Access Control in Rich Domain Model Web Applications

Extended Abstract

João de Albuquerque Penha Pereira
joao.pereira@ist.utl.pt

Instituto Superior Técnico
November 25, 2010

Abstract

Rich Domain Model (RDM) web applications present a serious challenge in the security and access control area. The Domain Model Authorization Policy Language is a Domain Specific Language which aims at the expression and management of access control policies in RDM web applications. It is inserted in a wider framework, the DMAPL framework, which also contains a model and a runtime engine. In this work, the DMAPL framework’s runtime engine was completed and integrated with the Fénix Framework. This allowed to implement and test the DMAPL framework in a real RDM web application, the FeaRS (Feature Request System). Taking advantage of the integration with the Fénix Framework, the DMAPL framework was further developed to support domain level access control.

1 Introduction

Nowadays web applications play a major role in information systems. Yet, the development process of these remains rather ad-hoc. To improve this process, developers are exploring new approaches, such as the Domain-Driven Design (DDD) [4]. DDD concentrates on the domain of a system and strongly supports that complex domains should be based on a model\(^1\), describing all its relevant entities and the relationships between them. DDD tries to leverage on all the advantages of the object-oriented paradigm, leading to a domain model whose entities contain both data and behavior. To such a model we call Rich Domain Model (RDM) [1]. RDM web applications present a serious challenge in the security and access control area. Several obstacles appear when trying to enforce

\(^1\)http://en.wikipedia.org/wiki/Domain_model
access control policies in RDM web applications due to the complexity of the systems and the different technologies working and communicating together.

Previous research on this subject was made by Dumiense [3]. In his work, he identified the common problems when trying to enforce access control policies in RDM web applications as being code scattering and tangling, expressing and enforcing complex rules, introducing dependencies between the code, and lack of support for the delegation of rights. He then developed a solution to solve such problems that is composed by three components: a model that supports authorization, amplification of privileges, and delegation of rights; a Domain Specific Language (DSL) called Domain Model Authorization Policy Language (DMAPL) based on the previous model with special constructs to help express complex access control rules; and a runtime engine to enforce the access control rules specified with the DMAPL upon the application. However, this solution still needs several improvements and an implementation in a real system for further study and analysis of its capabilities.

The remaining of this extended abstract is organized as follows. I define the goals of my work in 2. An overview of related work is laid now in 3. In 4, I describe the development made in the solution. A new type of access control rule is proposed in 5. The validation of the solution is then achieved and described in 6. Finally, in 7, the final conclusions are presented.

2 Goals

The main goal of my work is to continue the research and development of the solution described by Dumiense [3]. Specifically, I first intend to review the solution and do any development needed. Afterwards, I intend to validate the solution with an implementation in a real RDM web application. This is a crucial step in any piece of software currently being developed since it provides reliable feedback about the software’s performance, and allows to discover unforeseen problems. I plan to implement the solution in an RDM web application currently used by my University - the FeaRS (Feature Request System) web application. To use the solution in the FeaRS, I will first need to integrate it with the Fênix Framework. Another goal of my work is to have a framework that can be easily used by a team of Java developers when specifying access control policies, taking in consideration the Java programming language and the OOP paradigm, and how its developers usually work and develop solutions.

The ultimate goal of this work is to have a usable and extensible access control mechanism specially crafted for Rich Domain Model web applications.

3 Related Work

Policy Specification Languages (PSLs) are Domain Specific Languages (DLSs) designed specifically to define and express a policy of a system. The XACML [5] is an XML-based language most suitable to tie up large heterogenous au-
thorization systems, which falls out of the scope of this work. Ponder [2] is an object-oriented language making it suitable to specialize policies by inheritance. However, Ponder and Ponder2 [6] are not able to override policies with amplification of privileges like the DMAPL [3], which is a highly desired feature that enables the enforcement of access control policies in a compositional manner. Also, Ponder does not make use of Java annotations or wild-cards which is also desired not only by the advantages they present in avoiding code scattering and tangling, but also because they make the framework more suitable for Java developers by using familiar concepts. Ponder and Ponder2 are also oriented towards directory services instead of objects in a rich domain, like the DMAPL.

SPL [7] makes use of a different and more powerful construction of rules in comparison with Ponder, Ponder2, and the DMAPL. However, its syntax has no similarities with the Java programming language which can be a disadvantage when trying to build a framework suitable for Java developers. SPL also lacks of a specific mechanism for expressing delegation (although it supports the concept) which can confuse developers when trying to do so. Moreover, due to SPL’s enhanced algebra for composition of policies and the ability to express many different types of policies, the authors state that SPL is better used to assemble big policy blocks.

