**Security for software-aided event-facilitation**
**A holistic approach to comprehensive authentication & authorization**

by Marvin Botens
January 2019

Approved: _____ 22.02.2019
Prof. Dr. Stephan Karczewski (HDA) — Date
Committee Chair

Approved: _____ 22.02.2019
Prof. Dr. Christoph W. Wentzel (HDA) — Date
Committee Member

Approved: _____ 3/4/2019
Prof. Dr. Yan Shi (UWP) — Date
Committee Member

Suggested content descriptor keywords:

security, authentication, authorization, WebSocket, OAuth 2.0, RBAC, digital facilitation

# Security for software-aided event-facilitation

*A holistic approach to comprehensive authentication & authorization*

A thesis presented to the Graduate Faculty

**University of Wisconsin-Platteville**

In Partial Fulfillment of the Requirement for the Degree

**Master of Science**

by

Marvin Botens
2019

| | |
|---|---|
| Supervisor: | Prof. Dr. Stephan Karczewski (HDA) |
| Assistant Supervisor: | Prof. Dr. Christoph Wentzel (HDA) |
| Assistant Supervisor: | Prof. Dr. Yan Shi (UWP) |

# Declaration of Authorship

I, Marvin Botens, hereby declare that this thesis is written on my own solely by using the sources quoted in the thesis body. All contents (including text, figures, and tables), which are taken from sources, both verbally and in terms of meanings, are clearly marked and referenced.

In the context of the JIM Double Degree, this thesis is, in a similar form, also submitted to the Darmstadt University of Applied Sciences in Darmstadt, Germany.

Darmstadt, January 11, 2019

Marvin Botens

# Confidentiality clause

This thesis shall not be copied, published or disclosed to third parties, neither fully nor in parts, without the written agreement of the author.

Darmstadt, January 11, 2019

Marvin Botens

Hochschule Darmstadt

# *Abstract*

Fachbereich Informatik

Master of Science

**Security for software-aided event-facilitation**

*A holistic approach to comprehensive authentication & authorization*

by Marvin Botens

**Statement of the Problem**

Digital facilitation refers to the modeling and execution of facilitation workflows by use of specialized software systems. Originally limited to pure offline setups, the application environment of digital facilitation systems is now undergoing a transformation to online and hybrid deployments. During transformation, several challenges are arising in terms of those systems' security measures.

For one thing, the diverse set of protocols (HTTP, WebSocket, polling, etc.) used to achieve reliable real-time communication brings the need for comprehensive authentication of users with verifiable security characteristics.

Secondly, it is in the nature of things that people with different company or union affiliation (event organizer, agency, full-service provider, etc.) are collaborating to plan, realize and follow up on events. Thus, other than with company internal software, where authorization/access control represents internal and partially rigid policies, facilitation systems need to provide broad flexibility to reflect a very dynamic distribution of responsibilities.

Third and lastly, architectures that have been developed with a focus on pure offline usage are often not built to support the security requirements encountered in online environments. To avoid massive architectural changes during the transformation, there is need for non-invasive security enhancements that are capable of protecting a system, while requiring as minimal intervention to existing code as possible.

### Methods and Procedures

Initially, a literature research was conducted to find and evaluate methods and reference architectures for authentication at the HTTP and WebSocket protocols as well as available authorization concepts.

Subsequently, a case study of teambits:interactive, a product of the teambits GmbH in Darmstadt, Germany, was performed. It aimed to develop methods for applying security to facilitation systems that are evolving from pure offline to online/cloud-ready applications.

In that context, interviews with staff members of teambits were carried out to gain awareness about the requirements during the transformation. Based on the acquired insights, architectural concepts for authentication and authorization were worked out with a focus on least possible invasion into the system's current architecture and implementation.

Finally, the elaborated architecture was discussed and analyzed for its transferability to other systems.

### Summary of Results

The elaborated solution makes use of the OAuth 2.0 framework for authentication. It thereby separates distinguishable concerns such as sing-in procedures, credential processing and user administration and extracts them from application specific parts into a generic and reusable authorization service.

Specific solutions were then developed for the system's existing APIs to delegate authentication to the externalized service. In order to make those developments non-invasive on the part of the existing application, configurable concepts from the Spring Security framework as well as aspect-oriented programming (AOP) techniques were applied.

For authorization, a custom variant of the role-based access control (RBAC) model was derived. Its realization in the at-hand system architecture likewise builds on the usage of Spring Security and AOP techniques.

The derived solution was found to be transferable to a great extent. Although no actual transfer was carried out, it is perceived that modularization and the intense usage of framework components as well as AOP allows to apply the developed aspects to other facilitation systems without major modifications.

# *Acknowledgements*

I would like to express my gratitude to the wonderful family I have; my parents, Petra and Reinhard, whose support encouraged me on my way here, my little sister Fabienne, who is like a best friend to me, and my grandparents Emmi and Ernst, who are there whenever I need them.

Huge admiration goes to my girlfriend Romana for being understanding and generous in every situation.

I am especially grateful to my advisors Prof. Dr. Stephan Karczewski, Prof. Dr. Christoph Wentzel and Prof. Dr. Yan Shi. Thanks so much for your effort and time reading my drafts, providing feedback and answering my questions.

Special thanks also goes to my colleague Axel Guicking for providing valuable and most in-depth feedback to my writing and to Dr. Peter Tandler for supporting me with tips and ideas. I am grateful to all at teambits for giving me the chance to work and research in an exciting environment as well as for all the inspiring input I was provided.

# Contents

**Declaration of Authorship** i

**Confidentiality clause** ii

**Abstract** iii

**Acknowledgements** v

**List of Figures** ix

**List of Tables** x

**List of Abbreviations** xi

**1 Introduction** 1
- 1.1 Digital Live Facilitation - An Overview . . . 2
  - 1.1.1 Facilitation Basics . . . 3
  - 1.1.2 Classification of Facilitation Techniques . . . 4
  - 1.1.3 Depicting Facilitation Techniques in Software Systems . . . 6
  - 1.1.4 A World Café in teambits:interactive . . . 7
  - 1.1.5 Evolution of Digital Facilitation Systems . . . 10
- 1.2 Status-Quo & Current Knowledge . . . 11
  - 1.2.1 Basic Features . . . 12
  - 1.2.2 Security Aspects . . . 12
- 1.3 Knowledge Gap & Research Goals . . . 14
- 1.4 Outline . . . 14

**2 Literature Review** 18
- 2.1 Authentication Concepts . . . 18
  - 2.1.1 Authentication for HTTP . . . 18
  - 2.1.2 Authentication for WebSockets . . . 21
- 2.2 Authorization Concepts . . . 23
  - 2.2.1 Discretionary and Mandatory Access Control . . . 23

2.2.2 Role-Based Access Control . . . 24
2.2.3 Attribute-Based Access Control . . . 27

**3 Methods** **30**
3.1 Staff Interviews & Requirement Engineering . . . 30
3.2 Architectural Design . . . 31
3.3 Discussion & Transferability . . . 32

**4 Requirements** **33**
4.1 Situation Analysis . . . 33
4.2 Situation Evaluation . . . 34
4.3 Identified Needs . . . 35
4.4 Ranked Requirements . . . 36
Authentication . . . 36
Authorization . . . 37

**5 Authentication** **38**
5.1 Authentication Processing . . . 38
5.2 Central Authentication . . . 42
5.3 REST API Authentication . . . 46
5.4 Messaging API Authentication . . . 49
5.4.1 Message Handling . . . 49
5.4.2 Authenticating WebSocket Connections . . . 52
Current Architecture . . . 52
Spring Security WebSocket Support . . . 54
WebSocket Authentication via Tickets . . . 56
WebSocket Authentication via Personalized One-Time URLs . . . 58
5.4.3 Authenticating Polling Connections . . . 62
5.4.4 Establishing the Security Context for Message Processing . . . 63

**6 Authorization** **68**
6.1 Designing the Authorization Concept . . . 68
6.2 The Permission Evaluator . . . 72
6.3 Integration into the messaging API . . . 76
6.4 Integration into the REST API . . . 78

**7 Evaluation** **81**
7.1 Reflection & Discussion . . . 81
7.2 Transferability . . . 84

**8 Conclusion** **88**

8.1 Limitations . . . . . 89
8.2 Recommendations . . . . . 89
8.3 Further Work . . . . . 89

**A Transcript of Staff Interviews** **91**

**Bibliography** **98**

# List of Figures

1.1 A Word Cloud in teambits:interactive . . . 9
1.2 Client Types in teambits:interactive . . . 16
1.3 The Meeting Cockpit in teambits:interactive . . . 17

2.1 OAuth2 Architecture . . . 20
2.2 NIST models for Core & Hierarchical RBAC . . . 25
2.3 NIST models for SSD and DSD RBAC . . . 26
2.4 Attribute-based access control (ABAC) Authorization Architecture . . . 28

5.1 Spring Security Authentication Manager Architecture . . . 40
5.2 Authentication Manager Configuration . . . 41
5.3 Spring Security Filter Chain . . . 44
5.4 Spring Security OAuth2 for a REST API . . . 48
5.5 Messaging API . . . 50
5.6 Messaging API – WebSocket Connection . . . 53
5.7 Messaging API – WebSocket authentication via OAuth2-backed tickets . . . 57
5.8 Messaging API – WebSocket authentication via personalized one-time URLs . . . 59
5.9 Polling Connection Cycle . . . 63
5.10 AOP Proxies . . . 65
5.11 Messaging API with Authentication . . . 66

6.1 RBAC data model with parameterized UA . . . 72
6.2 Spring Security AccessDecisionManager Architecture . . . 74
6.3 RBAC PermissionEvaluator . . . 75
6.4 Spring Security – Security Interception . . . 76
6.5 Messaging API with Authentication & Authorization . . . 78

# List of Tables

5.1 Spring Security Filter Ordering . . . . . 45

# List of Abbreviations

**AA** attribute authority. 28

**ABAC** attribute-based access control. ix, 27–29, 68–70, 82, 84, 88

**ACK** acknowledgement. 53, 56, 58, 60, 61

**ACL** access control list. 24, 27, 68, 69, 74

**AOP** aspect-oriented programming. iv, 39, 65, 77–79, 86

**API** application programming interface. iv, vii, ix, 14, 18–22, 31, 39, 41–43, 46, 49–57, 59, 61–64, 66, 68, 76–79, 83–85

**BYOD** bring your own device. 10, 19

**CSRF** Cross-site request forgery. 54

**DAC** discretionary access control. 23, 24, 27, 68, 82, 88

**DAO** data access object. 41

**DMZ** Demilitarized Zone. 29

**DoS** denial-of-service. 63

**DSD** dynamic separation of duties. 25, 26

**HCI** human-computer interaction. 30

**HTML** Hypertext Markup Language. 43

**HTTP** Hypertext Transfer Protocol. iii, iv, vi, 2, 14, 18–21, 31, 38, 42–44, 46, 47, 49, 51, 52, 54–56, 59, 61–64, 73, 76–81, 88, 89

**I/O** input/output. 52

**ID** identifier. 53, 55, 57, 58, 60, 62

**IETF** Internet Engineering Task Force. 19

**IP** Internet Protocol. 55

**JDBC** Java Database Connectivity. 75

**LDAP** Lightweight Directory Access Protocol. 40

**M2M** machine to machine. 38

**MAC** mandatory access control. 23, 24, 27, 68, 82, 88

**NIST** National Institute of Standards and Technology. 25–27, 70–72, 82, 84

**OBS** objects. 26

**OOP** object-oriented programming. 39

**OPS** operations. 26

**OS** operating system. 35

**PA** permission-role assignment. 25, 26, 70–72

**PDP** policy decision point. 28

**PEP** policy enforcement point. 28

**PRMS** permissions. 26

**RBAC** role-based access control. iv, 24–28, 68, 70–75, 77, 79, 82, 84, 87–90

**REST** Representational State Transfer. vii, 12, 18, 31, 41–43, 46, 49, 51, 56, 62, 76–79, 83

**RFC** Request for Comments. 19, 52

**RQL** result query language. 46

**RTM API** Real Time Messaging API. 22

**RTSC** Real Time Strategic Change. 4, 7

**SaaS** Software as a Service. 1, 90

**SAML** security assertion markup language. 20, 28, 43, 45

**SOA** service-oriented architecture. 29, 46

**SoC** separation of concerns. 39, 41

**SoD** separation of duties. 24, 70, 82

**SpEL** Spring Expression Language. 73–75, 79

**SQL** Structured Query Language. 42

**SSD** static separation of duties. 25, 26

**SSO** single sign-on. 20, 35, 45–47, 83

**TCP** Transmission Control Protocol. 50, 51, 53, 62

**TCP/IP** Transmission Control Protocol/ Internet Protocol. 49

**TLS** transport layer security. 18, 19

**UA** user-role assignment. 25, 70–72, 75, 84

**UI** user interface. 31, 35, 36, 39, 43

**UML** Unified Modeling Language. 41, 50, 53, 57, 59, 66, 78

**URI** Uniform Resource Identifier. 13, 19, 21, 44, 45, 73

**URL** Uniform Resource Locator. vii, ix, 21–23, 44, 45, 51, 52, 54–59, 61, 62, 81, 88

**UX** user experience. 35, 47

**WAF** web application framework. 22

**XACML** extensible access control markup language. 28

# Chapter 1

# Introduction

Digital live facilitation strives to increase the added value of events by modeling the information flow between participants, facilitators and stakeholders in a software system. Very often those systems also cluster information and generate advanced insights from it. While this basic idea is already known, explored and practiced since more than a decade, it is currently undergoing changes in the environment that it is being used in. Traditionally, live events involved a strict offline-nature, in that most or all of the software utilized needs to run isolated, i.e. locally on the event location and without internet connection. This was mainly due to availability and security reasons. These days, however, a change in thinking is happening. An increased number of events is held remotely, with participants and facilitators distributed over various locations all over the globe. Also, due to the rise of modern cloud-hosted Software as a Service (SaaS) applications, expectations for global and continuous availability of applications as well as their ease-of-use are rising. This structural change in thinking brings new requirements to a software that aims to be utilized for the facilitation of modern live-events.

teambits GmbH, a company located in Darmstadt, Germany, is finding itself confronted with the aforementioned requirements. Starting development about 15 years ago, they were one of the pioneers in the evolving area of digital event-facilitation. These days however, with their software being designed for a pure offline environment, several challenges are arising during the transformation to a cloud-ready online application.

At the same time, new market players are providing lightweight, multi-tenant-ready and highly self-serviceable SaaS solutions and are thereby massively demanding big parts of the market. While the strength of the teambits software is a heavy toolset that allows massive customization of the information flow, an enhancement of usability as well as a rethinking of current security practices is required to further compete with those new players. More precisely, teambits needs to extend their software by

advanced authentication- and authorization mechanisms that support the previously introduced online scenarios.

This thesis examines authentication and authorization issues in software-aided event facilitation. It works out applicable solutions for a comprehensive security system with a particular view on the teambits software. The research will be conducted by a case study of teambits:interactive. Within this case study, the software's architecture shall be analyzed with a focus on security-relevant aspects. Finally, concepts shall be worked out to improve security and increase usability.

The following is a list of the more precise objectives. Initially, a research shall be conducted to find and evaluate methods and reference architectures around authentication within the HTTP and WebSocket protocols. Likewise, available considerations about access control concepts shall be examined. In terms of methodology, it is then intended to perform interviews with staff members of teambits to gather concrete requirements. Based on the latter, a solution shall be worked out that builds on the current application architecture. In a critical evaluation, the elaborated solution shall eventually be discussed and analyzed for its alignment with the gathered requirements. It shall also be abstracted from the details of teambits:interactive and assessed in terms of its transferability to other applications. To conclude the thesis, the general validity of the elaborated solution shall be evaluated and advices as well as next steps shall be recommended for teambits:interactive to succeed in the new market environment.

## 1.1 Digital Live Facilitation - An Overview

Conventional facilitation techniques make use of tools such as whiteboards or flip-charts for capturing and visualizing contributions and results. When an event involves interactions such as votings or brainstormings, information is often collected from participants on paper cards and processed by facilitators and/or editorial staff. Final results eventually need to be captured and made available to all participants in a process that is very often performed manually by editors [FB14]. While the usage of the aforementioned analog tools and techniques proofed to be very beneficial, it also bears some limitations, the most obvious being a natural limit of the group size and quantity of information that editors can efficiently manage [TKS13].

Digital facilitation addresses those limitations by supporting both the facilitation- and the editorial process through adequate soft- and hardware tools (e.g. software for votings and brainstormings, smartphones, tablets, screens and projections). In principal, those tools are substitutes for conventional utensils such as flip-charts, whiteboards

and paper cards, that automate the collection and visualization of ideas and opinions with larger quantities of participants.

In practice software products often tend to model specific facilitation techniques; see, for example, SixSteps, a tool that aims to reflect the most common steps in classical outcome-oriented staff meetings [Six].

In theory however, those products should be distinguished from the actual conceptual methods they support. Digital facilitation makes use of both conventional techniques – those that were traditionally practiced using whiteboards and paper cards – and advanced techniques that only arise from the possibilities introduced by the use of digital technologies [TKS13].

The remainder of this section will strive to provide a better understanding of the problem addressed by digital facilitation systems. Initially, it will zoom into the domain of facilitation and outline how it differs from other fields such as moderation. Further considerations will examine how facilitation evolved from the isolated treatment of small organizational sub-groups to the collective involvement of whole systems. A classification scheme for available facilitation techniques will be introduced that also helps to identify those methods that benefit most from the application of digital information processing. By looking at the use case of a so-called *World Café* in teambits:interactive, an example for digital facilitation processes will be given. Finally, to zoom out again from those in-depth inspections, focus will be shifted back to the technical objective of this thesis. In that context, some general observations will be made about the evolution of digital facilitation systems from local offline applications to cloud-based self-service platforms.

### 1.1.1 Facilitation Basics

The terms facilitation and moderation are often used in an interchangeable fashion. Yet, a differentiation for that terminology is actually possible and reasonable. The considerations given in [Bod05, p. 251] suggest the following distinction:

Moderation can be understood as a means to support and structure a particular work format. A moderator's primary responsibility in a group discussion, for instance, is to control the flow of information. Typical tasks of a moderator are the introduction of new topical elements, the probing for deeper understanding of participant statements, the exploration of discrepancies in held opinions and the mediation between group members. Facilitation, on the other hand, is described as a closer control of the group interaction. A facilitator more tightly steers the discussion by asking precise questions

or presenting content-related stimuli. He also facilitates the process of collecting data from the group.

In accordance with the aforesaid, [Lou10] describes the moderator as a less visible information manager while the purpose of the facilitator is seen in a rather prominent control of communication flow. Similar distinctions can furthermore be found in [Doo17]. Here, the author suggests moderation as the guidance of a discussion, while the focus of facilitation is on the implementation of more specific processes. Facilitation embraces a stronger aiming for precise results and decisions and strives to link participant responses to the goals of the defined process.

When talking about facilitation in the context of this thesis, we are referring to the work with large groups or whole systems as introduced by Ruth Seliger in [Sel15, p. 13f.]. In this context, emphasis is on the interaction with participants as part of result-oriented facilitation techniques in organizations. In [Sel15, p. 7f.], Seliger describes how organizational development in the german-speaking area of Europe went through a major transformation in the late 1990's, when methods for working with large groups spilled over from America:

Prior to the availability of such methods, it was common for consultants in Germany and Austria to split up organizations into teams and groups of approximately 12 people, align them in a sequential order of recurring workshops and then iteratively work through a topic. This approach was limited in terms of maximum group size and speed. With large organizations, there was high risk that things would change while the sequentially aligned workshops are still dealing with the initially defined set of goals. Moreover, the sequential nature of this methodology slowed down the progression of a process.

New methods developed by pioneers from the U.S. include *Future Conferences*, *Real Time Strategic Change (RTSC)*, *Open Space*, *World Café* and many more. Those techniques introduced the new possibility of working with the whole system of an organization collectively and simultaneously, instead of splitting it up into isolated workshop groups. By that, consultants and organizations are capable of handling complex topics and change processes in a time- and resource-saving manner. As a consequence, those methods do nowadays belong to the standard toolset of consultants that are dealing with organizational development [Sel15, p. 8,15,105].

### 1.1.2 Classification of Facilitation Techniques

Over the years, a large variety of techniques has evolved. In their advisor, *The Change Handbook*, Peggy Holman and her co-authors are talking about some 60 methods that

have already been available to facilitate whole system change processes in 2007. As a consequence, some sort of categorization became necessary to maintain an overview [HDC07, p. 16].

Holman elaborated a *Summary Matrix* containing seven characteristics for a quick, at-a-glance comparison and contrasting of available methods. The characteristics she relies on are *Purpose*, *Type of System*, *Event Size*, *Duration*, *Cycle*, *Practitioner Preparation* and *Special Resource Needs*. The following list gives a quick outline for each of those characteristics [HDC07, p. 16-21]:

**Purpose** The authors identified five dimensions of purpose from which one should be assignable to any facilitation technique. *Planning*, *structuring* and *improving* are the three most specific purposes a method can follow. *Adapt* and *support* are more generic dimensions. The former usually spans multiple specific purposes, the latter strives to enhance the efficacy of work, independent of its purpose.

**Type of System** The title of this characteristic might be somewhat misleading. The term system refers to the whole system or type of large groups addressed by a method. It should not be confused with the method type. Holman distinguishes two types of systems: On the one hand, there are *Organizations*. They usually have a clearer structure in terms of relationships between employees, customers and suppliers and thus a better awareness about who needs to participate in a facilitated development/change process. On the other hand, there are *Communities*, which are often practicing a rather diffuse structure and a less clear understanding about who the intended process actually concerns.

**Event Size** Methods can be distinguished by the size of events or groups for which they work best. Some may be suitable for small groups only, while others also or exclusively apply to larger systems. Methods for larger groups often benefit from proficient allocation of subgroups or use of adequate technology.

**Duration** This characteristic strives to provide approximate measures for the time required to prepare, carry out and follow-up on methods. It also tries to give an idea about the pace a method allows and the urgency for results it can satisfy.

**Cycle** The following cycles can be distinguished: *As Needed* refers to methods that are used to fulfill an intended purpose in an ad-hoc fashion. They do not foresee repetitive application if not explicitly required by the purpose they are used for. *Periodic* methods, in contrast, are repeated over time. They are typically used for planning purposes. *Continuous* methods, finally, are loosening the dependence on dedicated events. Instead, they are rather intended to integrate in the organizations' regular processes.

**Practitioner Preparation** This characteristic is about two things: The time required to prepare a method and the skills needed to succeed. Holman allocates methods into three classes: The class of *Self-Directed Studies* contains methods that allow any facilitator with background in group work to succeed. For another class, *General Training*, it is advisable to attend a workshop, specifically about the method intended to use. The final class is referred to as *In-depth Training*. It requires significant professional guidance. Often there is formal training and certification from a governing body for methods of that class.

**Special Resource Needs** Those are needs that go beyond the regular demands of events. It includes the reliance on an unusual amount of people as well as special technology requirements, such as specific hard- and software.

Together, those characteristics are capable of giving a general classification for old and new facilitation methods. Thus, when learning about new techniques, it can be helpful for professional facilitators to request or determine a short statement for each of the seven characteristics, in order to get a quick understanding about the technique and whether it is useful for the currently aspired goals. In their book, Holman and her co-authors are providing their *Summary Matrix* with a full classification for all of the approximately 60 methods that have been available at that time [HDC07, p. 22-27].

### 1.1.3 Depicting Facilitation Techniques in Software Systems

Companies in the domain of digital facilitation are regarding it as their duty to depict popular examples of the available facilitation methods in their software solutions. For this purpose, the classification introduced in the previous section can also be of substantial value.

For example, when looking at the *Event Size* characteristic, we find that methods for high numbers of participants are using conceptual elements that can be enhanced by digital information processing. Take the *World Café*, for instance. It is applicable for up to a thousand participants and is based on the idea of generating information in group discussions and then transferring it to the plenum in small presentations [Sel15, p. 105]. Transferring elaborated information from small groups to the plenum is actually a common idea and can be found in a variety of methods. It is, for instance, also incorporated in the *Open Space* technique [Sel15, p. 98]. Collecting and transferring information is usually done in an analog fashion using paper mediums such as flip charts. Especially in large groups, however, it may benefit from the use of digital information processing. In general, the *Event Size* characteristic appears to be a good

means for companies to find methods with high potential for a profitable depiction in their software tools.

A similar hint lies in the *Special Resource Needs* of a method. Other than with the size of an event, this characteristic indicates, independent of the participant count, how worthwhile it is to support specific methods by the use of technical infrastructure. Suitable methods are those that, already by design, intend the usage of digital tools for the collection, transfer and processing of information. An example therefore is the *21st Century Town Meeting*. According to Holman's matrix, it requires polling/groupware systems for interacting with participants [HDC07, p. 24].

Finally, promising aspects are also incorporated in the *Duration* and the *Cycle* of a method. Long events as well as those with a *periodic* or *continuous* cycle can leverage the technological support of digital facilitation to relieve participants' memory. Take *RTSC* as an example. This method is prepared by different teams in a time period of up two three months [HDC07, p. 23]. During the event, information is elaborated and transferred between a variety of groups with changing participants. Also external parties such as suppliers, customers, or experts can contribute information [Sel15, p. 62-71]. To facilitate this diverse information flow and simplify the processing and capturing of information over the duration of the event, software systems can be of great help.

So far, we looked at the basics of facilitation, explored a scheme for classification and overview over available methods, and outlined some criteria that makes a specific method suitable for the depiction in a software system. The following section will introduce a specific facilitation method in more detail and show how our case study company teambits usually models it in their software.

### 1.1.4 A World Café in teambits:interactive

The World Café is a frequently applied method for teambits, so it represents a good opportunity to demonstrate digital facilitation in action. Before examining how teambits depicts it in their software, however, the following will give a brief introduction of the general idea and structure of the concept.

Developed by Juanita Brown and David Isaac in 1995, the World Café is designed around the analogy of café conversations. An assumption of the technique is that people of an organization already have the wisdom and creativity to face tough challenges within them. In a café gathering, so the authors, those people will move faster than usual from ordinary to meaningful conversations, reveal their wisdom and create collective understanding as well as forward movement [BIC05, p. 4].

[Sel15, p. 109] gives a precise idea about the procedure of a typical World Café. The core idea of the technique is to split up the whole system into small groups in which three questions are asked that motivate engaging conversations about a specific topic. While the content of the questions aligns with the discussed topic, their orientations should generally follow the subsequent sequence:

**Question one** is meant as a warm up question.

**Question two** is about linking the ideas that came up in question one.

**Question three** aims at the deepening and further examination of the topic.

After each question, participants are switching groups. Finally, the revealed knowledge is brought back into the plenum.

In their publications about the World Café, Brown and Isaac are focusing on a set of design principles that aims at the propagation of talent and knowledge [BIC05, p. 4f.]. The last of those principles asks to share the collective discoveries made during the conversations [BIC05, p. 138]. This is where technological support can be attached to the concept.

Traditionally, the ideas of a discussion were often drawn on paper by participants or dedicated drawers (see [Sel15, p. 108]) and then attached to a board for presenting them to the plenum. teambits, in contrast, provides touch-enabled devices to each group, for the discussed ideas to be documented.

At those devices, ideas can be entered in a variety of forms, including plain text, keywords, or scribbles. The exact set of options provided to participants is often decided in advance by the event organizer. While the processing capabilities of scribbles are relatively limited, plain text and keyword documentations can benefit a lot from automated and editorial processing.

