

Gene Spafford

In this chapter we will focus on the theory of role-based access control. The insights from the scientific work will help us to build a fundamental understanding about what a role-based access control (RBAC) is and how its elements interact with each other. We will use this theoretical background in later chapters to develop a security concept for Merlin. The first part of this chapter gives an overview of the different access control policies and points out their characteristics. The second part describes the central elements and principles of RBAC. The third part explains how roles can be structured into hierarchies and what the benefits are. The last part concentrates on delegation of access rights in role-based access control systems.

2.1 Access Control Policies

Access control technology arose from research and development efforts supported by the United State Government Department of Defense starting from 1967 [Lat85]. This effort resulted in two fundamental types of access control: *Discretionary Access Control (DAC)* and *Mandatory Access Control (MAC)*.

In *discretionary access control systems* granting and revoking of access privileges is left to the discretion of the individual users [FK95]. Discretion in this context means, that a subject – a computer process acting on behalf of a user – with a certain access permission is capable of passing that permission to any other subject without the intercession of a system administrator [FK92]. This main characteristic of DAC systems has been deemed as a main drawback in the literature. Because the access restrictions stated by the authorizations can be easily bypassed, discretionary access policies do not provide real assurance on the flow of information in a system [SS94]. The fact that the end users do not *own* the information for which they have access permissions leads to the conclusion that discretionary access policies are not appropriate for many organizations. In this case, the corporations themselves are the actual *owner* of the system objects, which represent any resource accessible on a computer system (such as files, databases, or fine-grained entities like individual fields in database records) [FKC07]. The access is often based on employee functions or roles users take on rather than on data ownerships [FK92].

Mandatory access control systems restrict access to objects based on the sensitivity of the information contained in the objects and the formal authorization of subjects to access information of

a particular sensitivity level [FK92]. In other words, the access to an information is governed by the classification of subjects and objects in the system. Every user and information is assigned to a security level. The security level associated with an object reflects the sensitivity of the information contained in the object. The security level associated with a user shows his trustworthiness [SS94].

Neither discretionary nor mandatory access control systems are well suited for industry organizations that process unclassified but sensitive information. Security objectives in such environments often require support for higher-level organizational policies derived from existing laws, ethics, regulations or generally accepted practices. The need to control actions of individuals beyond the individual's ability to access information based on its sensitivity makes the role-based access control policies more adequate [FK95].

2.2 Role-Based Access Control (RBAC)

Role-based access control (RBAC) emerged in the 1990s as a proven technology for managing and enforcing security in computer systems [PSA01]. Role-based policies regulate the access to the information on the basis of the activities users execute in a system [SS94]. A role in a role-based policy can be thought of as a set of actions and responsibilities that a user can perform within the context of an organization [FK92]. Instead of specifying all access rights for every single user, role-based policies specify the access authorizations on objects for each role. Users are then assigned to roles. A user playing a certain role is allowed to execute all actions the role is authorized for [SS94]. Thus, the main concern within a role-based system is protecting the integrity of information by asking who can perform what acts on what information [FK92].

In the next subsections, we will discuss the basic elements and their associations among each other and will unveil the three main principles of role-based access control systems.

2.2.1 Basic Elements

A RBAC model defines three central elements: *Users*, *roles* and *permissions* [Fer01]. A *user* is a human being, any person who interacts directly with a computer system [SCFY96]. A *role* is a job function or a title within an organization describing the authority and responsibility assigned to a member of a role [FSG+01]. It can be seen as a semantic construct around which an access policy is built [Fer01]. A *permission* is defined as an operation on an object [FK92], i.e. a description of the type of authorized interaction that a subject can have with an object. Permissions are always positive and grant the holder the ability to perform some actions in the system. *Objects* are data object or resources within the computer system [SCFY96]. The basic concept of RBAC is that users are assigned to roles, which contain a collection of permissions. Users obtain permissions by being assigned to roles [Fer01]. User-role and permission-role assignments can be many-to-many, which means that the same user can be assigned to many roles and a role can have many users. Similarly, a permission can be assigned to many roles and a single role can have many permissions [FSG+01]. Figure 2.1 (p. 7) illustrates the facts.

2.2.2 The Security Principles

The role-based access control policy supports three well-known security principles: *Least privilege*, *separation of duties* and *data abstractions* [SCFY96]. The following sections explain these principles in greater detail.

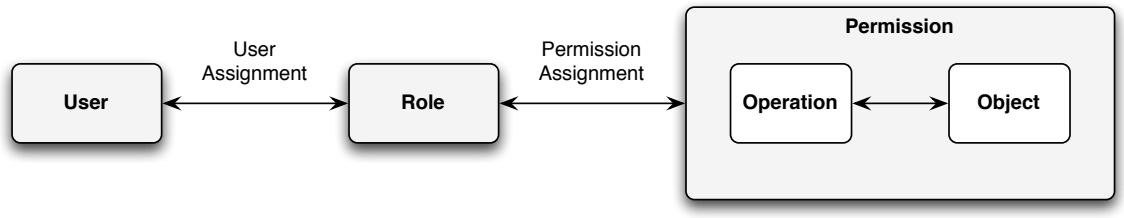


Figure 2.1: Basic elements of role-based access control (RBAC): *Users* (human beings), *roles* (functions within an organizational unit) and *permissions* (authorized operations on objects). In RBAC, users are assigned to roles (User Assignment), each of which contains a collection of permissions (Permission Assignment). Simplified figure inspired by [FSG+01].

Least Privilege

The principle of least privilege is one of the most important principles when designing protection mechanisms for secure computer systems [CC07]. This principle states that only those permissions that are required by the user to perform a task in a certain role are assigned to the role [SCFY96]. The system must guarantee that the user is only granted those permissions and no more [CC07]. Ensuring adherence to the principle of last privilege requires to identify the user's jobs and to determine the minimum set of privileges to perform the job [FK92]. Thus, it avoids the problem of an individual having the ability to perform unnecessary and potentially harmful actions [FKC07].

Separation of Duties

In some situations it might be required, that a sensitive task requires two or more individuals with mutual exclusive roles to be completed [SCFY96]. *Separation of duties* provides a powerful means of enforcing conflict of interest and other separation rules over RBAC elements [FSG+01]. A common example to illustrate this is a payment order that requires a role to initiate the payment and a role to authorize the payment [FK92].

Data Abstractions

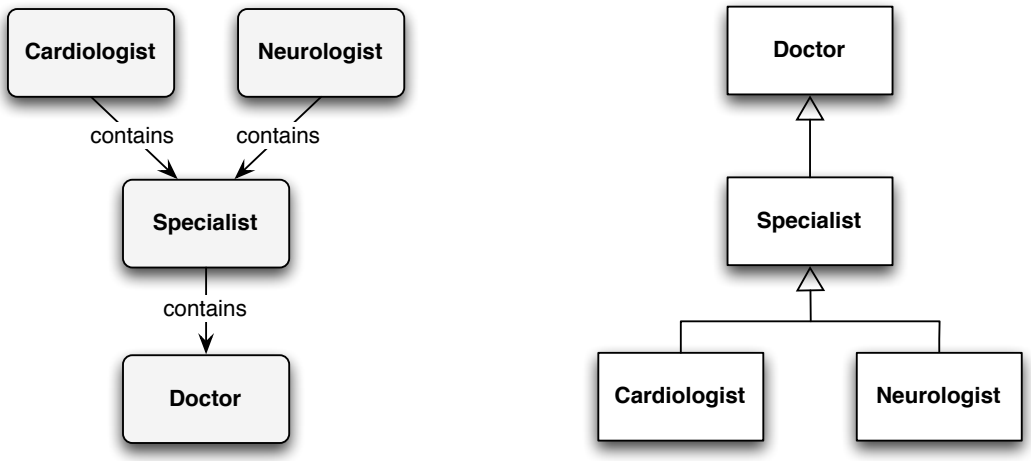
The third principle of RBAC is *data abstraction*. Instead of using the operating system's default permissions like read, write or execute, data abstractions allow the definition of abstract permissions such as credit or debit for an account object [SCFY96].

2.3 Roles and Role Hierarchies

As a major component of an RBAC system, role hierarchies go beyond the basic RBAC structures described in Section 2.2.1. The main motivation for role hierarchies is the observation that individual roles within an organization often have overlapping functions, responsibilities and duties, i.e. users in different roles may need to perform the same operations. Furthermore, there is a number of general operations performed by all or most employees within an organization [Fer95]. For example, general permissions may relate to operations such as accessing an internal Web page or sending an email [FKC07]. It would be inefficient and administratively tedious to specify these general permissions for every single role or to assign a large number of users to general roles. To improve the efficiency and to better reproduce the natural structure of an organization, RBAC

includes the concept of *role hierarchies*. A role hierarchy defines roles with unique attributes that may contain other roles, meaning that one role may implicitly include the permissions that are associated with another role [Fer95]. The collection of permissions needed for a job function can be defined by multiple subordinate roles, each of which can be reused in the sharing of common permissions [FKC07]. To illustrate the potential of overlapping permissions, consider four roles - cardiologist, neurologist, specialist and doctor - as depicted in Figure 2.2 (a). Because the cardiologist and neurologist are both specialists, it is appropriate to assume, that all permissions assigned to the specialist role would be assigned to the cardiologist and neurologist roles as well. Additionally, since a specialist performs many of the duties of a doctor, the permissions that are assigned to the doctor role would also need to be assigned to the specialist role. To emphasize the specialty of the cardiologist and neurologist roles, each of these roles has, besides the common set of permissions, a unique set of permissions necessary to perform their duties. In Figure 2.2 (a), the most powerful (senior) roles with the largest number of permissions are represented at the top of the tree, the less powerful (junior) ones are placed at the bottom [Fer95]. Senior roles contain more permissions than junior roles [NC00]. In our example, the roles neurologist and cardiologist inherit all permissions from the roles specialist and doctor. Due to the fact that the inheritance of permissions is transitive, any user who is assigned to the role cardiologist is *authorized* for the permissions that are associated with the role cardiologist as well as permissions associated with the roles specialist and doctor [FKC07]. Note that not all roles have to be related. The roles neurologist and cardiologist are not hierarchically related but they may share some or all of the same permissions [Fer95].

Figure 2.2 (b) shows a generalization approach discussed in [ML99]. This approach describes a *is-a* relationship between the roles based on their competencies: Cardiologist *is-a* specialist and specialist *is-a* doctor. Each of these roles is more general than the previous one. Some of the roles in the *is-a* hierarchy may be *virtual*, i.e. there are no users assigned to them. They are only defined to aggregate characteristics shared by the roles further up the *is-a* hierarchy [ML99].



(a) Role Hierarchy (b) Generalization

Figure 2.2: **(a)** shows a common way of representing role hierarchies by means of a *contains* relationship. The most powerful (senior) roles are shown at the top, the less powerful (junior) ones at the bottom. Senior roles inherit permissions from junior roles, i.e. they contain more permissions than junior roles. **(b)** depicts a *is-a* hierarchy using *generalization* to describe role relationships. In this example, a cardiologist *is-a* specialist, who itself *is-a* doctor. Figures inspired by [Fer95] and [ML99].

As you may assume correctly, with this short section on security systems we only scratched the surface when it comes to RBAC. The scientific work on role-based access control systems presents a lot more details, facts and sophisticated approaches to deriving roles from organizational structures and building roles hierarchies. However, a more detailed discussion on this topic would go beyond the scope of this thesis. In the next section, let's focus on the delegation of access rights in RBAC systems.

2.4 Delegation within RBAC

As discussed in section 2.3, a senior role in RBAC systems inherits permissions from a junior role as a consequence of a role hierarchy. In case a senior role fails to operate, it may occur that a junior role may not be able to continue its work because of the missing permissions assigned to the senior role. To ensure continuous operation, a junior role must gain the privileges reserved for a senior role, which are not granted to it explicitly. This can be accomplished by delegating access rights [NC00]. A *delegation* is a mechanism of assigning access rights to a user, i.e. it allows a user to assign a subset of his available access rights to another user. However, the user performing the delegation must be able to use the access rights. Delegation of access rights may occur in two forms in RBAC: delegation of individual *permissions* or delegation of *roles*. Delegating a permission gives the delegatee the ability to use the delegated permission. We note that individually delegating all permissions explicitly assigned to a role does not authorize the delegatee to act in that role. In order to be able to delegate a permission, the permission collection assigned to the delegator must contain the permission to be delegated [CK08]. A role delegation allows a user to delegate a role to another user in order to authorize the delegatee to perform operations that require permissions of the delegated role. As with permissions, a user is only authorized to delegate roles that are assigned to him [NC00]. To illustrate this, let us consider the following example. Imagine a hospital environment where a pharmacist has the permission to prepare medicine for patients, but a nurse does not have such a permission. In an emergency situation a doctor may request the nurse to prepare medicine for his patients. In this case, he delegates the pharmacist role, which a doctor role naturally contains, to the nurse who is then authorized to provide the remedy [NC00]. In other words, a delegated role gives the delegatee the ability to act in that particular role - the delegatee is authorized for the delegated role and thereby gains the ability to use the permissions assigned to it as well as permissions of roles that are junior to it [CK08].

Finally, delegation of privileges can be classified into two categories: *grant delegation* and *transfer delegation*. A *grant delegation* allows a delegated access right to be available to both the delegator and the delegatee. In *transfer delegation* models, on the other hand, the ability to use the delegated access right is transferred to the delegatee - the delegated access right is no longer available to the delegator [CK08]. As we will see in later chapters, the grant delegation will be relevant for our work.

Chapter 3

Requirements

"It is easier to change the specification to fit the program than vice versa."

Alan Perlis

After a theory block on role-based access control in Chapter 2, this chapter discusses the requirements for the Faculty information management system Merlin. We will start with the *general requirements* by illustrating the system context of our Web application. The second part of this section focuses on the description of general roles that can be adopted by various users while interacting with the system. In the second section, we will concentrate on the *functional requirements* to sketch out Merlin's core functionality by discussing general use cases. To complete this chapter, we present the *non-functional requirements* in the last section.

3.1 General Requirements

Since Merlin is intended to be embedded into an existing application landscape, there are some general requirements to be followed. By having a closer look at the system context, we will be able to identify the necessary interfaces needed to ensure the interoperability between Merlin and the present systems. A study of the University of Zurich's organization structure will help us to determine the various roles that employees must be able to adopt in order to accomplish their duties.

3.1.1 System Context

As already mentioned before, the publication management system Merlin is supposed to be designed in a way to be able to exchange data with other system in the University of Zurich's application landscape. Figure 3.1 depicts this situation. The following paragraphs describe each system separately and explain the role they play in the overall context.

ZORA

ZORA[1] is an acronym for *Zurich Open Repository and Archive*. It provides open and worldwide access to the research and scholarly output of the University of Zurich with the focus on qualified

[1] http://www.zora.uzh.ch

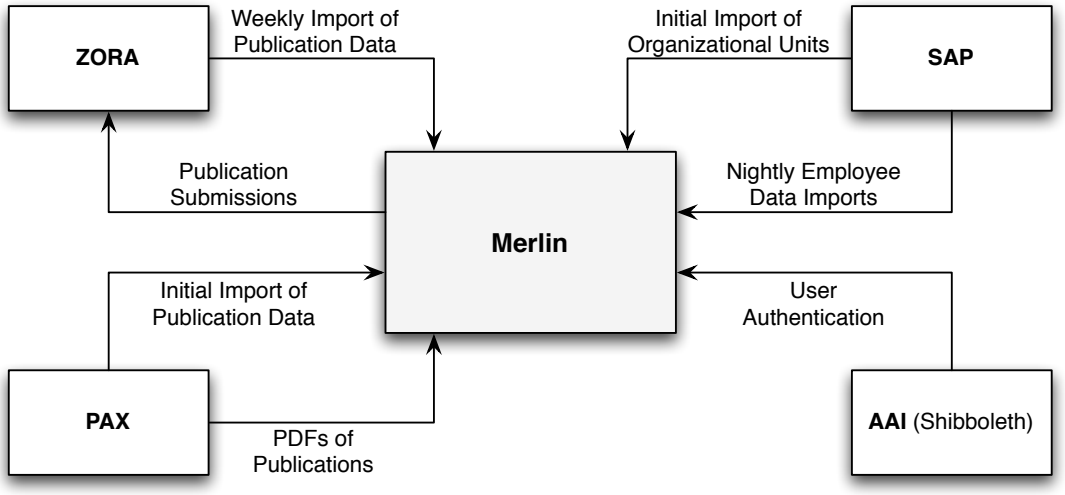


Figure 3.1: System context of Merlin: The University of Zurich's enterprise resource planning system *SAP* supplies Merlin with employee data, whereas the *AAI* service helps to authenticate its users. On the other hand, *ZORA* and *PAX* are used to exchange publication data.

scientific publications [zor10]. ZORA is based on the open source software EPrints[2], which has been modified to the needs of the University of Zurich. Besides being a repository for scientific work published at the University of Zurich, ZORA's main target is to guarantee a high quality of the published work. To achieve this, every single publication submitted to ZORA is reviewed by a team of editors who are responsible for correcting the submitted meta data and adding missing information if necessary.

In order to present a complete academic record for every employee of the Faculty of Economics, all publications from the ZORA repository are initially imported to Merlin. The main purpose of Merlin is to give its user the ability to enter new publications and submit them to ZORA for quality feedback. Changes to submitted publications made by ZORA editors are updated in Merlin by automatic imports from ZORA on a weekly basis. For further details on data exchange please refer to Mark Odermatt's original work [Ode10].

PAX

PAX[3] is a publication management system developed at the Department of Informatics of the University of Zurich within the scope of a diploma thesis. The *Semantically Annotated Publication Database* uses AJAX[4] to provide a rich user interface experience. It has been used for three years for management of various publication types published at the Department of Informatics.

The main purpose of Merlin is to redeem PAX as a publication management system. For a successful implementation of Merlin, all data from PAX has to be migrated to Merlin. Mark Odermatt's Master Thesis [Ode10] describes the approach taken to transfer all publication meta data including their PDF files to Merlin.

[2] http://www.eprints.org
[3] http://www.ifi.uzh.ch/pax/index.php/
[4] AJAX is a JavaScript and XML-based technology that uses asynchronous communication between client and server to create client-side dynamic and interactive user interfaces for Web applications [Her07].

SAP

SAP[5] is as well-known enterprise resource planning software widely used in the industry. It is the University of Zurich's main human resources, cost center and organization structure management system as well as an administration main frame for semester courses offered by the departments.

Since Merlin is an information management system developed for the research staff of the University of Zurich's Faculty of Economics, Business Administration and IT, SAP supplies Merlin with personal employee data, which are essential for user authentication. Further, it is necessary for Merlin to know the internal organization structure of the faculty in order to be able to assign published research work to the respective organizational unit. To achieve this, organizational units are initially imported from SAP. As with the organizational structure, after an initial import, nightly employee data imports are responsible for maintaining the employee's personal data up-to-date. It is a great concern to Merlin to keep the data quality as high as possible and therefore, only a restricted subset of the user's profile data can be actually edited by the user. This aims primarily to avoid the maintenance and synchronization of employee data on several systems. Details on implementation and used data formats are presented in Chapter 4.

AAI (Shibboleth)

AAI[6] is an acronym for *Authentication and Authorization Infrastructure* provided by the Swiss foundation SWITCH. AAI is one of many network-based, high-value services offered by SWITCH to all Swiss universities. It represents an easy and secure single sign-on mechanism for users to access different Web resources. The benefit of this technology is that a user registers only once with a *home organization* to which he is affiliated. The home organization is then responsible for maintaining the user related information and for providing him with his credentials – with other words, it carries out his *authentication*. The user's home organization can also provide additional information about the user to the resource upon resource's request and user's consent. The result is that all AAI-enabled resources are available to a user with a single set of credentials [aai10].

AAI's architecture is based on the Shibboleth open source project, which is a standard based software package for Web single sign-on across or within organizational boundaries. It implements the widely used identity standard SAML (Security Assertion Markup Language) to provide single sign-on and attribute exchange framework. Shibboleth simplifies management of identity and permissions for organizations by providing extended privacy functionality to control the attribute released to each application [shi10a].

There are two primary parts to the Shibboleth system: *identity provider* is a software run by an organization with users who wish to access a restricted service. After a successful authorization with his organizational credentials, the organization (identity provider) passes the minimal user's identity information necessary to a *service provider*, being the software run by the provider managing the restricted area, to enable an authorization decision [shi10a]. In our case, while the University of Zurich plays the role of the identity provider, the faculty information management system Merlin represents the service provider.

3.1.2 Roles

Merlin offers a wide spectrum of different functions, but, as with other modern systems, not all functions are available to all users. At the same time, not all users require the same amount of functionality to be able to accomplish their duties. In a real organization, every function has a

[5] http://www.sap.com/index.epx
[6] http://www.id.uzh.ch/dl/mobil/AAI.html

different area of responsibility. This fact gives rise to various roles that offer their assigned users a tailored subset of functions in order to be able to interact with the system the way they need.

Basically, Merlin distinguishes between five different roles: a *visitor* can browse the repository, search for publications or view information about an organizational unit without the need to be authenticated. Members of the faculty who act in the role of an *employee* have to log into the system first. They are provided with a custom profile page where they can manage their personal data. The *faculty* role has the access rights for entering new and editing or deleting existing publications. Users with this role are active researchers and obliged to confirm their personal and publication data for each reporting period. A *chair* is the head of a team and is able to manage his organizational unit. Finally, an *admin* is a system administrator, who is allowed to perform any operation offered by Merlin. Additionally, he is provided with functionality such as CSV data exports.

3.2 Functional Requirements: Use Cases

This section describes the functional requirements of Merlin in form of use cases. As already stated in Section 3.1.2, employees can interact with the system in different ways by being assigned to various roles. Each role offers a specific subset of functions that allow the associated users to carry out their work. To reflect their central importance, the structure of the following sections is based on roles. The description of the use cases will help us to understand the needs of each role and to define the scope of the required functionality.

Based on the theoretical background from Chapter 2, the roles presented here are organized in a hierarchy, as we will se in Chapter 4. As already discussed in Section 2.3, senior roles inherit access rights from junior roles and thus are allowed to perform operations that are explicitly permitted to junior roles. With this in mind, the role faculty, for example, is allowed to perform all actions of role employee, the role chair is allowed to access at least the same resources as the role faculty. Figure 3.2 provides an overview of the involved roles and their required use cases. Note, that this illustration depicts only the use cases relevant for this thesis. For a complete picture, please feel free to also consult Mark Odermatt's Master Thesis [Ode10].

