Javier Ochoa

Security for Java Web Applications Using Apache Shiro

Helsinki Metropolia University of Applied Sciences

Master’s Degree

Information Technology

Master’s Thesis

28 November 2014

Author(s) Title Number of Pages Date	Javier Ochoa Security for Java Web Applications Using Apache Shiro 74 pages 28 November 2014
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Peeter Kitsnik

Web applications have become a necessity to almost any organization worldwide, but these applications can considerably weak the corporation’s security network since they may be turned into security breaches by people with malicious intentions, causing damage to finances and to a company’s reputation.

Apache Shiro is a Java security framework built as a solution for developers to easily integrate security features such as authentication, authorization, cryptography, and session management, on any type of Java application. Its main objective is to reduce the complexity regarding to the management of an application’s security.

This study examined the integration and usability of Apache Shiro as a security framework for Java applications and it reached for an understanding of the framework where custom behavior was needed, instead of sensible defaults.

As practical work, a prototype was created to supply a security solution for a Java web application. The goal was to provide user management features requested, such as user role control access and the possibility of authentication through the use of user data stored within a rational database.

Besides the prototype being a success, this study has helped to gain a wider view of security and user management within Java web applications and further study will be placed to accomplish more reliable and secure applications.

Keywords	Java, security, user management, Apache Shiro

Contents

List of Figures

List of Listings

List of Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
DAO	Data Access Object
GSP	Groovy Server Pages
HTTP	Hypertext Transfer Protocol
JAAS	Java Authentication and Authorization Service
JCE	Java Cryptography Extension
JDBC	Java Database Connectivity
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface
JSF	Java Server Faces
JSP	Java Server Pages
LDAP	Lightweight Directory Access Protocol
JPA	Java Persistence API
MAC	Message Authentication Code
MVC	Model-View-Controller
OWASP	Open Web Application Security Project
POJO	Plain Old Java Object
SSL	Secure Sockets Layer
XML	Extensible Markup Language

1. Introduction

Nowadays, web applications are becoming one of the basic and necessary tools on business for enterprises, and at the same time, the biggest source of data breaches. Security breaches can create good opportunities for attackers to steal valuable data, such as customer information, to expose sensitive records or to go deeper into the organization network to reach internal resources and more, which can eventually ruin business reputation.

Any business that handles sensible information can be a target for security breaches at any time during its daily operations. Companies commonly implement many sophisticated defense systems, as firewalls and SSL (Secure Sockets Layer) encryption, into their web applications, which are useful for protecting application access, but that will not fully block all security breaches.

People often relate security breaches to single events from website hacking to information theft but in reality, security breaches have a wider range of an organization’s data, covering any type of business’s assets. Though security vulnerabilities can affect any business in terms of financial and data loss, strategic security measures and application security can be taken in order to eliminate potential attacks, reduce unexpected risks and help businesses retaining customers trust, and guarantee revenue.

Application security is a process that starts with the application development lifecycle to make sure that the best possible security is applied to each of the steps of the development process, including the hardware where the application may run on, or securing the network that the application may use to authenticate and authorize users. Planning software with application security in mind from the initial design leads to software with less security-related issues and less potential for vulnerabilities. However, this does not mean that applications which are designed with security in mind will ensure secure software. It means that fewer flaws will be identified during development, testing and production phases, and that fewer flaws related to the entire application are likely to be found.

This Master’s Thesis examines, in detail, Apache Shiro, a Java security framework which can be easily integrated into any Java application as a security solution. The aim is that the outcome of this thesis would be a prototype of a Java web application which integrates Apache Shiro as security framework.

1.1 Objective, Scope and Structure of the Thesis

The thesis sets the objective to investigate the integration of Apache Shiro in a Java web application, requiring multiple user roles management and authentication through credentials stored in a database.