The following example is a scenario that makes use of automated processing. Groups are asked to enter the most significant keywords about their discussion into a text field. The entered keywords are instantly showing up in a word cloud (see figure 1.1) that is either displayed directly on the input device or on an attached screen. The word cloud contains the entered keywords in a random order. Words that have been entered multiple times appear bigger.

It is a common scenario that multiple words with the same stem or meaning are entered. To avoid those being displayed as individual words, an algorithm is analyzing all contributions in real-time, identifies word variations and counts them towards the same stem.

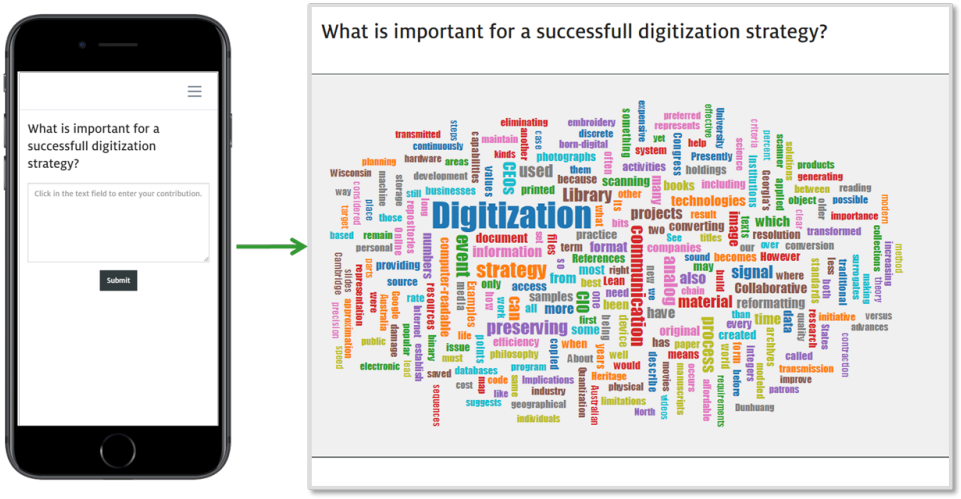



FIGURE 1.1: A Word Cloud in teambits:interactive [Tea]

How does that help to present group results in front of the plenum? Whenever a word is submitted by a group, it is directly transferred to a central server. During the presentation in front of the plenum, each group's individual word cloud can then easily be provisioned to stage projections, in any desired size and all without having to carry around and exchange large paper strips between individual presentations.

Additionally, there is another advantage arising from the application of word clouds. Since all groups are discussing the same topic and the same three questions, it is possible to merge the submitted keywords of some or all groups into a bigger set and observe how the resulting word cloud changes. This can, for instance, be a chance to identify differences and similarities in views within the organization.

Apart from word clouds, teambits offers a variety of additional formats for collecting, processing and displaying the ideas brought up within individual groups. For instance, information can also be entered in the form of longer text entries. Those ideas will then be analyzed and clustered into semantically related blocks, either by automatic algorithms or manually by an editorial team.

The preceding considerations outlined how two exemplary tools – word clouds and text clusters – can be used to support a World Café. Generally, however, the introduced tools are independent from the applied facilitation method. They can be assembled in various forms to support different facilitation techniques. On their website, teambits provides a complete list of supported tools [Tea]. The hints given in section 1.1.3 help to

chose facilitation techniques that benefit best from an application of software systems such as teambits:interactive.

Previous sections covered details and examples about the concept of facilitation and how it can be supported by software systems. The following section will zoom back out again to make some general observations about how the domain of digital facilitation is evolving, both in terms of its application areas as well as its technological foundation.

### 1.1.5 Evolution of Digital Facilitation Systems

In his article "Digital Facilitation" Dr. Peter Tandler points out the distinction between digital live and online facilitation. Live facilitation takes place synchronously and in real-time. The term online facilitation, on the other hand, is used in the context of asynchronous, internet-based information exchange as it is known from internet forums. Regardless of that distinction, digital live facilitation can take place in any of the following three environments [TKS13, p. 432f.]:

1. On-site, where it supports regular face-to-face meetings.
2. Online, where participants dial into virtual meeting rooms individually.
3. Multi-site, where several face-to-face groups or individuals connect to each other or a central venue via internet in order to hold a (partly) distributed meeting.

All of those environments benefit from digital facilitation by features such as automatic processing, aggregation and visualization of large amounts of data. Digital tools support open questions with lots of participants by algorithmic or editor-backed consolidation of data as well as real-time evaluations. Preciser handling of quantitative data and efficient depersonalization are only two examples from a far longer list of advantages introduced by digital facilitation [TKS13].

With the increasing popularity of cloud services, a shift is taking place also for digital facilitation tools. In the past, software was usually developed either for on-site or online events. While on-site events traditionally relied on local deployments of the event-relevant tools, software for online-meetings (see e.g. Cisco WebEx or Adobe Connect) has been internet-based for a long time [Web; Ado].

Today, the boundary between online and offline applications is becoming more fluid. The increasing interest in multi-site events makes it necessary to supply services via the internet. However, also classical on-site events are changing in that they are more often based on the bring your own device (BYOD)-principle, where participants use

their personal mobile devices to contribute to the event. Other than with rental equipment, those personal devices cannot easily be forced to connect to local networks. As a consequence, it becomes more appropriate to move services with direct participant interaction away from local deployments and to the internet. Besides better usability, this also increases flexibility and lowers infrastructure costs.

A lot of the companies that were founded after the year 2010, such as the Q&A and polling platform sli.do or the event app provider doubledutch, started their business with a focus on cloud-based self-service solutions [Sli; Dou]. Businesses that were founded before 2010, in contrast, such as the engagement/interaction platform SpotMe as well as our case study company teambits, commonly started with a product that was designed to run autonomously and isolated from other potentially inferring technologies and infrastructures [Spo; Tea].

Those older products don't leverage the potential of modern internet achievements such as the ability of synchronous collaboration over distributed locations or the ease-of-use introduced by self-service cloud applications. Moreover, due to the complexity of their products and the fact that competence in digital facilitation was not yet widely read by the time of their invention, most of those older companies, including the two aforementioned ones, started their businesses as full-service providers. Their offerings mainly focused on services such as the temporary deployment and operation of their products on venues, rather than the products themselves as a software solution.

As a consequence, those companies nowadays find themselves confronted with the challenge to transform their products to self-serviceable and cloud-ready solutions in order to meet the evolving demands of their customers.

## 1.2 Status-Quo & Current Knowledge

teambits launched the development of their central product teambits:interactive in the early 2000s as a result of the gathering of equally-minded researches and visionaries around the computer scientists Dr. Peter Tandler and Axel Guicking. When the software evolved from a research project to a marketable solution, there was, as stated above, not yet any widespread competence in digital facilitation, neither in the individual target industries nor with the event service providers. Therefore, customer demand focused on full-service offerings. The following sections explain teambits:interactive's status-quo of functionality and architecture and depict how the service-oriented approach that was followed until now significantly influenced its design as a product.

### 1.2.1 Basic Features

teambits:interactive is a client-server solution consisting of a monolithic server that is written in Java and based on the Spring framework, and various client applications (native, hybrid and web) written in Java as well as Javascript. The most important client types are depicted in figure 1.2 and 1.3. First of all, there is the *participant client* that allows attendees of an event to interact with the facilitators/editors (e.g. answer polls, ask questions, contribute to brainstormings, etc.) as well as with each other (e.g. chat or exchange business cards). Secondly, there is the *meeting cockpit*, which basically acts as a remote control of the participant clients. It serves as an interface for operators and editors to interact with attendees and process their contributions. The meeting cockpit, just like any other client application in the software, can be launched in parallel on as many machines as necessary. Multiple editors can therefore use it concurrently to consolidate contributions, cluster ideas, or sort and prioritize questions. At the same time, operators may launch their own instances of the meeting cockpit to release interactions to participants and send consolidated results to the stage. Alternatively, also a facilitator on stage can control the course of an event by the so-called *facilitator-* and *feedback clients*. Those clients allow to navigate through a predefined agenda of interactions as well as to select and display participant contributions from a list, optionally pre-filtered by the editorial team.

So far, the introduced features should already illustrate the software's immense flexibility. Further clients for even more advanced use cases shall be left uncovered at that point. With a full-service approach in mind, teambits clearly put their main focus on supporting a great range of customer scenarios rather than providing a self-explanatory operation experience. That wasn't a problem as long as it was guaranteed that only trained staff would deploy and operate the product. Nowadays, it is that very distinct flexibility and the complexity it implies that proves to be a limiting factor of the software's self-serviceability. Figure 1.3, for example, shows the feature-packed UI of the meeting cockpit that makes clear how beginners will likely have a hard time figuring out their way around with the software.

### 1.2.2 Security Aspects

Within this thesis attention shall be drawn to the topic of security and more specifically to the concepts of authentication and authorization. In teambits:interactive different mechanisms are used for the client-server communication. Apart from the download of static resources, clients usually interact with the server via a combination of a web-socket-backed messaging mechanism and (RESTful) HTTP endpoints. While

the former is used for the transmission of commands and data, the latter's purpose is primarily to provide prepared analysis results that are used, for instance, to display evaluation charts. The HTTP endpoints are currently not protected by authentication and are thus sensitive for data leakage through smart URI-guessing as well as client code analysis. The message communication, in contrast, can be protected by an optional authentication mechanism. Whenever, a client tries to connect to a sever that has authentication enabled, a challenge is sent back and a password prompt will be displayed on the client's UI.

Administrators can enable and disable password protection, define a global password for participants and one for operators, editors and facilitators. Depending on the client application a user tries to access (participant client, facilitator client, meeting cockpit, etc.) he needs to supply the conforming of the two passwords. No such thing as a username or group-identifier needs to be supplied in the first place. Thus, strictly speaking, this mechanism must rather be referred to as an access restriction than an actual authentication, as it cannot guarantee individuals' identities. This can be advantageous as it guarantees the participants' anonymity. From a security perspective it is furthermore widely accepted for short-lived offline environments, since those additionally restrict access to users that are physically present. In an online environment, on the other hand, network access is unrestricted and the system can be reached worldwide. In this case, not authenticating users widely withdraws control and knowledge (traceability) about who connects to the system and who performs which action. Indeed, the system can be configured so that the password request is followed by an optional identification procedure (users either enter their names or select them from a list), however, this process does not involve a challenge and so the validity of the claimed identities cannot be guaranteed.

The missing authentication implies another shortcoming. Without the ability to differentiate connected users or at least groups of users, no individual authorization can be established. As a consequence, the system grants access in an "all-or-nothing-fashion". As soon as one possesses the required password for a specific client type, he can perform any action the corresponding client is capable of. The shortcoming of that limitation becomes especially clear when looking at the meeting cockpit again. Previous sections (see 1.2.1) pointed out the different conceptual roles that leverage the meeting cockpit's various functionalities. Due to missing authorization, interference of different roles cannot technically be ruled out. In the past this didn't represent a significant risk, since the operator and editor roles were often exclusively performed by trained and trusted staff. And if a customer wanted to employ his own editors, they received a brief instruction to the relevant parts of the UI and were asked not to touch any controls that are not related to their current task. This worked fine, as long as those editors

could be supervised by trained staff. Yet, similar to the authentication problem described above, it becomes more complex in an online or multi-site event format, where a permanent supervision is not possible.

## 1.3 Knowledge Gap & Research Goals

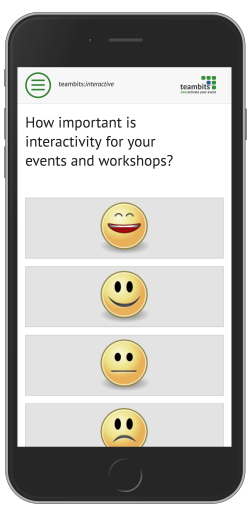
The preceding sections illustrate how several design decisions in the teambits software are aligned with the offline usage scenario and the full-service business model that the company started with. To cope with changing customer demand and the arising competitors, a technological transformation needs to be elaborated that turns teambits:interactive into an internet- and cloud-ready solution, while keeping the flexibility and reliability that is making up its competitive advantage. Security concepts in the area of authentication and authorization that are worked out and suggested within this thesis are perceived as a precondition to that transformation both generally as well as in the case of teambits.

While the general direction of transformation is clear, the details are not. Ideally, a worked out security concept aligns with two goals: Firstly, the improvement of workflows for staff and secondly, the market placement of the company and its services that sales and general management are targeting. At the beginning of this chapter, the objectives for this thesis were outlined. Those can be summarized in the following research goals. Analyze available methods in a literature review, gather requirements based on a team survey, work out an architecture based on those requirements, and finally evaluate and discuss the derived solution.

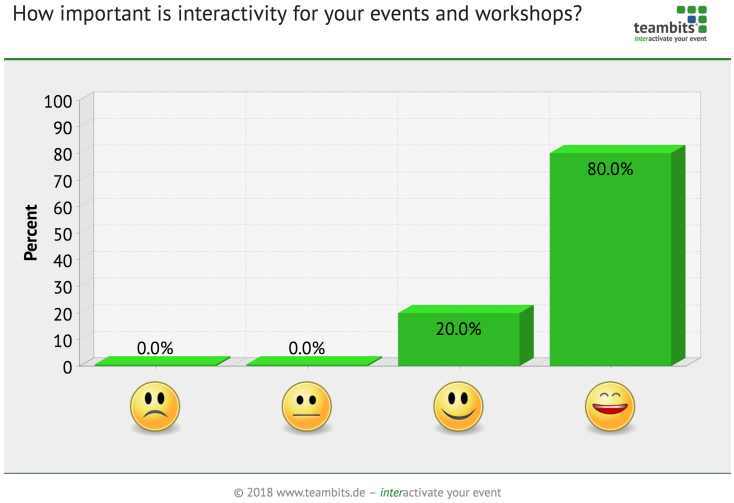
## 1.4 Outline

Chapter 2 starts with a literature review of authentication mechanisms for the HTTP and WebSocket protocol as well as different authorization concepts. This is followed by a description of the research methods applied within this thesis, see chapter 3. The latter includes, for instance, considerations about the survey structure used during interviews with staff members. In chapter 4, results from staff interviews are interpreted and respective requirements are defined. Chapter 5 then applies insights from the literature review and the derived requirements to work out a respective solution for authenticating users at the software's APIs. Chapter 6 does the same for authorization. In chapter 7, the derived solution is discussed and evaluated in terms of its

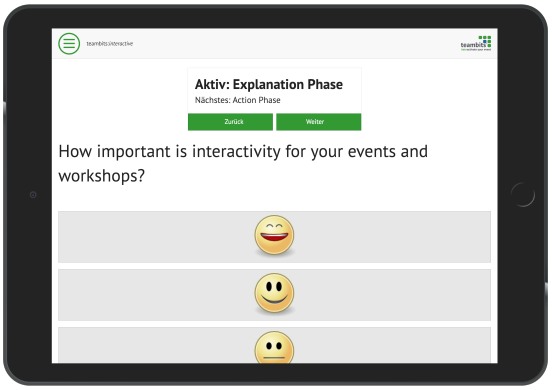
general validity and the possible level of abstraction from the characteristics of team-bits. Chapter 8 finally gives an outlook on the solution's applicability, its limitations as well as identified areas for future academic work.



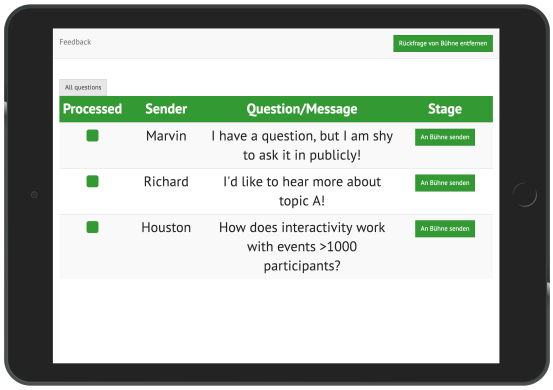

(A) Participant




(B) Stage




(C) Facilitator




(D) Feedback

FIGURE 1.2: Different client types of teambits:interactive during a basic smiley voting

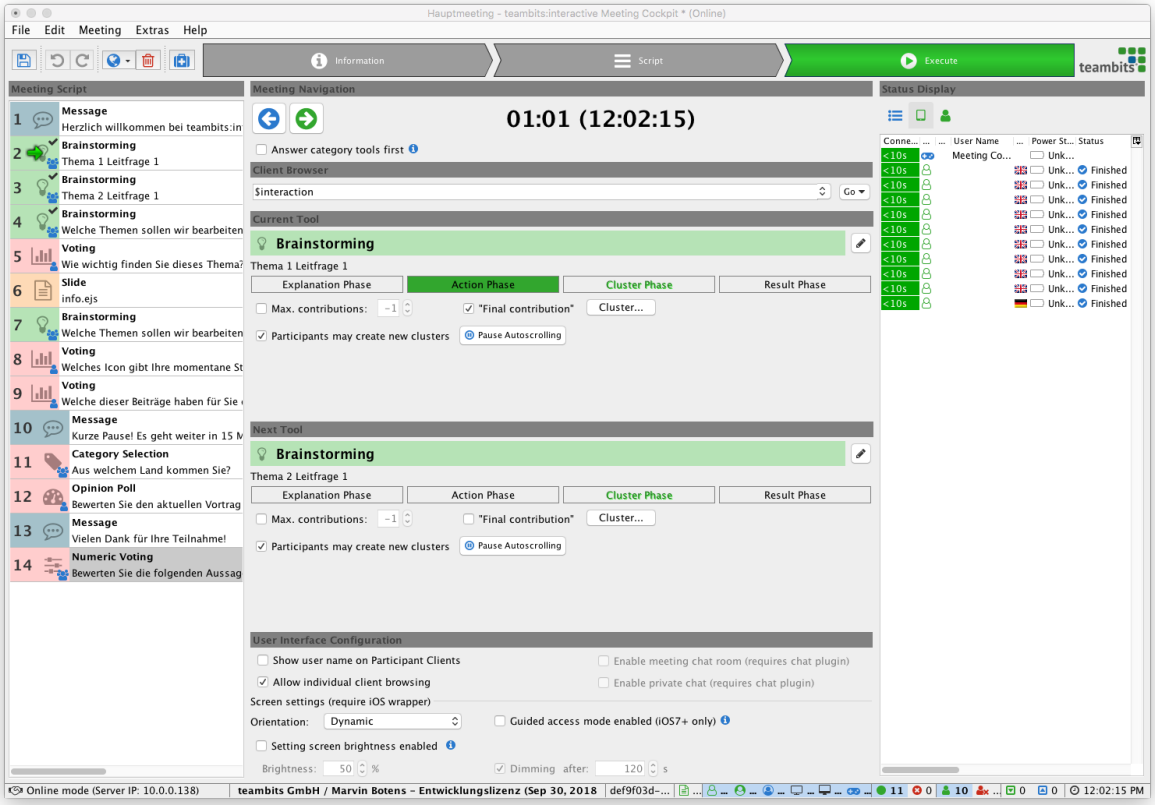



FIGURE 1.3: The Meeting Cockpit in teambits:interactive

# Chapter 2

# Literature Review

This thesis aims to work out a concept for securing an internet application that supports digital facilitation processes. From previously depicted information about the studied reference application's architecture and features, two challenges arise that are of primary interest for the application: First, user authentication needs to be established spanning multiple protocols used for client-server communication. Second, a suitable authorization system has to be designed to reflect the dynamic set of functional roles that are interacting with the application (see 4). The following sections will provide necessary background information on those questions, depict relevant research, and discover state-of-the-art approaches to related problems.

## 2.1 Authentication Concepts

In teambits:interactive clients communicate with the server via two distinct conceptual interfaces, a REST API and a messaging API. While the former is backed by classical HTTP endpoints, the latter uses WebSockets with a polling fallback for situations where WebSockets are blocked or not supported. This section analyzes and evaluates available options for securing those interfaces.

### 2.1.1 Authentication for HTTP

For classical web-/ HTTP communication there is a great variety of authentication mechanisms available, such as cookies, tokens, HTTP Basic/ Digest or TLS authentication. From the variety of those mechanisms, we are considering HTTP Basic, HTTP Digest, and token-based authentication as provided by the OAuth standard, as most relevant for securing an API. Alternative mechanisms such as TLS client certificates

and cookie-based authentication are considered unsuitable in that context. As discussed in [Coo], cookies are controlled by the user-agent and therefore provide limited flexibility to the application developer. Scripts in a browser environment cannot control which requests to send the cookies with and which not. Furthermore, cookies are bound to a domain, causing problems in case the software is reorganised into a microservice architecture in the future. TLS client authentication, as specified in [DR08, p.74], can be used domain-overlapping. However, it implies the necessity to manage trusted client certificates on the server, rendering itself useless for BYOD scenarios.

HTTP Basic and Digest authentication are specified in [Fra+99]. Both of the former can either be handled by the browser or by the application running in the browser. With basic authentication, username and password are sent over the network in cleartext with each request, making the protocol extremely sensitive for credential theft. Digest authentication uses cryptographic hashing to compensate that problem. Both client and server compute a hash based on a server-generated nonce value, username, password, HTTP method and request URI. The server grants access if both hashes match [Fra+99, p. 5ff.]. If the transfer of cleartext credentials is the most dominant problem with HTTP Basic, then the primary issue yielded by digest authentication might be the fact that the server needs to store cleartext passwords in order to calculate the hashes. Both protocols furthermore require credentials to be stored on the client for as long as the user session goes. With untrusted or compromised clients this again leads to increased risk for client credential leakage.

Token- and cookie-based mechanisms mitigate this problem by the concept of a server managed session. A client only needs to provide credentials to the server once when starting a session. It then obtains a token or cookie respectively to present with subsequent requests, indicating his association to the session. Besides the fact that client applications and requests are no longer aware about user credentials, this concept also introduces the possibility for a server to expire sessions or terminate them in case there is reason to believe they are compromised. As discussed above, cookies are not suitable in the context of an API. Consequently, tokens seem to offer the most appropriate way in our scenario.

OAuth 2.0 (hereafter referred to as OAuth2) is currently one of the most advanced access control frameworks utilizing tokens. It is specified by the Internet Engineering Task Force (IETF) in RFC 6749 [Har12]. Since its original idea is to provide third-party applications limited access to HTTP resources, the RFC claims authorization rather than authentication to be the framework's main purpose. Indeed, the standard specifies an architecture that allows logged in users to manage how other applications can access their data. It does neither specify how users should be authenticated, nor does it define a way for clients to verify the identity of authenticated users. As a consequence,

OpenID Connect was developed as an identity layer on top of OAuth2. It inter alia specifies an interface to be included in the OAuth2 authorization server (see figure 2.1) that allows clients to access its end users' profile information [Sak+14]. Despite its originally intended purpose, OAuth2 has become a widely utilized framework to achieve single sign-on (SSO) and has grown an alternative to the till then dominant security assertion markup language (SAML) standard [Den13].

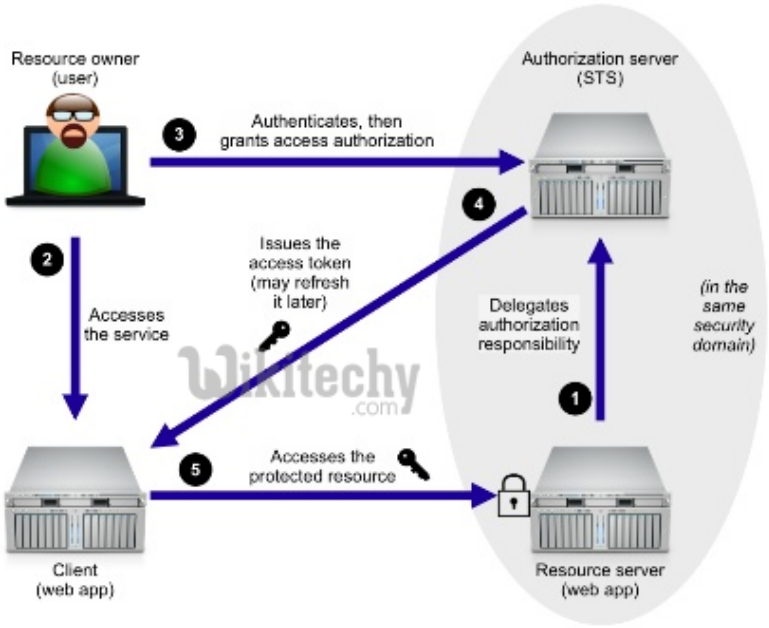



FIGURE 2.1: OAuth2 Architecture [Wik]

Figure 2.1 depicts the architecture as well as general roles and procedures to encounter in OAuth2 as introduced in [Har12, p. 6ff.]. The server hosting the protected application is referred to as the *resource server*. End-users, in turn, are *resource owners*. Users access data on the resource server via *client applications*. While communicating with the resource server's API, a client needs to present an access token. Access tokens are issued to a client by the *authorization server* after the resource owner has given it a so-called *authorization grant*. OAuth2 specifies different flows for the retrieval of authorization grants and access tokens, from which the following two are recommended for most end-user scenarios: Authorization code grant and implicit grant. Both are utilizing HTTP-redirects to forward the user between the authorization server and the client application. The difference between those flows lies in the way a client application obtains the access token from the authorization server. We'll look into this in more detail after a general analysis of the OAuth2 authorization flow.

The following outlines, according to [Har12, p. 7f.], how both flows can be used to establish authentication in the client application. Whenever a user wants to sign in to a new application, he is redirected to a web interface on the authorization server. If no session is currently active, authentication will be requested, otherwise the server immediately presents a confirmation dialog. At that point, the user can allow or disallow access and even specify so-called OAuth2 scopes, that restrict the client's permissions in connection with his account. After access is granted, the user is redirected back to the client application. During that redirect, the authorization server attaches a fragment parameter to the URI that makes a secret available to the client application. This is where it gets specific to the chosen flow. In case of the implicit grant, the access token is contained in the redirect-URL directly. With the authorization code grant, in contrast, an intermediate authorization code is instead attached that the client then has to supply to a HTTP endpoint in order to request the eventual access token.

Although it is not always clearly recognizable, the user perspective of this process should be fairly well-known from a wide range of internet applications that are offering authentication via social media accounts. The fact that with both flows client applications remain unaware of user credentials is what makes the OAuth2 framework very suitable to cross-platform authentication and third-party API access.

OAuth2 is flexible in that a single authorization server can be used to secure any number of resource servers. It is also secure in that it manages user data exclusively on the trusted authorization server giving the user full control over the way in that other applications may access his data on the resource server [Har12, p. 6]. As stated above, the standard does however neither specify how users should be authenticated at the authorization server, nor does it state how client applications can verify the identities of their users. This is something that developers need to take care of when relying on OAuth2 for cross-platform authentication, either via custom implementations or via other frameworks such as the previously introduced OpenID Connect [Den13].

### 2.1.2 Authentication for WebSockets

In contrast to HTTP, the WebSocket protocol, as specified in RFC6455, does not prescribe any particular way for authenticating clients [FM11, p. 53]. Instead, developers can either rely on generic HTTP mechanisms or develop custom protocols on application layer. Also a combination of both is technically possible. In the scientific community the problem seems to earn very little attention. [Erk12] essentially restates the RFC in terms of authentication and also [Kul13] only brings in the idea of a challenge

response theme without elaborating further details. Instead, blog posts and proprietary documentations such as [Are16] and [Sla] give a more extensive impression about concepts and best practices that are used in practice.