3.2.1 Visitor

A *visitor* is anyone who visits Merlin without the need of being authorized. The access rights are restricted to the public section of Merlin: visitors can browse the entire publication repository, search for specific publication types based on the entered search criteria and export the search result as BibTex. For more details on search, please refer to [Ode10]. Further, visitors are allowed to inspect the organizational hierarchy of University of Zurich, which is described in the next paragraph in more detail.

Browse Organizational Units

As part of the public section, a visitor is allowed to browse the organizational structure of the Faculty of Economics, Business Administration and IT without having to be logged in. A visitor can quickly and easily gain detailed information on a selected organizational unit, such as direct link to the homepage for additional information, number of publications published by the organization unit, its path through the organization chart or an elaborate list of employees with their functions, rooms, phone numbers, email addresses and publication counts.

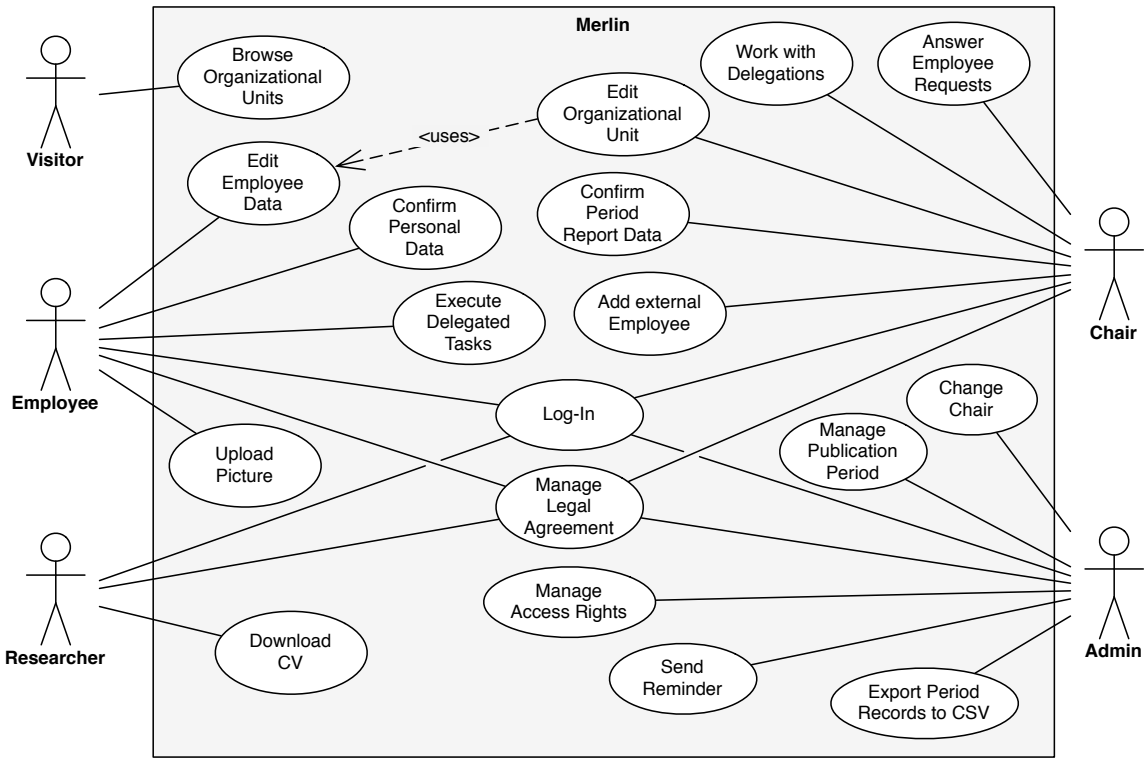


Figure 3.2: Overview of the required use cases. Each role has a defined set of responsibilities and access rights.

3.2.2 General Use Cases for Authenticated Users

There are some general use cases that are relevant for several roles. In avoidance of an unnecessary repetition in each of the affected roles, this section consolidates them in one place.

Login

Every employee attempting to access the protected area of Merlin has to login first. In order to be able to authenticate users from the Faculty of Economics, Business Administration and IT, and to provide them with their proper access rights a login action is required. As we will see in Chapter 4, the database of Merlin contains all employees of the faculty. After clicking on the login button, the user is redirected to the AAI Login[7] of the University of Zurich, where he has to enter his credentials. This login procedure is performed by the user's home organization, i.e. the identity provider based on the Shibboleth technology as discussed in Section 3.1.1. After a successful authentication with the identity provider, the user is redirected back to Merlin. Behind the scenes, the user's home organization encloses a number of user attributes in the login request. This part of the authentication aims on the user's identification at the University of Zurich and checks, if the user has a valid UniAccess credentials. Based on the attached attributes in the request and the persistent data of the Merlin database, the system is able to identify the user and to decide, whether to provide access to Merlin or to decline the user's request.

[7] https://aai-idp.uzh.ch/idp/Authn/UserPassword?lang=en

Answer Legal Agreement

After a successful first login in Merlin, the user has to accept or decline the Merlin's legal agreement first. The legal agreement states the general purpose of Merlin, explains what data is required from the user and how it will be processed. If the user accepts the legal agreement, he is redirected to his personal home page in Merlin and is able to access all resources according to his access rights. In case of a declination, the user is logged out and is not allowed to enter the protected area of Merlin. Furthermore, the chair of the user's organizational unit is informed about the refusal via email. If no affiliation is defined for the employee, the email is sent to the Merlin administrator.

Withdraw Legal Agreement

In case an employee decides to withdraw the legal agreement after he has accepted it, the system must provide a corresponding functionality to support the user's decision. This mechanism is supposed to protect the user from data abuse, i.e. if the user does not feel comfortable about his data being stored on the servers of the Department of Informatics or if he does not agree with the way his data is further processed or distributed by the University of Zurich.

After a withdrawal of the legal agreement, the user is logged out of the system and is no more able access the protected section of Merlin. As with the declination, the user's organizational unit head is informed about the withdrawal via email. This helps the chair to keep track of the happenings in his organizational unit.

3.2.3 Employee

The role *employee* represents an employee who is appointed at the Faculty of Economics, Business Administration and IT and has a valid UniAccess[8] account with a corresponding password. In order to be able to access the protected area of Merlin, a user has to authenticate himself with his credentials. Faculty staff such as librarians, administration or technicians act as an employee.

Edit Employee Data

Every employee of the faculty working with Merlin has an own profile containing his personal and publication data. To get a complete profile, each employee has to go through a setup wizard after his first login, which asks him to enter some required data concerning his share of activities, last appointment, education details and the organization unit he is affiliated to. The user can complete, edit or update the entered data anytime after a successful login.

Upload Picture

As with every modern Web 2.0 application, a user's profile has to provide the possibility to upload a profile picture. This helps a user to customize his profile and makes it appear more vital at the same time. Further, as we will see in section 3.2.4, a user portrait is a prerequisite for a complete curriculum vitae. The system is supposed to support various file formats such as .png, .jpg, .gif or .bmp and to adjust the size of the image automatically to some predefined formats.

[8] http://www.id.uzh.ch/dl/mobil/uniaccess.html

Execute delegated Tasks

Sometimes it may occur that a chair has to enter new publications into Merlin but is prevented from doing so by other important duties. In this case, he must be able to delegate a task to any employee within his organizational unit. As we will see in later sections, a task can be considered as an internal message sent from a sender to a recipient and comes always with the corresponding access rights needed to perform the described operations. Hence a chair can delegate the creation of a new publication to an employee, for example a secretary, who does normally not have the required access rights because of her assigned role *employee*. After accepting the task, the delegatee is assigned the necessary access rights and is then authorized to perform the task. When the task has been completed by the delegatee and confirmed by the delegator, the access rights are removed from the delegatee. This is to avoid a dilution of the access control system.

Confirm Personal Data Correctness

One of the main purposes of Merlin, besides being a faculty information management system, is to collect and consolidate publication period data for EQUIS[9] and AACSB[10] certificates[11]. Before the deadline of a publication period, every employee has to complete and confirm his personal data. This data is collected and further processed with the objective of a period report, which will be committed to the certification authorities mentioned above. Based on this report, the authorities decide whether to award an accreditation or not.

Before an employee confirms his personal data, i.e. at the beginning of each publication period, the system creates a copy of the user's profile. This so called *employee record* can contain different information than the actual employee profile. The rationale behind this decision lies in the fact that an employee could, for example, have earned his PhD degree and updated his profile accordingly at the time of his confirmation but has to be reported with his master's degree in the last publication period.

By pressing the *confirm* button, the employee ensures, that the entered data are complete and correct for the respective publication period. Thereafter, the employee record can no longer be edited by the user.

3.2.4 Researcher

Employees assigned to role *researcher* are part of the active research staff of the faculty. One of their main duties, besides teaching, is to conduct research and to publish they research work. Teaching and research assistants or professors count as faculty. The personal profile of the researchers contains additional information about the share of their activities. Furthermore, researchers are able to enter new and edit or delete their existing publications [Ode10].

Generate CV

It is common for the members of the research staff of the University of Zurich to publish a short curriculum vitae on their personal home page. Because Merlin stores all employee's personal and publication data, it is obvious to offer its users such a function. As a consequence, every user with a Merlin account should be able to download his personal academic curriculum vitae in form of a PDF file that is automatically generated by the system at the touch of a button. Besides the

[9] http://www.efmd.org/index.php/accreditation-/equis
[10] http://www.bestbizschools.com/AACSB-Accredited/default.asp
[11] For further information on EQUIS and AACSB accreditation please refer to Mark Odermatt's Master Thesis [Ode10]

profile picture, personal data and educational information it should contain the top 5 publications selected by the user.

3.2.5 Chair

Chair is a special role dedicated to professors, who are head of a research group or of an institute. As a head, a professor has additional duties such as managing the research group and its projects, leading the employees, monitoring the personal employee data maintenance or supervising all publications of his research group.

Answer Employee Requests

One of the Merlin's concerns is to care about data quality. This is very important when it comes to accreditations such as EQUIS or AACSB, because those certification authorities expect a superior data quality. In order to achieve this, Merlin is supposed to offer mechanisms to supervise the data flow. One of these mechanisms is the *employee request*. After a first login, the employees are expected to enter their personal data and to select the organization units they are affiliated to with the help of a wizards. The latter shall generate an employee request for each organization unit the user wants to subscribe to and send it to the assigned chair. The chair is then supposed to accept or to decline the employee request and therefore is responsible for maintaining the quality of his organizational unit as high as possible.

Work with Delegations

As already mentioned in Section 3.2.3 Merlin must offer a chair the possibility to delegate his work to other employees in his organizational unit. There should be two kinds of delegation available: a *task* and an *access right*.

A *task* represents a temporary access right that is supposed to be delegated to an employee with a corresponding task description. It should support the chair of an organizational unit in a onetime manner, meaning that if a chair needs to quickly delegate some work to another employee, he should just be able to create a new task with an appropriate description and send it to the desired recipient. Besides the description and the due date, a task should have one of the predefined scopes: create publications, edit employees of an organizational unit, create and edit external employees or edit organizational unit. Based on the scope of the task, the system is supposed to attach the required access rights – apart from access rights he or she already has –, so that the recipient of the task is actually able to accomplish it. If a task is accepted, the enclosed access rights will become valid. After the task is completed and *confirmed* by the chair, the validity of the enclosed access rights shall expire.

The delegation of an *access right*, as opposed to tasks, represents a *permanent* allocation of permissions to a selected user. Similar to tasks, the chair is supposed to select a delegatee and one of the four predefined scopes, which he or she would like to delegate. In contrast to tasks, the delegated access rights are supposed to be valid as soon as the delegation is submitted and should remain valid until the delegation is removed from the delegatee by the chair or an administrator.

Further, the system should be able to show all access rights assigned to a user for a better traceability. In addition, such a view can help a chair or an administrator to manage the user's permissions and to decide, whether a new task or access right delegation is necessary for an upcoming assignment.

Edit Organizational Unit Data

Basically, every team head is responsible for the management of his organizational unit. In Merlin, every chair should be able to edit the data of his or her organizational unit, such as the name, homepage, and, in case of a research group, academic info details (e.g. conference organized, faculty research seminars or scientific presentation for external audience) or current academic activity (e.g. professional experience, consulting or visiting professors). This data is not relevant for period reports but has a great value for internal use.

Additionally, after being nominated a chair by an administrator, the user automatically gains the required access rights that guarantee the ability to edit personal data of all employees in his organizational unit. This is justified by the fact that a chair must be able to edit every single peace of information concerning his unit. If an employee presents incorrect data in his personal profile, the chair must be able to correct it. Therefore, it serves as a control and data quality assurance mechanism at the same time.

Confirm Period Report Data

Before the deadline of each reporting period, every employee has to check and confirm his personal data stored in his employee record. As already stated before, it is a great concern to Merlin do keep the data quality as high as possible. Thus as a first quality check, before further processing of the employee records for a period report, the chair is supposed to examine and approve the employee's data. In case the employee should not have confirmed his data just before the period deadline, the chair must be able to remind him to do so. If the chair should not agree with the employee's entered data, he should be able to adjust it according to his point of view. Only data approved by the chair of the organizational unit shall be used for the period record.

Add external Employee

Besides conducting research, a research group's duty is to teach. It is common that external lecturers are commissioned to run a course offered by the research group. Thus, Merlin must provide the ability to add external employees and to trace their employee record for the EQUIS and AACSB accreditation. An external employee is supposed to be added by a chair of by administrators.

3.2.6 Admin

Based on the faculty's policy, an *administrator* can be any employee. Acting as administrator in Merlin means to have access rights to all resources and being able to execute all functions offered by Merlin. For example, an administrator is allowed to edit the personal data of every single employee, create, edit and delete publications, add new external employees to research groups or assign access rights to users.

Change Chair

To reflect the correct structure of the organizational units, an administrator should be able to set and change a chair for a selected unit by choosing an employee from the internal database. This employee should then gain additional access rights in order to be able to perform the duties of a team head. Inherently, a chair can be any user. Every chair can be the team head of several organizational units, but each unit can only have one chair.

Send Reminder

To support the admins in their work and to keep track of the data quality, Merlin should provide some management functions, which help the admins to supervise the users actions. It is a requirement that admins are able to monitor the acceptance of the legal agreement and to send *agreement reminder* to those employees, who did no yet answer it, i.e. did not log in for the first time. Further, an admin, like a chair, should be able to see whether all users confirmed their personal data for the current reporting period and if not, should be able to send a *personal data reminder* to all of them with a touch of a button. Last but not least, admins should be able to *remind* all employees who did not check their *publication data* for the period to be closed.

Manage Publication Period

Admins should be able to manage a publication period. As already stated, a publication period should have a deadline. On this date, all employee and publication data should be checked by the users and examined by the chairs. This is to ensure a high quality of data that is required for the creation of the annual report submitted to EQUIS and AACSB. Therefore, Merlin has to provide an interface to *set a deadline*. In addition, an admin should be able to *switch period* in order to be able to view the summary and the users confirmation overview of a past period. Finally, at the turn of the year the system is supposed to automatically create a new publication period and all corresponding employee records based on the employee's profile. For other details on publication period, see [Ode10].

Export Period Records to CSV

In order to be accredited for EQUIS and AACSB, the Faculty of Economics, Business Administration and IT has to submit annual reports to these certification authorities. To achieve this, Merlin has to provide an intuitive interface to export all required data in a simple way. With that said, the admins should be able to export all available *employee records* and *employee publications* in a structured CSV format.

The export result of the *employee records* should reflect the organizational structure of the faculty. Every organizational unit should be listed with its full name and contain all employee records of the affiliated employees. An employee record should present the user's personal data such as first name, last name, year of birth and should contain his or her share of activities performed at the University of Zurich as well as the appointment details.

Employee publications are supposed to be a customized output for each of the accreditation authorities. Every authority prescribes a number of publication types that have to be submitted and are essential for the accreditation request. Consequently, Merlin has to provide a mechanism to sort each employee's publications and sum them up to the required categories. Mark Odermatt's master thesis discusses the further requirements [Ode10].

Data Setup

A nice-to-have requirement would be the ability to setup the production environment in a customized way, so that the admins can select which data to load. Furthermore, after a successful deployment of Merlin, there is a great quantity of data to be initially loaded into the production environment. To separate the deployment from the setup of production data, Merlin should provide an interface for admins where they can choose whether they want to load employee data, publication data or both. Note, that this concern does not have any business characteristics, but has a great technical relevance and offers a setup convenience and the same time.

3.3 Non-Functional Requirements

After the introduction of functional requirements in the previous part, the last section of this chapter discusses the non-functional requirements. First we present the core technology that is supposed to be used for the development of Merlin and continues by describing further quality properties, which are expected to be met.

3.3.1 Core Technology: The Web Application Framework Grails

Probably the most important non-functional requirement of Merlin addresses the technology, which should be used for the implementation of the application. Merlin is supposed to use a modern technology that supports agile software development. One of those modern technologies that addresses this need is the Web application framework *Grails*[12]. Grails implements most of the principles known from the Ruby on Rails community and is based on an object-oriented programming language called Groovy – a dynamic Java. We will talk about Grails in more detail in Chapter 4. One of the leading reasons for the choice of this technology is the existing experience with Grails among the members of the s.e.a.l research group. This is an important fact, because the requirements in a life-cycle of such an application can change very quickly in today's turbulent environments [Hig06]. To support the maintainability and extensibility of Merlin a familiar technology should be chosen. This makes the implementation of new requirements easier.

3.3.2 Maintainability

Maintainability is always one of the main issues when developing a new software. Since Merlin is supposed to be operated for some time, it is required that the application is maintainable and flexible enough to support the implementation of new requirements. To facilitate this, the architecture of Merlin must be adoptable to changes that naturally occur in its environment due to evolving business needs. Further, the code of Merlin should be well structured, readable and extensible. Abelson and Sussman's quotation gets to the heart of this issue:

"Programs must be written for people to read, and only incidentally for machines to execute."

Finally, a good documentation of the inner life of Merlin helps to understand the rationale behind architectural decisions and explains the thoughts of the developers. It is an important entry point for everybody who wants to get an overview of the system or has to implement new features based on requirement changes.

3.3.3 Intuitive User Interface using modern Technologies

Merlin should be developed for a large basis of users with different computer skills. Therefore we are very concerned about the *usability* and *intuition* of the user interface. Wherever needed, descriptive texts should help the user to understand the presented information. To support this, a clearly structured layout and modern technologies such as JavaScript or AJAX should be used.

Javascript frameworks help to simplify the development and design of a desktop like user interface with interactive elements, which make the user experience more balanced, realistic and sophisticated. As an example, Merlin should present a hierarchical structure of the faculty's organization. It is obvious that an interactive tree view known from common file browsers is best suited for this kind of requirements.

[12] http://www.grails.org

Ajax is a JavaScript and XML-based technology supporting asynchronous communication between a server and its clients [Her07]. It is commonly used to enable HTTP-requests that update a part of a Web page without the need of reloading the entire HTML document. This technology can be used for *autocompletion* that supports the user during data entry. For example, if a user wants to update the organizational units in his profile he only needs to type the first letters of the unit name he is looking for and the system will autocomplete it. Behind the scenes, a HTTP-request is generated and sent to the server, which returns a list with results matching the user's input. Another usage of AJAX are *pop-up windows*. After the first login, the user is supposed to enter his personal data with the help of an assistant. To make this task more attractive, a pop-up based wizard can support the user in an interactive way.

Finally, to avoid an information overload, the data should be presented to the users in a coherent and structured manner. Thus, large forms and data amounts should be divided into clear and meaningful subsections.

3.3.4 Data Quality

As already mentioned several times, a high data quality is a very important characteristic of Merlin. To enforce this, the forms in Merlin should support the user's input by defining required attributes that must be provided. If some of the attributes should be missing, Merlin should not be able to save the resource and should show a comprehensible message at the same time. Further, some data may not be edited by the user. For example, the user's first name and last name are imported from the University of Zurich's human resources management system SAP. In avoidance of a decentralized data maintenance and unnecessary data synchronizations, this data shall be imported from SAP on a daily basis. So if a user wants to e.g. change his last name, the data update must be performed in SAP and is then automatically imported into Merlin.

3.3.5 Corporate Design

Because of the large number of approximately 600 users that are going to use Merlin on a nearly daily basis, the acceptance of the application is a great concern to us. Since Merlin will be accessible through a subdomain, it must follow the design rules of the University of Zurich in order to integrate seamlessly into the existing application landscape. At the time of development, the University of Zurich published a new corporate design guidelines, which are publically available on the Web[13]. We are proud to announce, that Merlin is one of the first Web applications implementing the new corporate design.

[13] http://www.cd.uzh.ch/index.html

Chapter 4

Implementation

"It's not a bug - it's an undocumented feature."

Anonymous

In this chapter, we describe the implementation of the requirements discussed in Chapter 3. We start by presenting the Web application framework Grails – the core technology used to develop Merlin. In this part, we summarize the main principles of Grails, then give an overview of the architecture and continue with some facts on the programming language Groovy. In the second part of this chapter, we unveil the security concept and present its implementation in detail. Subsequently, we concentrate on the domain model and describe some of the core business objects. Finally, we explain the data importers and exporters.

4.1 Agile Web Development with Grails

As already stated in Chapter 3, one of the main reasons for the choice of a particular technology is the experience that a development team has already gathered thanks to previous projects. The members of the Software Evolution and Architecture Lab, who are supposed to maintain and evolve Merlin, are already experienced Grails developers. Therefore, Grails is the technology to develop with.

The essence of Grails, as stated by the Grails project leader Graeme Rocher, is to dramatically simplify enterprise Java development and to take the Web development to the next level of abstraction. The developer should be able to utilize the richness and maturity of the underlying technologies, which we will get to know in a second. Finally, Grails shall bring back the fun of development on the Java platform [RB09].

4.1.1 The Principles

Grails implements many of the ideas introduced by the Ruby on Rails framework[1]. Grails should not be considered just a port of the Ruby on Rails framework to the Java platform, since Grails was influenced and inspired by many other platforms including Python, PHP and Java [LS09]. The following sections describe some these principles.

[1] http://rubyonrails.org/

Convention over Configuration

Convention over configuration is a design paradigm that helps to decrease the number of decisions a developer has to make in order to gain simplicity but maintain flexibility at the same time. In Grails, there are only a few configurations files. Grails makes most of its decisions based on sensible defaults based on the source code. For example, if you add a domain class called `User`, Grails will automatically create a table called user with all appropriate attributes in the database. Further, Grails exposes all actions defined in the controller `user` through the URL `/appName/user/action` and renders the corresponding views from the `/views/user` directory.

Note that this principle is called convention *over* configuration, not convention *instead of* configuration. This means that if a developer needs to adjust the defaults or use an existing Hibernate (we will talk about Hibernate in Section 4.1.3) configuration, Grails won't stay his way [LS09].