An overview of Apache Shiro’s architecture is included as part of the research, as well as a deeper analysis of its core and web support. Technical examples and code snapshots are included to help during the reading of the thesis. At the time of writing of this thesis, Apache Shiro 1.2.3 is the official stable release available and so this is the release used for framework analysis, code snapshots and examples. This thesis is not a guideline or tutorial for a full Java web application development, and no other programming language besides Java is covered.

The thesis is written in seven sections. Section 1 describes the case company and focus of this study. Section 2 overviews the theory background and context for this thesis. Sections 3, 4 and 5 discuss the concepts and features of the current framework to study. Section 6 presents the results of the study in the shape of a prototype for the project proposal. Section 7 presents an evaluation and summary of the study, as well as future steps for research and development. Section 8 summarizes this thesis with conclusions.

1.2 Company and Project Background

Northscreen is a Finnish company which a focus on solutions for monitoring road status through a solution called ArcticView. Currently Northscreen is developing a Java web application that will be used for internal management purposes. Due to the importance of data to be managed, the company requires the integration of a security framework to secure this application. This framework is Apache Shiro and the choice of it is due to the fact that this framework is already being used by the company in other solutions they provide. In this way, the expertise of the company is re-used and improved based on other requirements needed for this project.

This new application requires a wider and more advanced use of Apache Shiro in comparison to how the company is currently using it as their applications' security framework, demanding deeper understanding and more technical analysis of such tool for a successful integration.

There are two main requirements that are needed for a successful integration of Apache Shiro within this new project.

a) Multiple user roles management

A role, as explained in more detail in chapter 4 (section 2) is simply a title which defines an authority level for the user. The role will determine which actions a user can and cannot execute within the application. In this case, Northscreen needs two roles management, normal users and administrator users. As agreed with the company, both users will be able to execute any action within the application with the only exception of User Management. User Management is the feature within the application that allows creating, editing or removing users from the system, and in this case, this feature is allowed only for administrator users.

b) Authentication through credentials stored in a database

As agreed with the company, user credentials will be stored in a relational database which will be accessed when the user wants to log into the system. The credentials stored in the database include both user id and password. For security purposes, the password will be encrypted using the SHA-256 hash algorithm. Cryptography and best practices for storing credentials data are discussed in detail later in chapter 4 (section 4).

Research Question

How to integrate Apache Shiro as security framework for a Java web application, enabling the possibilities of multiple user roles management and allowing authentication through credentials stored in a database.

2. Security in Java

The use of Java security framework is the main focus of this thesis, and therefore it seems necessary to define what security means within the Java context.

Oaks (2001) tell how Java attracted the attention of programmers from all around the world when it was first released by Sun Microsystems, and how some of these developers were in fact attracted because of the security capabilities that Java came built with. However, the phrase “Java is secure” brought to these developers certain expectations which were not necessarily shared by the designers of Java itself [1].

Depending on the expectation, Oaks (2001) defines different meanings to the term “security” when applied to a Java application, including:

- Authenticated
  The identity of users involved in the application should be verified.

- Encrypted
  Data that the application may send and/or receive, over the network or through a persistent store such as a file system or database, should be always encrypted.

- Audited
  All potentially sensitive operations should always be logged.

- Well−defined
  A well−defined security specification should be followed.

- Verified
  Rules of operation should be set and verified. [1, 2]

These are important features that nowadays may be acknowledged and understood by any modern Java developer, due to the fact that technology has evolved considerably since the first release of Java.

Nevertheless it is important to mention that Java's security model did not, and still does not, implement all these features. For example, authentication was added to Java 1.1 release, and encryption is currently available to Java 2 platform as an extension.