WebSocket connections are initiated by the client via regular HTTP requests containing an Upgrade header as well as some protocol specific header fields. As a consequence, it is possible to reuse the aforementioned generic HTTP authentication mechanisms. This is especially tempting since most web application frameworks (WAFs) offer out-of-the-box support for those mechanisms. However, using them also involves some limitations. The JavaScript API, which is maintained in [Wha] and implemented in all major browsers, does not allow to customize WebSocket headers. This limits the choice of authentication mechanisms to those that are implicitly handled by the user-agent (i.e. HTTP Basic and cookies). In [Her] it is furthermore pointed out that shared authentication headers become difficult or impossible to use, when the WebSocket endpoint is located on a server that is separated from regular HTTP endpoints.

[Are16] suggests a way around this limitation. Authentication data can be added as query parameters to the WebSocket URL. The API in [Dat] is an example for this approach. In this particular case, the required authentication data consists of a username and an API key. Other implementations, however, might as well use tokens for authentication that the client must retrieve from a separate authorization service prior to opening WebSockets. In any case, as the author of [Are16] also points out, a new issue that arises from this method. Authentication data included in the URL might persist in proxy logs and may hence be accessible by administrators. Favoring tokens over username and password can avoid credential leakage; depending on token lifetime, however, the risk of temporary impersonation remains preserved.

The messaging platform Slack mitigates this issue in their WebSocket-based Real Time Messaging API (RTM API) by issuing single-use URLs. In order to connect to the API, clients have to make an authenticated call to a separate HTTP endpoint. The latter issues them a WebSocket URL that automatically invalidates after first use, rendering itself useless for skimmers. If not used to connect within 30 seconds after issuance, the URL automatically expires, additionally minimizing the chance for abuse in case of abandoned connection attempts [Sla].

Another approach to solve the WebSocket authentication problem is using a ticket-pattern. The general concept is described in [Her] as follows. Similar to the Slack example, clients have to make an authenticated request to a regular HTTP endpoint prior to opening a WebSocket. Instead of returning an individual URL however, the endpoint following this approach issues a ticket containing information like the user ID, the IP of the requesting client, a timestamp and any other information that might be required in the specific context. The issued ticket is stored in a database or cache and

returned to the client. Now, instead of using personalized URLs, all clients may connect via the same generic endpoint URL, but provide the previously acquired ticket as part of an initial handshake. The server then has to compare the ticket with the one stored in database or cache, check that the source IPs match, rule out expiration or previous usages, and make any other necessary verification. If all checks succeed, the connection is verified and the server may continue with post-authentication procedures [Her].

## 2.2 Authorization Concepts

Chapter 1 derived the need for a flexible authorization mechanism. Although the detailed requirements for access control will be derived in section 4, the following sections perform a literature survey to determine available concepts, identify the advantages and disadvantages they come with and find out which types of application they are applicable to.

### 2.2.1 Discretionary and Mandatory Access Control

Intensified interest in structured and schematized access control came up in the late 1960's. From a variety of access control models developed in the 60's and 70's, initially two concepts gained most prevalence: *mandatory access control (MAC)* and *discretionary access control (DAC)* [JKS12, p. 41]. While the notion of those concepts are rather generic, causing implementations to vary in details, a rough distinction can be as follows.

Generally speaking, MAC derives access authorization from security level classifications for subjects and objects, respectively referred to as clearance and sensitivity. While exact classifications are application-specific, examples known from governmental and military applications include terms such as confidential, secret and top secret. MAC comes with lower flexibility, but offers improved security, making it especially suitable for this type of applications. Additionally, information flow in MAC is restricted by the so-called read-down and write-up principles [SS94, p. 44f.] [SV01, p. 148ff.].

DAC, in contrast, makes access decisions based on user identities and granted authorizations. This concept shines through great flexibility and was hence frequently applied in commercial and industrial environments. Other than MAC, it does not limit

information flow. Instead, implementations often allow owners of information to discretionary grant authorizations to other users, hence the naming [SS94, p. 44] [SV01, p. 139ff.]. Substantiated models of DAC are introduced in [SV01, p. 139ff.] and include the so-called access matrix, authorization tables, capability lists and access control lists (ACLs).

Besides distinct advantages and disadvantages, both MAC and DAC are not suitable for all types of applications. According to [SS94, p. 46+48], MAC too rigidly reflects the structure of the U.S. government. The autonomous nature of DAC, on the other hand, is rather suited for academic research than for commercial enterprises, as its administrative control of information assets is not perceived effective enough. Indeed, DAC models often do not provide means to enforce constraints on the arrangement of access rights, leaving administrators with very little control over the distribution of information.

### 2.2.2 Role-Based Access Control

As a consequence to these shortcomings, research for role-based access control (RBAC) models has picked up speed. The general idea of this pattern is described in [SV01, p. 180ff.]. In RBAC, one uses roles in order to decouple the assignment of permissions to users. Roles represent working activities and a set of associated actions and responsibilities. Instead of assigning permissions to individual users directly, permissions are assigned to roles and users are authorized to activate roles. The authors of [SV01, p. 180] describe this decoupling as key to simplifying security management, especially when user responsibilities change. Further advantages mentioned in their article are the support of hierarchical roles as well as the principles of least privilege and separation of duties (SoD). Hierarchical roles are helpful to model generalization-specialization relationships. Least privilege and SoD are realized by allowing administrators to define constraints for role assignment and role activation. This addresses the aforementioned issue of missing control over information flow that is typically encountered in DAC systems.

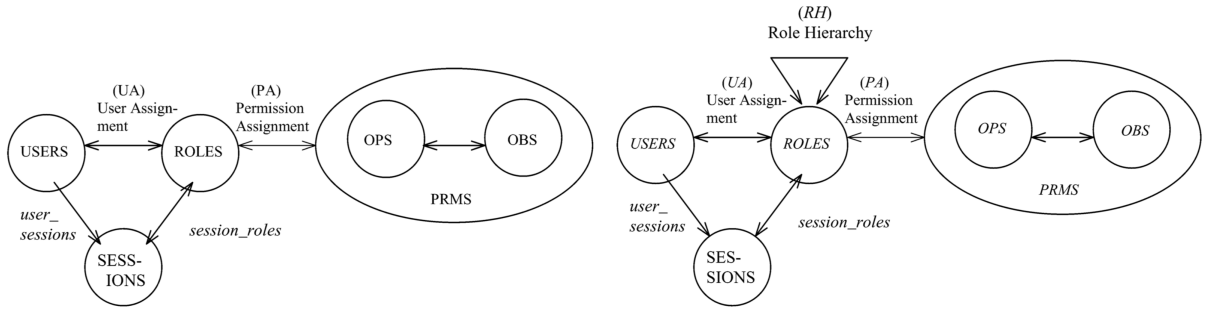
By the beginning of the 1990's RBAC got a lot of attention from both the scientific community as well as commercial software development. A lot of research such as [FK92] has been done, further refining the RBAC model and analyzing its superiority to DAC. Moreover, a lot of variants and extensions to the model were derived, introducing additional features such as parameterized roles and permissions [GO04, p. 259f.] as well as object-sensitive roles [Fis+09, p. 173ff.].

In 2001, the National Institute of Standards and Technology (NIST) made an attempt to standardize RBAC including only those features that are common in the major circulating models and implementations [Fer+01, p.224f.]. The resulting NIST standard includes four components, partly building up on each other:

- Core RBAC
- Hierarchical RBAC
- Static separation of duties (SSD) RBAC
- Dynamic separation of duties (DSD) RBAC

Figure 2.2 shows models for the Core and Hierarchical RBAC components. According to [Fer+01, p. 228ff.], those components have the following characteristics:

NIST's Core RBAC represents the basis for the three remaining components. It includes user-role assignment (UA) and permission-role assignment (PA) as well as the concept of role activation within user sessions. Hierarchical RBAC adds the concept of seniority by allowing reflexive relations between roles. It requires senior roles to acquire the permissions of their juniors and junior roles to acquire the users of their seniors.

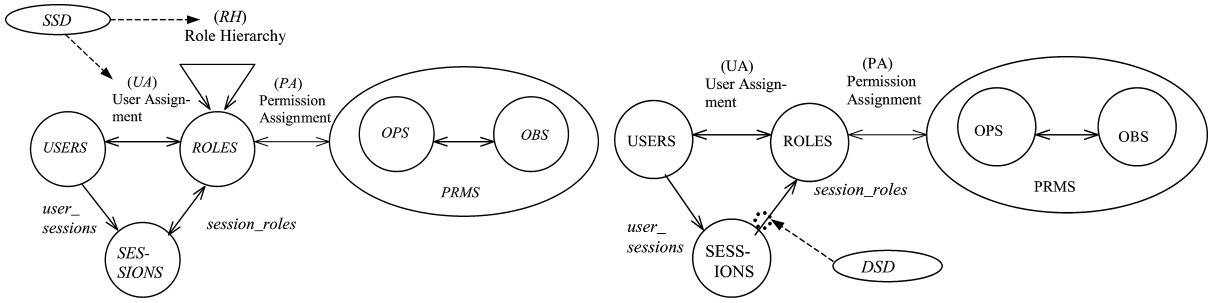



(A) Core RBAC (B) Hierarchical RBAC

FIGURE 2.2: NIST models for Core & Hierarchical RBAC [Fer+01, p. 232, 235]

Models for the SSD and DSD RBAC components are given in figure 2.3. In [Fer+01, p. 230f.], those components are described as follows. SSD RBAC adds exclusivity relations to user-role assignment (UA), allowing administrators to define constraints that prevent simultaneous assignment of conflicting roles to the same user. Eventually, DSD RBAC, being the last component in the standard, adds the same exclusivity to the activation of roles within individual user sessions. This can, for instance, be used to enforce the principle of least privilege by limiting simultaneous role activation and

requiring users to login with low-privileged roles whenever possible [Fer+01, p. 226, 231f.].




(A) Hierarchical RBAC with SSD (B) RBAC with DSD

FIGURE 2.3: NIST models for SSD and DSD RBAC [Fer+01, p. 239, 241]

Apart from their corresponding functional specifications, those four components depict the essential definitions given in the NIST standard. NIST's goal was to provide a universal set of elements, relations, functions and a consistent language, all representing the greatest common factor between major existing RBAC models and implementations [Fer+01, p. 226f., 232].

As a consequence, a lot of RBAC variants that were circulating at the time the standard was proposed, are intentionally left uncovered by NIST. The standard especially includes very little information about the different options to define access authorizations to individual objects. As illustrated in figures 2.2 and 2.3, in NIST RBAC the permissions (PRMS) available for PA arise from the operations (OPS) and objects (OBS), available in the system for which RBAC is deployed [Fer+01, p. 242].

In other words, roles include permissions to execute operations on individual objects or sets of objects. This apparently requires significant administrative work in systems where individual users need access to large groups of similar objects. [SS94, p.47] therefore introduces the concept of object-classes to group objects by their type (e.g. letters or manuals) or their application area (e.g. commercial letters or advertising letters). This eases up the configuration in scenarios where roles need to cover access to a whole category of information assets. The authors mention the example of a secretary position that needs access to specific types of letters. With object-classes a corresponding role can be defined to include permissions for those very types, including both current and future letters. This is in contrast to per-object permissions, in which case access needs to be granted for all current and future letters individually.

Object-classes represent an effective means against role explosion (see [EK10, sec. 2]) in systems where only a few users need access to all objects of specific types. In systems where many users need individual access to specific objects of the same type, however, the concept is of no help. This is for instance the case within applications that allow users to access their own data records while denying access to all other users' records. Using per-object permissions requires individual roles for all users to reflect this policy and thus preserves the role explosion problem.

As an extension to NIST RBAC, [GO04, p. 259f.] introduces parameterized permissions and roles as a concept to cope with this limitation. Although the authors use slightly different terms and abbreviations, we will at this point stick to the previously introduced NIST terms, for consistency. If $(x, m)$ defines a regular permission with $x$ being a specific single object and $m$ an operation (access mode), then the authors define parameterized permissions as $(x|x\{a_1, a_2, ..., a_n\}, m)$, where the parameters $a_1, a_2, ..., a_n$ isolate a set of objects to grant the permission on. Values for those parameters are not part of the permission assignment, but arise from user attributes that are supplied when a user activates a role within a session. If ordinary roles are referred to as (name, permission set), then parameterized roles (i.e. roles containing parameterized permissions) can be referred to as (name, permission set, parameter set), whereby parameter set represents the union of all parameters used in permission set [GO04, p. 259]. The solution provided by this approach allows to define fine-grained access control policies, such as granting users exclusive access to their individual account data, while avoiding the role explosion problem.

### 2.2.3 Attribute-Based Access Control

The previous section outlined how the features incorporated in NIST RBAC are limited. To reflect more advanced policies, individual extensions to the defined components are required. Parameterized- and object-sensitive roles, see [GO04] and [Fis+09], are only two examples from the variety of extensions that have been developed for the RBAC model. The limitations of the core model and its dependency on extensions are perceived as a shortcoming of RBAC. It has lead to rising demand for a more general model of access control.

ABAC innately aims to provide the flexibility required to reflect the greatest possible variety of policies, including, but not limited to, those currently depictable through MAC, DAC, and RBAC [JKS12, p. 43]. In ABAC, access control policies are modeled by constraining values and relationships of user-, object- and environmental attributes. With attributes such as identities and ACLs, for instance, one can model DAC policies. Relying on clearance and sensitivity instead, leads to a MAC policy. Consulting role

attributes finally helps if an RBAC policy is required. ABAC also allows to model arbitrary mixtures of those classical schemes. More over, since it includes environmental attributes, policies may even limit access based on properties of individual requests, such as user location, time of day, or type of authentication [JKS12, p. 42].

ABAC policies are defined using languages such as extensible access control markup language (XACML) or SAML [JKS12, p. 43] [OAS05a; OAS05b]. [YT05, sec. 3.3] explains a common authorization architecture for ABAC, which is also illustrated in figure 2.4.

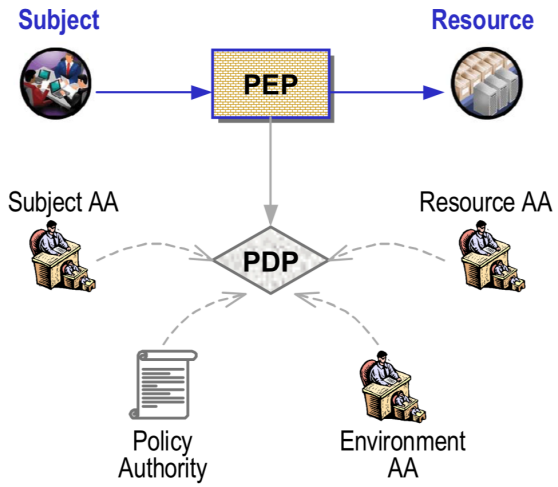



FIGURE 2.4: ABAC Authorization Architecture [YT05, sec. 3.3]

It is based on the following logical actors:

- Attribute authority (AA)
- Policy authority
- Policy enforcement point (PEP)
- Policy decision point (PDP)

All access requests are routed through the PEP, which delegates them to the PDP. The latter reads the configured policies from the policy authority. To calculate an access decision, it then obtains the required attributes from the respective AAs, including those of the requesting subject, the requested object as well as any necessary environment aspect [YT05, sec. 3.3].

One one hand, the ABAC model introduces great flexibility over the classical access control schemes. On the other hand, however, as [JKS12, p. 42] points out, it also

comes with new challenges such as attribute engineering and the complexities of policy expression and policy comprehension.

The research conducted in [YT05, sec. 1] reasons about fields and environments where the benefits of ABAC will overcome those complexities. The authors are considering web services in modern service-oriented architecture (SOA) environments as a potential field of application for this access control model. During the transformation from isolated systems to interoperable platforms, both in commercial as well as governmental environments, perimeter-based security measures such as Demilitarized Zones (DMZs), firewalls, and intrusion detection are reaching their technical limitations.

With classical applications, security architects had to deal with an exclusive set of known users. Potential offenders could simply be locked out by a proper application of the aforementioned measures. In the future, however, service discovery between consumers and providers will become more dynamic and less predictable. It is a tendency to open up systems to a wider range of people and other systems. At the same time, those systems need to maintain the confidentiality of the information they are managing [YT05, sec. 1].

For those requirements, traditional access control systems are perceived as being too static and coarse-grained. They are lacking the ability to take operational context, such as an at-hand transaction, into consideration when deriving their access decisions [YT05, sec. 1]. That is where ABAC's flexibility to construct multi-dimensional attribute policies can take effect.

The authors of the mentioned research are bringing up "deep" supply chain integration as an example where commercial systems can benefit from ABAC's advanced capabilities. For governmental environments, an equivalent example can be the process of exchanging information between different agencies, for instance in the event of a counter-terrorism operation [YT05, sec. 1].

# Chapter 3

# Methods

## 3.1 Staff Interviews & Requirement Engineering

In chapter 1, the motivation for an evolution of teambits' security infrastructure was derived. There is uncertainty, however, about what the exact requirements for this evolution are. As a consequence, the writer of this thesis decided to perform a series of interviews with staff members in order to collect, merge and prioritize existing needs that are valid for both the teambits software as well as comparable products. At teambits, there are essentially three groups of stakeholders that have a legitimate interest in the software's progression: Sales, services and development. Representatives of those departments shall be interviewed in order to gain a broad insight in what is perceived beneficial for the software's value creation potential. This section discusses the structure of the interviews. Appendix A provides a list of the questions asked together with the received interviewee responses. A discussion of the outcomes is conducted in section 4.

All interviews start with a situation analysis. The underlying idea to that is the gathering of insights about the so-called *mental models* that employees maintain about a software product. The concept of mental models has been proposed by a number of scientists. Inter alia by the philosopher and psychologist Kenneth J. W. Craik in 1943. Craik talks about small-scale models, individuals build of their surrounding reality, roughly consisting of a state, a set of possible actions and the effects those actions will have to the state. According to the philosopher, mental models enable individuals to perform more competent decisions based on the experiences they made in the past [Cra67, p.61].

The concept has made its way into various disciplines. It is frequently used to better understand influences on the performance of individuals and teams. In human-computer interaction (HCI), for instance, mental models are utilized to align a system's

UI with user expectations [Loe+13]. Researchers in the area of agile software development are furthermore investigating the concept of so-called shared mental models as a significant success factor for team performance [YP14].

In the context of our interviews, the state analysis is expected to reveal where team members and departments have consensus in their perception of the product and where angles are rather diverse. After describing different aspects of the software, interviewees are asked a series of evaluative questions. The answers to those questions shall give an idea about which aspects of the current security mechanisms are beneficial and which are rather detrimental. Following the evaluation, candidates are expected to identify concrete needs. For this purpose, a series of generic questions will be asked. Eventually, all identified needs are merged and presented in front of the product owners and stakeholders. The latter are then supposed to work out a set of concrete and ranked requirements to determine the primary focus for the research conducted within this thesis.

## 3.2 Architectural Design

Based on the elaborated requirements and the insights gained from the literature review in chapter 2, chapters 5 and 6 of this thesis will work out the architectural design for a comprehensive authentication and authorization system. The existing application as well as the requested features will be split up into smaller logical tasks and components that can be covered individually. Authentication and authorization shall be discussed separately. Both of those concepts can in turn be further split up.

For authentication, we will start by designing a solution for the processing of credentials. Subsequently, a central component shall be established that authenticates users by applying the formerly designed credential processing. Once the sign-in procedure is set up, focus shall be on integrating authentication in the REST- and messaging API. Here, we will utilize the research done about authentication in the HTTP and WebSocket protocols to design or apply suitable solutions.

Concerning the authorization, initial endeavor will be on the elaboration of an access control model. The insights gained from the analysis of available access control mechanisms will be used to pick those aspects that best fit the derived requirements and assemble a solution that is capable of reflecting the demanded policies. Hereafter, a technical architecture shall be developed that reflects the designed solution in team-bits:interactive and uses it to secure access to both of the system's APIs.

## 3.3 Discussion & Transferability

In order to evaluate the general validity of the research outcomes produced within this thesis, chapter 7 analyzes the designed solution for its transferability to other digital facilitation systems. After a discussion of its architectural characteristics and possible alternatives, we will examine the designed solution for aspects that are specific to teambits:interactive, such that are unrelated to teambits, but tied to a specific architecture or framework, and such that are completely detached from technical implementation details and thus fully transferable to other applications. To facilitate this evaluation, a comparison with corresponding features of competitor software is conducted. Inter alia, it aims to discover which of the competitors access control concepts could be realized by the designed solution and which require a divergent approach.

# Chapter 4

# Requirements

## 4.1 Situation Analysis

In the first part of the interviews, staff members were asked to describe how authentication and authorization works in the software. The answers to that question were surprisingly diverse. Even when abstracting from the answers' varying technical emphasis - which is of course owed to the employees' individual occupations - it is still worth noting that a general uncertainty exists about what security features the software offers, how they generally work and how they are connected. Interviewees commonly describe a variety of mechanisms the software supports, such as different password protections, auth codes, or user data retrievals from external sources. However, they don't seem to share a common model of how those mechanisms work together to build the system's overall security infrastructure.

This missing overview is likely based on a mixture of two reasons: First, there are discrepancies in what team members associate with the concepts of authentication and authorization. Second, the software's variety of facilities does de facto not provide a consistent big picture of its security infrastructure. In fact, interviewees report that as the software grew over time, various unrelated enhancements have been made to meet individual customer requirements. Today, it appears to be the missing consistency of those enhancements that causes a lot of the encountered ambiguity and complexity. This is critical, since it rises the chance for configuration mistakes and undiscovered programming errors. Apart from the following in-depth evaluation, those findings stress that it should be an overall goal to define a core authentication and authorization mechanism in that all current and future enhancements can integrate.

## 4.2 Situation Evaluation

When being asked for pros and cons of the previously described security features, interviewees often mentioned good usability as an advantage for participants. As described in section 1.2.1, meetings can be secured with a global password for participants. This password is easily communicable in respect of the particular event format. For instance, it can be displayed on stage projections, announced orally, or distributed via group calendar invitations. Moreover, it effectively limits the access to a defined audience by means of physical attendance or recipient lists. Besides easy password distribution, it is likewise appreciated that password protection can be disabled, making authentication optional. This is especially beneficial for development/ test environments as well as for anonymous offline events, where access to the system is limited on the network level and a double password burden on the user needs to be avoided.

Previous sections describe how diverse the applied security measures are and how that fact creates a risky complexity. In spite of this, team members also mention it as an advantage that the system is heavily hackable in terms of creating event-specific one-time mechanisms. In the past various of such highly individual mechanisms have been implemented to satisfy customer demands. Amongst them are, for instance, a user check-in process via barcode scanners as well as authorization decision makers based on arbitrary properties in the participant database. On the other hand, interviewees experience it as obstructive to re-authenticate using different sets of credentials while switching between admin panel, meeting cockpit, etc.

Being asked for an estimation of the systems overall security state, opinions are diverse throughout the departments. Sales and services mainly take the position of their customers, developers of course highlight technical considerations. According to the former, customer demands are satisfied with the presently available security measures. One rep, for instance, proposes the hypothesis of the system providing a good-enough security as long as customers are happy and nothing happens. The latter, in contrast, highlight a series of significant shortcomings that might only not have been an issue so far, because of the software's relatively low prevalence in the internet community. Mentioned shortcomings include, but are not limited to, the lack of individual user authentication as well as unsecured HTTP endpoints bearing the potential to reveal private customer information. According to some candidates authentication of individuals via personalized credentials is in fact requested by customers. Interviewees also agree that the current systems' authentication mechanism requires too much time and technical knowledge to support that requirement. Eventually, it comes to speech that the software lacks consistent logging of user actions. This might become vital in cases where misconduct or other incidents have to be investigated.

## 4.3 Identified Needs

In connection with the evaluation, interviewees are asked to describe what security features and improvements they could make use of. Project leaders, for instance, consider it a significant benefit if they could setup authentication autonomously via an admin interface. This should include the configuration of user accounts - either one-by-one or batched via a defined file format - and must be achievable without the need to modify any files on a host OS. Furthermore, there is a rising demand for the integration with central authentication providers, such as classical SSO services that are mostly encountered in enterprise environments, or social media platforms that are gaining popularity in the internet community. Another mentioned nice-to-have is passwordless authentication. It helps with both improving user experience (UX) and increasing security and thereby fulfills the common trend of rising expectations for both.

At the moment teambits does not support authorization of user actions. Section 1.2.2 explains how the system grants access to its backend UIs in an "all-or-nothing-fashion" and that staff members have to be trusted to only issue commands that are in conjunction with their current task. If authentication can be established, however, it is desirable to also define authoritative roles that restrict access to the features necessary in that conjunction. Interviewees came up with common roles, such as an administrator with access to all meetings on the server, editors, operators and facilitators with access to individual meetings, and participants. However, also some more diverse roles have been mentioned, that arise from an event's individual requirements. Take, for example, VIPs such as a company's general management, that are granted access to exclusive information and tasks. Such individual requirements are currently reflected in the system via hackable user properties. This is error-prone and often requires technical deep dive. In the course of designing an authorization system, it would be highly beneficial, if roles could be setup and assigned responsibilities dynamically.

In the same context, however, with lower prioritization, candidates have mentioned hierarchical mapping of roles as a nice-to-have feature for an authorization system. This roughly refers to the following idea. When a user is assigned a role, e.g. operator, it should be possible to assign him this role in the context of a meeting, a group of meetings, or all meetings on the particular system. This idea should be observed especially with regard to multi-tenancy or reseller support, which both might become of interest in the future. Those features, however, require additional considerations, such as the isolation of client teams, and are thus not given primary focus for the moment. Nonetheless, one should keep those ideas in mind when evaluating options for an authorization architecture.

So far, this section covered demands for authentication and authorization. Eventually, the interviews identified another need in terms of both concepts. First of all, in some cases personalized authentication of participants is not desirable. This is the case, for instance, when handling anonymous votings. In those cases it may be requested that participants do authorize themselves with a secret; the secret, however, shall not carry any information about their individual identities. Second and irrespective of how authentication is realized, any action requested by a user, especially including the authorization decision - i.e. if and why it was granted - should be logged to provide traceability in potential investigations.

## 4.4 Ranked Requirements

Subsequent to the interviews, all needs and demands identified by the team were merged and presented to the product's stakeholders. The latter worked out the following lists of requirements for the concepts of authentication and authorization, each with descending priorities. Ideas that were mentioned during the interviews (see the previous section), but considered out of scope by stakeholders, do not show up as concrete requirements. Nevertheless, those needs will be taken into account during the subsequent evaluation of architectures.

#### Authentication

**FR10 - Omnipresent authentication.** All client-server communication shall be secured in that it requires authentication to access any non-public information.

**FR11 - Overarching authentication.** All endpoints shall be secured by an integrated authentication system. Users shall authenticate at all of the software's endpoints\UIs using the same credentials. This includes the endpoints that are currently protected by separated individual authentication mechanisms, such as the admin interface and the messaging API for clients, but also those endpoints that are not yet protected at all, such as the HTTP endpoints for result retrieval and client material download.

**FR12 - Optional authentication.** In secured offline environments as well as for public online events, it must be possible to disable authentication.

**FR13 - Group authentication.** For scenarios where identities are not of interest, the system shall be capable of authenticating anonymous participants as affiliated to a group.

### Authorization

**FR20 - Common conceptual roles.** Privileges of users in the system shall be manageable by means of conceptual roles. Common roles include, but are not limited to, administrators, operators, facilitators and participants. Authorization decisions, i.e. whether a user is granted access to information or allowed to perform specific actions, shall be based on those roles.