DRY: Don't Repeat Yourself

Don't repeat yourself is another design paradigm that states that every peace of knowledge must have a single, unambiguous, authoritative representation within a system [HTC99], i.e. should be expressed in just one place [LS09]. This makes the maintenance of code very simple, because to effect a change, the code must be modified only in one spot. The alternative is to have the same peace of code in two or more places. If a developer then wants to change one, he has to change all. If he forgets to change of one them, the system might fail to operate due to contradictions. That is why DRY is one of the most important principles [HTC99].

Agile Philosophy

One of the biggest concerns of Grails is to be an agile Web framework. Agility means developing features in short and quick iterations and involving customers for fast feedback. Only the customers can tell if the application that is being developed corresponds to their needs and whether it adds value to their business. Further, being agile means to posses the ability to quickly react to changing requirements in today's turbulent environments. [Hig06]. Grails supports fast iterations by enabling quick code changes while running. With Grails, we can add new controllers, views or taglibs without having to restart the whole application. This is a whole new level of agility in the Java Web application development [LS09].

Scaffolding and Templating

Grails comes with a set of predefined commands, which make the creation of the application, addition of new controllers or views really convenient. By taping `grails create-app app_name`, a new application is created with the necessary application structure. This is a huge achievement compared to developing an application with the Spring MVC from scratch, where you have to struggle with jar files, bean definition files, a set of web.xml customizations, hibernate configuration files or database-creation scripts.

Scaffolding is a fantastic feature that helps to generate a set of views and controllers on the fly based on the attributes defined in the business model classes and provides basic CRUD – create, read, update and delete – operations without having to write a single line of code [LS09].

4.1.2 The programming Language: Groovy

Groovy is an object-oriented, dynamic programming language for the JVM that aims to bring the expressiveness and benefits of languages such as Ruby, Lisp and Python to the Java platform while still remaining Java friendly. It has a very similar syntax to Java, which makes it easy for Java developers to learn [RB09].

Let us have a look at some characteristics and interesting features of Groovy. Unlike Java, which mixes primitive and reference types, in Groovy *everything is an object*. This is done by autoboxing: Groovy automatically boxes primitive types into their object equivalents and vice versa. This allows Groovy to support interesting concepts such as operator overloading (operators in Groovy are just method calls that follow a naming convention). Further, Groovy's *GStrings* support a concept found in many other languages such as Ruby or Perl called string interpolation. This concept allows the embedding of variables within a string. Another powerful feature of Groovy is the *metaprogramming* capability. It gives Groovy the ability to add behavior to classes at runtime. This is achieved by creating a MetaClass for every class loaded by the Groovy runtime that is used when dispatching methods to the class itself. The last feature worth mentioning are *closures*. They can be essentially seen as reusable code blocks that can be assigned to a variable. Listing 4.1 shows a simple closure example.

```
[1,2,3].each { item -> println item }
```

Listing 4.1: Simple Closure

4.1.3 The Architecture

After the discussion on the principles in the last section, this part presents the Grails stack and the architectural design decisions living inside the framework.

The Grails Stack: Living in the Java Ecosystem

The design choices made during the development of Grails reflect its nature: the approach taken tries not to reinvent the wheel but concentrates on leveraging robust and trusted frameworks. As Figure 4.1 illustrates, Grails is powered by some of the most popular open source technologies in their respective categories.

Hibernate[2] is an Object/Relational (ORM) Mapping tool for Java environments. ORM refers to the technique of mapping the data representation from an object model (class) to a relational data model (table) with a SQL- based schema. Hibernate does not only take care of the mapping from Java data types to SQL data types, but also provides tools for data query and retrieval [hib10].

The *Spring*[3] framework is a modular platform, which provides a range of services such as dependency injection (technique for decoupling highly dependent software components by externally injecting a reference to a service inside a consumer), inversion of control (moving the system's flow control from a central piece of code to a caller-callee paradigm, where the callee decides how and when to answer the request) or aspect oriented programming (isolating auxiliary functions from the main business logic in one place) [spr10].

SiteMash[4] is a Web-page layout and decoration framework that helps to create large sites consisting of many pages for which a consistent look and feel, navigation and layout scheme is required. Basically, it intercepts the delivery of the HTML that is being returned to the browser,

[2] http://www.springsource.org/
[3] http://www.springsource.org/
[4] http://www.opensymphony.com/sitemesh/

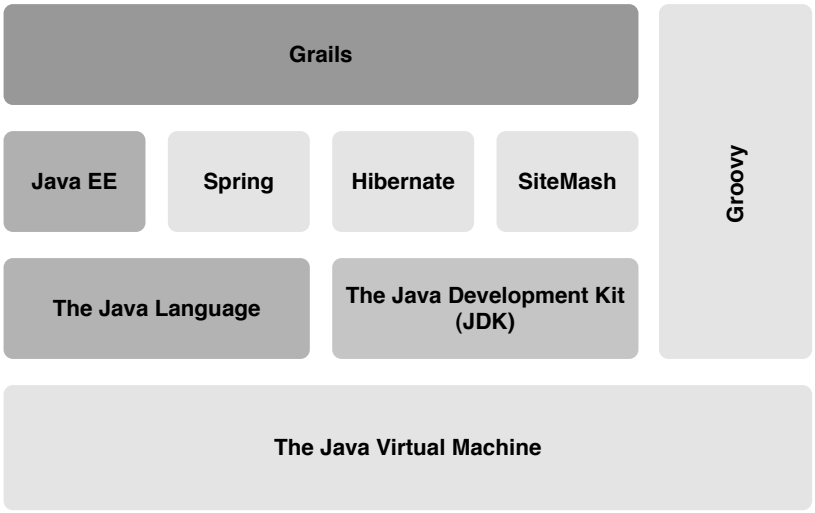


Figure 4.1: The Grails stack: Grails lives in the Java ecosystem and integrates some of the most popular open source frameworks such as Hibernate, Spring or SiteMash. The dynamic, object-oriented programming language Groovy was designed to seamlessly integrate with the Java Virtual Machine. Figure inspired by [RB09].

parses it, extracts the relevant content and generates an appropriate final page based on the decorator pattern [sit10]. Simply put, it is a stable and robust layout-rendering framework [RB09].

Grails was built to take full advantage of Java and the Java Virtual Machine. The main purpose was to reuse the amount of well-known and proven libraries that are available for the Java world without losses. The aim was to be able to implement the Web form processing classes in Groovy and, for example, the high-performance payroll calculations in Java [LS09]. Grails can handle both of them.

In addition, Grails has a plugin system built in that helps to provide solutions to extended problems that might not be covered out of the box [RB09]. Third party plugins such as the search plugin, which is built on full-text search engines Lucene[5] and Compass[6], or the scheduling plugin that is based on the scheduling service Quartz[7] emphasize the philosophy of using best-of-breed components [LS09].

Because Merlin is built with Grails, it adopts the architectural design of the framework. The following section describes the architectural patterns used in Grails.

Model-View-Controller

Grails is a Model-View-Controller (MVC) framework [RB09]. As depicted in Figure 4.2, the MVC pattern consists of three main components: the *model* refers to the representation of domain-specific business entities and functionality. The *controller* is instantiated upon a request and initiates the answer by making calls on model objects. It basically accepts user input and instructs the model and view to perform actions based on the input. Finally, the *view* presents the model in a form that is suitable for user interaction, in our case in a HTML document [TH07].

[5] http://lucene.apache.org
[6] http://www.compass-project.org/
[7] http://www.quartz-scheduler.org/

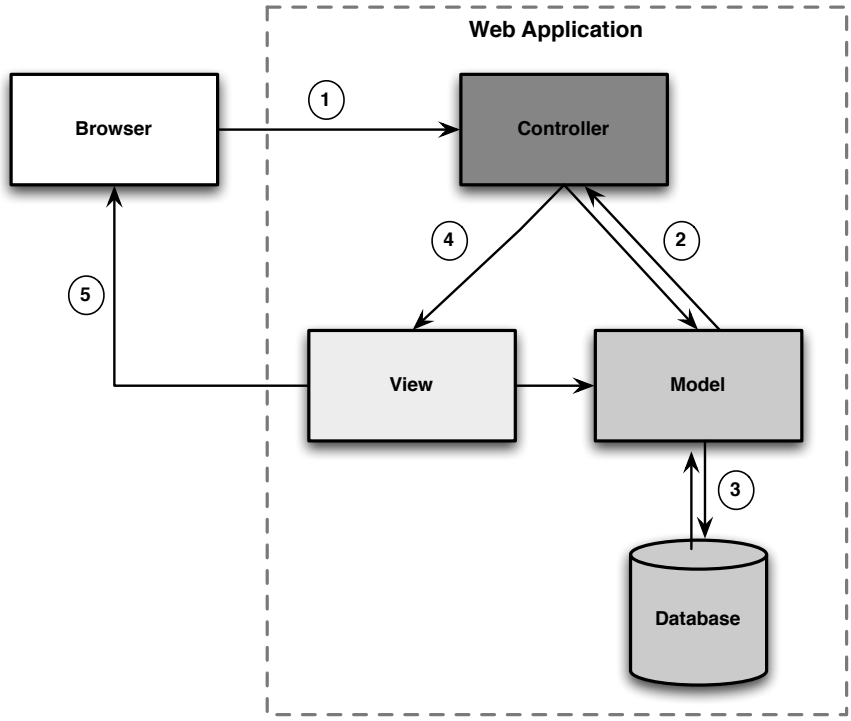


Figure 4.2: The Model-View-Controller pattern: (1) The browser sends a request to the Web Application. (2) The controller gets, updates, deletes existing or creates a new model object, which (3) is queried from or saved to a database using GORM (Section 4.1.3). (4) Then, the controller updates the view, which is rendered by the browser (5). Figure inspired by [TH07].

Domain Classes and Grails Object Relational Mapping

In an object-oriented application, a domain model represents the business entities that the application deals with. Developers often face difficulties in mapping the domain model objects to relational database. This is where Grails comes into play.

When we create a new domain model class, we create something that wraps a database table. *Grails Object relational mapping (GORM)* takes care of this. GORM essentially maps the name of the domain class onto a table name in the database. By default, all fields in a domain class are persisted, i.e. every property of a domain class is mapped onto a separate column with its type relating to SQL types. Each row in a table corresponds to a domain class object. But GORM has even more. Through Groovy's metaprogramming capabilities, every domain class is automatically enhances with a number of methods that allow us to query for domain instances. The simples of them is the `get()` method, which takes the identifier of a domain class and returns an instance if found in the database. Further methods are `save()`, `update()` or `delete()` just a name a few.

Dynamic finders are one of the most powerful concepts of GORM. They use the property names of the class to perform queries. Because they allow logical queries such as `AND`, `OR` and `NOT`, they are very flexible [RB09]. Listing 4.2 shows some examples. On line 1, we search for the employee with the name *Harald Gall*. The finder on line 2 returns all publications with word *Programming* in the title. The code on line 3 looks for all articles that have been created between today and last week.

```
1 def employee = Employee.findByFirstNameAndLastName("Harald", "Gall")
2 def publications = Publication.findAllByTitleLike("%Programming%")
3 def article = Article.findAllByDateCreatedGreaterThanOrEqual(new Date() - 7)
```

Listing 4.2: Dynamic Finders

Controllers

A Grails controller is a class, which is responsible for handling requests coming in to the application. The controller receives the request, processes it and finally decides what should happen next – it could execute another controller action, render a view or render information directly to the response. In other words, controllers are the orchestrators of a Grails application. They provide the entry point by coordinating the incoming requests, delegating to services or domain classes for business logic and rendering views [RB09].

A controller is prototyped. This means, that with every request a new instance is created. So we do not have to be as cautious about maintaining thread-safe code in a singleton controller. The class name of a controller must end with "Controller" by convention. Controllers do not need to extend any base class of implement any interfaces [RB09].

The default URL mapping configuration in Grails is simple. The first part of the URL corresponds to the name of a controller whereas the second, optional part of the URL corresponds to the name of an action defined in that controller. For example, the `employee/show/1` will map to the `show` action of the `EmployeeController` with a request parameter named `id` with the value of `1`. Specifying the action name is optional, that means if action name is left out of the URL, the default action for the requested controller will be executed (which by default is the `index` action) [RB09].

Services

A common pattern in the development of enterprise software is to provide layers of abstraction and reduce coupling between the layers within an MVC application. The so-called service-layer helps to encapsulate a set of business operations and therefore to centralize application behavior into an API, that can be used by other services and controllers.

Like other Grails artifacts services follow a convention and do not extend any base class. By executing the `create-service` command a new service class is created and put in the right place (grails-app/services) automatically.

Services in Grails are singletons by default, which means that there is only one instance of a service. To get a reference of a service within a controller, Grails uses Spring's *autowriting* concept as part of Spring's dependency injection that allows dependencies to be automatically injected by name or type [RB09]. Listing 4.3 shows, how this can be accomplished. As we can see on line 2, an instance of the `UserService` class is injected. We can then delegate the user creation to the `userService` by invoking the `create(...)` method (line 5).

```
1 class UserController {
2   def userService
3
4   def create = {
5     userService.createUser(params)
6   }
7   ...
8 }
```

Listing 4.3: Injecting a Service Instance into a Controller

Views

Grails uses GSPs (Groovy Server Pages) for its view layer and SiteMesh (4.1.3) to assist[8] in the page layout. GSP is very similar to JSP but uses Groovy instead of Java, which allows static and dynamic content to me mixed in the same document.

GSP has a number of built-in *tags* for performing basic operations such as logical if statements, looping or switching. Grails tags promote a clean separation of concerns and allow us to create well-formed markup [RB09]. The first example of listing 4.4 shows the *login* link, if the user is not logged in, i.e. there is no user in the session (line 1-3). The second example shows all publication titles of the `employee` instance (line 5-7).

```
1 <g:if test="${!session.user}">
2   <g:link controller="session" action="login">Login</g:link>
3 </g:if>
4
5 <g:each var="publication" in="${employee?.publications}">
6   <span class="tag">${publication.title}</span>
7 </g:each>
```

Listing 4.4: View Helpers in Grails: Tag Libs

A GSP *template* is a special GSP file, which contains only a fragment of a page. It can contain markup that is rendered from various places in an application in which case it the template would facilitate reuse. A template can be extracted from a page to simplify the containing page by breaking it down into smaller peaces [RB09]. Listing 4.5 shows, how Grails integrates the template `publications` located in the `employee` directory into a page by using the `<g:render>` tag.

```
1 <div id="publications">
2   <g:render template="/employee/publications"/>
3 </div>
```

Listing 4.5: Rendering the Publications Template

4.2 Security

This section presents the concept and implementation of the security architecture in detail. In the first part, we start with the description of the requirements that are necessary in order to meet the needs of the Faculty of Economics, Business Administration and IT. We then discuss the roles and their permissions needed to perform the duties described in Chapter 3, talk about the authentication and authorization procedures and explain the delegation of access rights. In the second part, we first deal with the domain model, then describe the SAP data importers and finally explain the functional principle of the CSV exporter.

4.2.1 Requirements

There is a wide range of requirements that should be satisfied in order to model the demands concerning security correctly. We will now look at them individually:

1. **Role-based Access Control**: As discussed in Chapter 2, role-based access control is the most suitable access control policy for the majority of organizations and is de facto standard when developing a security system for a Web application. It helps to map the organizational

[8] SiteMesh helps to merge all `.gsp` files into a file called `main.gsp`, which gives a consistent look to all pages [Kle09].

structure onto roles with corresponding permissions. Therefore, we want to implement a pure role-based access control without having to assign permissions directly to the users.

2. **Separation of authentication and authorization**: The security system must be able to separate authentication and authorization, i.e. there must not be a strong dependency between those two procedures. This is because we want to use the AAI Login of the University of Zurich. On the one hand, the use of the AAI service simplifies the authentication task for us (we do not need to implement a mechanism to authenticate the user), on the other hand we need a system that can reuse the authentication service of AAI and only provide the authorization service as an independent element.

3. **Instance level access control**: Since every Merlin researcher must be able to, for example, enter his publications, there must be an instance level access control providing access only to those publication instances that have been created by the user or that contain the user in the authors or editors list. To achieve this, we must ensure that every user is allowed to only edit a subset of all instances available in Merlin.

4. **Support for a role hierarchy**: Within an organization, individual roles often have overlapping functions and responsibilities (see Chapter 2). This means that roles can be structured in a hierarchy, like in a real organization, so that senior roles can inherit permissions from junior roles (multiple inheritance is possible). This fact helps to model the real word scenario in a more natural way. Further, the reuse of already existing artifacts (e.g. permissions) results in a much more clearer structure.

5. **Flexibility and extensibility**: The security system to be implemented must be flexible enough to support delegation of roles based on the theory from Chapter 2. It must either provide a mechanism for role delegation or be easily extensible in order to implement this requirement.

6. **Good and comprehensive documentation**: A good documentation is one of the most important prerequisite when using external software or plugins. If well explained, there is no need to try and error in order to understand the functioning of the security system, which might be very time consuming.

7. **Short learning period**: Due to a tight schedule, the learning curve should be as steep as possible.

4.2.2 A Word on Security Plugins

In software development, it is best practice to build a new system on existing libraries. Basically, there are two security plugins for Grails worth mentioning: the former *JSecurity plugin*, now called the *Shiro plugin*[9], and the *Acegi plugin*[10]. The following paragraphs will discuss these two security solutions in more detail.

Shiro

Shiro (former JSecurity plugin) is the most recommended and therefore the best documented plugin in the Grails literature [RB09, LS09]. The Shiro plugin builds on the JSecurity Library[11] and

[9] http://www.grails.org/plugin/shiro
[10] http://www.grails.org/plugin/acegi
[11] http://www.jsecurity.org/

provides authentication and authorization to a Grails application. Shiro combines a set of security filters with a *realm*, which is the bridge between Shiro and Grails [RB09].

Shiro provides a basic support for roles and permissions both of which can be assigned to a user. The possibility to assign permissions directly to a user disqualifies Shiro from being a pure role-based access control system. With Shiro, a user can have multiple roles and permissions. At the same time, several users can be assigned to the same role. However, there is currently[12] *no support for role hierarchies*. As discussed in Chapter 3, this is a relevant requirement for Merlin in order to be able to reflect the organizational structure of the faculty in the most natural way.

Another drawback of Shiro is the *tight coupling between authentication and authorization*. In order to use the authorization service, the user has to be authenticated. Shiro accomplishes this by assigning an authentication token with the username and password to the JSecurity subject interface, which can be though of as a central place where the authenticated user is held. The subject interface is then utilized in methods such as `hasRole(...)` or `isPermitted(...)`, which have to be implemented in the provided reamls. These methods form the basis for checking whether a user is authorized to access a particular resource. Without the registration of a user in the authentication method, the authorization cannot be performed. Since our aim is to incorporate the University of Zurich's AAI Login for authentication, we would have to find a workaround for this issue. The simplest would be to authenticate the user twice in order to register him with the JSecurity subject.

Shiro supports instance level access through its permissions. This construct is easy to understand and to work with. For example, if we wanted a user to grant edit rights for publications with the ids `1` and `9`, we would just add a wildcard permission `"publication:edit:1,9"` to the user. This permission contains the authorized controller `publication`, the action `edit` and a set of ids (`1,9`) as a string. In an application such as Merlin where a chair must be able to edit a large number of publications, this would result in an almost endless wildcard permission string or in a load of permissions with single ids (note that the ids in the wildcard permission are foreign keys stored as string) to be assigned to a user. This would imply a permission management overhead when adding new or deleting existing resources.

Finally, adding support for role delegations such as tasks would require an extension of the plugin based on an intense exploration and a deep understanding of the inner life of the JSecurity framework. Due to the concise documentation, this would be a very time consuming task.

Acegi

The *Acegi*[13] plugin builds on the Spring Security Framework[14] and provides authentication and authorization functions as well. It simplifies the integration of the Spring Security framework and by hiding the complexity of the underlying XML-based implementation. Acegi provides basic concepts such as users, roles and security filters (that are registered in the application context and the required web.xml file) and comes with all necessary Spring Security jar files [ace10b].

For authorization, the main idea behind the Acegi plugin is to create *request mappings*, which are basically mappings between application URL patterns such as `/publication/**` and defined roles, e.g. `ROLE_RESEARCHER`. There are two kinds of mappings: *static mappings* are defined in the `SecurityConfig.groovy` file and *dynamic mappings* are stored in the database in form of `RequestMap` objects. The Acegi plugin then provides access to resources by checking whether the logged in user has the necessary role to access the URL.

The Acegi plugin is a pure role-based security system. However, the current version does not support fine-grained access control through the definition of permission that would be assigned

[12] We are referring to version 1.0.1 of the plugin.
[13] http://grails.org/plugin/acegi
[14] http://static.springsource.org/spring-security/site/

to user roles. Further, because of the characteristics of the mappings between user role and URL pattern, there is no support for instance level access control (no description in the documentation of [LS09] or [ace10a]). Instance level access control means that different users are allowed to edit different subsets of resources. Since a role is a generic concept valid for every user in the system, we cannot map URLs containing specific resource ids to a user role.

Finally, the current version (0.5.2) of the Acegi plugin does not provide role hierarchy out of the box. As with Shiro, adding support for role delegations, as required for tasks, would imply another enhancement of the plugin.

Conclusion

Unfortunately, none[15] of the plugins discussed in the previous section meets all the requirements presented in Section 4.2.1. Both plugins provide security functions such as authentication and authorization by implementing different approaches. At this time, the Shiro plugin seems to be more mature since it provides fine-grained access control at instance level through permissions. Nevertheless, the implementation of the required functionality for Merlin would imply a customization and enhancements of the plugins. With the tight time schedule in mind, this might lead to unpredictable risks. We have therefore decided to accept the challenge of implementing a custom role-based security system based on our requirements.

4.2.3 Architecture

There are two interacting parts that are involved in the overall system setup. On the one hand, the Web application Merlin implements the business logic and provides an authorization service to filter access to resources. On the other hand, the SWITCH AAI service provides user authentication – a service that Merlin makes use of. As we can see in Figure 4.3, Merlin first checks, whether a user must be authenticated in order to access the requested resource (1). If that is the case, Merlin redirects the user to the AAI authentication service (2). After a successful authentication (3), the user is redirected back to Merlin (4) that answers the user requests based on his access rights (5).

As depicted in Figure 4.3, user requests are first filtered by the Merlin's authorization service, which can be thought of as a gate to the Web application. It represents a very important component in the security as it protects Merlin's resources from unauthorized access.

In the following section, we will describe the concept of the implemented RBAC system and will unveil and discuss the elements of the authorization service. The last part of this section summarizes our ideas by bringing all parts together in order to provide the big picture of the security system.