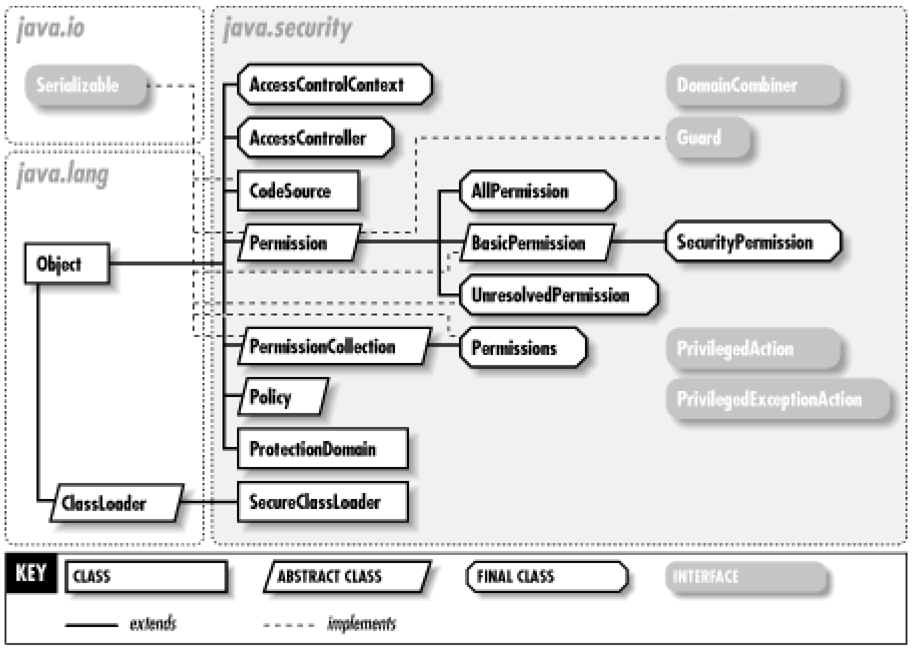

Figure 1 - The access control classes of the java.security package [2]

The java.security package contains all the interfaces and classes employed for the Java security architecture (see figure 1). These classes can be categorized in classes that implement access control, also referred as authorization, which avoid non-trusted code from performing delicate operations, and classes that implement authentication, which enable message digests and digital signatures and can authenticate Java classes and other objects.

2.1 Security in Web Applications

Bayse (2004) provides a description of the security challenges introduced by the use and implementation of web applications in its work *A Security Checklist for Web Application Design* [3].

Bayse (2004) describes web applications to be highly valuable to corporations because of the quick access they provide to internal resources and the fact that their deployment is almost effortless when executed remotely. However, these are the very same reasons that can turn into serious security risks, since a non-secure web application can provide quick and effortless access to private and valuable corporate data for unauthorized users through many different paths, as illustrated in figure 2 [3, Abstract].

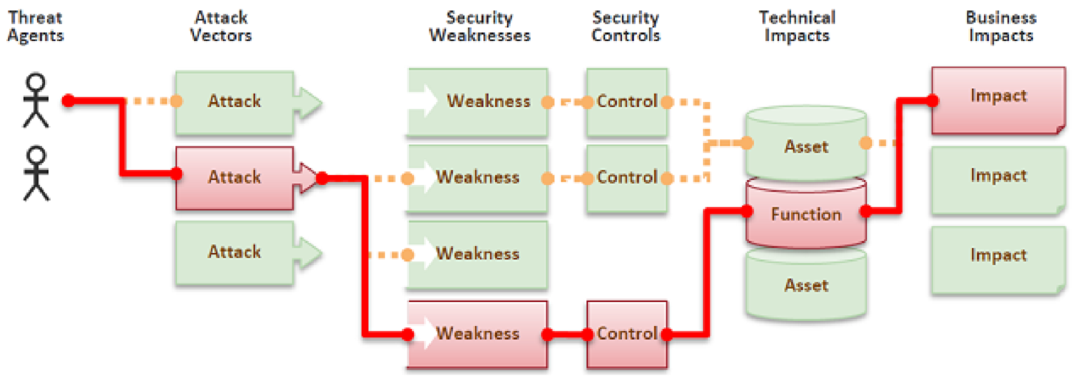

Figure 2 - Different paths through an application to do harm to the organization [4]

Bayse (2004) specifies a checklist of security concerns, which provide a basis for securing web applications, and the data sources they connect to, from malicious and unintentional abuse.