**FR21 - Dynamic roles.** Beyond the aforementioned common roles, the system shall also support dynamic ones to reflect event-specific positions, such as editors or analysts assigned by the customer. The responsibilities and privileges of individual roles shall be configurable by administrators.

**FR22 - Hierarchy mapping.** It should be possible to assign roles to users either in the context of a meeting or globally, i.e. for all meetings available in the respective system. Beyond that, this mapping capability shall be expandable to more diverse hierarchy levels that might be added in the future.

# Chapter 5

# Authentication

## 5.1 Authentication Processing

Before working out new concepts to address the derived requirements, it seems promising to gather some inspiration from the architecture of our case study of the teambits software. Section 1.2.2 describes how the software authorizes users via collective passwords before allowing them to identify theirselves by typing their name or selecting it from a dropdown list. Identification usually happens without further prove and hence relies on the participants' honesty. Participant data, i.e. the names available in the dropdown list, is retrieved from an externalized micro service. That micro service, henceforth also referred to as *user service*, provides extensible user retrieval strategies including a CSV import as well as an HTTP interface for machine to machine (M2M) interaction. It can be hosted independent from the main application and thus constitutes a flexible interface to customer specific systems.

The user service can be understood as a data source to allocate differentiable entities for the participants that we want to distinguish in the system. Now, in order to actually authenticate participants – as opposed to the little trustworthy identification process described above – and to benefit from proven security mechanisms, this data source must be integrated into Spring Security. As it retrieves data from internal systems of the customer, the user service will exclusively provision participant users. It cannot at the same time manage users that are external to the company, such as commissioned facilitators or staff of the hired full-service provider. Consequently, a second user data source, internal to the application, is required, solely for managing the different kinds of staff users that are interacting with the system as introduced in section 1.2.1.

In order to design a solution that reflects the aforementioned goals, some architectural concepts of Spring Security as well as the Spring framework in general should be reviewed.

A central concept in Spring is known as separation of concerns (SoC). SoC is a design principle in software engineering that strives to decompose software behavior into logical units responsible for distinct concerns [Lap07, p. 85ff]. It is based on the idea that a given problem can be separated into sub-problems (concerns). Handling those concerns individually decreases complexity and helps achieving engineering quality factors such as robustness, adaptability and reusability [ATB01, p. 1]. Different types of concerns can be distinguished as follows.

On the one hand, requirements such as allowing a user to login to an application, represent typical decomposable high-level concerns. Those can be split into sub-concerns representing distinct aspects of representation logic (UI), business logic, or data access logic, which leads to the commonly known idea of a layered architecture.

On the other hand, concerns such as logging, security and transaction management are non-separable and extend over multiple if not all structural pieces of the software. Those concerns are referred to as crosscutting concerns [ATB01, p. 2]. There is a difference in how to cover those types of concerns in an application. While decomposable concerns can be reflected by standard object-oriented programming (OOP) mechanisms such as inheritance and aggregation, crosscutting concerns require advanced concepts such as aspect-oriented programming (AOP) [ATB01, p. 2].

Figure 5.1 outlines Spring Security's architecture for processing authentication requests. It arises from the illustrations made in [Sprd] as well as the framework's reference and API documentation [Spre; Sprc]. The depicted architecture is a thorough embodiment of SoC, in that it decomposes the concern of authentication request processing into three sub-concerns:

1. Accepting authentication requests in order to delegate them to adequate authentication strategies,
2. retrieving user data form persistence layer, and
3. validating the provided credentials against the retrieved data.

In fact, some of those concerns are even further decomposed. For instance, password hashing is treated as a sub-concern of credential validation. However, in order to design a solution that satisfies the goals derived in chapter 4, the aforementioned set of concerns should be the right level of abstraction to look at. For each of those concerns, Spring Security provides concepts (i.e. interface definitions) to encapsulate the required application logic. While implementations are available for all popular use cases, each of those interfaces also acts as what Spring refers to as extension points for custom implementations [Sprd].

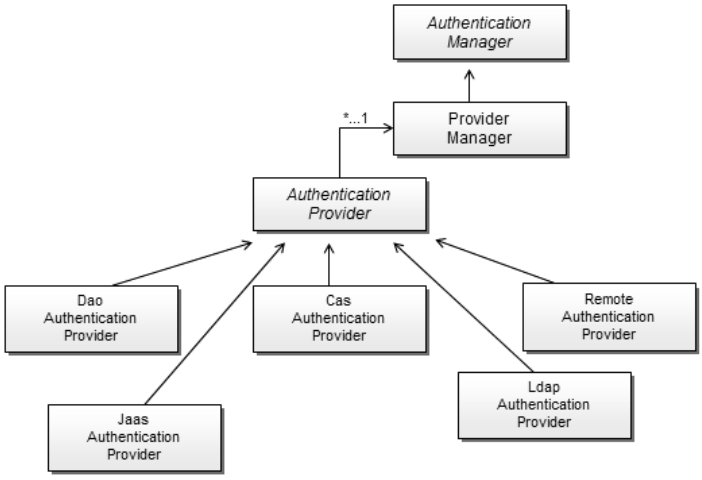



FIGURE 5.1: Spring Security Authentication Manager Architecture [Gig]

The following paragraphs are depicting the authentication processing architecture as it arises from [Sprd; Spre; Sprc]. The *AuthenticationManager* constitutes the topmost component in that architecture. Its purpose is to provide a unique interface to the software's authentication endpoints. While multiple *AuthenticationManagers* are available for different purposes, the most common representative is the *ProviderManager*. It is responsible for the first of the three aforementioned concerns. Its task is to accept authentication requests and delegate them to all suitable *AuthenticationProviders*.

Authentication requests can be distinguished by the data they provide. The most common variant is classical username/password credentials. However, also access tokens and one-time codes can be processed. Some systems are offering the latter in the form of so-called magic links that are distributed via email or text to allow password-free logins.

While iterating through the list of defined *AuthenticationProviders* the *ProviderManager* does two things: First, asking the provider whether it accepts the given type of request and second, in case the received answer was positive, triggering the actual authentication attempt. Providers are responsible for the second concern. An authentication request succeeds as soon as the first provider successfully validates the provided details.

*LDAP-* and *CasAuthenticationProvider* are two examples for providers that support authentication against specific remote systems. A more generic provider, however, is the *DaoAuthenticationProvider*, which is capable of validating classical username/password

authentication requests against arbitrary user backends. For retrieving users from a backend, it makes use of *UserDetailsServices*, a concept following the core J2EE pattern to abstract and encapsulate access to a data store via so-called *data access objects (DAOs)* [Oraa]. *UserDetailsServices* are covering the third of the defined concerns.

Earlier in this section, we stated how SoC helps to achieve the engineering quality factors robustness, adaptability and reusability. This takes effect with the *DaoAuthenticationProvider*. Separation of provider logic and data access logic supports reuse of a single provider implementation, including its mechanisms for password hashing, account validation, etc., for a variety of user backends. For one thing, this streamlines the DAO implementations provided by the framework, such as the *JDBC-* and the *InMemoryUserDetailsService*; the even greater advantage, however, lies in the granularity of available extension points. Adding custom *UserDetailsServices*, for instance, facilitates the integration of application-specific user repositories, while still taking advantage of the framework's proven security mechanisms [Sprd].

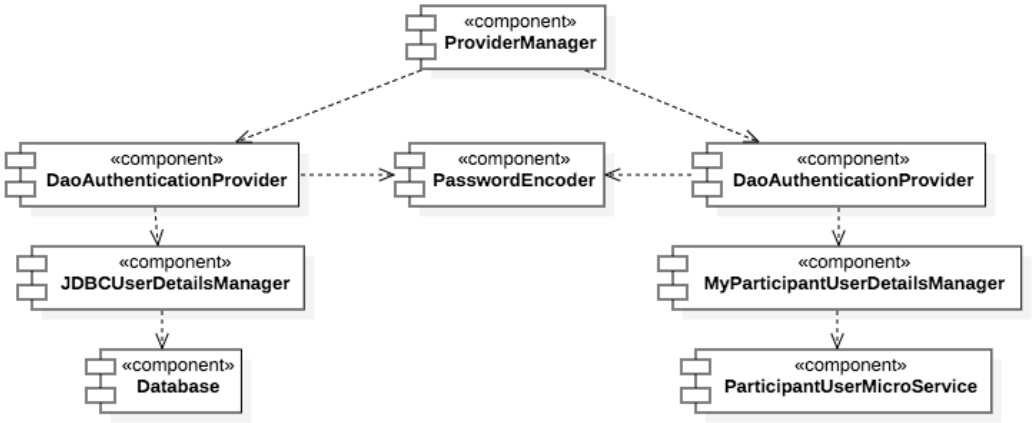



FIGURE 5.2: Authentication Manager Configuration for 2 User Data Sources

By utilizing both framework- and custom implementations for the introduced concepts, Spring Security supports arbitrary complex authentication configurations. It also allows to define multiple distinct configurations for isolated parts of the same application. This will become relevant in section 5.3, where authentication for the REST API is introduced. Based on the understanding gained from the previous paragraphs, the UML component diagram in figure 5.2 visualizes the target configuration for authenticating participant- and staff users in our case study of the teambits software. The

*ProviderManager* is set up to delegate requests to a set of two *DaoAuthenticationProviders*. One of them authenticates staff users against an internal SQL database. It retrieves those very users via the *JDBCUserDetailsService* which is provided by the framework. The second provider authenticates participant users that it retrieves from the external user service via a custom *ParticipantUserDetailsService*. As hinted above, both of the authentication providers are configured to do credential hashing during validation. They delegate this concern to a globally configured *PasswordEncoder*.

According to the requirements (see chapter 4), group authentication shall be offered as an alternative to the authentication of users. A very simple solution for that demand is to let the individual users of a group, e.g. participants or facilitators, authenticate via generic user accounts, *participants/ facilitators*. To reflect the group concept thoroughly, in contrast, advanced considerations are necessary, such as how group-authenticated individuals are reflected in the system. For now, we will stick with the previously introduced simple solution. The discussion in section 7.1, however, examines more details about alternative approaches.

## 5.2 Central Authentication

The requirements established in section 4 are demanding protection for both the application's REST API and its Messaging API. Since the latter are built on different underlying technologies, i.e. HTTP and WebSocket, they need individual consideration about how authentication can be integrated. Probably the most straight forward solution would be an integration of the previously configured *AuthenticationManager* into both APIs, allowing clients to authenticate at either of them using the same username/-password credentials. However, this is undesirable for a couple of reasons.

First, clients would either need to store credentials or connect to both APIs immediately after the user has entered his credentials. Both is not optimal, as it is neither good practice to keep credentials in memory for longer as absolutely necessary, nor should a client be required to open a web socket before it is required to fulfill primary application goals. Moreover, if a web socket connection is interrupted or a reload is required for some reason, credentials would have to be reentered. Finally, in case advanced authentication credentials shall be supported in the future, such as the previously introduced tokens or one-time codes, modification of both APIs would be inevitable.

To prevent those drawbacks, we are aiming to eliminate the use of personal user credentials for authentication at the various application endpoints. Instead, the idea is to provide a central authentication point that decouples from the remaining application

by associating the authentication with some sort of intermediate/pre-authenticated credentials (e.g. cookies, tokens or tickets as introduced in section 2.1). Those intermediate credentials can then be stored by the client and supplied with requests against the messaging- and REST API. This approach comes with the following advantage: Since the application level APIs remain unaware of the underlying user authentication mechanisms, we can connect new user data sources, support alternative credential types or implement new authentication frameworks, all without the need to modify the individual APIs. For instance, new user backends might be added, or the system could be configured as an OAuth2 or SAML client to support authentication via social-media or enterprise accounts.

This section works out a central component for establishing authentication using the previously configured *AuthenticationManager*. The subsequent two sections will then focus on bringing the established authentication to the REST and messaging API via intermediate credentials.

Spring Security provides several standard authentication mechanisms, including HTTP Basic and Digest as well as form logins. The latter is most suitable for the at-hand scenario. It relies on classic server rendered HTML forms that can be customized to reflect application styles. Furthermore, it is backed by cookie-based sessions and thus also comes with options for managing expiration and invalidations. Configuration of the same actually does not require any sophisticated knowledge about the underlying concepts. Nevertheless, for an easier understanding of the subsequent considerations, it pays off to have another look at what the framework does under the hood.

[Sprd] introduces the *springSecurityFilterChain*, a collection of Servlet filters that all of the framework's mechanisms in the web tier (UIs and HTTP backends) are based on. Servlet filters are part of the Java Servlet specification [Orab]. They dynamically intercept requests and responses before and after reaching the servlet, in order to transform the same or make use of any contained information. Figure 5.3 illustrates how Spring Security's filters integrate into the main filter chain via a so-called *FilterChainProxy*.

Table 5.1 lists the default security filters/ filter types and their ordering in the *springSecurityFilterChain*. The following paragraphs outline the interworking of those filters as it emerges from [Spre] and [Sprc]. Authentication processing filters - those filters responsible for processing credentials contained in a request - are lined up at position 4 of the chain. Examples for this type of filter are *UsernamePasswordAuthenticationFilter*, *CasAuthenticationFilter*, and *Basic-/ DigestAuthenticationFilter*. Generally all of them share a single task, which is to extract the credentials contained in the HTTP request, supply them to the configured *AuthenticationManager*, and - in case the latter's answer is positive - store the result in the *SecurityContextHolder*. Yet, filters vary in the location they look for credentials (request body or headers), the decoding of credentials (e.g.

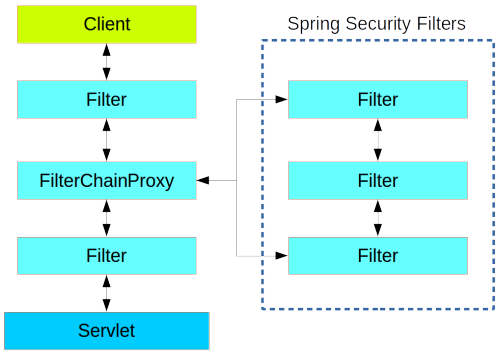



FIGURE 5.3: Spring Security Filter Chain [Sprd]

Base64 or plain), and the type of authentication request they process (username/password, token, etc.). Which of the discussed authentication processing filters are indeed contained in the chain depends on how Spring Security is configured for the individual application.

When applying the previously anticipated form login configuration, the type of filter used for authentication will be *UsernamePasswordAuthenticationFilter*. It is usually only invoked by one specific request mapping, most of the time defined by the URI *"/login"* and the HTTP request method *POST*. Apart from the previously outlined tasks, this filter's success and error handlers also redirect the user to respective success and error pages. The URLs for those redirects can either be configured statically or determined during runtime. We will make use of that feature in the following section. Before that, however, one more thing should be noted about the presented authentication scenario. Whenever an authentication request was completed successfully, the redirect response issued by the authentication filter has to travel back up the filter chain. This is when the *SecurityContextPersistenceFilter* (pos. 2) clears the *SecurityContext* and persists the contained user details in a *SecurityContextRepository* for restoration during the next request. In the default setup, this repository is backed by the HTTP session and thus, does not require any further configuration.

As previously mentioned, the management features of the underlying session implementation can be applied to control authentication sessions and perform remote logouts. However, also a direct logout mechanism is required. Spring Security provides a *LogoutFilter* that can, for instance, be mapped to the *"/logout"* URI. It also resides at position 4 of the ordering shown in table 5.1. Internally, this filter delegates tasks to a configurable set of handlers that, amongst other things, clear the *SecurityContext* and

| Pos. | Filter/Type | Description/ Example |
|---|---|---|
| 1. | *ChannelProcessingFilter* | Performs possible redirects to a different protocol |
| 2. | *SecurityContextPersistenceFilter* | Setup of a *SecurityContext* from the *Authentication* stored in the *HttpSession* at the beginning of a web request. Storing of changes to the context in the session at the end of the request. |
| 3. | *ConcurrentSessionFilter* | Updates the *SessionRegistry* to reflect ongoing requests from the principal. |
| 4. | Authentication Processing Filters | Processes contained credentials and adds the resulting *Authentication* to the *SecurityContextHolder*. Examples: *UsernamePasswordAuthenticationFilter, CasAuthenticationFilter, Basic-/ DigestAuthenticationFilter* |
| 5. | *SecurityContextHolderAwareRequestFilter* | Installs a Spring Security aware *HttpServletRequestWrapper* into the servlet container. |
| 6. | *RememberMeAuthenticationFilter* | If no earlier authentication processing mechanism updated the *SecurityContextHolder*, and the request presents a cookie that enables remember-me services to take place, a suitable remembered *Authentication* object will be put there. |
| 7. | *AnonymousAuthenticationFilter* | If no earlier authentication processing mechanism updated the *SecurityContextHolder*, an anonymous *Authentication* object will be put there. |
| 8. | *ExceptionTranslationFilter* | Catches any Spring Security exceptions so that either an HTTP error response can be returned or an appropriate AuthenticationEntryPoint can be launched. |
| 9. | *FilterSecurityInterceptor* | Protects web URIs and raises exceptions when access is denied. |

TABLE 5.1: Spring Security Filter Ordering [Spre, sec. 7.3]

invalidate the session so that authentication will no longer be restored during subsequent requests. Finally, the filter redirects the user to a preconfigured logout-success URI or back to the login page selectively.

A few words of retrospective: In summary, the configuration illustrated in the previous two sections represents a very basic authentication interface for staff and participant users. So far, it supports form based authentication by username/password credentials as well as a logout mechanism. However, future extensions can introduce further authentication options. For instance, the application can be configured as an OAuth2 client, allowing users to authenticate via a social media account. Alternatively, Spring Security SAML may be used to support integration into customers' SSO environments. Eventually, also a magic link authentication can be implemented as hinted earlier. Magic links, as described in [Mal16], are URLs containing short-lived one-time tokens that applications email to their users for password-free authentication. Any of those extensions will require advanced configuration or custom implementation of further security filters and *AuthenticationProviders*.

## 5.3 REST API Authentication

The application's REST API serves as the interface for any non-realtime communication. It is backed by regular HTTP endpoints and currently mainly used for administrative tasks via a dedicated web interface. Besides the REST API, there is furthermore the so-called result query language (RQL) endpoint, another HTTP-backed interface that is queried by clients to retrieve aggregated contribution data for the display of result charts. Technically, the RQL endpoint is not yet part of the REST API. However, since both are backed by the same underlying HTTP endpoint architecture they can be considered together in terms of authentication and authorization. Furthermore, it is favorable to unify all of the system's dynamic HTTP endpoints into one conceptual API. Hence, we will collectively use the term REST API for the entireness of those endpoints. We explicitly include all endpoints backed by Java Servlets and exclude those that deliver static resources such as script-, style-, or image files.

From the literature review in section 2.1.1, it appears that OAuth2 could be the most suitable mechanism for securing the application's REST API. When thinking about suitable mechanisms for a project, one should however evaluate whether the usage of an extensive framework such as OAuth2 implies any significant advantages over picking one of a variety of less common, but simpler authentication libraries. In the case of teambits:interactive the advantages are as follows.

First of all, OAuth2 is standardized and widely used, making it easy to find developers for both implementing clients against its services (OAuth2 consumers) as well as maintaining those services (OAuth2 providers). Its prevalence also results in widespread OAuth2 implementations with proven security being available for most languages and application frameworks, including Spring Security, which teambits:interactive relies on.

Second, OAuth2 client management capabilities are perfect for keeping control over teambits' various client applications. It provides a reliable means to prevent API access by unknown applications.

Third and last, as stated in the literature review, by using OAuth2 we can implement central user accounts and SSO, so that users are no longer required to authenticate at all clients individually.

Spring security can be set up as an OAuth2 provider in different ways. Both authorization and resource service may either be included in the same application or split across two applications [Spra]. Splitting them up is obviously beneficial when following a SOA approach. Within SOA environments, applications are split up into microservices performing small subtasks towards the overall application goal. Microservices can be

distributed over a server landscape and launched in arbitrary quantities to encounter evolving load [ZF09, p. 7ff]. In such environments, using a segregated authorization service to establish SSO over all microservices helps providing a seamless UX and increases scalability and security.

The advantages of decoupled applications, however, also come with overhead in terms of network communication and the high-level protocols it involves. In the case of the Spring Security Oauth2 implementation, decoupling authorization and resource services means that for every request processed by the resource server, an additional HTTP request is send to the authorization server for access token verification [Spra]. While this overhead is generally acceptable in highly-scalable online environments with extensible hardware resources, it should be avoided in offline-environments, where those resources are strictly limited. In full-service scenarios, for instance, it is not uncommon for teambits:interactive to run on a single notebook, serving events with hundreds of simultaneously interacting participants. Consequently, integrating both the OAuth2 resource service and authorization service in the main application seems to be the most adequate choice in this situation.

Figure 5.4 illustrates how both services integrate into the existing monolithic Spring application. The particular difference of this architecture to its distributed alternative is the token service. In a distributed setup distinct token services have to be used for resource and authorization service, whereby the one on the resource service is essentially an HTTP client validating tokens on the authorization service [Spra]. In our combined setup, on the other hand, both parties can access a shared token service via Java method calls, causing comparatively less overhead.

Both OAuth2 services are similar structured in terms of their Spring Security components. In each case, the Spring Security filter chain - a collection of filters that all requests travel through before and after reaching their endpoints [Spre, sec 10.1] - includes a filter for authentication (*AuthenticationProcessingFilter* / *SecurityContextPersistenceFilter*) and one for authorization (*SecurityInterceptor*). As said above, arbitrary authentication mechanisms can be implemented on the authorization server. The suggested architecture in figure 5.4 uses cookie-based HTTP sessions to store authentication between requests. This is beneficial, as explained in section 2.1.1, since cookies are automatically handled by the user-agent.

Whenever a registered client application needs to obtain an access token, it redirects to the *AuthorizationEndpoint*. In case the user is signed in, authentication details are fetched from the session-based *SecurityContextRepository* and stored to the *SecurityContext* for the remaining request lifecycle. Access is granted by the *SecurityInterceptor* to all authenticated users. Non-authenticated users are redirected to a sign-in dialog as described in section 5.1.

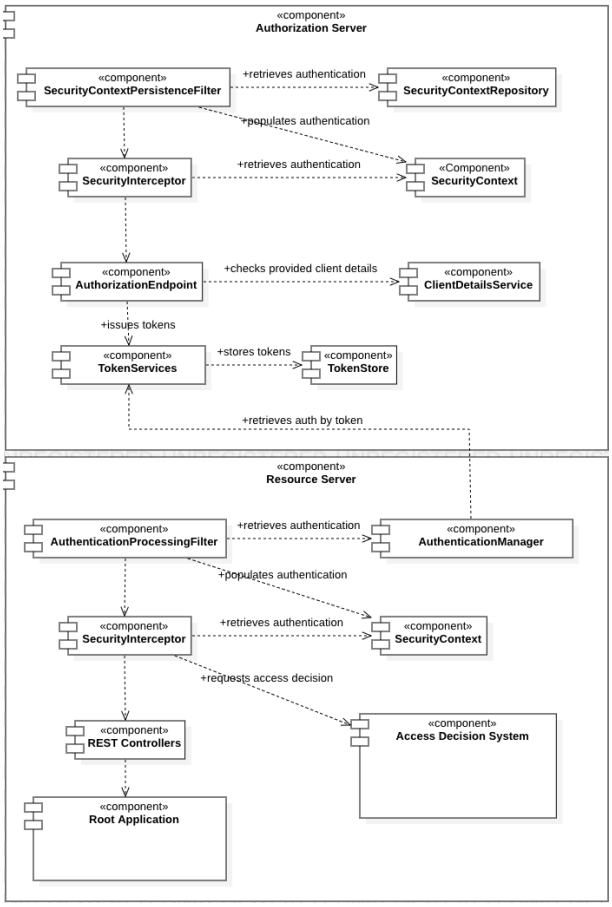



FIGURE 5.4: Spring Security OAuth2 for a REST API (UML Components)

Depending on the chosen flow type, the *AuthorizationEndpoint* either uses *TokenServices* to issue an access token directly (implicit grant), or generates an authorization code for the client to request an access token in an additional step (authorization code grant). Technically, both flow types are equally possible for the at-hand scenario. We are opting for the implicit grant, as it reduces the number of necessary requests. Either way, until expiration of the same, tokens are kept in the *TokenStore* together with the associated authentication.

On the *Resource Server*, requests are processed as follows. Similarly to the *Security Context Persistence Filter*, which restores authentication from a session repository, it is the *Authentication Processing Filter*'s responsibility to restore authentication for the provided token. It uses an *Authentication Manager* to validate the token at the *Token Services*

and retrieve the associated authentication details. The latter are then stored in the *Security Context* for the remaining request lifecycle. Eventually, the *Security Interceptor* is responsible for granting or denying access to the requested REST endpoint. It utilizes the information stored in the *Security Context* to consult the *Access Decision System* derived in section 6.

## 5.4 Messaging API Authentication

The messaging API in teambits:interactive is responsible for two-way communication between clients and the server. Clients connect to that API on startup, subscribe for a meeting, and use it to submit their contributions. Whenever events happen in a meeting (e.g. an operator unlocks new interactions), the server makes use of this API to notify subscribed clients. In contrast to the REST API, a lot of the components in the messaging API are custom developments and not based on standardized protocols and frameworks. Nevertheless, the messaging API shall also be designed to outsource user authentication to the central authentication component, in a way, similar to the OAuth2 setup outlined in the previous section.

In order to develop an adequate concept for integrating the previously worked out security system into the API, we will, in the first instance, have a look at its current design, elaborated by Axel Guicking in [Gui07, sec. 4.2.2].

### 5.4.1 Message Handling

Figure 5.5 provides a simplified overview. As mentioned earlier, the messaging API is backed by two underlying communication protocols: WebSockets and HTTP-based polling. On the transport layer, those protocols represent distinct technical realizations that strive to provide adequate alternatives for network environments that are subject to certain restrictions.

On the application level, however, both protocols are abstracted and thus widely indistinguishable. As opposed to the stateless nature of the REST API, the messaging API is based on the concept of connections that are kept up for the entire session lifetime. This is obvious for the WebSocket part of the API, which is indeed relying on longstanding physical TCP/IP connections. The polling approach, in contrast, uses separate connections for every poll, thus making it necessary to encapsulate coherent requests into a logical connection abstraction. teambits utilizes the concept of a *ConnectionRegistry* for managing all client connections. It abstracts the concrete connection type – WebSocket

or polling – and provides a unified interface for higher-level application components to use, so that the latter remain unaware of any technical communication details.