4.2.4 Authentication

When Merlin's authorization service notices that a user tries to access a protected resource it redirects the user to the AAI service in order to authenticate himself. Because the AAI service provides everything we need to check whether the user has a valid UniAccess account, we do not need to implement a credential matcher ourselves. After a successful authentication, the AAI authentication service redirects the user back to Merlin's `shibboleth` action located in the `SessionController` and sends a set of predefined attributes along with the HTTP request. The `shibboleth` action is then responsible for checking if the user is a valid member of the Faculty

[15] Please note that the discussion of security plugins in Section 4.2.1 refers to the plugin versions available at the start of our project. In the meantime, there has been some further development, especially for the Acegi plugin.

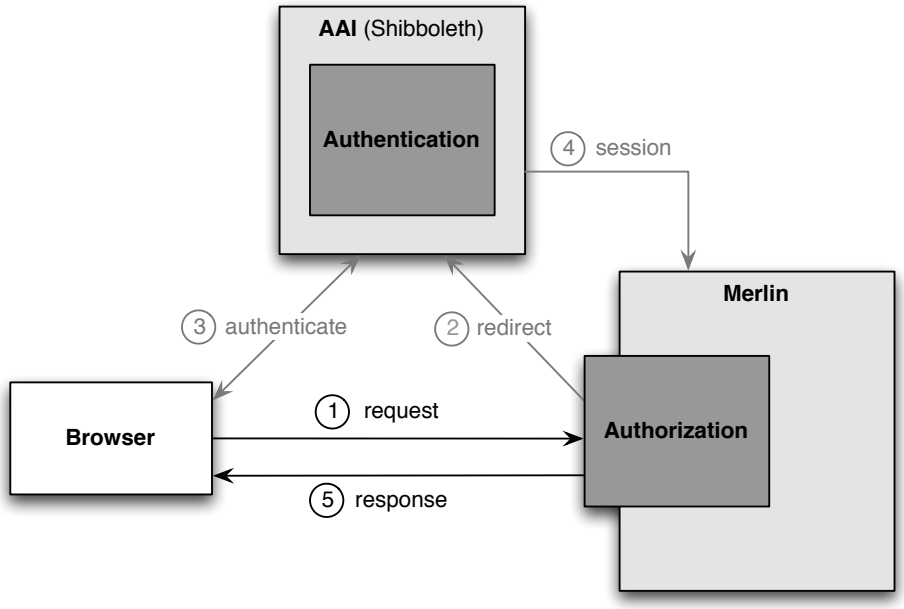


Figure 4.3: Architecture Overview: the Web application Merlin uses AAI for authentication. After a user sends a request to Merlin (1), Merlin first checks, whether the user needs to be authenticated in order to access the requested resource. If that is the case, Merlin redirects the user to the AAI authentication service (2). The user provides his credentials (3) and is redirected back to Merlin after a successful authentication (4). The identity provider AAI opens a session for the authenticated user and sends a defined set of user attributes to Merlin, which answers the user requests according to his access rights (5).

of Economics, Business Administration and IT based on the employee data stored in Merlin's database.

As we can see in Listing 4.6 we first verify whether the request contains a legal shibboleth ID in its header. This is to confirm that the request comes from the AAI service and has not been manipulated with the intention of breaking the security (line 5-7). If there is a shibboleth ID, we try to look for an employee that would match this unique ID. If no employee is found, we read the first name and the last name of the authenticated user out of the `request` object and save the corresponding shibboleth to the matching user (line 16)[16]. We then use the `sessionService` singleton instance to assign the user to its `currentUser` variable (line 23). This variable is the user by the system for further operations such as displaying of the correct user contents or authorization verifications.

```
1  class SessionController {
2    def sessionService
3
4    def shibboleth = {
5      def shibbolethID = request.getHeader("Shib-SwissEP-UniqueID")
6
7      if (shibbolethID) {
8        def employee = InternalEmployee.findByShibbolethID(shibbolethID)
```

[16]There was an attempt to match the user with his so-called "mNumber", which refers to the employee's personal ID. Since Shibboleth does not provide this information at this time, we have to match the user with his first name and last name.

```
9
10      if (!employee) {
11        def firstName = request.getHeader('Shib-InetOrgPerson-givenName')
12        def lastName = request.getHeader('Shib-Person-surname')
13        def employee = InternalEmployee.findByFirstNameAndLastName(firstName, lastName)
14
15        if (employee) {
16          employee.shibbolethID = shibbolethID
17          employee.save()
18        } else {
19          ...
20        }
21      }
22
23      if (employee?.agreed) {
24        ...
25        sessionService.setCurrentUser(employee.id)
26        ...
27      }
28    ...
29    }
30  }
31 }
```

Listing 4.6: Shibboleth Action in the SessionController

4.2.5 Basic Elements of RBAC

As discussed in Chapter 2, a RBAC model defines three basic elements: a *user*, basically a human being that interacts with a computer system, a *role*, which corresponds to a job title or function in an organization and has a set of authorities, and a *permission* that can be thought of as an operation on a resource.

As illustrated in Figure 4.4, Merlin's security is based exactly on those basic elements. The domain class `InternalEmployee` corresponds to the required entity `user` and has a many-to-many relationship with the class `Role`. For example, an internal employee can have the role `EMPLOYEE` and the role `RESEARCHER` at the same time. A role can have and belong to many permissions represented by the `Permission` class. This means that, for example, we can assign several permissions the role `EMPLOYEE` – one for creating academic degrees and another for editing them.

Since the user authentication is provided by the AAI Login (Section 4.2.4), we do not need to store the login and password information. We use the `shibbolethID` attribute for user identification instead. It holds the unique Shibboleth ID of each user. The `Employee` class provides some helper methods for role checking such as `hasRole(roleName)` or `hasAnyRole(roleNames)`. With the first method, for example, we are able to check, whether a user has the role `RESEARCHER` by just writing `employee?.hasRole(Role.RESEARCHER)`. The method then returns `true`, if the `employee` instance is assigned to this role. As we will see in later sections of this chapter, these helper methods are used by different services, methods and taglibs all over the system.

A `Role` has a single attribute `name`. Its narrow interface consists of four methods. For example, the `isPermitted(privilege, resource)` method checks, whether the role is permitted to perform the operation specified in `privilege` on the provided `resource`. For instance, if a role has the permission `EDIT` on the resource `Employee`, the method would return `true` meaning that the role is allowed to *edit employees*. A role can have none, one or many parents to inherit

permissions from, i.e. it provides support for *multiple role inheritance*. Thus, the role `RESEARCHER` can inherit permissions from the role `EMPLOYEE` and the role `AUTHOR` at the same time.

Finally, the `Permission` class in Merlin represents the permission element of a RBAC system, which is used for fine-grained access control. As we know, a permission is defined as an operation on an object. Therefore, the `Permission` class contains the `privilege` attribute, which represents the operation that can be performed on a resource. The `resource` attribute corresponds to the object, which will be manipulated. If we wanted a role to be able to *edit publications*, we would assign a permission with the *privilege* `EDIT` and the *resource* `Publication` to it. Both attributes are of the type `String`. The public interface of the class `Permission` consists of a single method `contains(priv, res)`, which checks, whether the permission contains the corresponding privilege and resource.

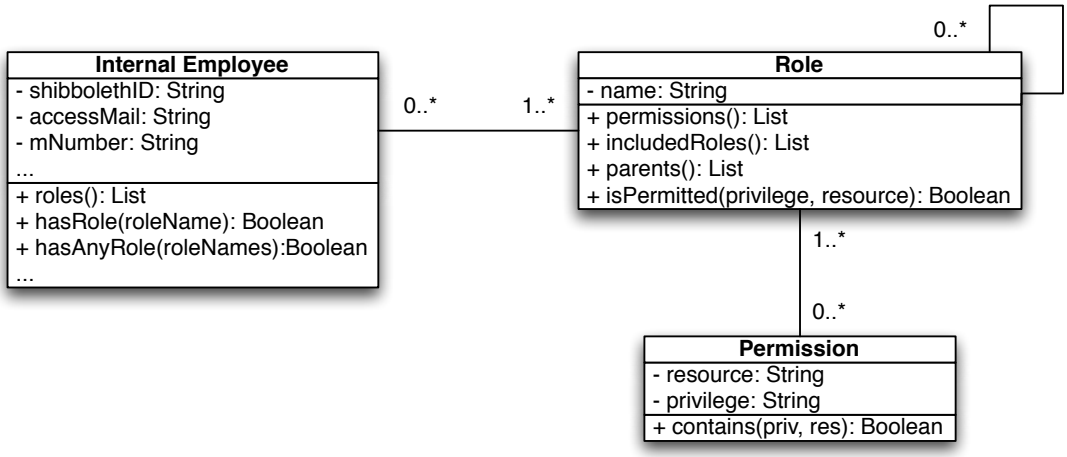


Figure 4.4: Implementation of the basic elements of RBAC in Merlin. The class `InternalEmployee` represents the user to whom roles are assigned (here modeled with the `Role` class). Instances of class `Permission` are allocated to roles.

4.2.6 Role Hierarchy

Because Merlin's access control system based on the concepts of RBAC models, its roles are a central component around which the access policy is built. In order to model the organizational structure of the Faculty of Economics, Business Administration and IT as natural as possible, Merlin supports roles hierarchies. As with other organizations, the roles in the academic domain can have overlapping duties and responsibilities. To avoid duplicates, role hierarchy provides a simple concept that helps to share permissions with the aid of the inheritance mechanism.

As depicted in Figure 4.5, Merlin's roles hierarchy goes beyond the basic tree-based model and provides *multiple roles inheritance*, which helps to share permissions even better. The role *chair* is the most senior role in this hierarchy. It contains permissions of the role *researcher* that in turn inherits permissions from the roles *author* and *employee*. The role *author* provides the users with basic rights to create new and edit existing publications whereas the role *employee* allows to edit the user's profile and manage his education. On the other hand, the role *chair* inherits permissions from another subset of roles being responsible for employee management such as editing of a subset of employees (role *internal employee manager*), creation of new employees (role

employee manager) or organizational unit management (*organizational unit manager*).

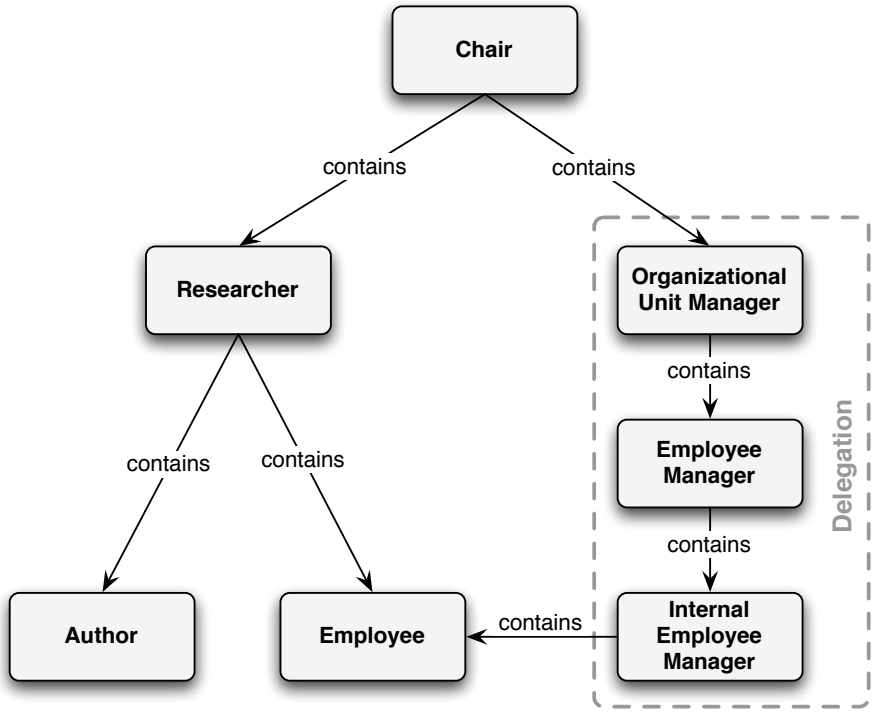


Figure 4.5: Merlin's role hierarchy supports multiple role inheritance. Role *chair* is the most senior role in the system. It inherits permissions from all roles that are underneath. Roles such as *organizational unit manager* or *employee manager* are used in for role delegations (see Section 4.2.9).

In Merlin, we use the role *admin* as a tagging mechanism for users who should be able to operate the system without any restrictions. It would be correct to put the admin role at the top of the hierarchy as the most senior role. The result would be the same only the implementation would be more complicated. Since the admin is allowed to access any resource, it is simpler and less computationally intensive to just return true in all methods when it comes to authorization questions (we will see this later in Section 4.2.7).

An advantage of this role hierarchy is its *flexibility*. If there is a requirement for a new role, it can be added to the hierarchy in a simple way. Further, it can inherit permissions from multiple roles or come with an entirely new set of permissions. The benefits of the hierarchy are also visible in the fact, that we can assign a user one single role, for example researcher, and he automatically get all permissions of the roles that are junior to this role. On the other hand, if a user has multi roles such as researcher and employee, we can remove the employee role from the user and he will still have all necessary access rights to perform his duties.

Finally, each role in the role hierarchy has its specific set of permissions. This helps to meet the *principle of least privilege*, which states that every role should have the least amount of permissions that the role requires in order to perform its duties. Table Table 4.1 gives an overview of all roles and the permissions that are specifically assigned to it.

Role Name	Permission	
	Resource	Privilege
EMPLOYEE	Employee	edit
	Academic Degree	create, edit
	EmployeeRecord	edit
AUTHOR	Publication	create, edit
	Author	create, edit
RESEARCHER	Appointment	create, edit
INTERNAL_EMPLOYEE_MANAGER	Appointment	create, edit
EMPLOYEE_MANAGER	Employee	create
ORG_UNIT_MANAGER	Organizational unit	edit
	Activity	create, edit
CHAIR	Organizational unit	edit
	Activity	create, edit
ADMIN	*	*

Table 4.1: Mapping of permissions onto roles. For example, the role `EMPLOYEE` has the permission to `EDIT` the domain class `Employee` or to `CREATE` and `EDIT` instances of the domain class `AcademicDegree`.

4.2.7 Authorization

Merlin's authorization system is the main gate to the Web application. It provides fine-grained access to protected resources based on users roles. Basically, there are four components that cooperate while authorizing users for access: the *security filters*, the *authorization service* with its *authorization helper* and the *security tags*. In the following section, we will discuss these components individually in order to understand their contribution to the access control. In the last section, we will provide the big picture of the system by putting all parts together.

Security Filter

Security is one of those problems that could be brought up as a prime example for crosscutting concerns of Aspect-Oriented Programming (AOP). Security rules often apply to multiple URIs, classes and even methods across the application. Typically, an application needs to authorize a user to execute certain actions, which can result in security being mixed with application logic – this is definitely undesirable [RB09].

The Merlin security filters use the filter construct provided by the Grails framework. They are consolidated in the `SecurityFilters.groovy` file, which lives in the `grails-app/config` folder. In their basic principle, security filters are code fragments that are executed before the controller actions. Listing 4.7 shows the authenticate before filter, whose code is executed before every action (accomplished by "*" in the controller and action declaration). What it does is to check, where there is a user ID in the session (line 4) or whether the user is attempting to access the public section of Merlin that does not need any authorization (line 5). If that is the case, the user can continue browsing, otherwise the user is redirected to the `login` action that triggers a redirect to AAI.

```
1 def filters = {
2   authenticate(controller: '*', action: '*') {
3     before = {
4       if (session.user || ((!session.user || session.userIdFromShibboleth) &&
5         params.action in PUBLIC_CONTROLLERS_AND_ACTIONS.get(params.controller))) {
6         return true
```

```
7       } else {
8         redirect(uri: "/login")
9       }
10    }
11  }
12  ...
13 }
```

Listing 4.7: Authenticate Filter

A shortened version of the public section can be found in Listing 4.8. As we can see, the public section is implemented as a hash map having the controller name as the key and a list with action names as the corresponding value. The authenticate filter uses this fact by checking whether the requested action is in the value list of the controller key. For example, a user is allowed to search publications or browse organizational units without having to be logged in.

```
private final def PUBLIC_CONTROLLERS_AND_ACTIONS = [
  "publication": ["show", "search", "list", "index"],
  "search": ["index", "results", "quickSearch"],
  "organizationUnit": ["show", "list", "index", "updateSelect", "showPublications"],
  ...
]
```

Listing 4.8: Public Section in Security Filters

Now let us consider an example of the `hasPermission(...)` method invocation in the authorization service. As we will see in Section 4.2.7, this method is responsible for checking, whether the current user has a particular permission. We first define some general privileges. As we can see in Listing 4.9, the `EDIT` privilege contains three methods, namely `edit`, `delete` and `update`. We can use such a privilege declaration for grouping controller actions. This helps to avoid code duplications.

```
 private final String EDIT = "edit|delete|update"
 private final String CREATE = "create|save"
```

Listing 4.9: Declaration of Privileges

We now can make use of the declared privileges. As shown in Listing 4.10, we can use the same authorization check for all actions defined in the `EDIT` privilege. If a user wants to edit, delete or update a particular employee instance, then one of his roles must have a permission with the privilege `edit` for the `Employee` resource. To verify this, we invoke the `hasPermission(...)` method of the injected authorization service, which takes a hash map with the required attributes as parameter (line 4). If the user has a role that contains such a permission and the user is allowed to edit the instance with the requested ID, the authorization service grants access to it. Otherwise, the access to the requested resource is declined and the user is redirected to the `unauthorized` action in the `SessionController` (line 5).

```
1 employeeEdit(controller: "employee", action: EDIT) {
2   before = {
3     if (session.user && !authorizationService.hasPermission(
4         [privilege: Privilege.EDIT, resource: Employee.class, id: params.id])) {
5       redirect(controller: "session", action: "unauthorized")
6     }
7   }
8 }
```

Listing 4.10: Edit Employees Filter

Authorization Service

To implement the authorization service, we make use of the Grails services. A service provides an easy way to centralize application behavior into an API that can be utilized by other controllers and services. This is achieved by dependency injection provided by the underlying Spring framework.

The authorization service is implemented as *singleton*, meaning that there exists only one instance of the service in the application. It provides a public interface containing five methods in order to check, whether a user has access to the requested resource. The methods are explained in Table Table 4.2.

Method Name	Description
`hasAnyRole(..)`	Checks, whether the user contains one of the roles provided in the parameter list.
`hasPermission(..)`	Checks, whether the user has a required permission or has permission to access a particular instance.
`hasRole(..)`	Checks, whether the user has a specific role.
`isAdmin()`	Checks, whether the user is an administrator.
`isChair()`	Checks, whether the user is a chair of an organizational unit.

Table 4.2: Public interface of authorization service.

Without doubt, `hasPermission(...)` is the most interesting method in the authorization service since it provides a mechanism to check whether a user is allowed to access a particular instance. Listing 4.11 presents the source code.

When a user requests access to a specific resource, we first check, whether he is an administrator (line 2). Since an administrator is allowed to do everything in the system, we grant access by returning true. Otherwise, we check if the roles assigned to the user contain the required permission (a privilege for a resource class) by calling the private `isPermittedTo(...)` method (line 9) in the first place. If that is the case and the `attrs` hash map contains an ID key, we query the instance from the Merlin's database (line 11). Then, we ask the `AuthorizationHelper` object to collect all instances for the requested resource class the user is allowed to access and check, whether the requested instance is contained in that set (line 14). This is accomplished by using Groovy's metaprogramming capabilities, namely the *reflection* feature. To understand the exact functioning, we first need to know that, as with Grails itself, there is a naming convention to follow. The `AuthorizatonHelper` class provides methods for each resource in the system that needs an instance level access control. With that in mind, we are now able to understand the code in Listing 4.11. For example, if a user wants to access a specific employee instance, the `hasPermission(...)` method first determines the name of the method (line 13) that is going to be invoked on the `AuthorizationHelper` instance called `helper` (line 12). In our example, the name of the method would be `employees()`. We then invoke the `employees()` method on the `helper` instance using Groovy's reflection mechanism to see, whether the instance is contained in the authorized employees set of the user (line 14).

```
1  def hasPermission(attrs) {
2    if (isAdmin()) {
3      return true
4    } else {
5      def privilege = attrs.privilege
```

```
6     def resourceClass = attrs.resource
7     def simpleClassName = simpleNameFor(resourceClass)
8
9     if (isPermittedTo(privilege, simpleClassName)) {
10      if (attrs.containsKey("id") && attrs.id) {
11        def resourceInstance = resourceClass.get(attrs.id)
12        def helper = new AuthorizationHelper()
13        def method = AuthorizationHelper.getDeclaredMethod(methodNameFor(simpleClassName))
14        return method.invoke(helper).contains(resourceInstance)
15      } else {
16        return true
17      }
18    } else {
19      return false
20    }
21  }
22 }
```

Listing 4.11: HasPermission() Method of the Authorization Service

Authorization Helper

The authorization helper lives in the `src/groovy` folder and its main contribution to the authorization system is to provide the authorization service with a set of entities that a user is authorized to access. This collection is always calculated at runtime. The `Authorization-Helper` class provides a method for every resource class that needs an instance based access control. Listing 4.12 shows an example of the `employees()` method that determines the set of employee instances a user is allowed to manipulate. As we can see, every user is allowed to edit his employee instance (line 3). If the user is a chair, he must be able to edit all employees in all organizational units he is chair of (line 5-7). This collection of employee instances in then used by the authorization service to either grand or decline access to the requested entity.

```
1  def employees() {
2    def result = []
3    result << currentUser
4
5    if (isChair()) {
6      currentUser?.chairedOrganizationUnits()?.each { unit ->
7        result << unit.employees()
8      }
9    }
10   ...
11   result.flatten().unique()
12 }
```

Listing 4.12: Employees() Method in the Authorization Helper

Security Tags

The Web page content that is presented to the user depends on the roles that he or she is assigned to. For example, if a user has the role `CHAIR` he should be able to see the "chair" menu in the top navigation bar or perform operations on his organizational unit by clicking on links, which are hidden from other users. In order to support dynamic content layouting, we defined a set

of security tags that provide an easy way for view customization according to the user's roles or permissions. They are basically wrappers for methods provided by the authentication service. Listing 4.13 shows some examples. Note that Grails tags start with a "g" as in the `<g:link>` tag example. To be able to distinguish Merlin specific tags, we decided to use the "m" static name scope.

```
<m:hasRole name="${Role.RESEARCHER}">
  <g:link controller="publication" action="newPublication">Create new Publication</g:link>
</m:hasRole>

<m:hasAnyRole in="${[Role.CHAIR, Role.ADMIN]}">
  <g:link controller="employees">Employees</g:link>
</m:hasAnyRole>

<m:hasPermission privilege="${Privilege.EDIT}" resource="${Employee.class}" id="${e?.id}">
  <g:link controller="employee" action="edit" id="${e?.id}">Edit Employee</g:link>
</m:hasPermission>
```

Listing 4.13: Security Tag Examples

4.2.8 Pulling it all together

Now that we know all parts of the Merlin's authorization system in detail, we look at their collaboration from a higher perspective. An overview of this interaction can be found in the examples shown in Figure 4.6. Basically, the authorization checks are performed by intercepting controller action invocation and by checking if the user roles contain the necessary permissions to access the requested resource. If not, the user is redirected to the `unauthorized` action in the `SessionController`.