Of the five PSLs presented, the DMAPL appears to be the most capable and well adapted to specify access control policies in RDM web applications. The DMAPL is also a PSL which clearly states to aim at the problems encountered when trying to specify access control policies in such applications. Therefore, I choose the DMAPL to support my following work.

4 DMAPL Framework and the Fénix Framework

I began my work by analyzing, and having a practical approach to the solution introduced by Dumiense in [3], the DMAPL Framework.

4.1 Overview of the DMAPL Framework

To give a brief overview, the DMAPL framework has three main components:

- The Model
- The Domain Model Authorization Policy Language (DMAPL)
- The Runtime Engine

The model supports the three types of access control rules: authorization; amplification of privileges; and delegation. The Domain Model Authorization Policy Language (DMAPL) is the language created to express and define those access control rules. Finally the runtime engine is the responsible for enforcing such rules in the designated application during its execution. In short,
a developer specifies the access control rules supported by the model in a policy file using the DMAPL, which will in turn be interpreted and enforced in the target application by the runtime engine. However, I came to realize that the DMAPL framework’s runtime engine lacked several fixes and further development to perform as expected.

4.2 Fénix Framework Integration

The Fénix Framework allows the development of Java-based applications that need a transactional and persistent domain model. Developing an application with the Fénix Framework starts by specifying the structure of its domain model using the domain-specific language called Domain Modeling Language (DML) [1]. After this step, the Fénix Framework generates a generic version of the domain classes. All the domain objects later created (objects that materialize the classes specified in the domain model), will be automatically persisted in a database by the Fénix Framework, and the rest of the application is developed in plain Java. After having the DMAPL framework ready to be used, to be able to implement it test it with the FeaRS, I had first to integrate it with the Fénix Framework.

During the development of the DMAPL framework, one important aspect always taken into consideration was to make it decoupled from the underlying application. This is important to prevent the DMAPL framework from setting restrictions to the application’s domain and development. Even so, the DMAPL framework needs to somehow be aware of the application’s users and roles to enforce the access control rules upon them. In a regular Java application, the mechanism the DMAPL framework uses to link its own domain with the application’s domain, is to enforce the application’s users and roles to extend two interfaces: the AccessControlUser interface, and the AccessControlRole interface. However, with the Fénix Framework between the DMAPL framework and the application’s domain, a different approach had to be taken.

4.2.1 Access Control Plugin

The Fénix Framework supports a plugin system that allows the developer to provide self contained components to the framework. Through the plugin system, it is possible to provide an access control domain, internal to the Fénix Framework, to link both the DMAPL framework’s domain and the application’s domain. Therefore, an access control domain model was created defining three classes:

ACRoot which is connected to all of the access control domain entities, making them available to the Fénix Framework application.

ACUser to represent the users under access control.

ACRole to represent the roles under access control.
Consequently, instead of having the application’s users and roles extending the DMAPL framework’s interfaces, they should now extend the ACUser (AC stands for Access Control) and ACRole classes accordingly. In turn, the ACUser and ACRole classes implement the DMAPL framework’s AccessControlUser and AccessControlRole interfaces. The access control hierarchy can be seen in Figure 1.

This way, if the class representing the users of the application extends the ACUser class, the application’s users automatically become subjects to the access control rules interpreted and enforced by the DMAPL framework. The same applies for the class representing the roles of the application when extending the ACRole class. As a result, it is extremely easy for the developer to place his users and roles under access control.

5 Domain Model Relations

Let us consider the example of the domain model shown in Listings 1 and 2 where two domain entities are defined (Student and Course), as well as the relations between them. The goal of the relation depicted in Listing 2 is to model the fact that a course can have several students, and a student can have several courses (relation CoursesHaveStudents). However, a developer should be able to define who in the system can enroll and remove students from courses. Using the DMAPL framework, how can we express this access control requirement? The granularity level of the DMAPL framework’s access control mechanisms is the Java interface method, and although methods to handle the relations are automatically generated by the Fénix Framework, one relation can still be modified through different methods in different points of the code. This makes
Listing 1: Example of a domain model where two domain entities are defined: Student and Course.

class Student extends ACUser {
    String id;
    Course course;
}

class Course {
    String name;
}

Listing 2: Definition of the relation between the domain entities Student and Course.

relation CoursesHaveStudents {
    Course playsRole courses {
        multiplicity *;
    }
    Student playsRole students {
        multiplicity *;
    }
}

5.1 Relation Rule

The creation of the relation rule is motivated by the desired possibility of enforcing access control at the domain level. With it, it is possible to specify who can alter a relation between two domain entities, and when.