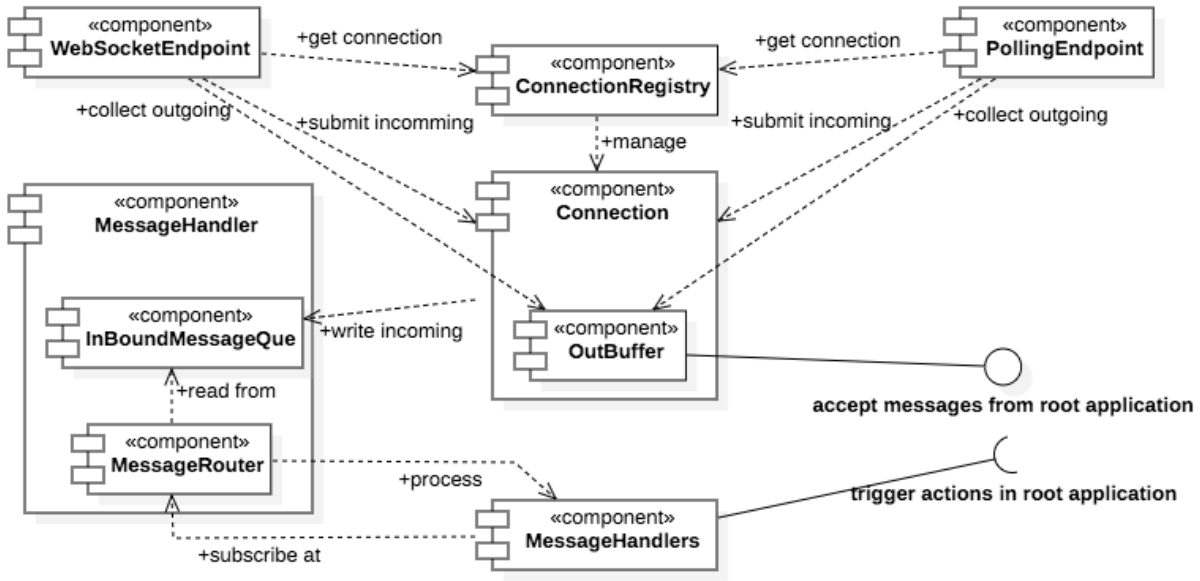



FIGURE 5.5: Messaging API (UML Components)

Besides this abstraction, another important concept within the messaging API is that of utilizing queues and buffers for incoming and outgoing messages. The system writes incoming messages to an ordered *InboundMessageQueue* before processing. Likewise, outgoing messages are stored in an *OutBuffer* before sending. Messages are furthermore kept in the *OutBuffer* until their receipt was acknowledged by the client. This technique is meant as an application level reimplementation of the so-called sliding window protocol. Usually being applied as a transport layer algorithm, this protocol fulfills the following three purposes [PD11, p. 117]:

1. Reliable delivery of messages over an unreliable network,
2. preservation of the ordering of messages, and
3. execution of flow control – i.e. throttling of senders by receivers

The algorithm is implemented to control data package transmission in the transport layer protocol TCP and thus usually requires no further consideration on application layer. During intense field tests however, a characteristic was discovered by teambits that is specific to the operation environment of live facilitation software. This kind

of software frequently operates in non-stationary network setups that are subjected to traffic peaks utilizing them to capacity. It proved an occasional scenario that hardware in such setups crashes and requires a reboot before resuming work. In those extreme events, the developers from teambits were able to observe data packages irretrievably getting lost, even when utilizing TCP. This lead to the decision to implement the aforementioned application level mechanism around the *OutBuffer*. It supports the first two of the three aforementioned purposes followed by the sliding window protocol.

The following outlines a typical connection lifecycle. When connecting, clients are required to provide their *clientId* as part of the endpoint URL. Depending on whether it is connecting the first time or reconnecting after an interruption, the client will be assigned a new or its previous logical connection instance. Each incoming message will be associated with and passed to the client's logical connection, which in turn submits it to the *InboundMessageQueue*. The processing of messages from that queue lies in the responsibility of the *MessageRouter*, which – depending on the concrete type of a message – further delegates it to individual *MessageListeners*. For processing, *MessageListeners* make use of services in the root application. In this sense, they might thus be seen as the messaging API counterpart for HTTP controllers in the REST API. Application services may generate outgoing messages to arbitrary clients and place them in the *OutBuffers* of the corresponding connections. This may either be triggered by incoming messages (e.g. an operator unlocking new interactions) or happen in response to any other external or internal event. From the *OutBuffers*, messages are regularly picked up and delivered to the clients by the respective mechanisms of the underlying physical connection.

Whenever a new connection is initialized, the system triggers an application level handshake for exchanging client parameters and determining the meeting to subscribe to. Currently also included in the handshake is the user identification process (e.g. free text or dropdown) and the optional challenge-response authentication as introduced in section 1.2.2. None of those mechanisms currently integrates into, or makes use of any Spring Security features. The handshake is triggered by the client as soon as the connection setup is completed and the server signals it is ready for receiving messages. It exclusively makes use of the previously described messaging flow.

For the subsequent security considerations, a few details should be known about how the processing tasks in the messaging API are utilizing various multithreading capabilities. Physical connections (both HTTP and WebSocket) are maintained within thread pools managed by the framework or the Tomcat container. This is decoupled from further parts of the system by the *InboundMessageQueue* and hence does not require any deeper inspection. The *MessageRouter* is running as a daemon in a dedicated thread. It picks up messages from the queue after they have been placed there by framework or

container threads. Handling of those messages then happens in one of the following two ways. Listeners that are typically causing low processing/ I/O load, execute directly in the *MessageRouter* thread; those performing more complex tasks, in contrast, are usually configured to use a managed thread pool for asynchronous execution.

From the preceding exposition, we can now conclude two challenges to cope with in the further course of this section:

1. The central authentication system shall be integrated into both connection implementations. This will allow pre-authenticated users to connect without resubmitting their credentials.
2. In the message processing threads, a strategy shall each time setup the *SecurityContext* with the processed message's respective authentication info. This will facilitate the use of Spring Security's authorization mechanisms to apply access control during message processing.

### 5.4.2 Authenticating WebSocket Connections

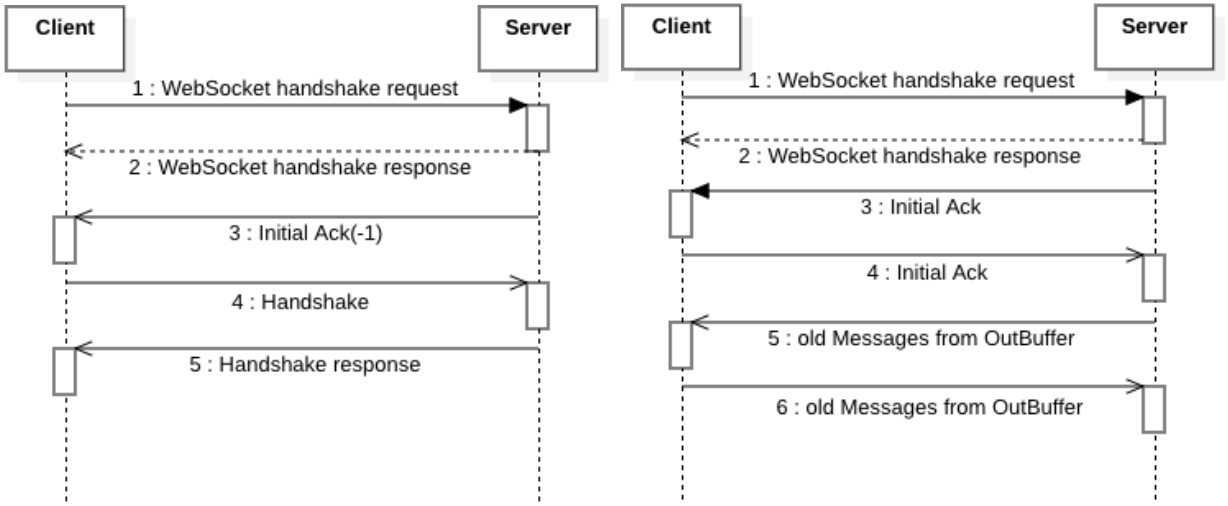
This section copes with the first of the aforementioned challenges. Specifically, it works out a solution to integrate authentication into the WebSocket part of the messaging API. The literature review in section 2.1.2 suggests personalized endpoint URLs or a ticket system as two adequate solutions for establishing authentication in a WebSocket connection. Before looking into the details of those solutions, however, we will take a closer look on how messages are currently handled by the messaging API's WebSocket connection implementation.

#### Current Architecture

Figure 5.6 depicts a sequence for the initialization of WebSocket connections as well as one for reconnections after an interruption. Both sequences start with a WebSocket handshake between client and server (1,2). As depicted in RFC 6455, the handshake is started by the client via an *HTTP Upgrade* request and completed by the server via the corresponding *Switching Protocols* response [FM11, sec. 1.3]. As soon as the WebSocket is initialized, the following two scenarios need to be distinguished:

**A** The client connects for the first time or after a longer period of time. The server does not currently maintain a previously used logical connection to the client.

**B** The client reconnects shortly after an interrupted connection. In this case, the server reuses the maintained logical connection, including the client's subscriptions, and thereby circumvents the need for a repeated application level handshake.




(A) Connect (B) Reconnect

FIGURE 5.6: Messaging API – WebSocket Connection (UML sequence diagram)

In both cases, the first message sent by the server is an initial acknowledgement (ACK). ACKs are known from TCP as a measure to provide reliability during communications over an unreliable network [Pos81, sec. 1.5]. At the messaging API, both clients and the sever issue ACKs containing the ID of the last received message. Message IDs are incremented independently per party (server/client) and per connection so that each party in a communication can track delivery of their messages and resend them in case they get lost. On top of that, the initial ACK has an additional purpose. Whenever a new logical connection is created (scenario A), the server sends -1 along as the ID. This indicates to the client to start with the application level handshake before sending any content-related messages. In case of a reconnect (scenario B), in contrast, the server sends along the ID of the last received message to indicate to the client where to resume sending. The client itself then also has to reply with an initial ACK, indicating the last message it received from the server. Only after receiving their respective ACK, client and server resume sending messages. Special attention should be payed to the following fact: When reusing a logical connection, both client and server are (re)sending messages from their *OutBuffers* that have been generated during previous

physical sessions. Without further protection, this handling is prone to information leakage and consequently requires special attention during authentication design.

Earlier we mentioned personalized endpoint URLs and a ticket system as two promising solutions for establishing authentication at a WebSocket connection. Both of those solutions have been shown to provide equally good security characteristics. Subsequent analysis will look at each approach in detail to find out which best integrates into the current implementation of the messaging API. For a start, however, Spring Security's integrated support for WebSocket authentication shall be analyzed, even though it would likely require major architectural changes to the current system.

### Spring Security WebSocket Support

Since version 4, Spring Security supports authentication and authorization at WebSocket connections. More precisely, the framework takes care of the following three aspects [Spre, sec. 10.11]:

- Enforcement of the same origin policy using CSRF tokens
- Setup of the *SecurityContext*
- Authorization of messages via Spring Security's default access control mechanisms

According to the reference documentation, Spring Security reuses the authentication information from the HTTP Upgrade request that initiates the WebSocket session. Thus, if Spring Security is configured to secure HTTP requests, and a user is already authenticated, that authentication will automatically propagate up to the WebSocket connection [Spre, sec. 10.11]. This is beneficial if we are authenticating users via cookie-based HTTP sessions, as those are automatically managed by the browser. In section 5.3, however, we decided to use OAuth2 tokens for authentication in order to gain cross-origin authentication support as well as increased control by the JavaScript client application.

Notwithstanding the above, Spring Security's WebSocket support is based on Spring's WebSocket abstraction. Currently, the messaging API does not make use of the abstraction, but instead relies on the bare WebSocket implementation of the underlying Tomcat server. Thus, applying Spring Security's WebSocket support would, as stated above, require some architectural changes. Moreover, it implies that all processing of messages would move from our custom threading model to thread pools managed by Spring. While this is generally a good thing, it conflicts with the queue and buffering mechanisms that are currently applied for increased reliability.

Thus, in order to integrate Spring's WebSocket abstraction while at the same time maintaining queues and buffers, we would have to chain those mechanisms. That way, each message would be processed once by Spring Security for performing authentication and authorization, then put in a queue, and eventually processed again. In summary, if we want to make use of Spring Security's WebSocket support, the following limitations apply:

- We are forced to restructure the messaging API to make use of Spring's WebSocket abstraction
- We would need to use cookies, HTTP Basic, or -Digest for authentication, limiting our control via the JavaScript client
- We would have to chain the existing queue/buffer and *MessageRouter* concepts behind the framework's processing mechanisms, introducing additional overhead

Altogether, Spring Security's WebSocket support does not integrate well with the software's current architecture as well as our architectural goals. Consequently, subsequent sections are considering the two alternative concepts that were mentioned at the beginning:

- Personalized one-time URLs and
- Tickets

Both of those approaches are making use of a separate HTTP endpoint that issues some sort of private data for authenticated users to supply at the WebSocket endpoint. With personalized URLs, that data is in the form of a random and unique secret, i.e. a token. Tickets, in contrast, contain actual session information such as user ID and IP address. With both approaches, the issued data is of a temporary and one-time nature, meaning it expires both after first use as well as a fixed amount of time. This is to reduce the risk for impersonation in case of disclosure, both accidentally or per concept. Personalized URLs, for instance, are per concept disclosing the secret after first use, as the URL may persist in the logs of proxy servers and firewalls. The ticket approach, on the other hand, uses the readily set up WebSocket to transfer ticket data. Thus, as long as communication happens encrypted, data stays private while using this approach.

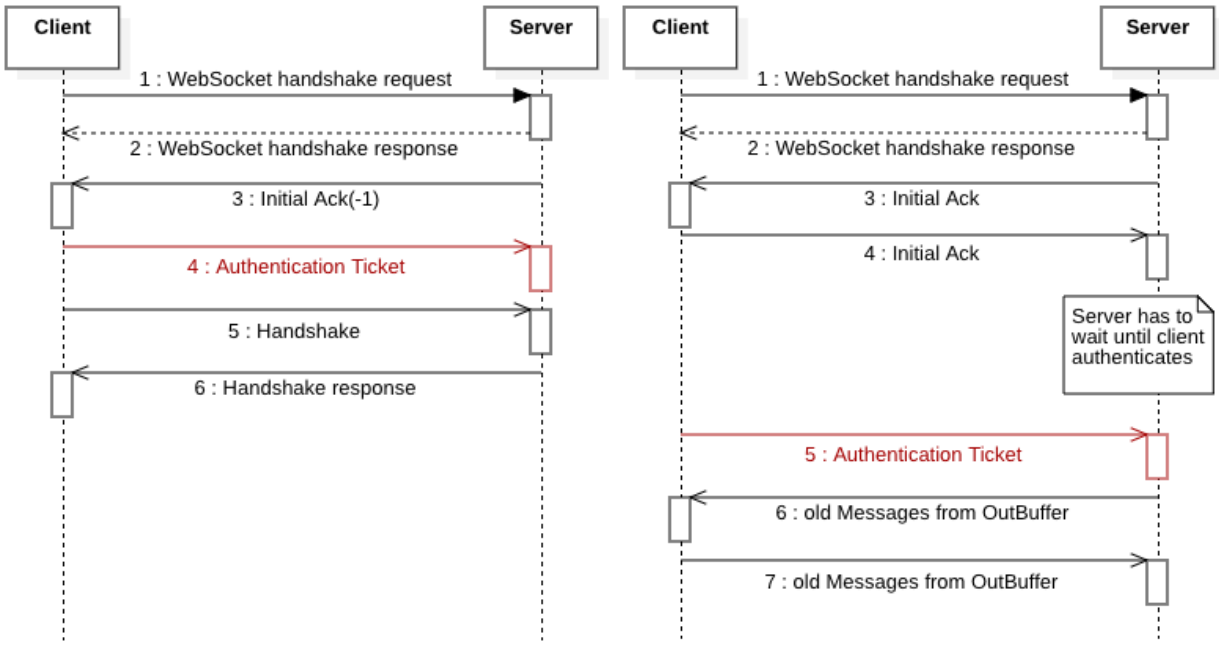
In the next step, attention should be on how well each of the aforementioned approaches integrates into the system's current architecture. Let's start by having a closer look at ticket authentication.

### WebSocket Authentication via Tickets

[Her] describes how tickets are commonly transmitted via an initial handshake. At the messaging API, that could, for instance, be integrated into the existing application level handshake. The latter currently includes the optional challenge-response-based password protection as well as the pseudo authentication outlined in section 1.2.1. Both of those could be replaced by equivalent steps for ticket evaluation. As mentioned before, tickets in the the stated literature are assembled from session, time and user information and apply a strict single usage policy. This is however not necessarily the only option to make use of the ticket approach. Instead of the aforementioned cleartext information, we might as well use a secret as ticket content, just as with the URL approach. In fact, it seems worth to consider reusing the REST API's OAuth2 tokens, as clients will obtain them anyway for accessing resources at the REST API. OAuth2 tokens, as introduced in section 5.3, are issued for a medium-long time period and allow unlimited reuses. Disclosing them, thus poses a risk for impersonation. Reusing those tokens at other parts than an application's HTTP endpoints is indeed not intended in the OAuth2 standard; however, as transmission via the WebSocket channel happens encrypted, it implies the same level of security as supplying the token within the HTTP authorization header. After all, reusing the OAuth2 token, instead of explicitly generated tickets, reduces the amount of necessary HTTP request by one per authenticating user. With a lot of users, this easily causes significant savings in terms of authentication-related traffic.

Figure 5.7 extends the previously introduced connection and reconnection sequences by use of OAuth2 tokens. Sequence A again outlines steps that apply whenever a client connects for the first time. Initially, the WebSocket connection is till opened without authentication. After an indication by the server that a new connection was created (initial ACK of -1), the client sends the authentication ticket containing the OAuth2 token as part of the application level handshake (step 4). In case of an invalid ticket, the server may immediately close the connection. Otherwise, the handshake completes successfully and messaging starts regularly. If no authentication ticket is supplied in step 4, the connection will be set up as anonymous. As functional requirement *FR12* (see section 4) explicitly requires authentication to be optional, anonymous connections must not directly be refused and instead should be allowed to proceed to the authorization system for further handling. The authorization system will be developed in section 6. It is responsible for appropriate handling of unauthenticated connections. Depending on the configuration of individual meetings, it may for instance decide to deny anonymous connections or asign them a reduced set of privileges.

So far, this solution appears solid. The OAuth2 token is sufficiently protected to avoid disclosure and risk for impersonation. Sequence B, however, reveals that – regardless




(A) Connect (B) Reconnect

FIGURE 5.7: Messaging API – WebSocket authentication via OAuth2-backed tickets (UML sequence diagram) – authentication elements are highlighted in red

of the used secret – a vulnerability arises from the messaging API's behavior when clients are reconnecting after an interruption. The system relies on queues and buffers for reliability and to allow quicker resuming of messaging after reconnects. This feature is super sensitive to information leakage during what we call a temporary downgrade from an authenticated to an unauthenticated session during communication setup.

The following example outlines this vulnerability in more detail. Say, an honest client H is maintaining an authenticated WebSocket connection to the server. At some point in time, network issues are causing the connection to abort. When the client wants to reconnect, it regularly identifies itself at the server using its client ID (step 1). The server accepts the request and retrieves the corresponding logical connection from the *ConnectionRegistry*. However, even though the previous connection was an authenticated one, we can no longer rely on the client's identity. As the client ID is transmitted via the URL, it can, as stated multiple times before, not be considered trustworthy. Beyond that, client IDs are generally not treated as a secret in the software, as they

are displayed to operators and other users in cleartext. Consequently, whenever an authenticated connection between H and the server is interrupted, a malicious client M could use H's ID to reconnect. To prevent M from impersonating the user behind H, we need to enforce re-authentication for all connecting attempts that are trying to resume proceeding sessions.

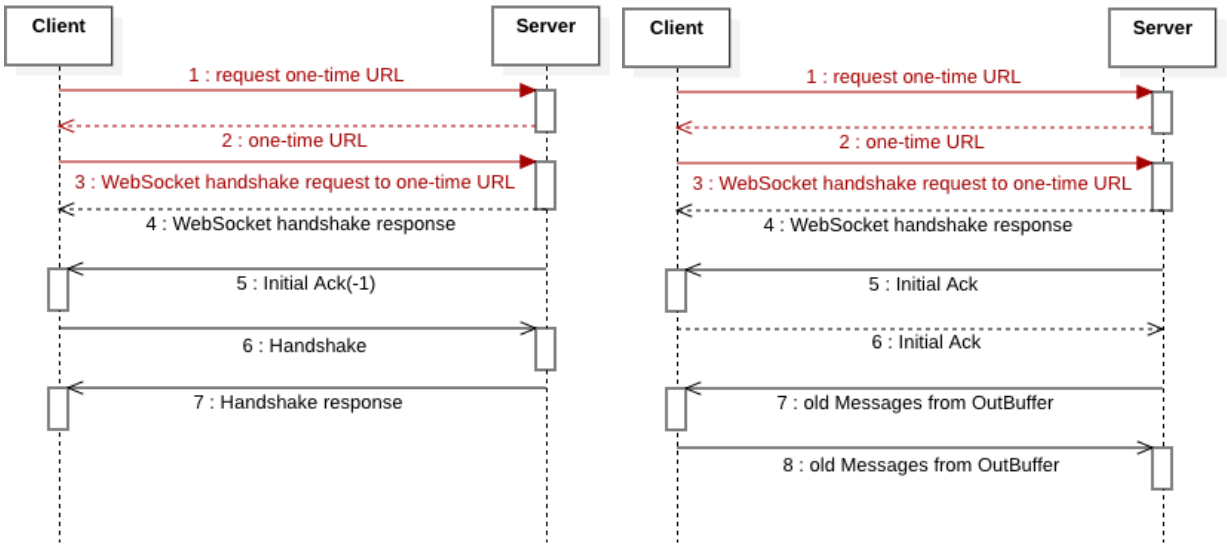
While this is technically achievable, difficulties arise from a characteristic of the currently used sequence. When H connects, the server immediately retrieves its previous logical connection. H is then required to resend its authentication ticket. If and when the client follows this requirement, however, is out of the server's control. Optimally, the ticket should be transmitted during the following application level handshake; however, if M is pretending to be H, it could easily delay or omit authentication completely. Thus, for the time between the retrieval of the logical connection until the authentication ticket arrives at the server, we are speaking of a temporarily downgraded connection. During that phase, we cannot be sure about the connecting clients actual identity. As a consequence, to avoid information leakage, the server must not accept or send any messages via the authenticated logical connection during that time. This is contrary to the current behavior, which intends to resume sending messages from the *OutBuffer* as soon as the client's initial ACK was received. While it is certainly possible to make the server wait for an authentication ticket to arrive before resuming message delivery, the solution would imply some interfering modifications of the application logic. This is not per se an exclusion criteria, however, coupling of security and application logic often bears risk for silent breaks during later refactoring and is thus preferably avoided whenever possible.

#### WebSocket Authentication via Personalized One-Time URLs

As an alternative to the ticket mechanism, we will finally look at an approach based on personalized one-time URLs. Instead of leaving authentication to a custom messaging protocol, personalized URLs include credentials as path or query parameters that are supplied to the server with the WebSocket handshake request. This causes verifiable user information to be available right from the start of a connection and thus rules out the aforementioned problems around temporary downgrades during connection setup. Compared to the ticket solution, this approach consequently requires less modifications to the application logic and hence supports better encapsulation of security concerns. Personalized URLs make use of randomly generated secrets. Other than tickets, however, URLs are disclosing this secret on transmission and thus require it to expire after first use. As a consequence, reusing OAuth2 tokens is not an option with this approach. Instead, clients will have to request a new one-time token for each

WebSocket connection they intend to open. Also, the server cannot make use of any of the previously depicted mechanisms, such as the OAuth2 *TokenServices* and *AuthenticationProcessingFilter*. Instead, it requires implementation of dedicated *TokenServices*, *-Stores* and filters to generate, keep, extract and validate one-time tokens.

Figure 5.8 extends the WebSocket endpoint's connection and reconnection sequences by use of personalized URLs. Anytime before starting a connection, as well as before reconnecting after an interruption, clients have to obtain a new one-time URL. One-time URLs or their contained tokens can be provided via regular OAuth2-protected HTTP endpoints. On request (step 1), the server generates the token, associates it with the authentication in the *SecurityContext* and, before returning it to the client (step 2), stores it within a *TokenStore*. When subsequently opening the WebSocket, the client provides its token as part of the endpoint URL (step 3). To reuse as many parts of Spring Security as possible, we should then implement a custom *AuthenticationProcessingFilter* that extracts the token from the URL, validates it using custom *TokenServices* and again sets up the *SecurityContext*. The associated authentication will then be made available by the framework within the WebSocket session. It can subsequently be used during the second of the previously established challenges: Setting up the *SecurityContext* within the message processing threads.




(A) Connect (B) Reconnect

FIGURE 5.8: Messaging API – WebSocket authentication via personalized one-time URLs (UML sequence diagram) – authentication elements are highlighted in red

After step 3, we could easily make use of standard Spring Security mechanisms and reject incoming requests that are missing an authentication token. As mentioned earlier, however, *FR13* requests to make authentication optional. Thus, we are not applying authorization at this point and instead accept all incoming requests, associate them with a new or an existing logical connection and return the WebSocket handshake response (step 4). Subsequent steps remain unchanged to the previously outlined sequences. Whenever a client connects the first time (sequence A), the server sends its initial ACK (step 5) containing -1 to trigger the application level handshake. In case of a reconnect (sequence B), in contrast, the initial ACKs from client and server (step 5 and 6) both inform about the respective id of the last received message to indicate where to resume sending messages from the corresponding *OutBuffer*.

The previously derived technique establishes trustworthy user authentication right from the beginning of a connection and thus avoids the complexities arising from the temporary downgrade phenomenon of the ticket approach. However, it does not guarantee the consistency of identities between individual connections. In fact, authentication information might change between physical connections and - due to *FR13* - also a permanent downgrade from an authenticated to an anonymous connection is technically possible. To illustrate the challenges resulting from this context, we are reusing the previous example of an honest client H and a malicious client M:

H maintains an authenticated connection to the server. If, at one point, the connection aborts, M could uses H's ID in order to connect and try to hijack the authenticated logical connection. In order to pass security, M may either completely omit authentication or use another user's identity. According to the formerly worked out configuration, Spring Security will accept the connection in both cases. Without further interception, M would be assigned H's authenticated logical connection and receive potentially private messages from the latter's *OutBuffer*. Furthermore, depending on the implementation of authorization (see section 6), M might even escalate its privileges and trigger actions in H's name. To eradicate this privacy and security risk, an additional check must be carried out during reconnects. The authentication supplied with new physical connections is obliged to match the one stored within the targeted logical connection. In case authentication in the new physical connection differs, or the new connection is anonymous, one of the following two procedures should be triggered:

1. The affected physical connection is closed with an adequate error. In this case, the logical connection may be maintained for subsequent connection attempts.
2. The logical connection is discarded and replaced by a new one to associate with the new physical connection's authentication.

Procedure 1. maintains the client's subscriptions and avoids reiterating over the handshake in the event of connection related problems (e.g. expiring of tokens due to increased latency). On the other hand, the approach may lead to a temporary lockout in case of updated authentication info (e.g. a modified username). Procedure 2., in contrast, avoids lockouts, but may instead lead to mistakenly discarded connections which requires retransmission of client-, user-, and subscription data. Regardless of which procedure is eventually chosen, its implementation is sufficiently trivial compared to the ticket approach. Using the only just elaborated URL approach, the connection setup, including all authentication aspects, can securely be completed before the server sends out its initial ACK. No temporary downgrades are delaying the handling decision, making the approach significantly less fragile to refactoring errors.

To conclude our considerations, it is worth doing a brief recap of the authentication alternatives discussed within this section. We analyzed Spring Security's integrated mechanism and found that it is not applicable for the at hand application. For one thing, it requires significant architectural changes to the application and secondly, it establishes authentication using default HTTP mechanisms and propagates it to the WebSocket connection. Since JavaScript does not allow customization of HTTP headers on a WebSocket handshake request, using those Spring mechanisms would limit us to browser-managed authentication.

Ticket systems were found to be a more suitable approach. They transmit authentication information only after the WebSocket is established and are thus decoupled from regular HTTP authentication mechanisms. However, when applied to the messaging API's reconnection mechanism, this approach causes what we defined as a temporary downgraded connection, leading to a complex and thus potentially fragile implementation.