Instance level access is accomplished by calling the `hasPermisson(...)` method, which takes a hash map with attributes as parameter. Based on that information, it checks, whether the user has the necessary permission and whether the requested resource instance in contained in the set of his authorized entities. This set is calculated upon invocation of the resource methods provided by the `AuthorizationHelper`.

Chair or administrator checks are performed through the methods `isChair()` and `isAdmin()`. They both return a boolean value according to the roles the user is assigned to.

Finally, the to check if a user has access to, for example, create a publication, the security filters invoke the `hasRole(...)` method on the authorization service, which returns true if the user has the required role.

4.2.9 Delegation

As discussed in Chapter 3, delegation within Merlin is supposed to support two types: a temporary delegation called *task* that should support a chair in a onetime manner and a permanent allocation of permissions called *access right*. According to theory, in a RBAC system a user can only delegate roles that are assigned to him to another user.

In Merlin, a delegation is more than just a forwarding of roles. We implemented a construct that can be thought of as a carrier for further information that needs to be sent along with the delegation. Basically, delegating permissions in Merlin means sending internal messages to other users. Figure 4.7 provides an overview of the domain classes involved in the delegation.

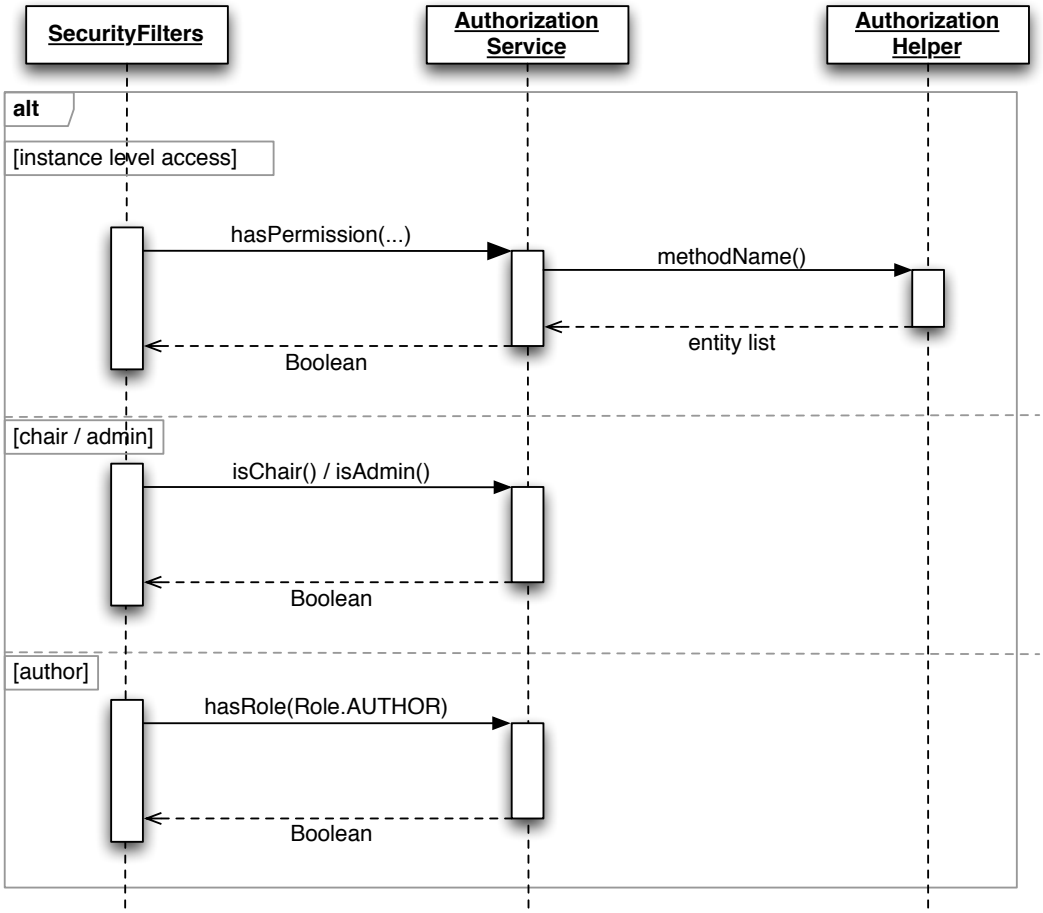


Figure 4.6: Interaction between the elements of the authorization system. `SecurityFilters` intercept the invocation of controller actions and forward the request to the `AuthorizationService`, which in turn asks the `AuthorizationHelper` instance to calculate the user's authorized entity set and checks, whether the requested instance is contained in it. If not, the access to the requested resource is denied.

Access Right

We start with the simplest delegation first. When a chair wants to delegate an *access right*, he has to select the scope of the access right first. Based on the scope, the role that is assigned to the access right is determined. This role provides the delegatee with new access rights that he might not have had before. There are four types of delegation scopes: *create publications*, which provides the delegatee with the ability to enter new publications, *edit employees* that makes the delegatee able to edit all employees of the selected organization units, *create and edit external employees* that, in addition, provides the access right of creating new external employees for the chosen organizational units and finally *edit organization unit*, which adds the right to edit the selected organizational units. Table 4.3 gives an overview about how roles are mapped to delegations based on the selected scope. As we can see here, all roles that are adhered to a delegation are junior roles to the chair role. This fact helps us to meet the precondition stated in the theory: a user can only delegate

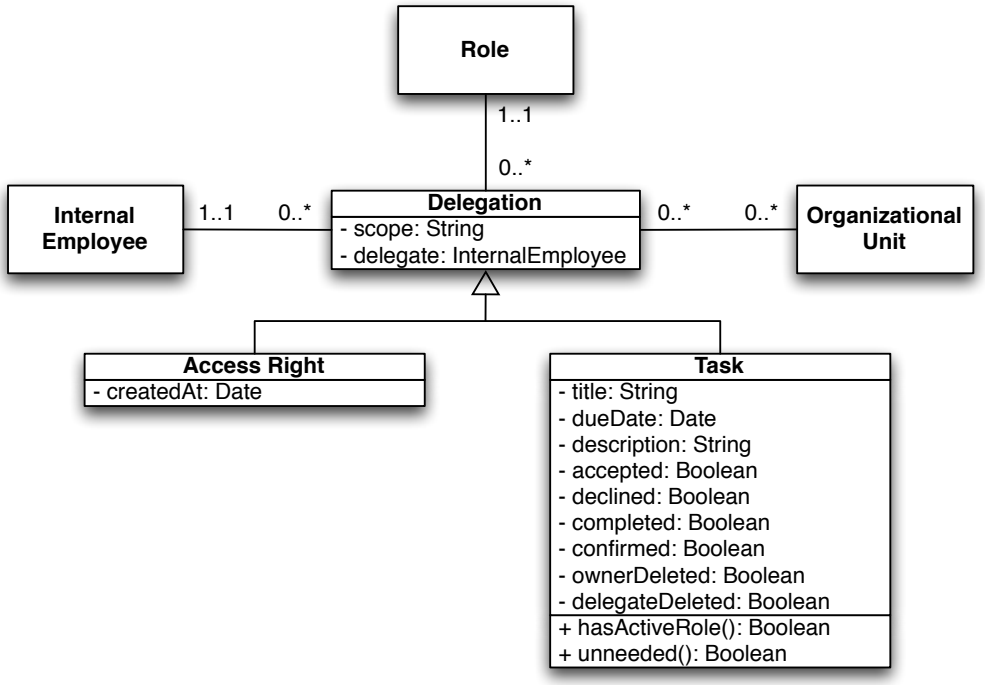


Figure 4.7: Delegation in Merlin: An `InternalEmployee` can have many delegations, an instance of the `Delegation` class has an owner, who initiates the delegation, and a delegate who receives the delegation. A delegation has one role assigned to it and can have many organizational units allocated. `AccessRight` and `Task` are specializations of the `Delegation` class.

roles that he or she owns. Since in Merlin only chairs[17] can send delegations, this requirement is met due to the fact that the chair role inherits from those assigned to the delegation, i.e. the chair has these roles implicitly because of the role hierarchy.

Scope	Role
Create publications	`AUTHOR`
Edit employees	`INTERNAL_EMPLOYEE_MANAGER`
Create and edit external employees	`EMPLOYEE_MANAGER`
Edit organizational unit	`ORG_UNIT_MANAGER`

Table 4.3: Roles are assigned to delegations according to the selected scope.

[17] Admins can send delegations also. But as we have seen in Section 4.2.6, the admin role is used to tag the users that should be able to use the system without restrictions.

Task

In contrast to the simple access rights, the anatomy of a task is a little bit more complicated. As we can see in Figure 4.7, a *task* contains, besides a role, detailed information such as title, due date, task description and set of organizational units. A task is much more complicated in the handling than an access right. As depicted in Figure 4.8 a task can have different states that are handled by the `delegationService`. After a task has been sent by a chair and accepted by the delegatee, it changes its state from *pending* to *accepted*. This activates the enclosed role. After the delegatee has completed the task, he marks it as completed (state *completed*). The chair then acknowledges the completion of the task by confirming it (state *confirmed*), which deactivates the attached role. As we have seen, the system decides based on state of the task, whether the assigned role is active or not. For example, if a user does not have time to accomplish the task until the due date and therefore he declines the task, we do not want the system to provide him with the access rights that are sent along with the task – this would dilute the security system.

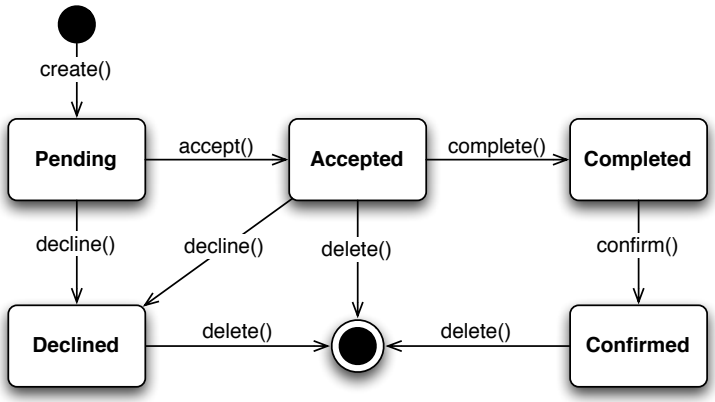


Figure 4.8: Task states: After the task has been *accepted* the provided role becomes active. When the delegatee has completed the task, he informs the delegator by changing the state to *completed*. If the task has been successfully accomplished, the owner set the state to *confirmed*, which deactivates the role – the user is no more able to perform the actions requested by the task.

We also provide a separate user interface for task handling. As you can see in Figure 4.9, every user in the system has its own task box where he can manage the tasks assigned to him. Different task states are represented by different icons. For example, the clock icon informs the delegator that the task is still pending and has neither been accepted nor declined.

4.3 Domain Model and Basic Functionality

After having discussed the security concept and its implementation in detail, we will now concentrate on the domain core and the functionality that Merlin offers its users. To get a better understanding, we will start with an overview of the domain model. We then move over to the employee specific domain classes and will present some of the basic functions related to employees. The next part of this section focuses on the implementation of the organizational units. Finally, we will pull all things together to see how employees can be assigned to their organizational units.

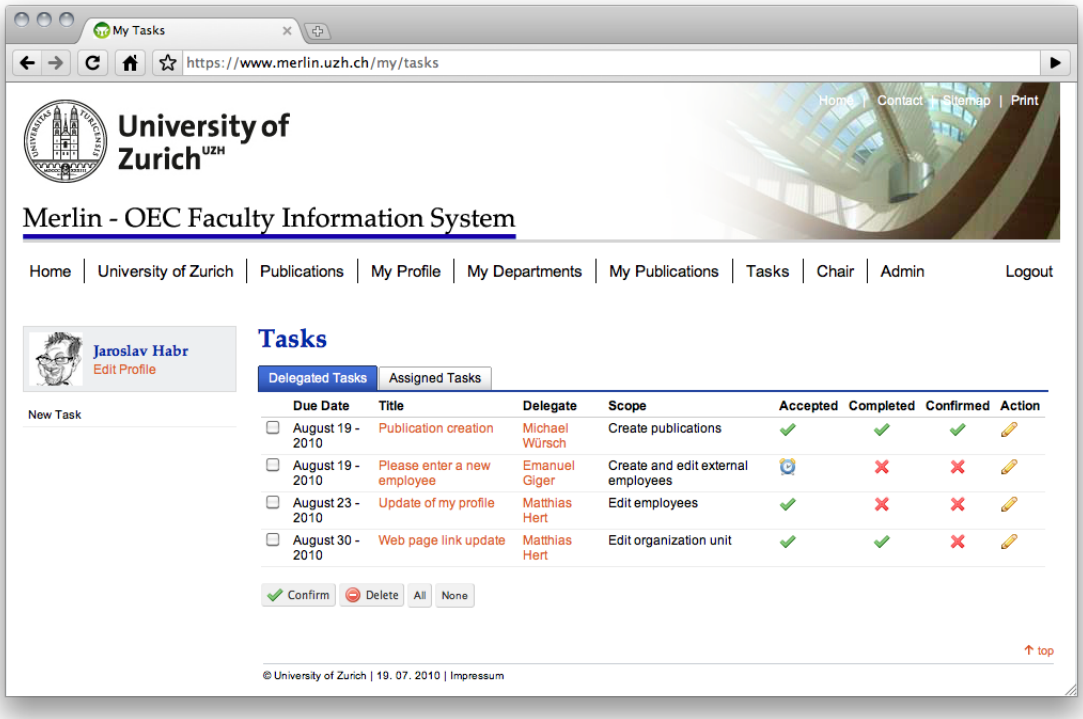


Figure 4.9: User interface for task handling: every chair has an inbox and an outbox containing incoming and outgoing tasks. The states of the task are represented by various icons.

4.3.1 Overview

Figure 4.10 shows a very simple version of the domain model. Basically, there is a class `Employee` that models the members of the Faculty of Economics, Business Administration and IT. The education of the employee is represented by the class `AcademicDegree`, which holds all relevant information about the employee's highest academic degree. Details about the employment are saved in an `Appointment` object. With theses three domain classes, we are able to represent every information about the user that is relevant for Merlin. Note, that the relationships between these classes are always one-to-one relationships.

An employee can belong to many organizational units and an organizational unit normally has more than one employee. We implemented this many-to-many relationship with the aid of the `Membership` class, which is basically a mapping table between those two classes. However, the `Membership` class contains another detail about the type of the relationship – if an employee is a chair of a particular organization unit, we save this information in the `chair` attribute (boolean) of the `Membership` instance. This way we are able to track, whether an employee is a chair or not.

Finally, to provide the administrators with the necessary data for the EQUIS and AACSB accreditation, every employee has many employee records. An employee record in an instance of the `EmployeeRecord` class and contains all details about the employee. We can think of it as a clone of the employee instance that consolidates every peace of information about the employee available in the system. But why would we want to duplicate this information? Imagine, for example, that a member of the active research staff receives his PhD degree in the current reporting

period (a time span with a defined start and end date within which employees publish their publications). He then might update the education information in his Merlin profile. If we did not have a construct such as employee record and the admin then exported the employee information for the last period directly from the profile, our employee would be listed with his new academic degree – this would falsify the period report that the faculty sends to the accreditation authorities. Thus, since an employee record belongs to a reporting period, it holds the correct employee information for that specific time span.

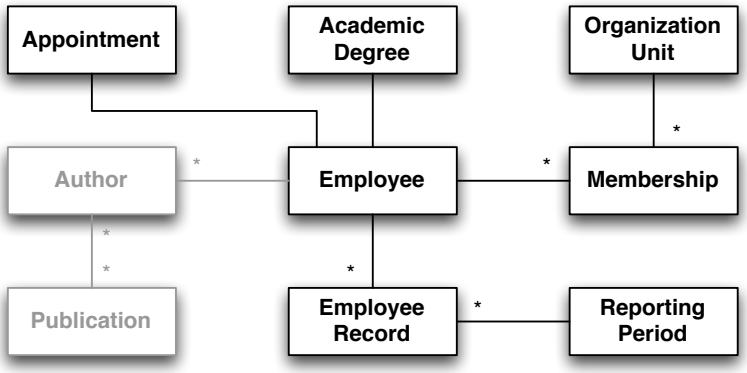


Figure 4.10: The simplified domain model shows the `Employee` class with its relationships to its corresponding domain classes such as `AcademicDegree`, `Appointment` or `Membership`.

4.3.2 Employee

Besides the `Publication` class, the `Employee` class is one of the central elements, which the whole application is built around. The employee class represents a real person in Merlin. As we can seen in Figure 4.11, there are two types of employees: the class `InternalEmployee` represents the members of the Faculty of Economics, Business Administration and IT, whereas the `ExternalEmployee` class corresponds to employees that are external to the faculty but relevant for Merlin (e.g. external experts teaching a course).

The `Employee` superclass contains general attributes such as first name, last name, sex or email, which are inherited by the subclasses. It provides basic methods for publications and organizational units.

The `InternalEmployee` class is the most interesting class in this context. As we can take from Figure 4.11, instances of this class hold the `shibbolethID` information, which we get from the authentication service AAI and the employees `accessMail`, that is composed from the imported `mNumber` (see Section 4.4.1 for further details on importers). Finally, the `addToken()` method is used for user identification needed while confirming a new email address.

External employees are employees who are external to the faculty but are involved in the teaching such as external lecturers. Thus, they do not need access to Merlin. Nevertheless, it is important for chairs to be able to track them because of their teaching activities. They are represented by the `ExternalEmployee` subclass, which contains the `credits` attribute in addition. The `semesterWeekHours()` method calculates the required ratio out of the provided ECTS credit number.

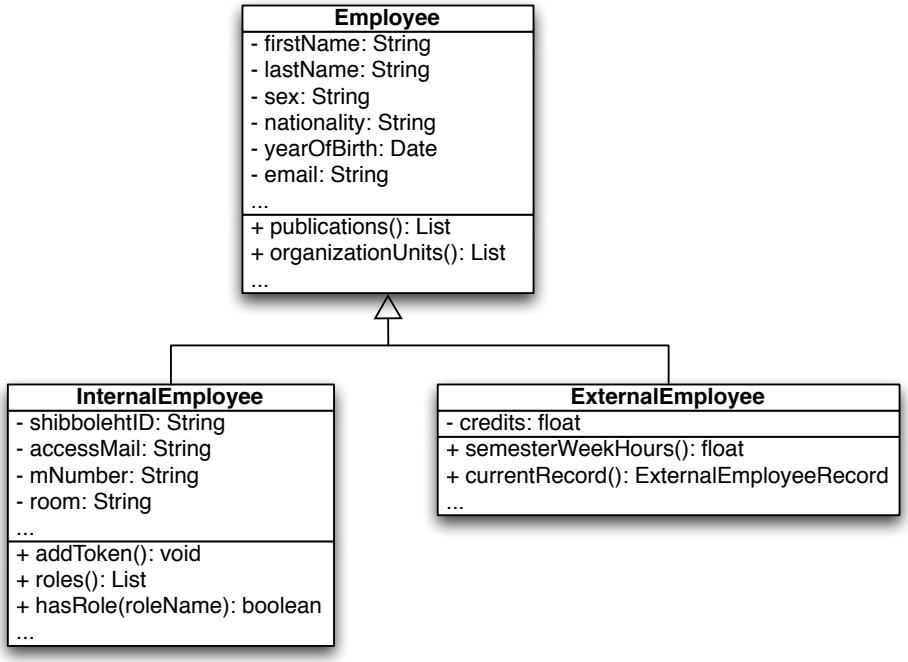


Figure 4.11: The superclass `Employee` provides basic attributes, which are inherited by the subclasses. `InternalEmployee` represents the internal members of the Faculty of Economics, Business Administration and IT. Instances of the class `ExternalEmployee` correspond to external employees such as lecturers.

Setup Wizard

After knowing the internal structures of the employee implementation, let us consider some higher-level functionality and features that are provided by Merlin. First, we want to make sure that we have a complete profile of every user. To achieve this, we have implemented a setup wizard that every user has to go through after the first login in Merlin. As we can learn from Figure 4.12, the wizard provides the user with intuitive forms and asks questions about his personal data, education, appointment or organizational unit affiliation. Based on the entered information, the wizard dynamically adapts to the user input. For example, after the user has selected *teaching and research assistant* as his function within the organization, the setup presents the *share of activities* window as the next step. If his function was *technician*, he would have been redirected directly to the education window.

To support the user as effectively as possible, the setup saves the user input after every step. This is an important feature that helps to prevent data loss, for example in case of a client crash – the user does not have to worry about entering the data over and over again. One last feature worth mentioning is the fact that, in the background, the setup wizard assigns access rights to users according to their input. This is a very convenient feature for administrators who are not obliged anymore to assign access rights for every user manually.