- Risk Assessment
- Authentication
- Authorization and Access Control
- Session Management
- Data and Input Validation
- Cross Site Scripting (XSS)
- Command Injection Flaws
- Buffer Overflows
- Error Handling
- Logging
- Remote Administration
- Web Application and Server Configuration [3, Abstract]

Additionally, and in relation to security for web applications, there is a project called OWASP Top Ten project [5], directed by Open Web Application Security Project (OWASP) [6]. OWASP is a non-profit organization which objective is to help corporations and individuals to improve the security in their software, providing them with more visibility over software security so that decisions can be made based on real software security risks. The OWASP Top Ten project provides a document for web application security, which list the ten most critical web application security risks considered by the project.

On June 12, 2013 the OWASP Top Ten for 2013 was officially released and it is as follows:

- A1 - Injection
- A2 - Broken Authentication and Session Management
- A3 - Cross-Site Scripting (XSS)
- A4 - Insecure Direct Object References
- A5 - Security Misconfiguration
- A6 - Sensitive Data Exposure
- A7 - Missing Function Level Access Control
- A8 - Cross-Site Request Forgery (CSRF)
- A9 - Using Components with Known Vulnerabilities
- A10 - Unvalidated Redirects and Forwards

The methodology of how this top ten is produced is documented on the OWASP website [7].

2.2 Java Security Frameworks

There are several security frameworks available for Java applications besides Apache Shiro. Here there is a brief overview of the most commonly used frameworks.

HDIV

HDIV is an open-source framework that aims to avoid or reduce web security risks that currently exist in some of the most used JVM web frameworks because of their design. HDIV integrates to work from within the application instead following the behavior of traditional web applications working as firewalls. It is advertised that this framework covers all web risks described by OWASP Top 10. As for web support, it can be integrated with many web frameworks, including Spring, Grails, Struts and Java Server Faces (JSF) [8].

Spring Security

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. In its website it is announced that the real power of Spring Security is found in how easily it can be extended to meet custom requirements [9].

Some of the features described are:

- Comprehensive and extensible support for both Authentication and Authorization.
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc.
- Servlet API integration.
- Optional integration with Spring Web MVC [9].

Java Authentication and Authorization Service (JAAS)

JAAS is a security framework which provides a user-centric authentication and authorization API, and that can be integrated in small applications as well as in enterprise applications. For authentication purposes, JAAS uses a service provider approach. This means that the underlying authentication logic is something the application remains unaware of, and so, it is then possible to configure multiple modules for login purposes without requiring any code changes. JAAS was introduced with JDK 1.3 as an optional package, and fully integrated as part of JDK 1.4 [10]

jGuard

jGuard is an open source library that provides easy authentication and authorization for Java web and desktop applications, and it is built on JAAS framework. The primary feature of jGuard is that it allows the security system to be independent of the persistence choice for the credentials of the user to be stored: relational databases, XML files, or accessed through LDAP servers. This opens up the possibility of easily shifting from one data source to another at any time with minimum effort and changes [11].

3. Apache Shiro Overview

Apache Shiro (pronounced “shee-roh”, which means ‘castle’ in Japanese language) is a powerful and flexible Java security framework that offers developers a comprehensive solution to authentication, authorization, session management and cryptography, and can be used to secure any type of Java applications, from command line applications to the largest web and enterprise applications, and even mobile applications [12].

One of its goals, if not the first and most important, is to be easy to use and understand, since security can be quite complex at times. Thus, Apache Shiro masks complexities where possible and exposes a clean and intuitive API that minimizes the developer's effort to turn any application into a secure application [12].