Eventually, personalized one-time URLs were discussed. While requiring additional implementation effort (*TokenStore* and *TokenServices*), this approach makes best use of proven Spring Security mechanisms and allows reasonable separation of security- and application logic. In spite of the aforementioned browser restrictions, it can apply authentication directly when opening a WebSocket and thus mitigates the temporary downgrade problem to a permanent downgrade, which is manageable with less complexity. In the context of the messaging API's specific sequence design, personalized one-time URLs consequently seem to represent the most promising solution in terms of security and robustness.

### 5.4.3 Authenticating Polling Connections

Comprised by the authentication mechanisms derived for WebSockets, this section works out an adequate counterpart for the polling connections. It also belongs to the first of the two previously established challenges: Integrating central authentication into the messaging API. Polling connections are meant as a fallback for network environments where WebSocket connections are blocked by proxies or firewalls. As the connection type shall be abstracted on application level, we need to construct a mechanisms that - similar to the WebSocket connection - authenticates the user and maintains the authentication info within the logical connection. A significant difference, however, lies in the fact that the polling connection, in contrast to WebSockets, is only a logical connection comprising from individual physical TCP connections that are opened and closed before and after each poll. Other than with WebSockets, there is no guarantee that messages sent via multiple polls do indeed always originate from the same sender. Incoming polls are handled by the *PollingEndpoint* (see figure 5.5). The latter identifies the related connection by an ID that is specified as a path parameter in the URL. Currently that ID is enough to associate the poll with an active connection. Since the ID is not kept as a secret, but accessible at least to logged in operators and administrators, it could easily be abused to impersonate other users.

Consequently, to achieve security, we need to authenticate each poll individually before assigning it to its logical connection instance. Luckily however, since polling happens via HTTP, we can use one of the standard authentication mechanisms available for that protocol. Since OAuth2 is already setup for the REST API, it comes in handy to reuse it for authentication at the polling endpoint. That way, clients do not need to request another ticket or token, but can actually reuse the header configuration required for requesting resources via REST.

Figure 5.9 outlines the steps included when processing a poll with OAuth2 authentication. As described in section 5.2, an *AuthenticationProcessingFilter* within the *springSecurityFilterChain* is responsible for extracting the access token from the request, authenticating the associated user and setting up the authentication info within the *SecurityContext*. The request is then passed on to the polling endpoint, which compares the authentication info from the *SecurityContext* with that, kept by the logical connection. Only if both authentication infos match, all incoming messages contained within the request body are passed to the connection, which eventually writes them to the *InboundMessageQueue* (see figure 5.5). If no connection exists for the specified ID, it will be created using the authentication that is currently in the *SecurityContext*. Outgoing messages from the connection's *OutBuffer* are handed over to the *PollingEndpoint*. Here they are added to the response body. After clearing the *SecurityContext*, the response is finally sent to the client.

The previously outlined cycle repeats for every incoming poll. In case the comparison in step 8 fails, an adequate handling is required. For instance, the referenced logical connection could be destroyed, causing the application level handshake to re-trigger. However, this is sensitive for denial-of-service (DoS) attacks, as it can be abused to deter legitimate users from accessing the application. Alternatively, the poll could simply be blocked by returning an HTTP 403 response. This, in turn, may - just as for WebSocket connections - cause temporary lockouts in case authentication details change during a session. Irrelevant of the chosen solution, it must be ensured that no messages from the compromised request are written to the *InboundMessageQueue* as well as no messages from the *OutBuffer* are sent to the client.

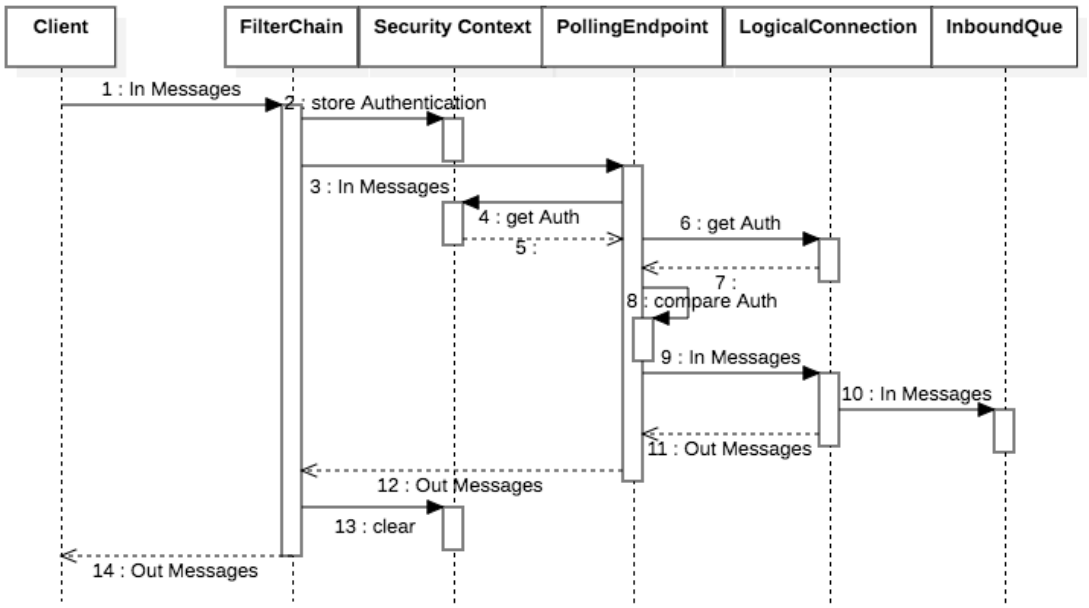



FIGURE 5.9: Polling Connection Cycle (UML Sequence)

### 5.4.4 Establishing the Security Context for Message Processing

Thanks to the achievements of the preceding two sections, the messaging API is now capable of authenticating users in both WebSockets and polling connections. This section finally focuses on the second of the previously derived challenges: Establishing the *SecurityContext* within an adequate scope to make it available within message processing threads.

While this section focuses on authentication, rather than authorization, the following info about access control in Spring Security might be necessary to understand the succeeding considerations. Spring Security performs authorization via so-called *SecurityInterceptors* [Spre, sec. 8.1.5]. *FilterSecurityInterceptor* and *MethodSecurityInterceptor* are the two main implementations for that concept. The former resides in the *springSecurityFilterChain* (see table 5.1) to secure HTTP requests, the latter can be used to secure method calls. Independent of which interceptor we are using, Spring Security relies on the authentication stored within the *SecurityContext* to make authorization decisions. The *SecurityContext* is usually configured as a *ThreadLocal* variable, meaning it is available to all methods that are called within a single thread of execution [Spre, sec. 8.1.2].

Earlier in this chapter, it was derived how the processing of messages is decoupled from their incoming and outgoing channels by the use of queues and buffers. The messaging API authenticates users within logical connections. Incoming messages are assigned to their respective connection before being written to the *InboundMessageQueue*. Other than the connection assignment, there is no further processing happening before messages are written to the *InboundMessageQueue*. This is, inter alia, done for performance reasons and should not be altered. Consequently, authorization decisions – i.e. whether a user is allowed to perform the requested action – have to be made when the messages are pulled from the queue for processing. As previously stated, dequeuing is done by a *MessageRouter* and the actual processing tasks are performed by *MessageListeners*. The threads these components are running in differ from those used by the connection endpoints. Thus, even if Spring Security manages the *SecurityContext* at the endpoints - see for instance the OAuth2 integration for polling - the context will not be available within the message processing threads. Also a propagation of the context as described in [Spre, sec. 13.12.4] is out of question as threads are decoupled through the *InboundMessageQueue*.

Consequently, in order to integrate Spring Security's authorization system into the message processing, we need to derive a custom strategy to set up the *SecurityContext*. A solution could be to setup the context within the *MessageRouter* when it picks up a message from the queue and clear it after processing. That would cover those messages that are processed directly within the router's thread. Those that are processed by threads within a managed thread pool, however, would still require additional consideration. Indeed this problem could be solved by propagating the *SecurityContext* down to the threads from the thread pool. However, it is perceived an even cleaner approach to set up the context right where it is required, namely in the *MessageListeners* themselves.

The following solution strives to achieve exactly that. Making authentication info

available is a crosscutting concern, that stretches over the variety of available *MessageListeners*. Hence, it is a typical application for AOP mechanisms. AOP was previously introduced in section 5.1 as a technique for centralizing functionality that is otherwise distributed over various application modules. We can use the technique to dynamically attach our *SecurityContext*-setup-strategy to all *MessageListeners*. Before designing a solution for that endeavor, the following two paragraphs briefly outline central concepts and terminology in AOP [KH01]:

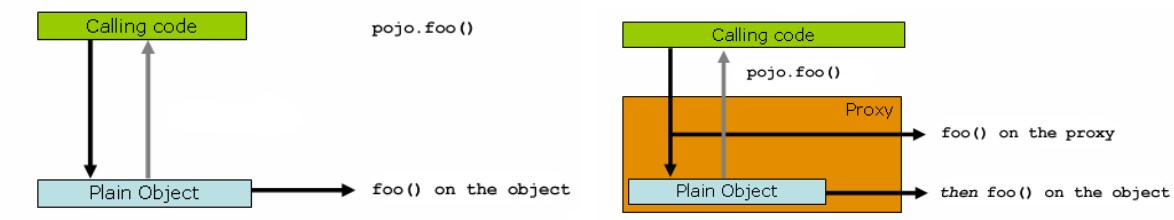
**Aspect** - module providing centralized functionality for crosscutting concerns.

**Joinpoint** - well defined point in a program's execution (e.g. constructor call, method call, field access, etc.)

**Pointcut** - collection of joinpoints designated e.g. by method names or wildcards.

**Advice** - the action to be taken before or after execution of the joinpoint.

The Spring framework offers an AOP implementation that is based on so-called proxy objects. An AOP proxy is an object created by the Spring framework in order to append an aspect's functionality to the original business object [Sprb, sec. 5.1]. Figure 5.10 illustrates how those proxies work. Instead of calling methods on the target object directly Spring AOP requires developers to call methods on a dynamically provided proxy object. The latter shares the original objects interface and thus provides the same methods. Each of the proxy's methods eventually delegates the call to the respective method on the target object; however, before and after that call it can perform additional actions. Spring AOP uses that to inject code that was provided within advices and aspects.




(A) Plain object call (B) AOP proxy call

FIGURE 5.10: How AOP proxies work [Sprb, sec. 5.8.1]

In figure 5.11 we are using AOP to realize a *SecurityContext*-setup strategy. In this connection, we are introducing an aspect containing one advice, the *SecurityContextSetup* advice. The latter's task is to retrieve a pending message's authentication info from

the corresponding connection, set it up within the *SecurityContext* and delegate to the *MessageListener* for processing. After the listener completes, the *SecurityContextSetup* advice is furthermore responsible for clearing the *SecurityContext* again so that the next message is not accidentally processed with a wrong context. To make sure the advice is executed for all message types, a pointcut needs to be defined that covers the respective methods of all *MessageListener*'s as joinpoints.

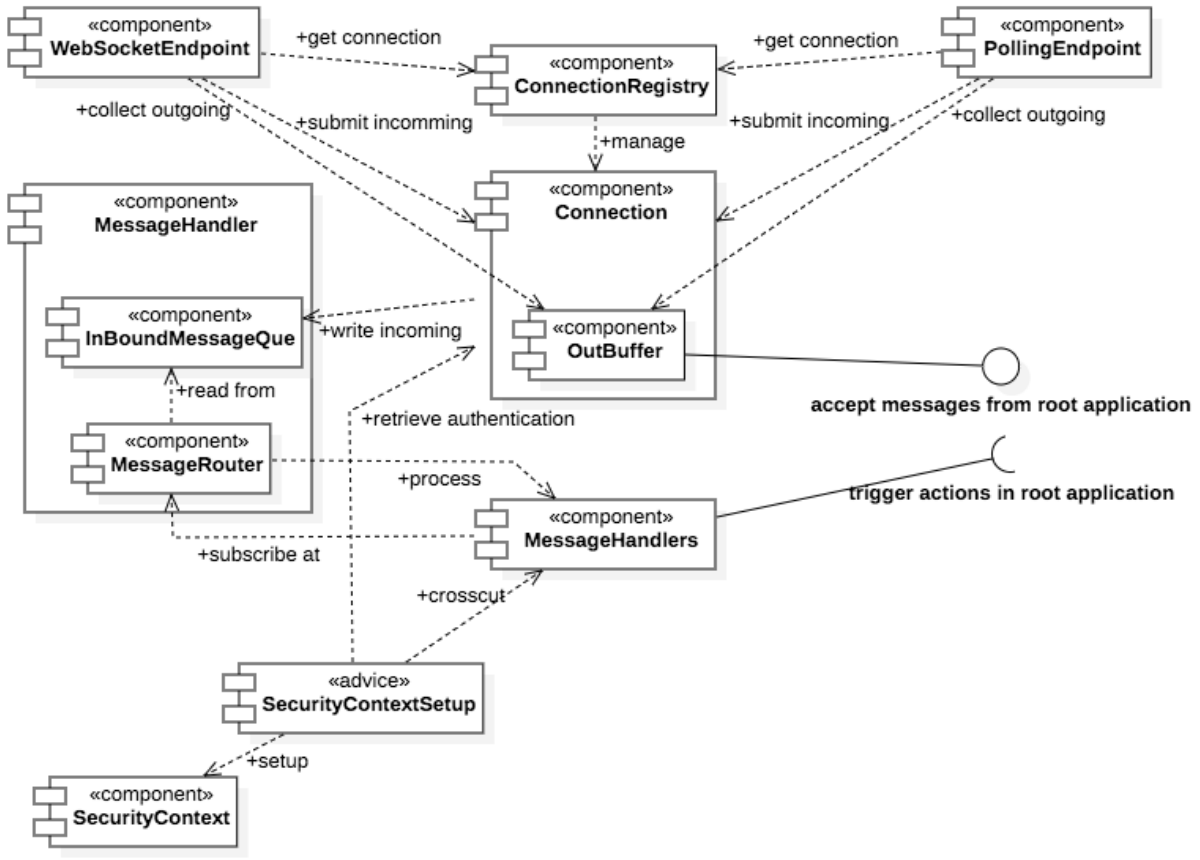



FIGURE 5.11: Messaging API with Authentication (UML Components) - Extension of Figure 5.5

Brief recapitulation: In the previous sections, we outlined the current architecture of the messaging API with a focus on the connection types it supports and their respective mechanisms for handling incoming and outgoing messages. We also looked at the application level reimplementation of the sliding window protocol by queues and

buffers and the thereto related decoupling of message processing from message sending and receiving. On top of the current architecture, we derived individual mechanisms for establishing authentication both at the polling and the WebSocket endpoints. We assigned that authentication to logical connection instances and finally worked out a strategy that sets up and clears the *SecurityContext* before and after a message is processed. The achievements made within this section are a prerequisite for the following considerations about making authorization decisions during message processing.

# Chapter 6

# Authorization

From the functional requirements introduced in section 4, this thesis so far covered everything associated with authentication. Authentication has been established at all of the system's APIs, and the previously derived *SecurityContext*-setup strategy makes sure that it is available at those parts of the system where access decisions shall be made. The following sections will derive concepts and architectures to fulfill those of the functional requirements that deal with authorization.

## 6.1 Designing the Authorization Concept

Taking a quick look again at the requirements from section 4, we can state the following as the most urgent demands: First, we need roles that can flexibly be assigned to any user. Second, the roles shall be assigned to users in one of two ways: Either in a global context, or in the context of specific meetings. Roles that are assigned to a user in the context of a meeting shall limit the involved privileges to exactly that meeting. Roles assigned without that context shall enable the user to execute the involved privileges for all meetings within the system.

The literature review in section 2.2 presents the major widespread authorization frameworks MAC, DAC, RBAC, and ABAC. Since MAC consists of hierarchical security level classifications and rigid information flow principles (read-down and write-up) it does not provide the required flexibility to reflect the aforesaid role architecture. DAC provides this flexibility, but comes with limitations in control and scalability. Let's think of DAC in terms of ACLs. An ACL is a list or table where each row grants access to one specific user for one specific object. The table's columns usually specify which operations the user is allowed to perform at the object. In the context of a meeting, possible operations for instance include *updateMeetingProperties*, *navigate*, etc. On the one hand,

this concept allows very granular permission management. On the other hand, however, it produces a lot of identical entries for systems where many users obtain equal permissions. Consider a facilitation-system where $5.000$ users are joining a meeting. If none of those users obtains any special privileges, it requires $5.000$ identical ACL entries to grant those users the required access permissions. Configuring those entries manually is a cumbersome and error-prone endeavor. Of course, configuration tasks can be automated in batch jobs; however, while the number of lines won't technically be a problem, setting up so many identical entries does not fit well with the idea of having conceptual roles such as participants, facilitators, etc. Moreover, whenever participant privileges change, all of those individual ACL records will have to be reedited. That's unnecessary overhead.

ABAC allows to define access control on the basis of arbitrary attributes from the subject, the object and the environment. It is especially suitable for implementing fine-grained access policies based on the relationship of a system's business data. An example ABAC policy could be along the following lines. Employees from a service department are granted access to their department's projects, but only during work hours and only from within the company network. Project managers, in contrast, gain access to their project at all times and also from outside the company network. In this scenario, employees are the subjects and projects the objects. Both have attributes such as the department they belong to. Project managers have an additional attribute for the project they manage. Network and time of access are the environmental attributes required for that particular policy. ABAC is perfect for modeling policies of that type. teambits:interactive, however, is usually not that deep integrated into company networks. It doesn't have information about the company's hierarchy model or the staffing of projects. As a consequence, the required attributes for modeling policies such as the above are not available within the system. Instead, the software often needs to be extremely dynamic in terms of granting access to a diverse set of users. For instance, freelancers, external facilitators and operators frequently require access to diverse parts of the system. Modeling attribute data and corresponding policies to grant those users the required access could easily become a complex task, even though the policy is comparatively straightforward. To grant the respective privileges to control a meeting, for example, we would need to extend the data structures of the meeting or the user to represent the corresponding relation between the two. The strength of ABAC lies in the modeling of advanced policies including entangled attribute combinations from subject, object, and environment. For the teambits software, however, it is rather desirable to simply add new users to the system and assign them one of the conceptual roles outlined within the functional requirements. Therefore, in spite of the mightiness introduced by ABAC, for teambits:interactive, it does not seem to constitute a suitable approach for access control management.

Instead, RBAC appears to be a promising alternative. It provides the right amount of flexibility and keeps the added complexity at a minimum. While implementing ABAC in teambits:interactive would require to extend the system's existing data structure, authorization with RBAC instead appends additional data structures that are independent from any business data. RBAC does not rely on attributes of the system's subjects and objects and therefore allows more dynamic management of access control rules. Moreover, separation of authorization and application data implies greater decoupling of the respective system parts, which – as mentioned before – is generally beneficial for security. The literature review introduced NIST RBAC as a standardization approach that covers the most widespread features of the access control concept. It includes entities for users, roles, and permissions (see figure 2.2). Permissions are associated with roles via so-called permission-role assignments (PAs), roles are assigned to users via user-role assignments (UAs). Furthermore, the standard also includes a session concept. Within a session, users activate a subset of their assigned roles. During the course of the session, execution is then limited to those permissions associated with active roles. Finally, NIST RBAC includes separation of duties (SoD) constraints for restricting UA and role activation. This can be used to avoid parallel assignment or activation of conflicting roles.

Using sessions and SoD constraints with RBAC adds an extra layer of control for administrators. While that is of severe importance for mission critical enterprise applications, it is not too helpful for a digital facilitation system that strives to cope with the user experience of modern internet applications. Being required to switch between roles in order to perform different actions adds unnecessary complexity. It would likely have a negative impact on the product's acceptance both by participants and facilitators and therefore shall be refrained within the teambits software. Moreover, constraints on role assignment and activation are not demanded by the at hand requirements. As a consequence, sessions will be disregarded within the subsequently derived RBAC model.

Permissions in NIST RBAC are assembled from operations and objects. In a facilitation system, an exemplary permission might consist of the operation *navigate* and the meeting *m0815*. Being assigned to a role, that permission enables users to trigger navigation events within the specific meeting *m0815*. It is a characteristic of this approach that individual roles have to be configured for every meeting. This causes a lot of manual configuration effort and is furthermore susceptible to the role explosion problem. In order to fulfill the defined requirements, we thus need an alternative to the NIST modeling. Parameterized roles, as developed in [GO04, p. 259f.] and introduced in section 2.2.2, seem promising in that regard. By adding parameters to permissions and roles, the concept allows to reuse the same permission and role definitions for all objects of the same type. Similar to ABAC, however, parameters in

[GO04, p. 259] take values from attributes of subjects and objects. Subject attribute values, for instance, are applied to parameters when activating a role in a session. As mentioned before, this solution is especially suitable for enterprise systems where all relevant information for access control is already present in the business data. For the teambits software, in contrast, it introduces some difficulties as illustrated in the following example. [GO04, p. 259] gives the subsequent scheme: A parameterized permission $(x|x\{a_1, a_2, ..., a_n\}, m)$ makes use of the parameters $a_1, a_2, ..., a_n$ to isolate a set of objects from type $x$ on that method $m$ shall be allowed to execute. A role can be referred to as $(name, permissionset, parameterset)$. The following is a concrete example: Take a role $(operator, [navigate, ...], [operatorId, ...])$. It contains a permission $(Meeting\{operatorId\}, navigate)$ that makes use of the parameter $operatorId$ to isolate the meetings for that the currently logged in user shall be allowed to navigate. The parameter $operatorId$ is currently not available in the meeting data structure and must consequently be added in order to setup the referred permission. Obviously, this represents a very inflexible solution for a digital facilitation software, as it requires modifications at the meeting data structure for almost every policy change. It fails to comply with the demand for dynamic roles that are configurable during runtime. Moreover, the concept does not adequately decouple application- and security data, which we previously noted as beneficial.

The subsequent concept represents a solution that copes with the aforementioned issues. Instead of supplying parameter values during role activation, we are now adding those values as part of UA. In fact, to cover the demand for a hierarchy mapping of roles, only a single parameter is necessary. *FR22* requests that a role can be assigned to a user in two ways: Globally or in the context of a meeting. If we add a parameter *meetingId* to UA, we can achieve exactly that. Whenever the parameter is set, the assignment counts for the specific meeting, otherwise it counts globally. Figure 6.1 depicts the corresponding data model for that solution.

The entities *User, Role* and *Permission* are taken from the NIST model (see figures 2.2 and 2.3). UA and PA are also from NIST; however, since both are many-to-many relationships, figure 6.1 explicitly models them as the intermediate entities they represent when the schema is implemented in a relational database. The main difference to NIST RBAC is that UA is now a tertiary relationship. It connects a user, a meeting and a role. A user can be assigned the same or different roles independently for individual meetings. The following outlines how a role can be setup in this scheme: The tuple *(p1, navigate)* represents our *navigate* permission known from the previous examples. *(r6, operator)* is a role and *(p1,r6)* is a PA assigning the *navigate* permission to the *operator* role. The UA *(u16,r6,m0815)* assigns the *operator* role to user *u16* within the context of the meeting *m0815*. UA *(u16,r6, )*, on the other hand, achieves the same assignment within the global context.

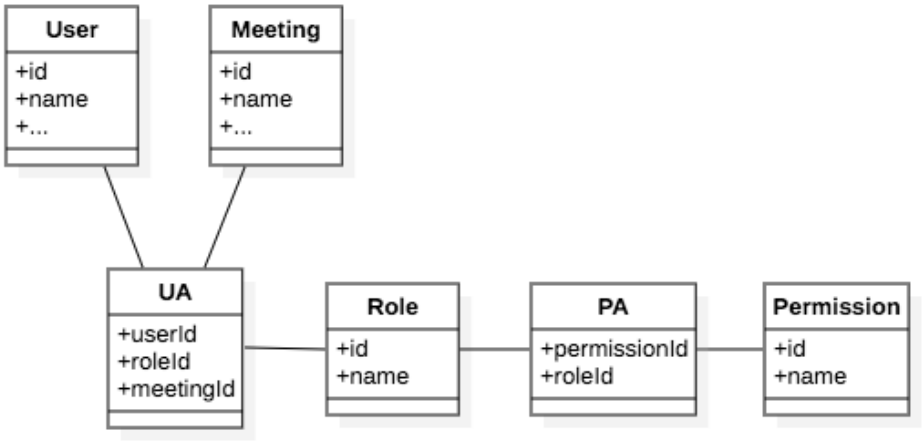



FIGURE 6.1: RBAC data model with parameterized UA

Other than within NIST, permissions in this new model are no longer specific to an object. In contrast to [GO04, p. 259f.], also roles are not accompanied by parameters. Only the assignment of a role to a user decides about the context in that the concerned permissions apply. As outlined in the previous example and requested by *FR22*, that context can currently either be global or local to a meeting. In the future, however, intermediate hierarchy levels, i.e. collections of meetings, can easily be added.

*FR21* requests dynamic configurability of roles and permissions. In our custom model, permissions are defined and associated with operations at implementation time. Anything besides permissions can be configured at runtime. Roles, PA and UA, for instance, are reflected in a dedicated RBAC data store. Users and meetings, in contrast, are part of the system's business data. They are referred from the RBAC data store; however, neither the user-, nor the meeting data structure need modifications in order to realize the schema outlined in figure 6.1. Consequently, the aspired decoupling between application and security-relevant data is guaranteed by this approach.

## 6.2 The Permission Evaluator

In the previous section, we derived a theoretical concept and a data model for access control in teambits:interactive or similar facilitation software. The subsequent sections will now work out how this concept can be implemented with Spring Security and integrated into the application.

Figure 6.2 depicts the architecture for making authorization decisions in Spring Security. In section 5.1 the *AuthenticationManager* was introduced as the central concept for authentication in Spring Security. The respective counterpart for authorization is the *AccessDecisionManager*. It is responsible for granting or denying access to a HTTP endpoint or a method invocation. Whenever access is denied, the manager throws an *AccessDeniedException* to be caught by the invoking environment. Similar to how authentication can be delegated to multiple *AuthenticationProviders*, access decisions can be derived by involving votes from multiple *AccessDecisionVoters*. *AffirmativeBased*, *ConsensusBased*, and *UnanimousBased* represent *AccessDecisionManagers* with distinct strategies for aggregating the votes they obtain from their voters. Two default voters in Spring Security are the *AuthenticatedVoter* and the *RoleVoter*. The former enforces a minimal authentication trust level for the requested resource (anonymous/unauthenticated, remember-me authenticated, or fully authenticated), the latter verifies whether the user holds all required roles. Both minimal authentication trust level as well as required roles can be configured individually for resources. However, roles in the context of the *RoleVoter* should not be confused with RBAC roles. *RoleVoter* roles are authorities that Spring Security can assign to users. They always have a global context and cannot dynamically be assembled from permissions. Instead, those roles are usually hard-coded in the security configuration to restrict access to specific application parts. For instance, a typical configuration using authority-based roles could express that access to the */admin/*** URIs should be exclusive to users holding the *ROLE_ADMIN* authority [Spre, sec. 11.1].