Employee Profile and Custom Homepage

In Merlin, every internal employee has his *own profile page* that he is able to freely manage. The profile is the central point for employee specific data and functions. As we can see in Figure 4.13,

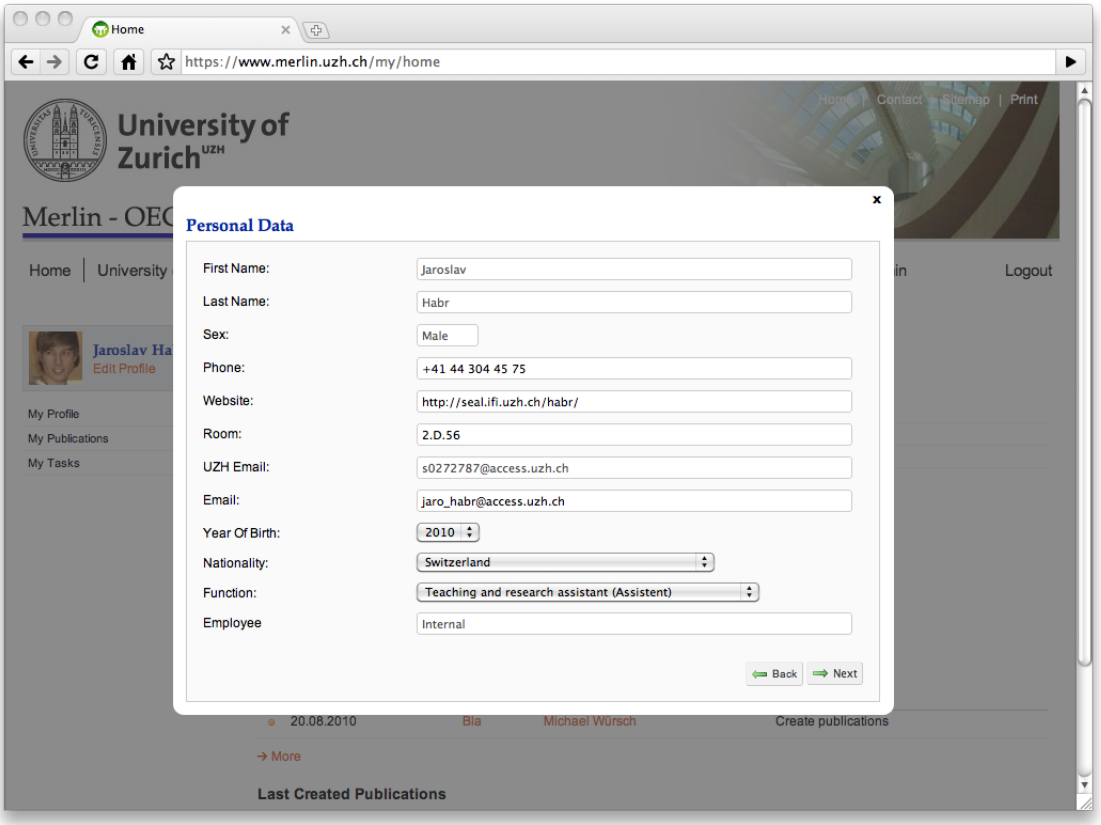


Figure 4.12: The setup wizard guides the user through data entry after his first login. Its dynamic behavior helps to provide a custom form sequence based on the user input.

every profile may additionally contain a profile picture. This helps to personalize the employee page even more.

Underneath the profile picture we can find a set of functions that help the user either to manage his profile or to perform other tasks such as personal data confirmation or the automatic generation of the curriculum vitae.

Another feature worth mentioning is the personalized *homepage*, which will be displayed after every login. The purpose of this page is to inform the user about latest news in his academic environment. For example, there is general information about the current period, which, among other things, informs the user about remaining days before the period deadline. Further, the user is reminded of open confirmations and new tasks or can view his five latest publications.

Being a Chair

Merlin provides specific functions for chairs. For example, a chair is allowed to manage his or her organizational units. To be able to do so, the system automatically assigns the correct access rights upon chair definition and removes them when the chair of an organizational unit is assigned to another person.

A chair can delegate tasks or access rights to other members of his organizational units. The chair only has to select the scope of the delegation and the system, again, automatically delegates

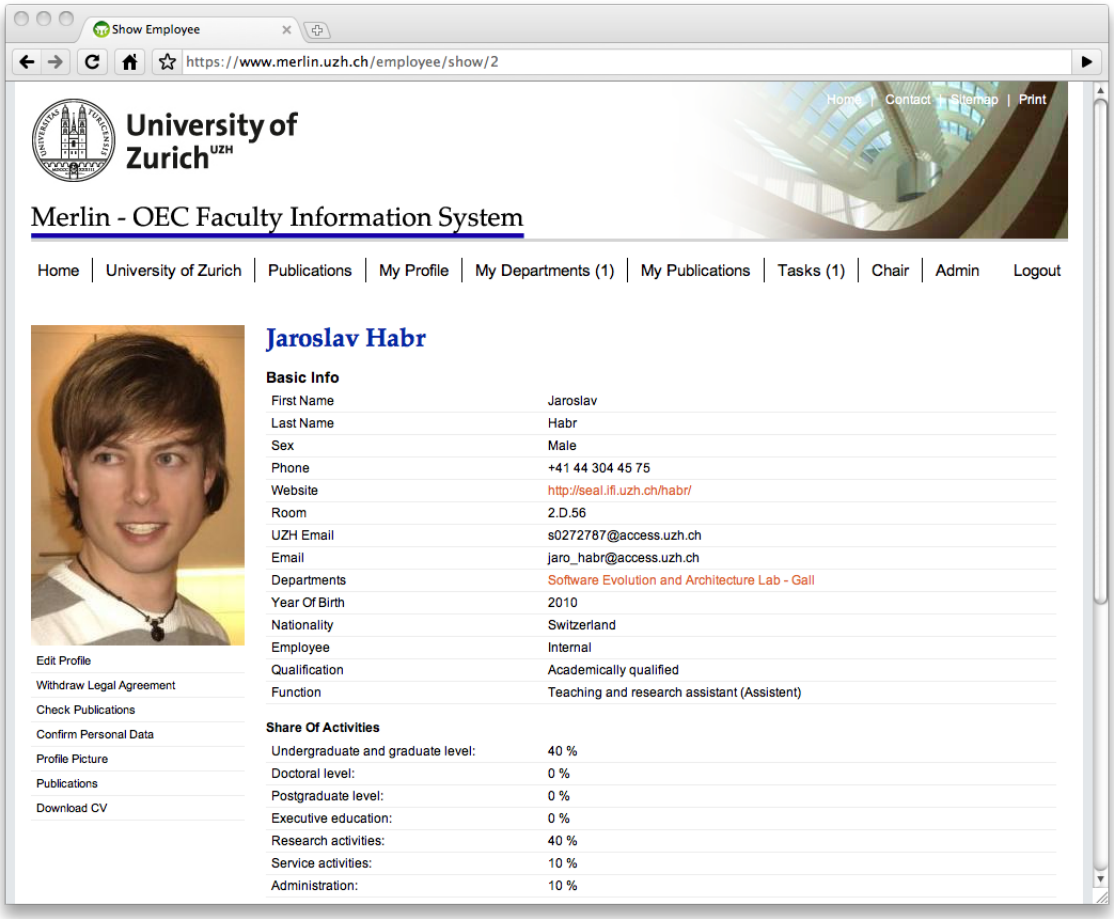


Figure 4.13: Employee profile consolidates and presents employee specific information in a clearly laid out manner.

the corresponding access rights to the delegatee. The chair does not have to care about the access rights of the recipient of his delegation. Everything is set up by Merlin – it makes sure that the delegatee is allowed to perform the defined task even if he does not have the corresponding access rights yet. However, the chair is able to see the roles of all members of his organizational units and all delegatees in a separate access rights management view– just to be informed about their permissions.

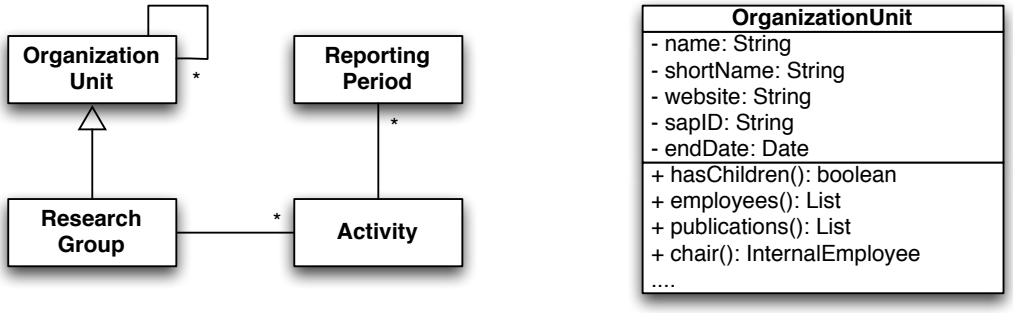
4.3.3 Organizational Unit

The domain model excerpt of an organizational unit is even simpler. As depicted in Figure 4.14 (a), it basically contains three classes: the superclass `OrganizationalUnit` represents all departments and institutes, whereas the subclass `ResearchGroup` corresponds to research groups – the leaves in the organizational structure.

The internal of the `OrganizationalUnits` are presented in Figure 4.14 (b). An instance of this class always has a name, a short name that corresponds to its abbreviation, a website, a sap ID, which is imported from SAP and stored in order to be able to form the organizational structure and an end date attribute that is used for deactivation in case the organizational unit should be

emitted.

Every `ResearchGroup` has many activities. An activity represents the academic occupation such as consulting or professional development. The activity of each research group is stored per reporting period.



(a) Domain Model Excerpt (b) Class OrganizationUnit

Figure 4.14: Organizational unit model. **(a)** shows an excerpt from the domain model with focus on the organizational unit and the involved classes. **(b)** presents some attributes and methods provided by the `OrganizationUnit` class.

Organizational Structure

To provide its users an intuitive for navigation in the organizational structure, the units are presented as a tree. This is a natural way of representing such structures. The tree is implemented using JavaScript, which provides the ability to expand and collapse each intermediate node. Thus, information overload is prevented by only displaying the relevant information to the user.

Organizational Unit Profile

As with employees, Merlin provides a customized profile for every organizational unit. The information is divided into related sections, which are presented in form of tabs. In the fist tab, each profile contains some *basic information* such as full name, short name, home page, the name of the chair, number of employees or number of publications published within this unit. This information can be enhanced by the chair with detailed academic information such as scientific presentations for external audience, visiting professors or researchers funded by third party. The second tab provides an overview and basic info (first name and last name, function, email, room, phone and number of presentations) about *employees* who are member of the unit. Finally, the third tab shows an *organization chart*, which helps to better find the selected unit in the organizational structure.

4.3.4 Assigning Employees to Organizational Units

Let us now put all things together by summarizing, how employees can be assigned to organizational units. After a user has been guided through the setup wizard, he is ask in the last step to enter the organizational units he is affiliated to. Upon a successful submission, Merlin creates

an instance of the `Request` class behind the scenes. As illustrated in Figure 4.15, the class `Request` is associated with the class `InternalEmployee`. Thus, a request instance contains (two) instances of the `InternalEmployee` class, namely a requester and a recipient, which is represented by the chair of the organizational unit. The request is then sent to the chair that is informed about the incoming inquiry both by an internal message within Merlin and by email. After clicking on the provided login link in the email, the chair sees a new open request on his personalized homepage. He then decides, how to proceed by either accepting or declining the request.

But why are the things so complicated? Why can an employee not just be added to an organizational unit without the chair having to confirm the request? Well, since the data provided by SAP does not allow us to map all employees to their corresponding organizational units, we have to implement a mechanism that helps us to control the affiliation of each employee. The reason behind this decision lies mainly in the data quality concern of Merlin and the correct reproduction of the organizational structure, which is not available in any other system of the University of Zurich. This is what makes Merlin so important: it is going to be the first application that has a complete and correct structure of the organizational units of the Faculty of Economics, Business Administration and IT and its employees.

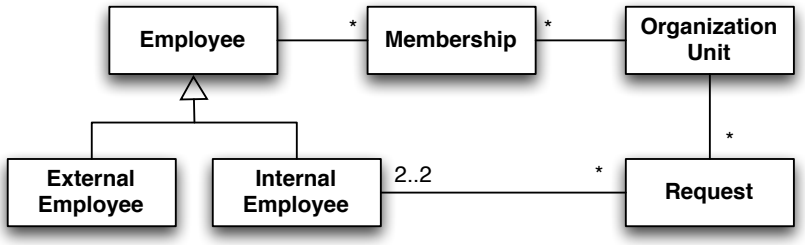


Figure 4.15: Relationships between employees and organizational units. The `Request` object helps to provide a control mechanism for the employee affiliation.

4.4 Data Source: SAP

As discussed in Chapter 3 Merlin will be embedded in an existing application landscape. In order to communicate with the other application in the environment, we must either provide specific interfaces or the ability to import data via cron jobs and importers.

SAP is the most important data provider when it comes to employees and organizational structures. In order to be able to identify users and to assign them to an organizational unit, we need to import this information into Merlin. The first possibility to achieve this would be the specification and implementation of an interface to SAP. However, this approach would be very tedious and time-consuming and would require knowledge of the internal SAP structures and data types. With our tight time schedule in mind, this would imply an overhead whose benefit could not be justified. We therefore decided to implement two data importers, one for employee data and the other for organizational data structures.

Before we go ahead with the discussion on how these importers are implemented, we would like to say a few words about data quality. As already stated, data quality is a big concern to Merlin. Since the SAP structures its employees and organizational units according to its cost centers, there is no clean data structure that we could build our system on. After this insight, we

decided to initially import the organizational units and then to build the organizational structure ourselves.

In the next section, we show how the data importers are implemented and what data formats they have to deal with. We then continue with the description of the cron job that is responsible for nightly employee data imports.

4.4.1 Receiving Data from SAP

Basically, there are two data formats that are exported from SAP. The organizational structure comes in form of an XML file whereas the employee data is contained in a CSV file. We therefore need two different importers to write. Figure 4.16 shows the classes involved in the data import.

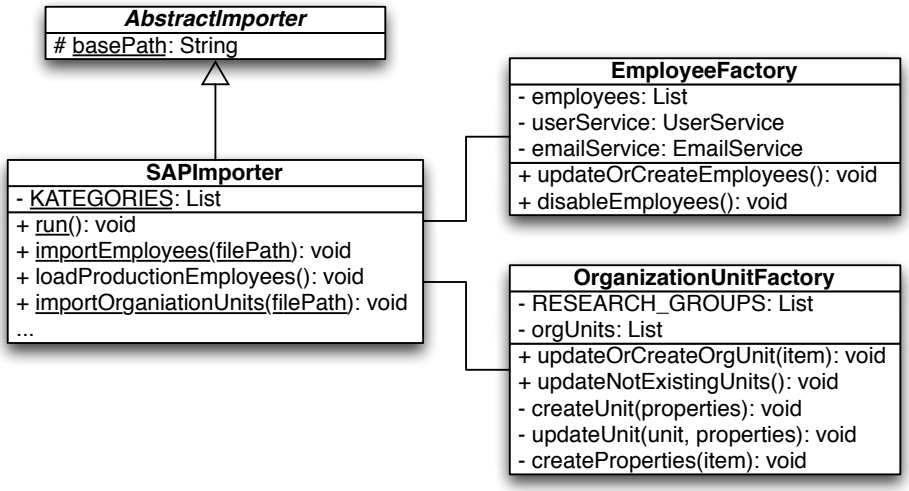


Figure 4.16: The SAPImporter uses instances of `EmployeeFactory` and `OrganizationUnitFactory` to create new or update existing entries.

The class `SAPImporter` is the central element, which holds instances of the `EmployeeFactory` and `OrganizationUnitFactory`. The first is, as we can assume from the name, responsible for importing employees whereas the latter provides Merlin with new organizational units. The `SAPImporter` offers a static method `run()`, which uses the `importEmployees(...)` method to start the import of employee data. Listing 4.14 depicts some details of the implementation. As we can see on line 8, the `updateOrCreateEmployee(...)` method is responsible for creating new employees. If an employee with the provided properties already exists in the Merlin database, his entry will be updated. Otherwise, a new employee is created.

```
1 static def importEmployees(filePath) {
2   ...
3   def employeeFactory = new EmployeeFactory()
4
5   new File("${AbstractImporter.basePath}/${filePath}").eachLine { line ->
6     def properties = []
7     line.split(";").each {properties << it.trim()}
8     employeeFactory.updateOrCreateEmployee(properties)
9   }
```

```
10  ...
11  }
```

Listing 4.14: ImportEmployees() Method in SAPImporter

As already mentioned, the organizational structure is exported from SAP in form of a XML file. Listing 4.15 shows an example of the "Software Evolution and Architecture Lab" unit and the information tags that are contained in each XML item. Every item has a unique SAP ID (line 4). In order to form the structure of the organization we use the `<SOBID>` tag, which helps us to find the parent of the organizational unit. This way, we can rebuild the structure of the organization in Merlin[18].

```
1  <item>
2    <MANDT>001</MANDT>
3    <OTYPE>O</OTYPE>
4    <OBJID>50041203</OBJID>
5    <SHORT>LL-35112</SHORT>
6    <STEXT>Software Evolution and Architecture Lab - Gall</STEXT>
7    <SUTXT9000/>
8    <SUTXT9002>Software Evolution and Architecture Lab - Gall</SUTXT9002>
9    <KATEGORIE>LL</KATEGORIE>
10   <KOSTL>0000035112</KOSTL>
11   <RSIGN>A</RSIGN>
12   <RELAT>002</RELAT>
13   <SOBID>50000492</SOBID>
14   <URL>http://seal.ifi.uzh.ch/</URL>
15 </item>
```

Listing 4.15: Example of an Organizational Unit in XML

4.4.2 Cron Job for Nightly Imports

To be able to import employees on a daily basis, we have to implement a cron job that would trigger the employee import of the `SAPImporter`. For this task, we make use of the Quartz Plugin[19], which offers an easy way for invoking methods based on time triggers. As we can see in Listing 4.16, there is not much of a code to achieve this. In the `EmployeeImportJob`, we first define a trigger with the Quartz cron expression (line 3), which then executes the code defined in the `execute()` method. Here we simple invoke the `loadProductionEmployees()` method to start the import.

```
1  class EmployeeImportJob {
2    static triggers = {
3      cron name: "Import employees every day at 2 o'clock", cronExpression: "0 0 2 * * * *"
4    }
5
6    def execute() {
7      SAPImporter.loadProductionEmployees()
8    }
9  }
```

Listing 4.16: Cron Job to trigger nightly Employee Imports

[18] Please note that the data in the XML is very incomplete. Thus, we are able to rebuild the structure only partially.
[19] http://grails.org/plugin/quartz

4.5 CSV Export of Employee Records

One of the main features of Merlin is to supply the administrators with data for reports that are sent to the EQUIS and AACSB accreditation authorities on a yearly bases. To support this feature, we implemented the `CSVRecordExporter`, which is responsible for exporting the information such as first name, last name, current appointment or education from employee records into a CSV file. Note that in this section we describe only the export of the employee data, which is only one part of the overall data export capabilities of Merlin. For details on CSV publication data export, please refer to Mark Odermatt's master thesis [Ode10].

Since in Merlin we distinguish between internal and external employees, there are two types of employee records with different information sets. As we can see in Figure 4.17, we implemented the `IExporter` interface, which declares three basic methods that every exporter must provide. The `exportHeader()` method exports headers in form of attribute names. To collect all employe records for a organizational unit, the `records(unit)` method must implement the code. Finally, the `properties(record)` method should provide the code for collecting all properties of a record. The classes `InternalEmployeeExporter` and `ExternalEmployee-Exporter` implement this interface. They are instantiated by the `CSVRecordExporter`, which provides a method `exportRecords(unit)` to export all records for a particular unit.

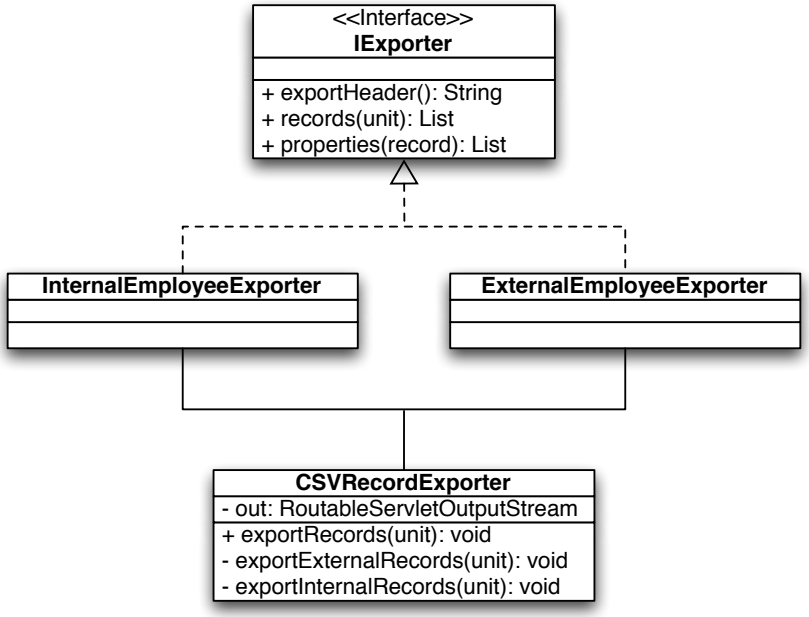


Figure 4.17: CSV exporter provides the `exportRecords(unit)` method that is responsible for exporting all employee records for a organizational unit. It uses instances of the `InternalEmployeeExporter` and `ExternalEmployeeExporter`, which implement the `IExporter` interface.

Chapter 5

Deployment Environment

"The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."

Tom Cargill

This chapter is dedicated to the deployment environment. In the first part, we will discuss the setup of the Apache HTTP-Server and the servlet container Tomcat. The second part explains the installation and setup of the Shibboleth packages for the operating system Debian. For a detailed description of the deployment process of the Grails application Merlin into this environment please refer to Mark Odermatt's master thesis [Ode10].

5.1 Server Setup

The first section of this chapter describes the involved server components. First, we will talk about the *Apache HTTP-Server* and look at some configuration examples. The second part concentrates on setup details of the *Tomcat* container. In the last part, we will describe how these two components communicate together and will provide some details on the protocol configuration.

5.1.1 Apache HTTP-Server

Apache HTTP-Server is robust, commercial-grade, featureful and freely available open source implementation of an HTTP Server. It supports a variety of features, many of which can be added as compiled modules to the core functionality. For example, the *mod_ssl* module helps to encrypt the communication between a browser and the Web server. The Apache HTTP-Server is available under the Apache license [apa10].

Let us now proceed with the Apache configuration files. In order to understand the details, we need to provide the big picture first by explaining the integration and collaboration of the Apache configuration files under Debian.