A relation rule is composed by a:

Subject that refers to whom the rule applies. Can be either a user or a role.

Change type to qualify the type of change allowed in the relation. It may be
Listing 3: Relation rule specifying that the role Management can add and remove elements from the relation CoursesHaveStudents.

```
CourseStudentAssignment:
   allow role Management
to change relation CoursesHaveStudents
```

the value add-to when the rule applies only to the addition of relations between entities, remove-from when the rule applies only to the removal of relations between entities, and change when the rule applies to both addition and removal of relations between entities.

Target relation which is the fully qualified name of the target relation of the rule.

Optionally, a relation rule may have a constraint introduced with the keyword where. Simply put, a constraint is a Java expression between brackets that evaluates to the boolean values true or false.

In Listing 3 it is presented an example of a relation rule, stating that users holding the role Management can change the relation CoursesHaveStudents, and therefore assign or remove students from courses.

6 Validation of the DMAPL framework

To implement and test the DMAPL framework I used the FeaRS2 (Feature Request System). The FeaRS has been fully implemented and used in a real environment for some time now, making it a good candidate to test our solution and provide us reliable feedback with real data.

6.1 The FeaRS

The FeaRS is a real RDM web application developed to manage suggestions of features for several systems. Its purpose is to gather suggestions inputed by users, and through a voting system, rank those suggestions. The ultimate goal is to improve the services provided by a system.

6.2 The FeaRS access control requirements

The FeaRS has the following access control requirements:

- Only registered users can vote, add features, and add comments.

2http://fears.ist.utl.pt
• A registered user can remove votes, but only his own votes.
• Only administrators can create, edit, and delete projects.
• Only administrators can see the current list of administrators.
• Only administrators can grant a registered user the role of administrator, and revoke it (except for themselves).
• Only administrators can assign a registered user as a project administrator, and remove it (except for themselves).
• Only administrators and project administrators can change the project’s features state.
• Seeing existing projects and features requests is public.

To express the FeaRS access control requirements, and test the different DMAPL framework’s access control mechanisms, three exercises were made.

In the first exercise I tried to express the FeaRS access control requirements using only authorization rules, tickets, and amplification of privileges. This meant to access control 18 methods, which resulted in an access control policy with 17 authorization rules, one ticket, and one amplification of privileges. Using both the authorization rules, tickets, and amplification of privileges, provided great expressiveness to specify access control rules. As a result, all of the FeaRS access control requirements were successfully covered by the DMAPL framework’s access control mechanisms.

In the second exercise I used the same authorization rules, tickets, and amplification of privileges, together with annotations. However, not all FeaRS access control requirements were satisfied with annotations. For example, methods with the same annotation but with different access control constraints, are targets of one single authorization rule where we can only specify one constraint. This suggests that annotations are hard to use with constraints. Also, it was not possible to use annotations with the ticket and the amplification of privileges. Similar to methods with different access control constraints, tickets and amplification of privileges are usually more specific cases of access control. This is opposite to the annotations purpose, which is to be used in more generic cases. Using annotations resulted in an access control policy with six authorization rules (three of them using the access control annotations), one ticket, and one amplification of privileges.

In the third and last exercise, I tried to express the FeaRS access control requirements using only relation rules. However, access control requirements that did not affect any domain relation were not possible to express using relation rules. The semantics of tickets and amplification of privileges are also not suitable for relation rules. In [3], the constraints introduced to be used with the access control rules, can include the arguments passed on to the access controlled methods giving these constraints greater expressiveness. When using relation rules, part of this expressiveness is lost. This exercise resulted in
an access control policy with 21 rules (16 relation rules and five authorization rules), one ticket, and one amplification of privileges.

7 Conclusions

In this work, I completed the development of the DMAPL framework’s runtime engine since it was still not ready to be used as it could be. Next, to be able to use and test the DMAPL framework with any of RDM applications used by my University, I integrated the DMAPL framework with the Fénix Framework. This lead to the creation of an access control plugin providing an easy for developers to use the DMAPL framework’s access control mechanisms in any application using the Fénix Framework. Taking advantage from the integration with the Fénix Framework, I also introduced a new type of access control rule: the relation rule. The creation of this new type of rule was motivated by all the emphasis given to the domain model of an RDM application by the DDD development approach. With the relation rules, developers are now able to express domain level access control within the DMAPL framework. Finally, I implemented and tested the DMAPL framework in a real RDM web application, the FeaRS. Four exercises were done testing the several DMAPL framework’s access control mechanisms, where all the FeaRS access control requirements were successfully satisfied.

References