The authority-based role mechanism provided by Spring Security is not suitable to implement our RBAC concept. As always, a possible solution is the implementation of custom *AccessDecisionVoters* or even a custom *AccessDecisionManager*. However, the framework provides another useful extension to its authorization capabilities. Since version 3.0, it is possible to use Spring Expression Language (SpEL) in addition to configuration attributes and voters for configuring access control. SpEL can be used to define complex boolean expressions for making access decisions [Spre, sec. 11.3]. When setting it up, Spring Security adds an additional voter to the *AccessDecisionManager*. Its type depends on the invocation to be secured. *FilterInvocations* (HTTP requests) are secured by the *WebExpressionVoter*, *MethodInvocations*, in turn, are secured by the *PreInvocationAuthorizationAdviceVoter*. That differentiation is required, since the voter needs to work with the individual invoked secure object in order to extract the information required for evaluating the SpEL expression [Sprc].

Expression-based access control offers the same functionality as classical configuration attribute- and voter-based access control. For instance, SpEL provides expressions such as *isAnonymous()*, *isRememberMe()*, *isFullyAuthenticated()*, or *hasRole()* to cover the functionality traditionally provided by the *Authenticated-* and *RoleVoter*. On top of those,

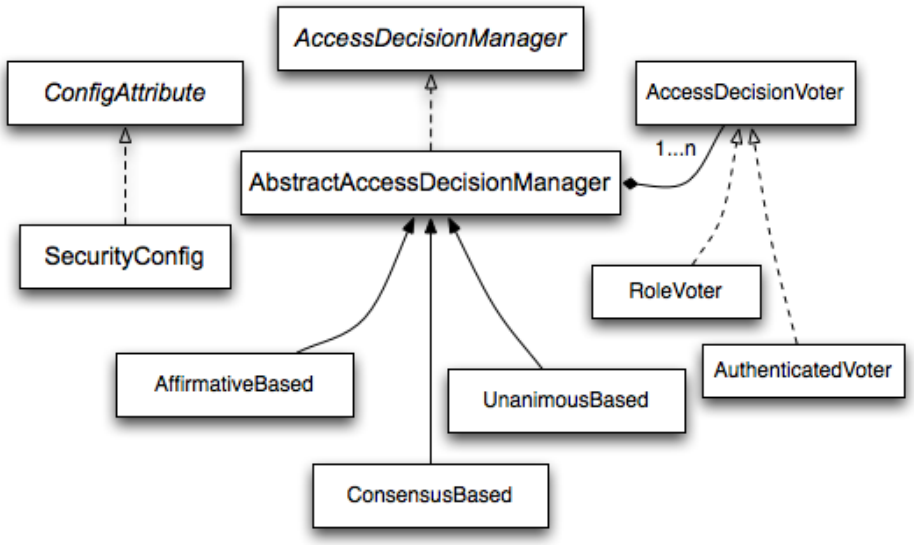



FIGURE 6.2: Spring Security AccessDecisionManager Architecture [Sprb, sec. 11.1.2]

however, SpEL also includes a new functionality that is only available at method invocations: The expression *hasPermission(Object target, Object permission)* can be used to check whether the authenticated user holds a specific permission in the context of the requested object. It is built for use with Spring Security's ACL system. To evaluate the *hasPermission(..)* expression, the respective *AccessDecisionVoter* makes use of an *AclPermissionEvaluator*. The latter is meant as a bridge between the expression system and the ACL system. It extracts all required information from the secured object and eventually queries a backend service – the *AclService* – for retrieving the corresponding ACL entries from a data store [Spre, sec. 11.3].

While the permission system was build for ACL, it has no explicit dependencies on it. In fact, the ACL system can be swapped out for an alternative implementation by replacing the *AclPermissionEvaluator* with a custom evaluator that bridges to another underlying module [Spre, sec. 11.3.3]. This architecture seems just about the right place for attaching our composed RBAC concept. For a start, we define a *RbacPermissionEvaluator*. It will, for instance, extract relevant information, such as the concerned meeting, from incoming messages and evaluate the authenticated user's permissions by querying an underlying RBAC service. Analog to the *AclService*, we introduce the *RbacManager*. The term *manager* indicates, corresponding to regular Spring terminology, that the service supports both reading from as well as updating the respective

RBAC data store. The store holds the RBAC schema depicted in figure 6.1. It can be realized as part of the system's existing relational database and connected via JDBC.

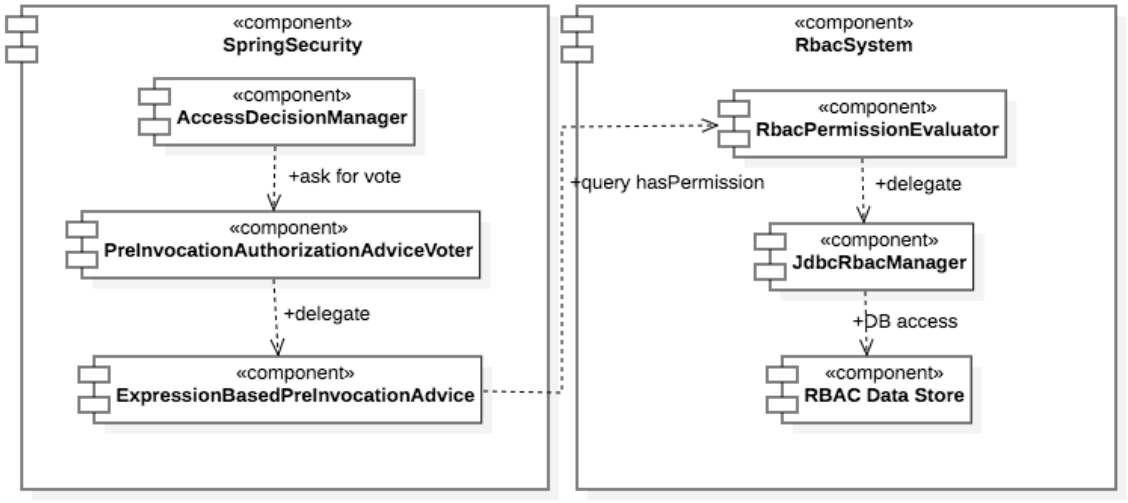



FIGURE 6.3: RBAC *PermissionEvaluator*

Figure 6.3 shows how the RBAC module eventually integrates into Spring Security. To derive its decision, the *RbacPermissionEvaluator* queries the *RbacManager* in order to find out whether the authenticated user holds the required permission for the given meeting. Authentication details are available in the *SecurityContext* - thanks to the previously developed setup strategy - and will be auto-injected into the *PermissionEvaluator* by Spring Security. Since permissions obtained via global roles propagate down to every meeting, the evaluator has to check both global and meeting-specific UAs. Based on the results it obtains from the *RbacManager*, the evaluator will then return true or false for whether the user has or does not have the requested permission for the at hand meeting. The *PermissionEvaluator*'s response, however, is not to be confused with the final access decision. It is returned to the calling expression handler - i.e. the *ExpressionBasedPreInvocationAdvice* - which uses it as part of the overall SpEL expression. The boolean result of the latter is eventually passed to the respective *AccessDecisionVoter* and returned to the *AccessDecisionManager* which derives the final access decision based on its strategy and, if necessary, throws an *AccessDeniedException*.

## 6.3 Integration into the messaging API

As mentioned several times in the previous sections, Spring Security includes two main mechanisms for applying authorization to application parts: HTTP security and method security. While the former's responsibility is to secure all kinds of HTTP endpoints, such as web applications, REST APIs or file downloads, the latter is mainly meant to apply additional security to business objects on the service level. Notwithstanding the above, method security can also be applied on API level. It is thus perfectly suited to enforce authorization during message processing at the messaging API.

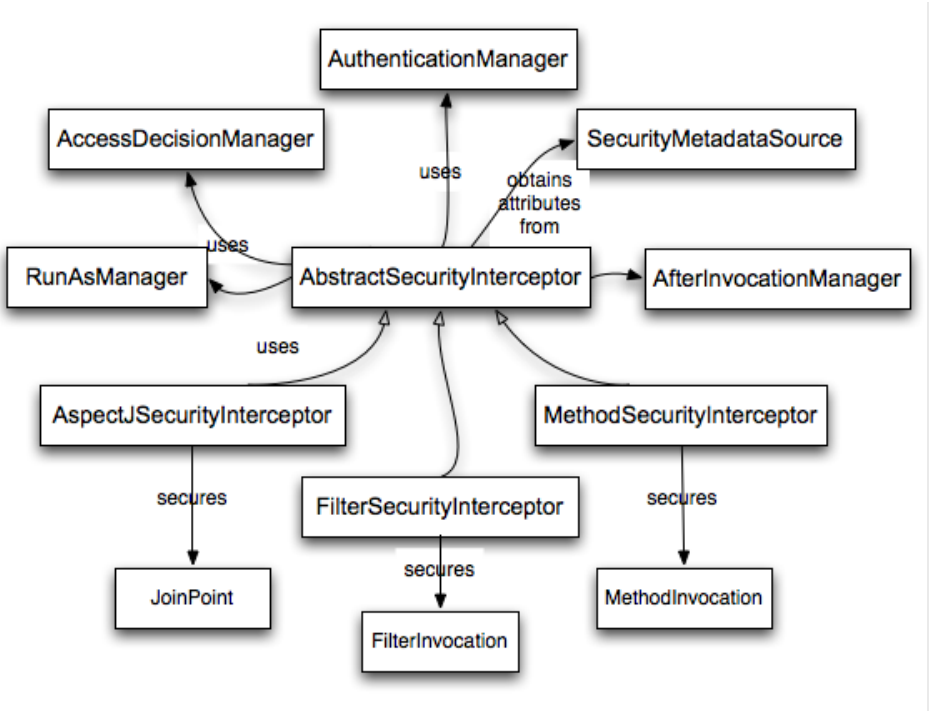



FIGURE 6.4: Spring Security – Security Interception [Spre, sec. 8.1.5]

Figure 6.4 outlines how Spring Security applies authorization via so-called *SecurityInterceptors*. *FilterSecurityInterceptor* and *MethodSecurityInterceptor* are the two types of interceptors responsible for securing *FilterInvocations* (HTTP endpoints) and *MethodInvocations*. The following are the general tasks of a *SecurityInterceptor*, independent of its concrete type. When intercepting the invocation of a secure object, it is responsible for obtaining the authentication details of the currently active user from the *SecurityContext*. It then looks up information related to the secure object, such as a list of *ConfigAttributes* required by the *AccessDecisionManager*, from the *SecurityMetadataSource*.

*ConfigAttributes*, for instance, include the required roles and authentication trust level (i.e. anonymous, remember-me or fully authenticated). When all required information is available, the *SecurityInterceptor* uses the configured *AccessDecisionManager* to authorize the invocation request. If access is granted by the latter, control is passed back to the concrete subclass to proceed with the execution of the secure object. On the other hand, if the manager throws an *AccessDeniedException*, the secure object will not be executed and instead, the exception travels up the stack of callers until being caught. [Sprc]. When invoked from an HTTP environment, exception handling can be left to Spring Security. Position 8 of the *springSecurityFilterChain* (see table 5.1) contains the *ExceptionTranslationFilter*, which converts any Spring Security exception into an adequate HTTP error response. Invocations from outside an HTTP environment, in contrast, require custom handling of *AccessDeniedExceptions*.

To authorize messages during processing at the messaging API, we will add method security to the individual *MessageListeners*. Thanks to the previously worked out RBAC integration, we can then use the *hasPermission(..)* expression to express the permission, a user requires for sending the at hand message. The configured *AccessDecisionManager* validates the expression and checks permission using our custom RBAC module. If the requested permission is associated with the authenticated user, access is granted and the message will be processed normally. If, in contrast, the user does not hold the permission, the message will be discarded and we need to handle the corresponding *AccessDeniedException*.

Similar to setting up the *SecurityContext*, handling this authorization-related exception is again a crosscutting concern. It consequently appears appropriate to make use of AOP again and extend the previously created aspect by an advice that takes care of the exception handling. Obviously, this advice has a lot in common with the one for setting up the *SecurityContext*. Since both share the same scope we can reuse the previously defined pointcut to cover the respective methods of all MessageListener's as joinpoints. The advice's responsibility is then to catch *AccessDeniedExceptions* thrown as part of the method invocation process and take appropriate response actions. For an incoming message that is not processed due to access right restrictions, it seems to ben an adequate measure to reply with a message that informs the sender about the negative access decision. In contrast to the REST API, communication at the messaging API is asynchronous. Consequently, the response message needs to contain some information that allows the sender to associate it with the original message and the requested action. Also, a reason for denial, such as the missing permission, could be part of the response. In order to supply all this information in a format that is easily processable by clients, we introduce a new message type to the application, the *AccessDeniedMessage*. Figure 6.5 extends the previously introduced component diagram to depict the new structure of the messaging API. It depicts the two custom advices

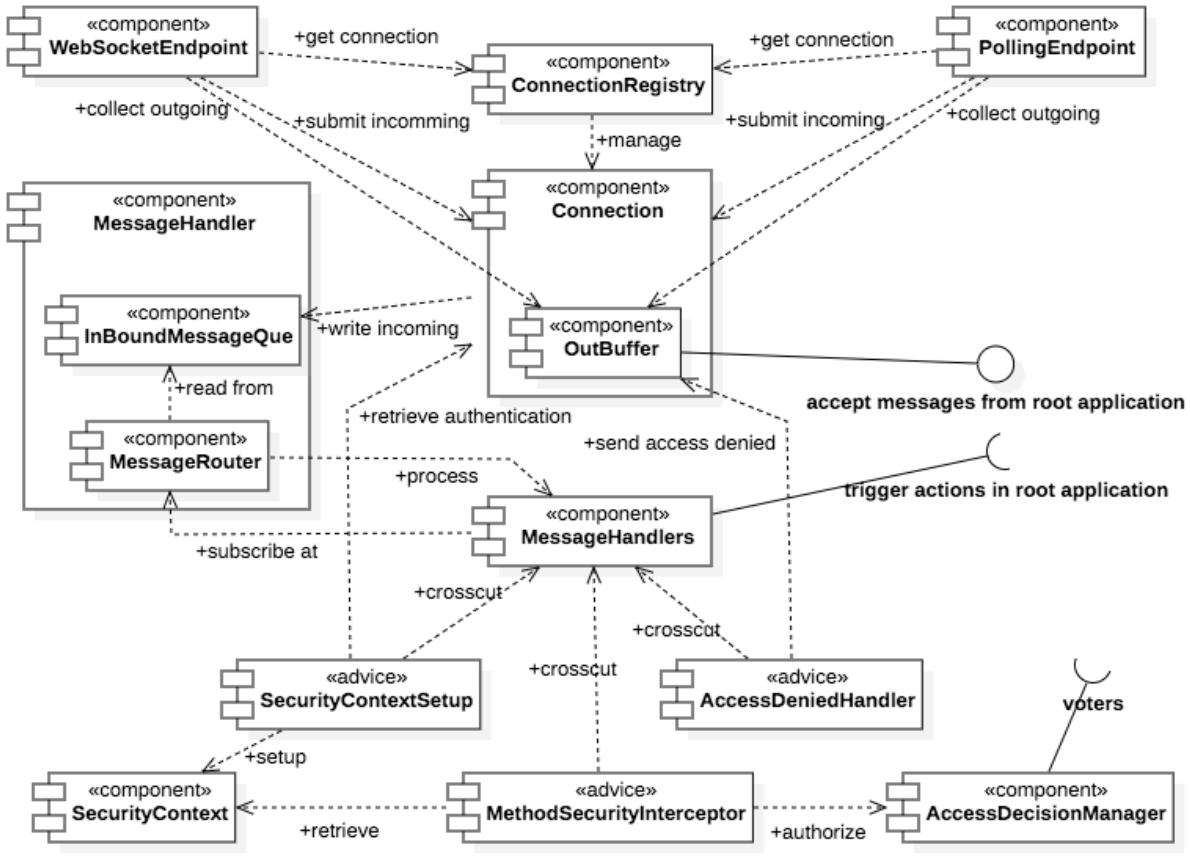



FIGURE 6.5: Messaging API with Authentication & Authorization (UML Components) - Extension of Figures 5.5 and 5.11

*SecurityContextSetup* and *AccessDeniedHandler* as well as the *MethodSecurityInterceptor* which Spring Security also realizes via its internal AOP infrastructure.

## 6.4 Integration into the REST API

Now that authorization is established at the messaging API, the following section focuses on adding access control to the REST API. As outlined before, Spring Security offers HTTP security and method security as the two main mechanisms for applying access control to different application parts. Usually, Spring-based applications are making use of HTTP security for securing web-based endpoints such as APIs, web

apps and static resource downloads. However, also method security can be used to apply access control at HTTP controllers. From the applying developer's perspective, the only difference is that with HTTP security, access rules are maintained in a centralized security config, while method security allows to annotate those rules directly at individual controller methods. A more technical perspective, in contrast, reveals that both solutions differ in the type of *SecurityInterceptor* that Spring Security applies behind the scenes. HTTP security makes use of the *FilterSecurityInterceptor* (see pos. 9 at table 5.1), a Java servlet filter within the *springSecurityFilterChain*. Method security, in turn, relies on the *MethodSecurityInterceptor*, introduced in the previous section as part of Spring Security's AOP infrastructure [Sprc].

While both interception mechanisms are backed by the same underlying *AccessDecisionManager* architecture, they still differ in some features. For instance, even though both mechanisms support the use of SpEL in their configuration, only method security can evaluate *hasPermission(..)* expressions [Spre, sec 11.3.3]. Consequently, since our derived RBAC system is built on that expression, we are obliged to use method security, also for securing the application's HTTP endpoints.

By this, the authorization process itself is comparatively similar between the REST- and the messaging API. It is the *MethodSecurityInterceptor*'s responsibility to retrieve the authentication info as well as related configuration attributes from the *SecurityContext* and the *SecurityMetadataSource* and pass both to the configured *AccessDecisionManager* for authorization. As outlined in the previous section about the messaging API, the latter makes use of the respective *AccessDecisionVoter* to parse the SpEL expression and handle it via our custom RBAC module. If access is granted by the manager, the controller method is executed regularly. In case of an unauthorized request, in contrast, the manager will throw an *AccessDeniedException* before the method even executes. Differences to the messaging API only arise from the fact that communication at the REST API happens synchronously. This means that dispatching and processing requests as well as sending corresponding responses is, other than at the messaging API, not decoupled via queues and buffers. Instead, the entire request-response lifecycle is processed in one coherent thread, managed by the Spring framework. As a consequence, we can this time not only make use of Spring Security's generic interception mechanism, but also leave the *SecurityContext* setup as well as the handling of *AccessDeniedExceptions* to the framework. Section 5.3 explained how authentication in a servlet environment works. Spring Security sets up the *SecurityContext* by one of two filters from the *springSecurityFilterChain*: The *SecurityContextPersistenceFilter* (pos. 2 at table 5.1) or one of multiple available *AuthenticationProcessingFilters* (pos. 4 at table 5.1). The exact interrelation of those components is outlined in figure 5.4. In the context of our OAuth2 protected REST API, the respective filter is of type *OAuth2AuthenticationProcessingFilter*. In case a user is authenticated, but access

is denied by the *AccessDecisionManager*, the corresponding *AccessDeniedException* will be caught by the *ExceptionTranslationFilter* (pos. 8 at table 5.1) and automatically converted into an HTTP unauthorized response.

# Chapter 7

# Evaluation

The domain of digital live facilitation has several specific security aspects. Challenges especially arise during the transformation from a pure offline business towards online and cloud environments. The trade-off between security and usability differs between environments. As participant involvement is key to success for digital facilitation, usability is often seen as the top concern in offline environments. Within online environments, however, it is mandatory that security keeps up with current standards. Software for digital facilitation has to provide real-time communication. At the same time, it has to cope with the challenge of unreliable and overloaded non-stationary network setups. When custom techniques are implemented to compensate this lack of reliability, such as the sliding window protocol outlined in section 5.4, those techniques require additional security considerations.

## 7.1 Reflection & Discussion

Within this thesis, we performed a case study of a specific software, teambits:interactive, during which we looked at the software's architecture, the requirements it faces, as well as concepts that can make up potential solutions. Chapter 2 did a literature review to gain insights about available techniques and best practices that can be used to cope with the mentioned challenges. It introduced and analyzed different options for authenticating users at HTTP and WebSockets. While for HTTP, a vast amount of solutions and standardized frameworks exists (see cookie-based sessions, tokens and OAuth2), WebSockets do not yet offer a common approach towards authenticating users. Instead, rather proprietary solutions were developed over the years and eventually lead to best practices, such as the discussed approaches on tickets and personalized one-time URLs.

Apart from authentication, chapter 2 also analyzed different access control concepts. Specifically, we looked at the concepts MAC, DAC, RBAC, and ABAC. Being introduced in the late 1960's MAC and DAC have been two of the very first generic access control systems available. It was found that MAC, as it is targeted to military and governmental environments, is rigid and mainly specialized to enforce strict information flow principles. It binds access control to information categories and externalizes configuration of the latter to administrative departments. Thereby, it keeps control away from those people that are actually working with the information in the system. DAC, on the other hand, makes access control configuration a discretionary decision of the information owners. Instead of categories or classifications, DAC considers pieces of information individually and allows their owners to configure access privileges for other users. Quite contrary to MAC, administrators in DAC are usually not given any means to restrict the propagation of privileges and therefore largely lack comprehensive control over information flow.

The analysis of RBAC revealed that it compensates the missing administrative capabilities of DAC by incorporating so-called SoD constraints that administrators can utilize to restrict the assignment of access rights. The NIST standard was introduced, which incorporates four basic manifestations of RBAC. It was also discovered that a lot of extensions have been developed for RBAC to reflect more advanced access control scenarios. With object-classes and parameterized roles, we introduced two examples therefore.

Finally, ABAC was outlined as an access control system that has been designed to cover advanced scenarios by default. We depicted how it applies attributes of subjects, objects and the environment to define and enforce arbitrary complex policies. ABAC shines through its great flexibility and its independence of extensions. It also became clear, however, that the system comes with certain complexities. Challenges of modeling attributes, defining new policies and maintaining existing ones make it hard for non-administrative users to make effective use of the system and are hence rendering it inappropriate for dynamic self-service applications.

Chapter 4 built on the domain knowledge introduced in the beginning of this thesis and performed an extensive analysis of needs and demands that are important during the pursued transformation from a highly technical offline system towards an easy-to-use, cloud-ready self-service application. Interviews have been performed with carefully picked staff members of teambits to gain insights about the divergent demands that exist throughout the sales, services and development departments. The results of those interviews and the gathered domain knowledge were then used to elaborate a set of requirements that forms the basis for the transformation that the company is striving at.

Based on the insights gained in the literature review, chapters 5 and 6 designed a solution that brings advanced authentication and authorization capabilities to the individual system APIs. More precisely, an *AuthenticationManager* configuration was worked out that allows authentication of principals against two distinct user data sources: One internal staff user data source and one external participant user data source (e.g. spreadsheets, participant management-, or company specific systems). Subsequently, a login mechanism was derived that provides a form-based login process for users to authenticate against the provisioned *AuthenticationManager*. To secure requests against the REST API as well as the polling part of the messaging API, we designed a solution based on the OAuth2 authorization framework. For WebSocket connections, in turn, we developed an authentication solution based on one-time tokens. Together those solutions fulfill both functional requirement FR10 – enforcing authentication at all of the systems public interfaces – and FR11 – providing access to all parts of the system via one central user account. Beyond the defined requirements, the elaborated configuration also supports SSO at all of the system's distinct client applications.

FR12 requests authentication to be optional. This is a feature that Spring Security provides by default. When no credentials are supplied, the framework establishes a so-called *AnonymousAuthentication*. It is then to be configured within the authorization system, which parts of the system should be accessible to unauthenticated users and which not.

Finally, a capability for authenticating users as part of a group, instead of authenticating them as individuals, is requested in FR13. This requirement is only partly met by the designed solution as it requires a work around. In order to authenticate as participants or facilitators, one would create an account *participants* or *facilitators* and supply it to the group of users that shall be authenticated as such. Technically, this fulfills the requirement. However, the solution is not optimal as it causes the system to treat the individual users of a group as one indistinguishable entity. Editing properties of a user, for instance, is not possible with this solution, as it would edit them for all users of the group. Apart from that, it might seem counter intuitive for people to sign in with a username, when one is actually authenticating as part of a group. A solution for that could be to allow authentication via passcodes. A passcode could identify an individual as part of the desired group, create a temporary, anonymous user and automatically assign it the corresponding roles. This would come with several advantages compared to the described work around. For instance, it would allow all users, although anonymous, to be treated as individuals in the system. That, in turn, makes it possible to edit properties of individual users, or even upgrade them from anonymous to authenticated individuals. Alongside those advantages, however, the hinted approach would also add additional complexity. In favor of better assertiveness of the

overall solution, it was thus renounced as part of this iteration and postponed to future security revisions.

In chapter 6, a concept for establishing efficient access control was designed and integrated in both of the application's APIs. Building on the insights gained in the literature review, the designed solution for configuring and deriving access control policies is based on the NIST RBAC model. Although ABAC is a likewise promising concept, it implies too many additional complexities, especially in terms of attribute engineering, and hence appeared less appropriate for the at-hand scenario. RBAC with its dynamic roles, instead, fits to the demands expressed within the requirements. It supports predefined roles, as requested in FR20, as well as dynamic roles that are defined at runtime, see FR21. On the other hand, as none of the analyzed RBAC mechanisms supports mapping roles to different hierarchy mappings, see FR22, we derived a custom enhancement that extends the user-role assignment (UA) by a context parameter. Currently, this parameter allows an assignment to be made either in a global context or in the context of a meeting. Eventually, the derived model was incorporated into the existing application. Therefore, an *AccessDecisionManager*, *PermissionEvaluator*, as well as other components specific to the Spring framework, were implemented, configured and integrated into both of the system's APIs.

In summary, the derived solution covered all of the elaborated requirements. Alongside our case study of the teambits software, the objective of this thesis, however, also was to develop a conceptual solution that is, at least in parts, transferrable to other systems in the domain of digital live facilitation. Therefore, the following section will perform an analysis of the general validity and applicability of the derived concepts. It aims to determine how and to which extent those concepts and the elaborated knowledge can be abstracted from the teambits software and transferred to other facilitation systems.

## 7.2 Transferability

Naturally, other companies in the domain – see for example the competitors, SpotMe, Sli.Do, and Mentimeter [Spo; Sli; Men], as introduced in chapter 1 – are not disclosing a lot of information about the internal structure of their systems. Nevertheless, the challenges they are facing are likely very similar to those encountered during the analysis of teambits:interactive. Hence, it is expected that a lot of the derived solutions are in fact transferable between those systems.

We are starting this analysis by looking at the usage of OAuth2. As outlined in the literature review, OAuth2 is a framework that is widely used to offer social media logins at online services, mostly together with OpenID Connect (see section 2.1.1). While in those cases, applications are delegating authentication of their users to external service providers, the concept is also suitable for achieving SSO inside the barriers of a distributed or multi-client platform. In this case, the platform itself has to include a component that acts as a service provider, see the authorization service in the authentication concept derived in section 5.2.

Delegated authentication, i.e. the concept of performing authentication at a central place in order to issue pre-authenticated tokens to other parts of the system, is a universally applicable practice to improve both security and usability. It increases modularity and abstracts diverse and possibly complex authentication techniques (username/-password credentials, magic links, etc.) away from individual application parts. As this significantly reduces the required modifications at those individual applications parts, delegated authentication can also be seen as a generic approach for the non-invasive addition of new authentication methods to existing systems.

Apart from its utilization for first-party clients, OAuth2 is also beneficial to control API access through external, trusted applications. That may be interesting in case a platform shall be opened to third-party developers so that they can implement alternative client applications. Those aspects are not specific to teambits:interactive. Rather, they are applicable to any platform that is structured into multiple application components and/or frontends and could hence likely be transferred to competitors to a great extent.