The main Apache server configuration file `apache2.conf` is located in the `/etc/apache2/` directory. This file is the central point for configuration where all settings from single distributed files are brought together. In Debian, each site can have its own configuration files, which are located in the `/etc/apache2/site-enabled` folder. As we can see in Listing 5.1 (line 3), these configuration files are included in the main `apache2.conf` file with a simple `include` command [deb10].

```
1 ...
2 # Include the virtual host configurations:
3 Include /etc/apache2/sites-enabled/
```

Listing 5.1: Apache2.conf File

For Merlin we configured two separate files. The first is named `merlin` and contains configuration for standard communication over the HTTP-Protocol. The second file, `merlin-ssl`, declares settings for secure communication over the HTTPS-Protocol. Since both files are located in the `/etc/apache2/site-enabled` directory, they are included in the main `apache2.conf` file. Listing 5.2 shows an excerpt of the `merlin` configuration. As we can see on line 1, we defined a `VirtualHost` with the address *:80, meaning that if the server receives a request on port 80, it will look for all virtual hosts with this address. One of them is the virtual host for Merlin. If the requested host name was "www.merlin.uzh.ch" (line 2), the server will use the directives specified in this file to serve the request. When users try to access a protected site in Merlin, Merlin's security system will redirect him to the *login* action. This triggers the directives specified on line 4 – the user is redirected to port 443, which is a secured communication channel (line 7).

```
1  <VirtualHost *:80>
2    ServerName www.merlin.uzh.ch
3    ...
4    <LocationMatch "^/login">
5      AuthType shibboleth
6      ShibRequireSession On
7      ShibRedirectToSSL 443
8      ShibUseHeaders On
9      require valid-user
10   </LocationMatch>
11  ...
12 </VirtualHost>
```

Listing 5.2: Apache Config for Merlin (Excerpt)

The configuration for virtual host that listens to port 443 is shown in Listing 5.3. This directive uses Merlin's certificate for its identification (verification of the public key) as service provider and the private key for the decryption of the ciphertext communicated between Shibboleth and the Apache Server (line 5-6). This procedure is necessary for a save transfer of the Shibboleth headers that contain a user's sensible data.

```
1  <VirtualHost 130.60.156.139:443>
2    ServerName www.merlin.uzh.ch
3    ...
4    SSLEngine On
5    SSLCertificateFile /etc/ssl/certs/www.merlin.uzh.ch.crt.pem
6    SSLCertificateKeyFile /etc/ssl/private/www.merlin.uzh.ch.key
7    SSLCertificateChainFile /etc/ssl/certs/QuoVadis_Root_CA_2.pem
8    SSLOptions StrictRequire
9    SSLProtocol all -SSLv2
10   ...
11 </VirtualHost>
```

Listing 5.3: Apache SSL Config for Merlin (Excerpt)

5.1.2 Apache Tomcat

Apache Tomcat is an open source implementation of the JavaServlet and JavaServer Pages technologies [tom10]. It is a widely used application server for Web application development. Grails uses Tomcat in its default development environment.

Compared to the setup of the Apache HTTP-Server, the configuration of Apache Tomcat seems to be very simple. The configuration file with the name `server.xml` holding the Tomcat settings can be found in the `/etc/tomcat6/` directory. As we can see in Listing 5.4, the only thing that has to be defined in the config file are the properties of the virtual host. In Tomcat, virtual hosts are defined within a `<Host>` tag, which has some descriptive attributes such as `name` for the unique virtual host name, `appBase` for the specification of path where the Web application is located, `unpackWARs` for automatic extraction of war files or `autoDeploy` for their automatic installation [Nie04].

The `Context` tag represents a Web application within a virtual host. A host can contain several context definitions. The `path` attribute of the `Context` tag corresponds to a unique context path that must be supplied with the URL, whereas the `docBase` attribute describes the path to the root directory of the Web application [Nie04]. As we can see on line 3 in Listing 5.4, we provided a `docBase` with the name "merlin-01", but did not define a context path. To illustrate the setup, let us consider the following example. If a request with the URI "www.merlin.uzh.ch" is sent to Tomcat, it internally maps the URI to the `/var/lib/tomcat6/webapps/merlin-0.1` directory where the Merlin application lives. This is because there is no context path information provided in the URI (it maps to the root).

```
1 <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true"
2   xmlValidation="false" xmlNamespaceAware="false">
3   <Context path="" docBase="merlin-0.1"/>
4 </Host>
```

Listing 5.4: Tomcat Config

5.1.3 Connecting Apache to Tomcat

The deployment of an Apache HTTP-Server with an Apache Tomcat is common practice. The requests are not served by Tomcat directly but are by handled by Apache first [Nie04]. Besides the need for Shibboleth Service Provider, this setup has several benefits such as faster delivery of static content or load balancing. Tomcat is able to communicate with various Web servers, one of them being the Apache HTTP-Server, through a corresponding connector.

The most common approach to interconnect Apache with Tomcat is the use of a *JK-Connector* in Tomcat with the corresponding modules for Apache. Figure 5.1 illustrates the communication, which is based on the *AJP protocol* (Apache JServ Protocol).

The communication setup is quite straight. After the installation of the *mod_jk* module in Apache, we have to activate the *JK-Connector* in Tomcat first. Listing 5.5 shows how this can be accomplished. All we have to do is to define a `Connector` as a child element to the `Service` tag in the `server.xml` file and specify its attributes (line 3). The `port` attribute sets the port number of the connector. `Protocol` specifies the protocol to be used with the connector, which is AJP/1.3 in our case. Finally, incoming requests that require a SSL connection are redirected to the port number defined in the `redirectPort` attribute [Nie04].

```
1 <Service name="Catalina">
2   ...
3   <Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>
4   ...
```

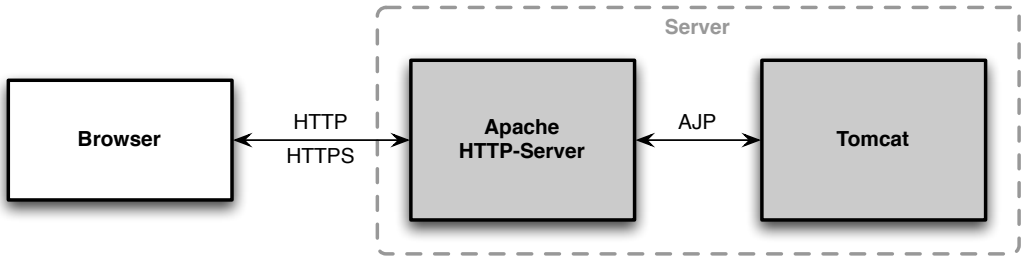


Figure 5.1: Communication between Apache HTTP-Server and Apache Tomcat based on the AJP Protocol. Figure inspired by [Nie04].

```
5 </Service>
```

Listing 5.5: Activation of AJP Protocol in Apache Tomcat Config

In order to tell Apache what port number to use for the communication, we create and set up a so-called `worker.properties` file for a Tomcat worker. A *Tomcat worker* is a Tomcat instance that is waiting to execute servlets on behalf of a Web server such as Apache, which can forward servlet requests to a Tomcat process (a worker) running behind it [wor10]. An example of a `worker.properties` definition can be found in Listing 5.6. As we can see on line 1, we first define a list of workers. In our case, the Tomcat worker instance has the name `tomcat01`. Then, we set the `port` property to the port number where the Tomcat worker is listening for `ajp13` requests and specify the name of the `host` afterwards. Finally, we set the protocol `type` to `ajp13`.

```
1 worker.list=tomcat01
2 worker.tomcat01.port=8009
3 worker.tomcat01.host=localhost
4 worker.tomcat01.type=ajp13
```

Listing 5.6: Worker File Properties

The last thing that remains to do is the update the Apache configuration file (`etc/apache2/sites-available/merlin`). This can easily be achieved by specifying the path for the `JkWorkersFile` (line 1). Additionally, we can define `JkLogFile`, which is used for log entries of the `mod_jk` module.

```
1 JkWorkersFile "/var/lib/tomcat6/conf/jk/workers.properties"
2 JkLogFile "/var/log/apache2/merlin.mod_jk.log"
3 ...
4 <VirtualHost *:80>
5   ...
6 </VirtualHost>
```

Listing 5.7: Declaration of Worker File in Apache Config

5.2 Shibboleth Setup for Authentication

A detailed explanation of the Shibboleth setup can be found on the SWITCH's home page [shi10b]. The setup basically comprises of three major steps, namely the *setup of a profile*, *installation of Shibboleth RPM packages* and *configuration* the `shibboleth2.xml` file, which is interactively being customized based on the user input.

The configuration of the Shibboleth Service Provider (SP) is concentrated in the `shibboleth2.xml` file, which is located in the `/etc/shibboleth/` directory. For documentation purposes, we will present some excerpts of its internal structure and the defined settings for Merlin. Besides containing some standard setup that comes with the xml file, a detailed description of all its tags of would go beyond the scope of this thesis. For details on the implementation of the native SP please refer to [shi10c].

Basically, there are three sections of the `shibboleth2.xml` that have to be overridden with the application specific configuration. In Listing 5.8 (line 3) we first register the host in the `<InProcess>` tag, which contains settings governing the portion of the SP that runs inside the Web server. Its internal `<ISAPI>` tag provides the ability to obtain canonical scheme, host, and port information about an incoming request [shi10c].

```
1 <InProcess logger="/etc/shibboleth/native.logger">
2   <ISAPI normalizeRequest="true">
3     <Site id="3" name="www.merlin.uzh.ch"/>
4   </ISAPI>
5 </InProcess>
```

Listing 5.8: Shibboleth Registration of new Site

To map incoming Web requests to configuration settings and the application associated with them [shi10c], we assign the host name "www.merlin.uzh.ch" to the `applicationID` *merlin* (Listing 5.9, line 3).

```
1 <RequestMapper type="Native">
2   <RequestMap applicationId="default">
3     <Host name="www.merlin.uzh.ch" applicationId="merlin">
4     </Host>
5   </RequestMap>
6 </RequestMapper>
```

Listing 5.9: Shibboleth Request Map for Merlin

Finally, since we operate another applications on the same server that uses the Shibboleth authorization service, we need to override the settings by specifying the `<ApplicationOverride>` tag with the previously defined `applicationId` "merlin" (Listing 5.9, line 3) and define the credential resolver, which contains the path to Merlin's private key and certificate (Listing 5.10, line 3-4).

```
1 <ApplicationOverride id="merlin" entityID="https://www.merlin.uzh.ch/shibboleth"
2    homeURL="http://www.merlin.uzh.ch/">
3   <CredentialResolver type="File" key="/etc/ssl/private/www.merlin.uzh.ch.key"
4    certificate="/etc/ssl/certs/www.merlin.uzh.ch.crt.pem"/>
5 </ApplicationOverride>
```

Listing 5.10: Shibboleth Application Override for Identification of a second Application

Chapter 6

Evaluation

"There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone."

Bjarne Stroustrup

This chapter describes the evaluation of the Web application Merlin that is based on user tests carried out within the Faculty of Economics, Business Administration and IT. We first discuss the test setup by providing a detailed description of the test environment, the involved users, the time schedule and the selected approach. We then continue with the user feedback by revealing the main problem areas and provide some improvement suggestions. We close the chapter with an overall conclusion.

6.1 Test Setup

The test setup has three main areas that are worth mentioning: the *test environment*, the *user base and time schedule* and the *approach* selected for the execution of the test. The following section will describe these areas individually.

6.1.1 Test Environment

The Web application Merlin has been deployed on a low-end[1] server machine running the operating system Debian 5.0. The server setup used for our tests corresponds to one described in Chapter 5. Basically, we used the Apache HTTP-Server in connection with the servlet container Apache Tomcat. For data persistency, a MySQL server was configured and preset with production data representing almost 3000 publication and nearly 600 user entries.

The tests were executed in a distributed manner meaning that each user was asked to accomplish his tasks from his office or home computer with the browser of his choice. This was an important precondition in order to test Merlin with different browsers.

6.1.2 User Base and Time Schedule

The user base contained 12 users from 4 different departments including the Software Engineering, the Empirical Research in Business, Industrial Relations and HRM, the Services and Opera-

[1] Intel(R) Pentium(R) 4 CPU 2.80GHz, 4 GB of system memory, 32-bit

tions Management and the Performance Management research groups. The aim was to build a wide spectrum of users with different computer skills in order to test the usability of the system.

The tests were executed within the time frame of one week. The exact date and time of the task accomplishment was left to each user. The time effort was approximately 30 minutes on average.

6.1.3 Approach

The users were divided into different groups according to their function in the organization. For each group a set of specific tasks has been defined. The task description containing the exact steps to be executed along with a questionnaire was sent to each test user via email. The users were asked to carry out the tasks assigned to them and to provide feedback to each of the steps with focus on user interface, feasibility and usability. The following section describes the task of each user group.

Administration

The first user group contained faculty's members of *administration* who do not belong to the active research staff such as *technicians*, *librarians* or *secretaries*. They had to execute the following tasks:

1. Login to Merlin with their UniAccess account
2. Completion of the setup wizard with personal data
3. Confirmation of the entered email address
4. Execution and confirmation of one or both of the following delegations: registration of an external employee or entry of a new publication

Researcher

The user group *researcher* contained members of the faculty who belong to active research staff. User with a function such as *senior and research assistant* or *teaching and research assistant* were assigned to this group. They were asked to execute the following tasks:

1. Login to Merlin with their UniAccess account
2. Completion of the setup wizard with personal data
3. Confirmation of the entered email address
4. Selection of five top publications
5. Validation of the generated curriculum vitae
6. Execution and confirmation of one or both of the following delegations: registration of an external employee or entry of a new publication

Chair

The group *chair* covered members of the faculty who are a chair of an organizational unit, all of them being *full professors*. The following tasks were on their list:

1. Login to Merlin with their UniAccess account
2. Completion of the setup wizard with personal data
3. Confirmation of the entered email address
4. Confirmation of incoming employee requests and validation of employee data in their organization unit
5. Entry of 1-3 new publications
6. Selection of five top publications
7. Registration of a new external employee
8. Delegation of new publication entry to a member of his organizational unit
9. Delegation of new external employee registration to a member of his organizational unit
10. Validation of the generated curriculum vitae

6.2 User Feedback and Improvement Suggestions

Most of the users returned their feedback within the specified time period. The quality of the feedback, however, varied from an incomplete feedback form to a detailed feedback on task completion, general look and feel and improvement suggestions. We got the impression that this fact correlates with the time the user spent to become acquainted with Merlin. Nevertheless, all of the feedback was considered for the final analysis. Please note that the evaluation of the user feedback in this thesis mainly focuses on the *user management* and *security related concerns* such as the feasibility of delegated tasks. For evaluation of publication related feedback please consult Mark Odermatt's master thesis [Ode10].

The following sections are structured as follows: the first part summarizes the feedback related to tasks that the users had to execute. The second part gives an overview of additional issues, suggestions and improvements.

6.2.1 Task related Feedback

As described in Section 6.1.3, users of different test groups had to execute specific tasks based on their function in the organization. Tasks such as login, completion of the setup wizard or confirmation of the email address were the same for all test users. Other tasks, such as confirmation of incoming employee requests, could be only executed by chairs. There are two main areas that can be isolated from the feedback: *login and setup wizard* and *delegation*.

Login and Setup Wizard

Fortunately, all users were able to locate the *login* button and to log into the system with their UniAccess credentials successfully. This leads to the conclusion, that this very important authentication procedure works as expected.

Most users understood the idea of the *setup wizard* and its importance related to data completion and quality. However, some of the users did not entirely fill in their personal data and complained about not being able to perform tasks such as entry of a new publication, which should be possible according to their functions. By using the administrator's `fakeLogin`[2] functionality we were able to find out that these users did not complete the setup wizard, which is necessary for the system to assign the correct access rights. This issue needs a further investigation on the motives behind the canceling of the wizard, but a possible solution would lie in a stricter setup wizard that does not allow the use of Merlin unless it has been successfully completed.

Not all users did enter a private email address while filling out their personal data. Those who did were able to successfully *confirm their email address* by clicking on the link provided in the email that is sent by the system if an email address update has been noticed. There were no issues reported related to email confirmation.

All users were able to successfully *request membership* for an organizational unit they belong to. At the same time, most of the chairs of these organizational units were able to find and to confirm the incoming requests within Merlin. All requests that have been confirmed lead to a correct connection between the requester and the organizational unit he requested for. However, there has been an *issue concerning sending multiple membership requests*. Because pending requests are only visible in the "My Departments" menu item (with a number of pending requests behind the label and a separate section for detailed information on pending requests), some users have overseen this menu item and requested their membership several times. This resulted in multiple requests of the same requester for the same organizational unit, which turned out to be disturbing for the chair of the unit. A *solution* of this issues would imply two changes: first, we have to ensure that pending requests are more visible and are not overseen by the users by improving the user interface, i.e. placing the pending requests into the user profile. Second, we have to prevent the system from sending multiple requests with the same requester and organizational unit. Both improvements can be quickly implemented and imply only small changes in the code.

Delegation

The user tests showed that there are some *usability issues with delegations*, most of them aiming on the *comprehensibility* and *workflow*. Some users did not receive any tasks. This was due to the fact that no tasks have been sent by the chairs. One user has been sent a task with a *wrong scope* so that he was not able to accomplish the described steps. On the one hand, this implies that the system works correctly, because the user should no be able to access a resource he is not allowed to. On the other hand, there seem to be a comprehensibility issue concerning the labeling of the scope of the task. A further investigation must be done in order to understand the circumstances of this situation.

One user was not able to perform the delegated task, because he *did not accept* it. This gives rise to the assumption that he was not able to figure out that a task must be viewed and accepted first in order to activate the delegated access rights. With other words, the *workflow* of the task delegation was *not understood*. A possible *solution* to this issue would be an improvement in presentation of new tasks in the task inbox, for example by making the font bold or by providing a description of the task workflow in the inbox via help text.

[2]This function was implemented for administrators so that they are able to log in as another user in case of occurring problems.

Another user reported that, after he accepted the task successfully, an *error message* was displayed after he tried to add an external employee to his organizational unit. An intense analysis of this issues revealed that the external employee has been successfully created but with *no affiliation*. This error message was caused by the fact that, after saving a new external employee, the system redirects the user back to the edit form. Because the created employee did not belong to the organizational unit the user had access rights for, he was no able to edit the newly created external employee. Consequently, the system worked correctly. Currently, Merlin allows to save employees without an organizational unit, which is justified by the fact that employee data coming from the SAP system often do not have an affiliation. If we set the `memberships` attribute of an employee to `nullable`, it would cause an amount of errors while importing data from SAP. A possible *solution* would preset the organizational unit when creating a new employee. However, this would not be possible if the user had the permission to add employees for more than one organizational unit. Another solution to this issue would be an implementation of a validator that explicitely checks, whether an organizational unit has been selected before saving. This solution has been implemented in the meantime.

6.2.2 General Feedback

This section concentrates on the overall user feedback that goes beyond the assessment of the distributed user tasks. We summarized the feedback into five categories that will be discussed individually in the following paragraphs.

Personal Data Management

Two users complained about a *loss of the entered personal data* after editing their profile information. Since the testers did not provide a detailed description of this issue, it was a difficult task to exactly reproduce the circumstances that would lead to this issue. Finally, we were able to locate the problem. It was caused by the Grails' `URL validator` that we used to validate the entered homepage URL – when an incorrect URL format was provided by the user the employee data failed to save. This issue has been closed in the meantime.

Another issue that has been mentioned by one user and was already known before the tests was the *malfunctioning* of the *personal data confirmation* functionality, which would not allow to confirm a user's personal data. In the mean time, this issue has been closed.

Finally, one user complained about not being able to upload a profile picture. Unfortunately, no details on the used browser or setup information has been submitted. After logging in with the user's name, we were able to upload a profile picture without any difficulties.

Layout and Usability

Three of the users testing Merlin suggested to make the *login button* bigger in order to better locate it in the page. Some users found the naming of the menu items not consistent enough. For example, they suggested to rename the *Tasks* menu item into *My Tasks*, which would certainly make the menu list look more consistent. Finally, the layout of the generated curriculum vitae has been criticized. This issue arises from a known problem with the used PDF Plugin (and the latest Grails version) that is responsible for rendering the PDF file out of a HTML page.

More Descriptions

Several users expressed the wish for more information on Merlin in form a *FAQ* that would explain the most critical functions and would serve as a guide at the same time. We think that this is

an excellent idea that should be implemented in a future work. Further, more help text should be provided on personal data attributes such as *email*, employee *qualification* and *organizational unit*, which seemed not to be self-explanatory. Finally, one user suggested to provide a *more detailed description of the purpose of Merlin* that should be placed on the welcome page.

Offered Information and Comprehensibility

Two users shared the opinion that the *general information about the current reporting period* on their personal homepage had no additional value for them. One of them did not understand the meaning of the number of days to deadline. We think that more investigation on this subject should be made in order to evaluate, if such information should be removed. Further, one user did not understand the meaning of the *personal data and publication confirmation*. In our opinion, a detailed FAQ section should solve this issue.

Additional Information

Two users requested additional information. The first request concerns the user's *academic degree*. Currently, Merlin supports only one academic degree for each user. If the user has for example two PhD degrees, there is no possibility to enter them both. This request is certainly of great value and should be implemented in a future version of Merlin.

The second request described the need for *additional information for organizational units* with the main focus on *teaching*. It should be possible to enter the number of ECTS points, which would reflect the teaching activities of the unit or the number of supervised master theses. A future discussion on this matter should clarify whether there is a need for such a requirement.

6.3 Conclusion

We have learned a lot from the user tests. Based on the discussion above we can conclude that the basic and most critical functionalities such as *authentication*, *authorization*, data entry in form of the *setup wizard*, *email confirmation*, *membership requests* and *confirmations*, and the *delegation* (the process of creating and sending of tasks and access rights) worked as expected. We have seen that the workflow of the task delegation was not entirely understood by all users and needs some improvements especially in the user interface area. We encountered an issue with saving external employees and validating homepage URLs. Both issues have been closed in the meantime. Further, the users requested more descriptive information in form of a FAQ and help texts. Finally, some additional information on the teaching activity of an organizational unit and the possibility to save several academic degrees have been expressed.

Chapter 7

Conclusion and Future Work

"The computer was born to solve problems that did not exist before."

Bill Gates

In this chapter, we want to retrospect the project by presenting some final thoughts. We start with our experience with the technology of choice – Grails – and discuss its advantages and drawbacks in the context of Merlin. In the second part, we bring up some general issues concerning the course of the project, the supplied data quality and its consequences. Then, we summarize our contribution. Finally, the last part of this chapter presents improvement suggestions for future work.

7.1 Developing with Grails

Grails makes Web applications development with Java easy. There is no more onerous editing of configuration files, customizing web.xml files, writing injection definitions, tweaking build scripts, dealing with tons of jar files, modifying page layouts and restarting applications [LS09]. Grails is without doubt a "next-generation" Java Web development framework, which offers the developers a lot of functionality out of the box and a great number of convenience methods. In the following paragraphs we pick out and discuss some aspects of Grails that we think are worth mentioning.

7.1.1 Agility

The advocates of the agile manifesto would describe agility as a way of life to provide a suitable and quick response to changing business requirements. It is the ability to create and to respond to change in turbulent environments [Hig06]. Compared to developing a Java Web application from scratch, Grails supports agility in a powerful way. It is easy to quickly change the code of a controller action or to update a page layout without having to restart the application. However, as we will see in a minute, there are some drawbacks to agility in Grails. Let us start with the object-relational mapping layer.

GORM

Grails' object-relational mapping (GORM) is a powerful mechanism to map domain classes onto database tables. By convention, they have the same name as the corresponding domain class.

Every domain class attribute is mapped to a separate column with its appropriate SQL type in the database. This is a very powerful feature that Grails offers out of the box, since the developer does not have to deal these mappings himself. Further, domain classes are automatically enhanced with a number of methods that allows us, together with the provided dynamic finders, to query for domain instances. This abstraction layer provides the developer with a convenient way to work with persistent data – they do not have to deal with low-level SQL statements.

Every good thing has its down sides. Because the object-relational mapping is handled by GORM, it is hidden from the developer. This fact leads to drawback when it comes to data type changes in the domain model. To explain this, we have first to go deeper into the *data source* configuration provided by Grails. Grails' data source configuration lives in the `grails-app/conf` directory in form of a `dataSource.groovy` file. Besides properties such as driver, user name and password, the data source config supports three types of autogeneration: *create*, which creates the database automatically from the domain model on application load, *create-drop* that (re)creates the database upon (re)start and *update*, which preserves the database tables between application runs.

If a developer works with the predefined *create-drop* setting in the development mode, the database is created on every application startup and dropped after stop. This can be very time consuming, especially when the domain model is not trivial. Added to the compilation time of the source code and the startup time of the provided apache tomcat application server, it can take several minutes before the application is ready to be used. This fact can be very annoying, especially when working with domain classes. Updates on domain classes always result in a recompilation of the source code and need a restart of the application.

So why do we not use the *update* option? Update does not consider changes to domain class attributes – it only updates (deletes) existing or creates new entries. Working with the update option in the development mode can result in strange errors, for example when changing the data type of a domain class attribute. Since update does not recreate the database, there is a mismatch between the domain class data type and the SQL type of the attribute in the database table. If the developer is not aware of this, it can lead to frustration and hours of debugging.

The Web application framework *rails* uses another approach to address this issue. It provides a mechanism called *migration* that helps to update the database *incrementally*. This is a much more fine-grained control of the database build-up. If you want to recreate the whole database with rails, you just need to rerun all migrations. If you just want to update the data type of one attribute in a domain class, you only need to run the latest migration – there is no need to drop the whole database [TH07]. In our opinion, this is a much more agile approach that helps to save valuable time and prevents from frustration.

Writing and Executing Tests

Grails basically offers two types of tests. On the one hand, there are *unit tests* that are used to test single classes or controllers. *Integration tests*, on the other hand, are meant to test multiple classes working together.

Unit tests are fast-running tests but they require to make extensive use of mocks and stubs. *Stabs* are classes that mimic the real behavior of methods by returning hard-coded values. *Mocks* do basically the same, but they additionally have *expectations* that specify the expected code behavior. Unit tests do not provide dynamic methods such as `save()` or `update()` and are therefore not suitable for testing persistency issues [RB09].

Integration tests bootstrap the whole environment including the database in order to provide the ability to test interactions between several classes. The loading of the entire environment is very resource intensive and time costly. For example, if you want to test the hierarchy of an organizational unit, which is basically just one method `hierarchy()` provided by the `organi-`

`zationalUnit` class, you have to write an integration test for it because the hierarchy is based on relationships between orgnazational units. Therefore, in order to test one simple method with about 10 lines of code, you will have to wait for nearly 2 minutes until you get the result. In projects with a tight time schedule and high time pressure, this much waiting time is definitely not acceptable.

7.1.2 Grails Plugins

Plugins are a very important concept for frameworks such as Grails because they provide the possibility to extend the framework with new functionalities. This helps to accelerate the software development by reuse of code. Further, plugins are usually designed by developers with a broad domain experience and are well tested. As a result, the application developer has to write and test less code himself.

Grails provides simple commands for installing and uninstalling plugins. In most of the cases, they can be found in the Grails' central plugin repository. If that is not the case, they cap be simply installed by providing the `install` command with the URL or the absolute path to where the plugin is located. In Grails, plugins are Grails applications themselves that supply hooks, which are injected into the plugin runtime. In Grails, plugins live in a separate directory external to the Grails application.

There is great number of plugins available for Grails, which are provided by the Grails community. Our experience showed us that there are mature plugins, which provide a wide spectrum of functions such as the Searchable Plugin[1], the RichUI Plugin[2] or the Quartz Plugin[3]. In their base conception, they are wrappers for existing frameworks or services. For example, the excellent Quartz Plugin is entirely based on the Enterprise Job Scheduler[4].

Other plugins used Merlin provide base functionality but are not that mature yet. For example, as we have seen in Chapter 4, there is no appropriate security plugin that would meet all requirements of a sophisticated application such as Merlin. As stated in [Ode10], we had to fix the File Uploader Plugin[5] ourselves to be able to profit from its functionality in Merlin. Another example is the PDF Plugin[6], which we used in Merlin for automatic PDF generation and which is no longer supported by the community. Due to issues with the latest Grails version 1.3.3, we had to replace this plugin with the Rendering Plugin[7].

Grails uses a separate directory for plugin storage that is external to the Grails project. The directory is located somewhere in the Grails installation folder on the machine. When installing a new plugin, Grails only updates the `application.properties` file by adding a new line with the plugin name and version (`plugins.quartz=0.4.2` as an example). After a fetch from the source code repository, the Grails application of a co-developer then downloads and installs the plugin automatically, if the plugin is available in the central Grails plugin repository. This fact makes changes to plugins more complicated. If we want to modify the plugin so it better meets our needs, there is no possibility to commit these changes to our source code repository directly from the project. In order to do so we have copy the source code of the plugin into the `plugins` folder, which must be crated first, and then tell Grails to integrate the plugin by modifying the `BuildConfig.groovy` file. This is way to complicated for small source code changes such as bug fixes.

[1] http://www.grails.org/plugin/searchable
[2] http://www.grails.org/RichUI+Plugin
[3] http://grails.org/plugin/quartz
[4] http://www.quartz-scheduler.org/
[5] http://grails.org/plugin/file-uploader
[6] http://grails.org/plugin/pdf
[7] http://gpc.github.com/grails-rendering/docs/manual/index.html

7.1.3 Groovy is cool

Groovy provides a very compact way of writing code. When compared to Java, an experienced Groovy developer can write the same code with about 50% less lines of code [RB09]. This is possible due to features such as closures or Groovy Strings (Chapter 4).

Another thing worth mentioning is Groovy's *rich API*, which provides classes such as `XMLSlurper`. XMLSlurper provides the ability to parse XML files into a document tree that may be traversed similiar to XPath expressions. Listing 7.1 shows an example of the `importOrganizationUnits(...)` method, which uses the XMLSlurper to easily traverse the organizational units XML file. We can parse the XML file with single line of code (line 7) and traverse the hierarchical tags contained in the XML by simply chaining tag selectors (line 9). This makes the XML file handling very simple.

```
1  private static final def KATEGORIES = ["LL", "LP", "LA", "LG", "F", "I"]
2
3  static def importOrganizationUnits(filePath) {
4    def orgUnitFactory = new OrganizationUnitFactory()
5
6    def file = new File("${AbstractImporter.basePath}/${filePath}")
7    def xml = new XmlSlurper().parse(file)
8
9    xml.values.GT_DATA.item.each { item ->
10     if (item.KATEGORIE in KATEGORIES) {
11       orgUnitFactory.updateOrCreateOrgUnit(item)
12     }
13   }
14   ...
15 }
```

Listing 7.1: Groovy's XMLSlurper

7.2 Other Aspects and Project Experience

Besides the choice of a technology and the implementation of concepts and ideas there are other relevant aspects related to software development that have a significant influence on the progress of the project as well. Let us have a look at some aspects related to external resources, supplied data quality or changing requirements.

7.2.1 External Resources are Barriers

The more external resources are involved in a system, the complicated the things will get. When a number of applications are involved in a system, dependencies are introduced. As discussed in Chapter 3, Merlin is supposed to cooperate with other systems in the University's application landscape such as *Shibboleth* or *SAP*.

Shibboleth

In order to make Merlin to work with the Shibboleth service, we must setup the Apache HTTP-Server configuration files accordingly (see Chapter 5 for details) and install additional packages on the server. One problem we experienced is the compatibility issue with different distributions of Linux. The precompiled packages are available for a variety of distributions but Shibboleth

does not provide a version for the distribution Fedora, which is operated on the server Merlin was originally intended to be deployed on. One possibility is to compile the required packages ourselves, which requires experience and knowledge of the Fedora distribution and, of course, time. Due to our tight time schedule we decided to deploy Merlin on another server running Ubuntu, where the packages were already installed.

Another drawback in working with Shibboleth was related to header information provided by the identity provider. In order to be able to identify employees, we requested the attribute called `userID`, which should contain the identification number of an employee (`mNumber`). Unfortunately, the identity provider was not able to provide us with this important information, so that we were forced to reimplement the code in order to indentify the users by their first name and last name.

SAP and Data Quality

One of the most important data suppliers for Merlin is the enterprise resource planning software SAP. It contains relevant information on employees and organizational structure, which should be integrated into Merlin.

After a tedious, four month lasting forwards and backwards with the legal department of the University of Zurich, we were able to agree upon two data exports: a CSV file for employee data and a XML file for organizational units. A detailed analysis of this data led to the conclusion that there is no possible way to programmatically rebuild the organizational structure with a correct assignment of the employees to their organizational units due to the bad quality and incompleteness of the provided data. Consequently, we had to redesign the whole workflow and discard the implemented functions. We were forced to find another way to assign users to organizational units. This was a quiet annoying experience because we had to deal with a new and much more complicated situation and had to provide a simple solution to the problem. The idea of a setup wizard was born.

Unfortunately, we had to realize that there is no system in the University of Zurich's application landscape that would have a clean organizational structure with a correct employee affiliation. This was a surprising discovery, which makes Merlin the first system providing this important information.

7.2.2 Changing Requirements

One of the most challenging aspects of software development is the fact that today's business environments are turbulent. This leads to unstable requirements, which change frequently. Further, customers often do not exactly know what functionality they really need to be implemented. In most of the cases, fuzzy requirements get clear in the course of the projects.

One way to address this issue is to use an agile software management method, which provides the software developer with basic but very important rules. One of these methods is *extreme programming (XP)* that was originally developed by Kent Beck, a co-author of the *Manifesto for Agile Software Development*. One of the practices of XP is the *simple design* guideline:

> *"Designing for the functionality that has been defined, not for potential future functionality."*

With other words: do not guess about the future – create the best, simple design you can today [Hig06]. It is a waste of time to design the system for future requirements that will probably never be needed.

Because the requirements for Merlin concerning user functionality and organizational structure were partially unclear at the beginning of the project, we decided to follow this guideline.

With the growing domain knowledge base things got clearer during the project. The strategy was to implement the simplest working design based on the level of knowledge, which worked well until some big changes were required. As an example, one of the changes was the insight that an employee needs to have a clone of its data that is supposed to be exported every reporting period. Another change was triggered by the recognition of the requirement for a cross-role delegation with access rights. Yet another modification was initiated by the conclusion on the provided SAP data quality. These changes led to permanent redesign of the system, which prevented the project to progress constantly.

7.3 Reflection on Contribution

In the last five month we implemented a sophisticated information management system from scratch. We started with little domain knowledge and had to struggle with new technology at the beginning.

7.3.1 Security

In order to provide the demanding requirements concerning security, we had to implement a non-trivial, *flexible*, *role-based* access control policy, which uses the University of Zurich's *AAI Login (Shibboleth)* for authentication and supports *role hierarchies* with *multiple permission inheritance*, *instance level access control* and *automatic role delegations*. This was a very challenging task since none of the Grails plugins provides such an advanced functionality. We had to design a security concept first and then find the best possible way to implement it. We came up with the idea of using the Grails filters as controller actions interceptors and implemented an *authorization service* based on the *singleton* design pattern which uses Groovy's *metaprogramming* capabilities (*reflection*) to calculate a set of entities at runtime that a user is authorized to access.

7.3.2 Functionality

Based on the security system, we provide different roles with different access permissions. A *visitor* does not have to authenticate and is allowed to browse the private section of Merlin. This includes browsing and searching the publication repository as well as the organizational structure of the University of Zurich in a simple and intuitive way.

Members of the Faculty of Economics, Business Administration and IT must authenticate first in order to use the wide functionality of Merlin. An *employee* is a staff member that does not contribute to the research activities of the faculty. He is allowed to edit his own profile, confirm his personal data, upload a profile picture to personalize the profile page and execute tasks that are delegated to him by chairs. To support the user after his first login, we provide an intuitive setup wizard, which automatically enhances the user with the corresponding access rights according to his data input.

Researchers are allowed to enter their publications, define top five publications, upload their research work to ZORA with a single click and download a personalized academic curriculum vitae containing their personal data, education information and top five publications.

A *chair* is the team head of an organizational unit. To be able to perform his duties, Merlin offers a set of chair specific functionalities such as answering employee requests (for maintaining data quality and tracking of employee affiliations), delegating tasks and access rights to other users with a simple, predefined set of scopes, adding external employees to his organizational units, managing personal and publication data confirmations and editing organizational units.

Finally, an *administrator* is able to manage publication periods, change chair of an organizational unit, send reminder to employees who have not confirm the legal agreement yet, view and manage access rights of every employee by just updating his user profile and export period records to CSV.

7.3.3 User Interface and Design

Merlin is the first application in the University of Zurich's application landscape implementing the new corporate design guidelines. It provides its users with intuitive, clear design with the concern of not overloading the users with too much information. Therefore, Merlin structures its information into coherent and well-arranged sections as we can see on the setup wizard example (Figure 4.12). The use of modern Web technologies such as JavaScript (e.g. scriptaculous framework for dynamic visual effects) and AJAX provides a desktop-like user experience.

7.3.4 Conclusion

In our opinion, Merlin was a success. We implemented all of the required functionality and provide even more features that were not in the scope of the system at the beginning. We were able to manage the frequent requirement changes, delays caused by the legal department and setbacks triggered by the poor SAP data quality or the redeployment of Merlin on a different server by implementing an agile project management approach. We share the opinion that under the prevailing circumstances and with our high engagement we have reached far more than the implementation of the initially required functionality.

7.4 Future Work

A software solution of the size of Merlin will never by perfect for every user. There might be ideas for improvements that address features, design, usability or performance.

One aspect that should be scrutinized is the *performance* under a full load. We did not have the opportunity to test the behavior and response times of the system when for example 100 users send a membership request for an organizational unit at the same time. This is due to the fact that the production server Merlin is intended to run on is not known yet. However, we have done some performance improvements related to autocompletion response time issues.

Another aspect that is worth mentioning are *automated tests*. From our point of view, the automated test basis should be extended. We implemented several tests that address some critical functionality such an employee import, organizational structure, role inheritance or the export of employee records to CSV. Due to the high time pressure and the testing issues described in Section 7.1.1, we were not able to test each and every functionality provided by Merlin.

Many projects (and almost all open source ones) use a technique called *continuous integration*, whose basic idea is that a server continuously checks out the latest source code for the project, builds it and then runs a set of defined tests. If there is failure, it sends a notification so that the team gets a fast feedback in case the build is broken. There are several products available on the market, but none of them has a direct support for Grails applications. Instead, we would have to use a shell script, batch file, Ant build or something similar in order to execute the relevant Grails commands and scripts [LS09]. We hope that the current products available on the market will add support for Grails application soon.

Finally, there might be some possible improvements related to *workflow and usability*. Admittedly, we had a small user basis testing the application but in order to draw an overall conclusion,

the application must be tested in a real world scenario with a large number of users. The future will show whether the implemented operations layout conforms to the high expectations of every single user of Merlin.

Appendix A

A List of used Tools

A.1 Source Code and Project Management

- **Jetbrains IntelliJ IDEA** - Integrated development environment (IDE)[1]
- **Git** - Version Control System[2]
- **Redmine** - Web application for project management [3]
- **Capistrano** - automated execution of repeatable tasks for Web application deployment[4]

A.2 Grails Plugins

- **Grails Mail** - Sending mails with Grails[5]
- **Quartz Plugin** - A plugin for job scheduling[6]
- **Searchable Plugin** - Full-text search for Grails applications[7]
- **Burning Image** - Image manipulation and image upload handling[8]
- **PDF Plugin** - Plugin for PDF generation from existing pages [9]
- **RichUI Plugin** - A set of AJAX components for a rich user interaface[10]

[1] http://www.jetbrains.com/idea/
[2] http://git-scm.com/
[3] http://www.redmine.org/
[4] http://www.capify.org/index.php/Capistrano
[5] http://grails.org/plugin/mail
[6] http://grails.org/plugin/quartz
[7] http://www.grails.org/plugin/searchable
[8] http://www.grails.org/plugin/burning-image
[9] http://www.grails.org/Pdf+Plugin
[10] http://www.grails.org/RichUI+Plugin

A.3 Javascript

- **Prototype 1.6** - JavaScript Framework[11]
- **Scriptaculous** - JavaScript library for dynamic visual effects[12]
- **dTree** - A free JavaScript tree menu[13]

[11] http://www.prototypejs.org/
[12] http://script.aculo.us/
[13] http://destroydrop.com/javascripts/tree/

Appendix B

Content of the CD-ROM

Each copy of this thesis containts a CD-ROM. The contents of the CD-ROM are:

B.1 Thesis

- **Masterthesis.pdf**
  A copy of the written thesis.
- **Abstract.pdf**
  English version of the abstract of the thesis.
- **Zusfsg.pdf**
  German version of the abstract of the thesis.

B.2 Framework and Application

- **Grails_1.3.3.zip**
  Binary release of the grails framework.
- **Merlin.zip**
  Source code of Merlin including all plugins.
- **Merlin.war**
  Compiled source code of the Merlin Web application.
- **Merlin_SQL.zip**
  SQL-dump of a working Merlin-database.

B.3 SAP Data Files used for Import

- **Employees.csv**
  Personal data of all employees of the University of Zurich's Faculty of Economics, Business Administration and IT.

- **OrganizationalUnits.xml**

  Organizational Units data of the University of Zurich's Faculty of Economics, Business Administration and IT.

References

[aai10] About aai, August 2010. https://www.switch.ch/aai/about/, (last visited: 10. August 2010).

[ace10a] Acegi plugin, August 2010. http://grails.org/plugin/acegi, (last visited: 18. August 2010).

[ace10b] Grails - acegisecurity plugin - implementation overview, August 2010. http://www.grails.org/AcegiSecurity+Plugin+-+Implementation+Overview, (last visited: 18. August 2010).

[apa10] About apache, August 2010. http://httpd.apache.org/ABOUT_APACHE.html, (last visited: 22. August 2010).

[CC07] Liang Chen and Jason Crampton. Inter-domain role mapping and least privilege. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 157–162, New York, NY, USA, 2007. ACM.

[CK08] Jason Crampton and Hemanth Khambhammettu. Delegation in role-based access control. *Int. J. Inf. Secur.*, 7(2):123–136, 2008.

[deb10] Debian administration, August 2010. http://www.debian-administration.org/articles/412, (last visited: 22. August 2010).

[Fer95] D.Richard Kuhn David F. Ferraiolo. Role-based access control (rbac): Features and motivations. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 1995.

[Fer01] David F. Ferraiolo. An argument for the role-based access control model. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 142–143, New York, NY, USA, 2001. ACM.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[FK95] David Ferraiolo and Richard Kuhn. An introduction to role-based access control. *NIST CSL Bulletin on RBAC*, 1995.

[FKC07] David F. Ferraiolo, Richard D. Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control, Second Edition*. Artech House, Inc., Norwood, MA, USA, 2007.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[Her07] Michael Hermann. A semantically annotated publication database using ajax. Master's thesis, University of Zurich, February 2007.

[hib10] Hibernate, August 2010. http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/, (last visited: 14. August 2010).

[Hig06] Jim Highsmith. *Agile software development ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[HTC99] Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 1999.

[Kle09] Dave Klein. *Grails: A Quick-Start Guide*. Pragmatic Bookshelf, first edition, 2009.

[Lat85] Donald C. Latham. Department of defense standard department of defense trusted computer system evaluation criteria, 1985.

[LS09] Glen Ledbrook, Peter Smith and Brenda Smith. *Grails in Action*. Manning Pubn, 2009.

[ML99] Jonathan D. Moffett and Emil C. Lupu. The uses of role hierarchies in access control. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 153–160, New York, NY, USA, 1999. ACM.

[NC00] SangYeob Na and SuhHyun Cheon. Role delegation in role-based access control. In *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*, pages 39–44, New York, NY, USA, 2000. ACM.

[Nie04] Stephan Niedermeier. *Cocoon 2 und Tomcat*. Galileo Press GmbH, 2004.

[Ode10] Mark Odermatt. Managing publications with the faculty information system merlin, August 2010.

[PSA01] Joon S. Park, Ravi Sandhu, and Gail-Joon Ahn. Role-based access control on the web. *ACM Trans. Inf. Syst. Secur.*, 4(1):37–71, 2001.

[RB09] Graeme Rocher and Jeff Brown. *The Definitive Guide to Grails*. Apress, second edition, 2009.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29:38–47, 1996.

[shi10a] About shibboleth, August 2010. http://shibboleth.internet2.edu/about.html, (last visited: 10. Augutst 2010).

[shi10b] Deployment of shibboleth service provider (sp) 2.3.1 on linux with rpm packages, August 2010. https://www.switch.ch/aai/docs/shibboleth/SWITCH/2.3/sp/deployment/linux-rpm.htmlsetupprofile, (last visited: 22. Augutst 2010).

[shi10c] Nativespshibbolethxml, August 2010. https://spaces.internet2.edu/display/SHIB2/NativeSPShibbolethXML, (last visited: 22. Augutst 2010).

[sit10] Sitemesh overview, August 2010. http://www.opensymphony.com/sitemesh/, (last visited: 14. August 2010).

[spr10] Introduction to spring framework, August 2010. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/overview.html, (last visited: 14. August 2010).

[SS94] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32:40–48, 1994.

[TH07] Dave Thomas and David H. Hanson. *Agile Web Development with Rails*. The Pragmatic Programmers, Raleigh, North California, second edition edition, 2007.

[tom10] Apache tomcat, August 2010. http://tomcat.apache.org/, (last visited: 23. Augutst 2010).

[wor10] The apache tomcat connector - generic howto, August 2010. http://tomcat.apache.org/connectors-doc/generic_howto/workers.html, (last visited: 23. Augutst 2010).

[zor10] Zurich open repository and archive, August 2010. http://www.zora.uzh.ch/, (last visited: 10. August 2010).