A bit more specific than the OAuth2 configuration outlined in section 5.3 are the concepts derived around the messaging API, as introduced in section 5.4. The solutions that teambits developed for that API are, amongst others, addressing the following two main challenges: Firstly, missing reliability of network environments on event venues. This is countered by teambits through their queue and buffering concepts. Secondly, the blocking of WebSocket connections in certain company or public networks, bypassed by teambits through their fallback to a polling mechanism.

Both of those concepts require certain considerations in terms of security. While we cannot presume that competitors of teambits are building on similar solutions, it should be expected with high certainty that they are facing the same challenges around unreliable networks and blocked protocols. Unverified customer reports about connection problems that are frequently arising during the application of competitor systems, lead to the assumption that at least some of the alternative providers for digital facilitation are not implementing respective counterparts to the presented teambits mechanisms. From an economic perspective, those circumstances are making the developed concepts constitute a competitive advantage for teambits that is worth protecting. From

a scientific point of view, on the other hand, the situation will actually lead to an increased interest about the derived solution's transferability. This includes how certain concepts, as well as their security relevant extensions derived within this thesis, can be abstracted and applied to other facilitation platforms with different internal architectures.

As information during a facilitation process frequently needs to be delivered to participants in a server-initiated fashion, the requirement for a two-way communication between server and client lies in the nature of digital live facilitation. It is thus presumable that also the software of competitors relies on WebSocket connections and/or some sort of polling mechanism. Independent from exact manifestations of their communication design, it is likely possible to transfer parts of the developed concepts to other solutions. For instance, the idea of one-time tokens being requested prior to a connection request is broadly independent of other application parts or the applied frameworks and libraries. So it is with the concept of logical connections that authentication information is assigned to. Differences however exist at mechanisms such as the setup of the *SecurityContext*. Those concepts are still transferable; however, they are targeted at the idea of processing messages in isolated message processing threads, rather than in a framework-managed connection thread pool. Moreover, the solution is obviously specific to Spring Security and will differ more or less when transferred to other frameworks.

On the other hand, the use of AOP during the implementation of security aspects stands out through universal applicability. While it is true that the message processing architecture of teambits:interactive is specific to the company, the integration of crosscutting concerns such as the *SecurityContext*-setup strategy via advices around the affected methods is a very generic technique. Similar as with the above discussed concept of delegated authentication, using AOP for the integration of security concerns leads to minimal- or even non-invasive solutions, as it requires little to no modification of existing code. Consequently, even if the application architecture of other systems varies, the approach used to derive the strategy around message processing (see section 5.4.4) will be transferable to other facilitation systems, most likely even to systems of other domains.

Let's eventually look at the derived access control concepts. From competitors' documentation, it very often does not become clear how they are authorizing user actions. However, it appears that most of the succeeding cloud platforms for digital facilitation are providing some sort of team feature. Sli.Do, for instance, allows to add users or guests to an account that are then allowed to support each other in the management of meetings [Sli]. Also Mentimeter offers organizational accounts that enable the collaboration of members at interaction design [Men]. Which access control concept fits

best for other platforms heavily depends on the targeted use-cases. The RBAC variant developed within this thesis focuses – pursuant to the at-hand requirements – on maximum freedom in the assignment of access rights for very fine-grained parts of the system. The granularity achieved through dynamic roles and context-based role assignment might be overwhelming for users and therefore undesirable in platforms that aim at fast expanding bulk selling.

Platforms such as SpotMe, on the other hand, are aiming to support more custom solutions and reflect very specific requirements of their various client industries [Spo]. For those, the derived RBAC system might actually represent an attractive solution. The contained hierarchy mapping comes with significant flexibilities. For instance, different hierarchy levels could be added for teams, departments, business divisions and whole organizations. Hence, in terms of the bare concept, our RBAC model appears to be highly suitable for a transfer to other systems. When it comes to the concrete application components, however, the solution is again specific to the Spring Security framework. That includes, for instance, the custom *RbacPermissionEvaluator* as well as the *RbacManager*.

In summary, this analysis revealed that significant parts of the concepts and implementation developed within this thesis are transferable to other products. The extent in which such a transfer would be beneficial for the target products heavily depends on the use-cases those products are aiming to support. Other parts of the developed solution, however, are very specific to teambits' architecture. They can and should not be transferred in order to maintain the company's competitive advantages.

# Chapter 8

# Conclusion

Pursuant to the objectives of this thesis, we researched available methods for authentication at the HTTP and WebSocket protocol. It was found that a variety of scientifically founded authentication mechanisms are available for HTTP reaching from cookie-based sessions over HTTP Basic and Digest up to more complex frameworks such as OAuth2. For WebSockets, on the other hand, the topic of authentication did not gather a lot of attention in the scientific community. Instead, we discovered a ticket- and a one-time URL-based approach that have evolved as best practices over time. The situation is different, when it comes to authorization mechanisms. Several well-established concepts such as MAC, DAC, RBAC and ABAC have been analyzed for their characteristics, their advantages and disadvantages as well as how they fit into different application environments.

Another objective was to identify concrete demands for the enhancement of teambits: interactive's security infrastructure. Interviews with company members have been performed to analyze the needs of teambits and define a set of requirements to focus on. On the basis of those requirements, we worked out a solution that provides comprehensive authentication and authorization mechanisms over all of the software's interfaces. It makes use of OAuth2 for HTTP authentication, a custom one-time URL mechanism for WebSocket authentication and a variant of RBAC for overall access control. The designed architecture is based on the insights gained during the literature review.

Eventually, the elaborated solution was examined for its general validity and transferability to other systems. Only a few of the worked out concepts were found to be specific to teambits, following from the fact that the underlying communication mechanisms are proprietary developments. Most of the worked out concepts, in contrast, turned out to be transferable to a great extent.

## 8.1 Limitations

The developed solution does not include an in-depth analysis for all individual aspects of the combined authentication and authorization mechanisms. This is out of the allocated time for the performed research, as the primary scope of this thesis was to assemble a comprehensive approach to security in digital facilitation, rather than to perform an exhaustive discussion on smaller architectural and implementation details.

In the future, it will likely not be enough to assign roles and privileges on a per meeting basis. Indeed, the worked out RBAC model can easily be extended by multiple hierarchy levels; apart from that, however, it might become relevant to define more fine-grained access policies that cannot be depicted with the current model. A plausible scenario could be the introduction of affiliation-based access control, i.e. policies that allow participants to edit and delete their own submissions, such as contributions, questions, etc. When such scenarios become a requirement, further enhancements of the developed access control model are required.

## 8.2 Recommendations

teambits should evaluate the developed solution and, in case the company's strategic goals have changed, revalidate it against their current requirements. The derived authentication concepts are perceived as the most urgent goal in terms of implementation; however, also the access control model will add value to the software and increase the variety of its possible applications. Setting up comprehensive access control is the first step towards making the software ready for multi-tenant cloud offerings. It will also be necessary to deploy long-running instances of the software within company networks where information shall be kept private to individual departments.

## 8.3 Further Work

WebSocket authentication is a field that would highly benefit from further academic research. The protocol should be accompanied with more precise considerations about the establishment of trust between client and server. That research might in large parts be based on the model of HTTP authentication, where both simple mechanisms as well as advanced frameworks are available. It might also include the already available best practices that have been analyzed and used within this thesis.

Apart from that, also the topic of access control in multi-tenant cloud applications could benefit from further investigation. While this should be highly relevant for all providers that are running SaaS applications in the cloud, there is also comparative very little scientific literature available on the topic. Presumably, a lot of vendor-specific systems have been developed both by the SaaS providers themselves as well as by infrastructure providers. In order to make access control in multi-tenant applications more transparent and avoid future problems such as vendor lock-ins, more openly accessible research should be done in this field.

Both of the foregoing suggestions are generic to the topic of authentication and authorization. The following is more specific to the performed case study. In the evaluation chapter, we suggested an alternative for the requirement of group-based authentication that was renounced in favor of a more assertive solution. Future iterations and security revisions should revalidate the requirement and elaborate a more sophisticated solution for this demand.

Finally, as hinted in the previous section about limitations, the elaborated access control system from our case study will need enhancements to support more fine-grained policies, specifically to establish affiliation-based access control. In doing so, one might either extend the derived RBAC variation or combine it with another access control model so that also access policies can be implemented that allow users to manipulate or delete data of that they are the creator.

# Appendix A

# Transcript of Staff Interviews

Although the interview questions have been drafted in English language, the interviews themselves were conducted in German. Answers provided in this transcript were translated from the notes and audio recordings captured during the interviews. Translation happened to the best of the author's knowledge and belief.

Furthermore, it should be noted that some of the answers given by the interviewees state wrong or misrepresented information. Also some answers contradict each other and some of the mentioned expressions represent internal terms or names of specific parts of the software. The reader does not need those terms in order to follow the discussion. The interview results are interpreted in chapter 4 of this thesis. Relevant expressions and tool names are explained and covered in thematically corresponding chapters.

## Understanding the current system state

### From your perspective, how does authentication and authorization currently work for staff (administrators, operators, etc.)? (How are users configured, how are access rights configured?)

**Interviewee 1** What the system provides is actually not an authentication mechanism, but rather a password protection. You can set a participant password and a facilitator password (the latter is optional). You can also set an auth code in the users.xls file. There is an admin password for the web interface. Reseller logs and databases can be encrypted. The system does not provide authorization (no management of privileges possible). Restrictions in terms of allowed actions are

enforced by requiring users to authenticate at different interfaces (different views for different roles).

**Interviewee 2** There is an admin user for the web frontend. It secures access to protected resources. For other parts of the system, we have a separate facilitator password.

**Interviewee 3** There are global passwords per role/ group of users. The concept of individual users is not implemented in the system.

**Interviewee 4** There is username/ password protection for the web interface. However, only one default password is usually used. It rarely varies between individual events. Authorization happens via the user. A user is not individually identified; instead, group-users are implemented.

**Interviewee 5** Operators get access by a password. The operator can configure passwords for participants and facilitators. No authentication of individual staff users is possible since all are sharing one password.

## From your perspective, how does authentication and authorization currently work for participants? (How are users configured, how are access rights configured?)

**I1** We can define participants in the users.xls file or via BES scanners. We can also define a participant password. Authorization is not actually possible in the system, but can partly be achieved via hacking.

**I2** Participants have a participant-password. There is a token-mechanism for event-material download, but this only works with the old IOS app.

**I3** Usually, there is no authentication and authorization for participants. Global passwords for WebSocket connections are possible, but content such as menu entries and agenda is available before the password request. There is a pseudo-personalization mechanism (e.g. via a username dropdown or a PIN in the participant list). Authentication may also be realized by event IDs that have to be entered into a portal/ app.

**I4** Authorization happens via user properties or via tool phases in the meeting script. Authentication may happen via the selection of a username in a drop down. Otherwise, users remain anonymous. Authentication may alternatively happen via a group assignment, via dropdown or via a pre-defined assignment. Authentication may also happen when a participant enters his name or via staff that hands

out rental devices with pre-established authentication. Finally, authentication may as well happen via random user assignment.

**I5** Participants can be assigned individual PIN codes. The operator can configure passwords for participants. Authentication and authorization can happen via the event ID in the app.

## Where in the system (which interfaces) does authentication and authorization currently apply, where not?

**I1** APIs are not secured; you can, for instance, access them via developer tools.

**I2** Facilitator and participant passwords are requested with the first message. Some HTTP endpoints are unsecured. The server license password is requested on server start.

**I3** The web interface includes authentication via admin credentials, web app clients are using passwords (facilitator or participant password). Also the meeting cockpit is protected by a password. Protocol exports are secured by the credentials from the admin interface. No authentication is implemented at the client ZIP download [an archive containing event-specific content such as stylesheets and the agenda]. WebSockets have no direct authentication as well. Indeed, a password is transmitted on request; however, in the first place, the WebSocket can be opened without specifying credentials. Same is true for the alternative polling connection.

**I4** The meeting cockpit is not secured, it can be opened without a password. The WebUI is secured via a username and password. I am not sure how result queries are secured. The feedback view in the Meeting Cockpit is also not secured. Finally, the server startup can be protected via a simple password, no username.

**I5** Client and web interface are secured.

## Please name things that you think are good and things that are bad about the current authentication and authorization mechanism?

**I1** The system is easily hackable to fulfill specific customer wishes. This is good. Bad is that it might not in all parts be sufficiently secured. Likewise, it is a drawback that authentication of individual participants is not possible.

**I2** Good is that the code for security is not massive. The control flow is easy and comprehensible. Rather negative is that the system does not entirely rule out access to unauthenticated participants. There is no traceability of access, which is also not good. The distinction between participant and staff members and the restriction of user actions through different station types is not fine-grained enough. The user service is only secured by access to 127.0.0.1, which means anybody with server access is able to issue requests against it.

**I3** The global passwords can easily be communicated to participants and operators. No individual credentials need to be distributed. That is positive. Negative, in turn, is that individual authentication of participants cannot easily be established, although it is sometimes requested. Moreover, legal issues might arise when the GDPR takes effect in the first half of this year. Authentication via personal accounts, also for staff cannot be realized, no clear role definition (admin, operator, editors) can be configured. Some parts of the system are open and only obscured by URLs. Thus, participants who know URLs may be able to gain access other events. Also, malicious users could analyze clients and extract its unprotected resources.

**I4** Good is that authentication is optional. It can be enabled and disabled according to the requirements of different environments. Negative is that while the admin interface has password protection, not all of the contained links are likewise secured. If you know the system, you can bypass the web interface and access clients directly. A password protection is missing for the meeting cockpit.

**I5** The software works for what I want to do with it. I feel save using it and I am not aware of any issues with the system. However, a user management for the web interface is needed so that admins can create meetings that only some people can see and others not. Indeed, such demand did not yet come up on the market, but it may arise in the future.

## Do you see any usability issues with the current security system?

**I1** For participants, it is super easy. No authentication is easier than any authentication. For staff users, it would be nice if they can switch easier between different views. Actually, for staff it is very complicated to handle the different UIs and their individual sign-in requirements.

**I2** For participants, it is not transparent which data is collected and how it is processed. The login window for users is not aligned with the layout of the software. Users

likely don't get the impression of good security. The system does not provide adequate transparency.

**I3** It is easy for participants, especially if an event ID is used as password. Authentication timeouts are too fast.

**I4** No, there are no usability issues. Only documentation issues. Passwords that one needs for different interfaces need to be documented, which is currently not done consistently.

**I5** Authentication is not organized centrally. The server admin uses different credentials than operators at the meeting cockpit, or participants at their client.

# Understanding Current and Future Needs

## Which conceptual authentication mechanisms are relevant/ requested by customers now and in the future?

**I1** Connection to participant management systems are required sometimes. Also XLS files for user management are currently very relevant. In the future, I believe that mechanisms such as social media logins or logins via public identity systems, that are already spread over the internet, will eventually also become relevant in our domain.

**I2** GDPR compatibility will be relevant in the future. Apart from that, clients will continue to request events without personal authentication. The demand for coupling the software with various participant management systems will grow. Finally, password credentials for both group authentication and personal authentication will continue to be relevant.

**I3** Customers are happy with unsecured clients as well as with individual pins/passwords. So far, only one customer requested authentication via in-house employee accounts (he pushed the required configuration via MDM to the mobile devices). Another client requested a SSO approach and one requested a connection to DocCheck, a community system for doctors.

**I4** Currently relevant is the capability to connect authentication to existing user admin systems, keyword SAML login. Users don't want to create accounts, they want to reuse their companies existing IT infrastructure. Users also want to be

auto-logged in. Username and password are a security risk. In the future, the importance of externalized identity management will grow.

**I5** General problem: Clients often need access to a local network in order to access server. A solution therefore is relevant. After connecting to a password protected network, participant should automatically be redirected to server and not required to enter additional credentials. Customers often seem to have low request for security and a high demand for simplicity. However, security concerns are likely to rise in the close future.

## What are currently the conceptual roles staff members and participants can have in the system (administrator, operator, etc.)? What could be future roles?

**I1** Master, facilitator, feedback guy on stage, director guy clustering participant answers, table facilitator (collects ideas from participant), group/forum facilitator (collects answers from table facilitator), main facilitator (collects all results), feedback director, result previewer, participant manager (edits excel file), operator, admin.

**I2** Operator, table facilitator, facilitator, stage (without staff), administrator. For participants, there are usually no differentiable roles, however VIPs with increased voting weight are sometimes reflected via user properties. In the future, arbitrary roles should be declarable dynamically per event. For instance, customers may want to see questionnaire results spontaneously or managers might demand a result preview before going on stage for presentation. Those are just examples that require roles with respective access rights. Not all use cases are foreseeable. Thus, roles should be dynamically manageable via an admin interface.

**I3** Administrator, operator, facilitator, technical roles (stage, preview), director team, client, agency (for seeing results). In the future, we require dynamic roles that are declarable according to the needs of individual events, such as, to allow only a specific group of people to vote. Also, it should be possible to grant global administrators the privilege for creating new meetings and local ones the privileges for managing a specified subset.

**I4** Operator, facilitator, stage (no natural person, but a role). Participants can be grouped. In the future, roles for resellers will be required.

**I5** Administrator, feedback director, operator of meeting cockpit, operator of cluster tool, manager of participants. There are no roles for participants; however, participants can be segmented by criteria on their properties. In the future, a role for event providers is required that enables them to receive meta data such as how many participants contributed to a voting.

## Ideas for Improvement

### What are your ideas to improve authentication and authorization in the system?

**I1** For development and testing purposes, it would be nice if authentication could further on be completely disabled.

**I2** Authentication of individual participants and groups should be equally supported. Messages from all clients should be subject to an authorization check. Likewise, any HTTP endpoints should do fine-grained authorization for clients. Different admin roles should be available.

**I3** The system should provide a basic configuration that does not require a lot of configuration. In addition, it should offer extensive grouping support.

**I4** See my answer about roles for resellers.

**I5** See answer about roles.

# Bibliography


[Ado] *Adobe Connect*. Accessed: 2019-01-05. URL: https://www.adobe.com/products/adobeconnect.html.

[Are16] Craig Arendt. "Journey into WebSockets Authentication/Authorization". In: *Stratum Security Blog* (2016). Accessed: 2018-09. URL: https://blog.stratumsecurity.com/2016/06/13/websockets-auth/.

[ATB01] Mehmet Akşit, Bedir Tekinerdoğan, and Lodewijk Bergmans. "The six concerns for separation of concerns". In: *Proceedings of ECOOP*. 2001.

[BIC05] J. Brown, D. Isaacs, and W.C. Community. *The World Café: Shaping Our Futures Through Conversations That Matter*. 0. Berrett-Koehler Publishers, 2005. ISBN: 9781609940393. URL: https://books.google.de/books?id=J2h0kxwsLNMC.

[Bod05] Clive Boddy. "A rose by any other name may smell as sweet but "group discussion" is not another name for a "focus group" nor should it be". In: *Qualitative Market Research: An International Journal* 8.3 (2005), pp. 248–255.

[Coo] Aug. 2018. URL: https://stackoverflow.com/questions/17000835.

[Cra67] Kenneth James Williams Craik. *The nature of explanation*. Vol. 445. CUP Archive, 1967.

[Dat] *DataSift WebSocket API*. Accessed: 2018-09-24. MediaSift Ltd. URL: https://dev.datasift.com/docs/platform/api/streaming-api.

[Den13] Zach Dennis. *Choosing an SSO strategy: SAML vs OAuth2*. 2013.

[Doo17] Claire Doole. *Moderator, Facilitator and Master of Ceremonies: What is the Difference?* http://www.doolecommunications.com/moderator-facilitator-master-ceremonies-difference/. Accessed: 2019-01-07. 2017.

[Dou] *doubledutch*. Accessed: 2018-08-08. URL: https://doubledutch.me.

[DR08] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[EK10] Aaron Elliott and Scott Knight. "Role Explosion: Acknowledging the Problem." In: *Software Engineering Research and Practice*. 2010, pp. 349–355.

[Erk12] Jussi-Pekka Erkkilä. "Websocket security analysis". In: *Aalto University School of Science* (2012), pp. 2–3.

[FB14] Joachim Freimuth and Thomas Barth. "Workshop-und Sitzungsmoderation als Handwerk und Mundwerk". In: *Handbuch Moderation: Konzepte, Anwendungen und Entwicklungen* 18 (2014), p. 123.

[Fer+01] David F. Ferraiolo et al. "Proposed NIST Standard for Role-based Access Control". In: *ACM Trans. Inf. Syst. Secur.* 4.3 (Aug. 2001), pp. 224–274. ISSN: 1094-9224. DOI: 10.1145/501978.501980. URL: http://doi.acm.org/10.1145/501978.501980.

[Fis+09] Jeffrey Fischer et al. "Fine-grained access control with object-sensitive roles". In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 173–194.

[FK92] David F Ferraiolo and D Richard Kuhn. "Role-Based Access Control". In: *15th National Computer Security Conference* (1992).

[FM11] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt.

[Fra+99] John Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2617.txt.

[Gig] *Spring Security Guide*. =https://docs.gigaspaces.com/xap/9.7/dev-java/introducing-spring-security.html. Accessed: 2018-10-28.

[GO04] Mei Ge and Sylvia L Osborn. "A design for parameterized roles". In: *Research Directions in Data and Applications Security XVIII*. Springer, 2004, pp. 251–264.

[Gui07] Axel Guicking. "Electronic Meeting Support in Heterogeneous Environments". 2007.

[Har12] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor, 2012. URL: http://www.rfc-editor.org/rfc/rfc6749.txt.

[HDC07] Peggy Holman, Tom Devane, and Steven Cady. *The change handbook: The definitive resource on today's best methods for engaging whole systems*. Berrett-Koehler Publishers, 2007.

[Her] *WebSocket Security*. https://devcenter.heroku.com/articles/websocket-security. 2017.

[JKS12] Xin Jin, Ram Krishnan, and Ravi Sandhu. "A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC". In: *Data and Applications Security and Privacy XXVI*. Ed. by Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquin Garcia-Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 41–55. ISBN: 978-3-642-31540-4.

[KH01] Gregor Kiczales and Erik Hilsdale. "Aspect-oriented programming". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 26. 5. ACM. 2001, p. 313.

[Kul13] Achin Kulshrestha. "An empirical study of HTML5 websockets and their cross browser behavior for mixed content and untrusted certificates". In: *International Journal of Computer Applications* 82.6 (2013).

[Lap07] Phillip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

[Loe+13] Diana Loeffler et al. "Developing Intuitive User Interfaces by Integrating Users' Mental Models into Requirements Engineering". In: *Proceedings of the 27th International BCS Human Computer Interaction Conference*. BCS-HCI '13. London, UK: British Computer Society, 2013, 15:1–15:10. URL: http://dl.acm.org/citation.cfm?id=2578048.2578069.

[Lou10] Nadejda Loumbeva. *Difference between Moderation and Facilitation*. https://loumbeva.wordpress.com/2010/06/09/difference-between-moderation-and-facilitation/. Accessed: 2019-01-07. 2010.

[Mal16] Nick Malcolm. *Everything you need to know about passwordless logins*. https://thisdata.com/blog/an-introduction-to-passwordless-logins/. Accessed: 2018-11-05. Sept. 2016.

[Men] *Mentimeter*. Accessed: 2018-12-28. URL: https://www.mentimeter.com.

[OAS05a] Standard OASIS. "Assertions and protocols for the OASIS security assertion markup language (SAML) V2. 0". In: *Organization for the Advancement of Structured Information Standards (OASIS)(March 2005)* (2005).

[OAS05b] Standard OASIS. "Extensible access control markup language tc v2. 0 (XACML)". In: *Organization for the Advancement of Structured Information Standards (OASIS)(February 2005)* (2005).

[Oraa] *Core J2EE Patterns - Data Access Object*. https://www.oracle.com/technetwork/java/dataaccessobject-138824.html. Accessed: 2018-11-02.

[Orab] *The Essentials of Filters*. https://www.oracle.com/technetwork/java/filters-137243.html. Accessed: 2018-11-04.

[PD11] Larry L Peterson and Bruce S Davie. *Computer networks: a systems approach*. Elsevier, 2011.

[Pos81] Jon Postel. *Transmission Control Protocol*. RFC 793. RFC Editor, 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[Sak+14] Nat Sakimura et al. "OpenID Connect Core 1.0 incorporating errata set 1". In: *The OpenID Foundation, specification* (2014).

[Sel15] Ruth Seliger. *Einführung in Großgruppen-Methoden*. Vol. 3. Heidelberg: Carl-Auer Verlag GmbH, 2015.

[Six] *SixSteps*. Accessed: 2018-09-15. URL: https://www.sixsteps.com.

[Sla] *Slack API*. Slack Technologies, Inc. URL: https://api.slack.com/rtm.

[Sli] *sli.do*. Accessed: 2018-08-08. URL: https://www.sli.do.

[Spo] *SpotMe*. Accessed: 2018-08-08. URL: https://spotme.com.

[Spra] *OAuth 2 Developers Guide*. http://projects.spring.io/spring-security-oauth/1.x/docs/oauth2.html. Accessed: 2018-10-16.

[Sprb] *Spring Reference*. https://docs.spring.io/spring/docs/5.1.3.BUILD-SNAPSHOT/spring-framework-reference/core.html. Accessed: 2018-11-19.

[Sprc] *Spring Security API*. https://docs.spring.io/spring-security/site/docs/5.1.1.BUILD-SNAPSHOT/api/. Accessed: 2018-10-17.

[Sprd] *Spring Security Architecture*. https://spring.io/guides/topicals/spring-security-architecture/. Accessed: 2018-10-29.

[Spre] *Spring Security Reference*. https://docs.spring.io/spring-security/site/docs/5.1.1.RELEASE/reference/htmlsingle/. Accessed: 2018-10-17.

[SS94] Ravi S Sandhu and Pierangela Samarati. "Access control: principle and practice". In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

[SV01] Pierangela Samarati and Sabrina Capitani de Vimercati. "Access Control: Policies, Models, and Mechanisms". In: *Foundations of Security Analysis and Design*. Ed. by Riccardo Focardi and Roberto Gorrieri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 137–196. ISBN: 978-3-540-45608-7.

[Tea] *teambits*. Accessed: 2018-08-08. URL: https://www.teambits.de.

[TKS13] Peter Tandler, Katja Königstein, and Stefan Schnitzler. "Digitale Moderation-An jedem Ort und zu jeder Zeit Betroffene zu Beteiligten machen". In: *Handbuch Moderation. Konzepte, Anwendungen und Entwicklungen, Göttingen* (2013), pp. 431–450.

[Web] *WebEx*. Accessed: 2018-08-08. URL: https://www.webex.com.

[Wha] "Web sockets". In: *HTML - Living Standard* (2018). Accessed: 2018-09. URL: https://html.spec.whatwg.org/multipage/web-sockets.html.

[Wik] Accessed: 2018-10-15. May 2017. URL: https://www.wikitechy.com/tutorials/oauth/oauth-architecture.

[YP14] Xiaodan Yu and Stacie Petter. "Understanding agile software development practices using shared mental models theory". In: *Information and Software Technology* 56.8 (2014), pp. 911–921.

[YT05] Eric Yuan and Jin Tong. "Attributed based access control (ABAC) for web services". In: *Proceedings. IEEE International Conference on Web Services*. IEEE. 2005.

[ZF09] Klaus Zeppenfeld and Patrick Finger. *SOA und WebServices*. Informatik im Fokus. Berlin, Heidelberg, 2009. ISBN: 9783540769903.