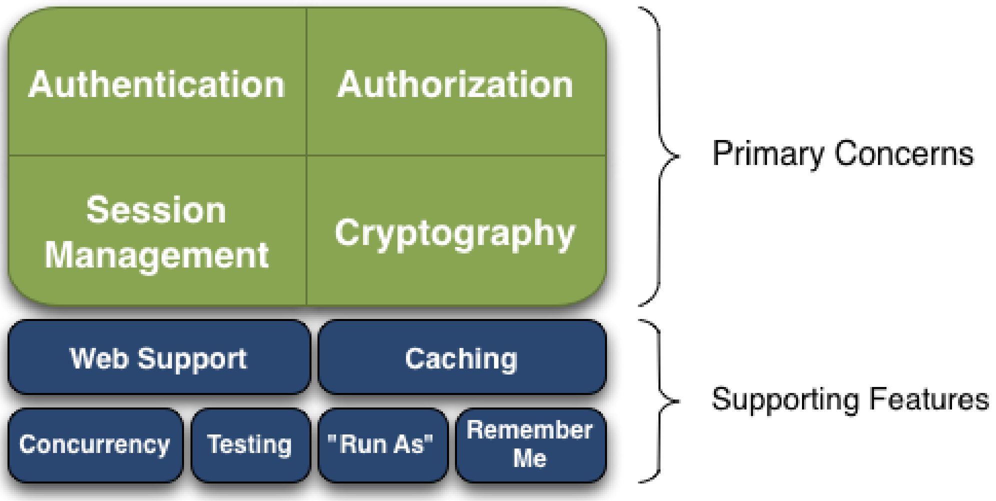

Figure 3 - Apache Shiro features [12]

Shiro provides the application security API to perform the following aspects as figure 3 illustrates:

- Authentication: to provide user identity, commonly known as user ‘login’.
- Authorization: to manage access control.
- Session management: per-user time-sensitive state.
- Cryptography: to protect and hide sensible data from curious eyes.

Shiro additionally supports certain auxiliary features, such as web application security, unit testing, and multithreading support, but these mainly exist to boost the above four primary concerns.

- Web Support: Shiro's web support APIs to easily help securing web applications.
- Caching: this feature ensures that security operations remain fast and efficient.
- Concurrency: Apache Shiro supports multi-threaded applications.
- Testing: Test support exists to help writing unit and integration tests and ensure that any code is secured as expected.
- "Run As": A feature to allow users to assume another's' user identity, which can be useful in certain scenarios.
- "Remember Me": Remember users' identities across sessions so that logging in is only needed when mandatory [12].

3.1 Architecture

Apache Shiro's design goal is to reduce the complexity of securing an application meanwhile being intuitive and easy to use. Shiro's core is design following the traditional thought about application security within the context of someone (or something) interacting with an application [13].

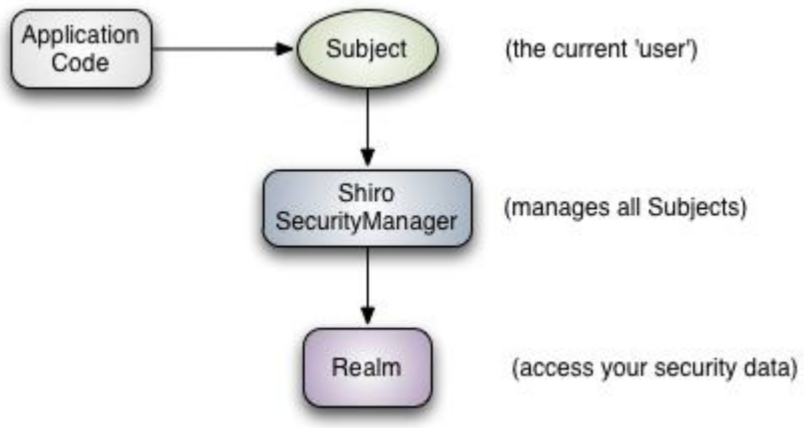

Figure 4 - High-level architecture overview [13]

As illustrated in figure 4, Shiro's architecture has three primary concepts: the Subject, SecurityManager and Realms.

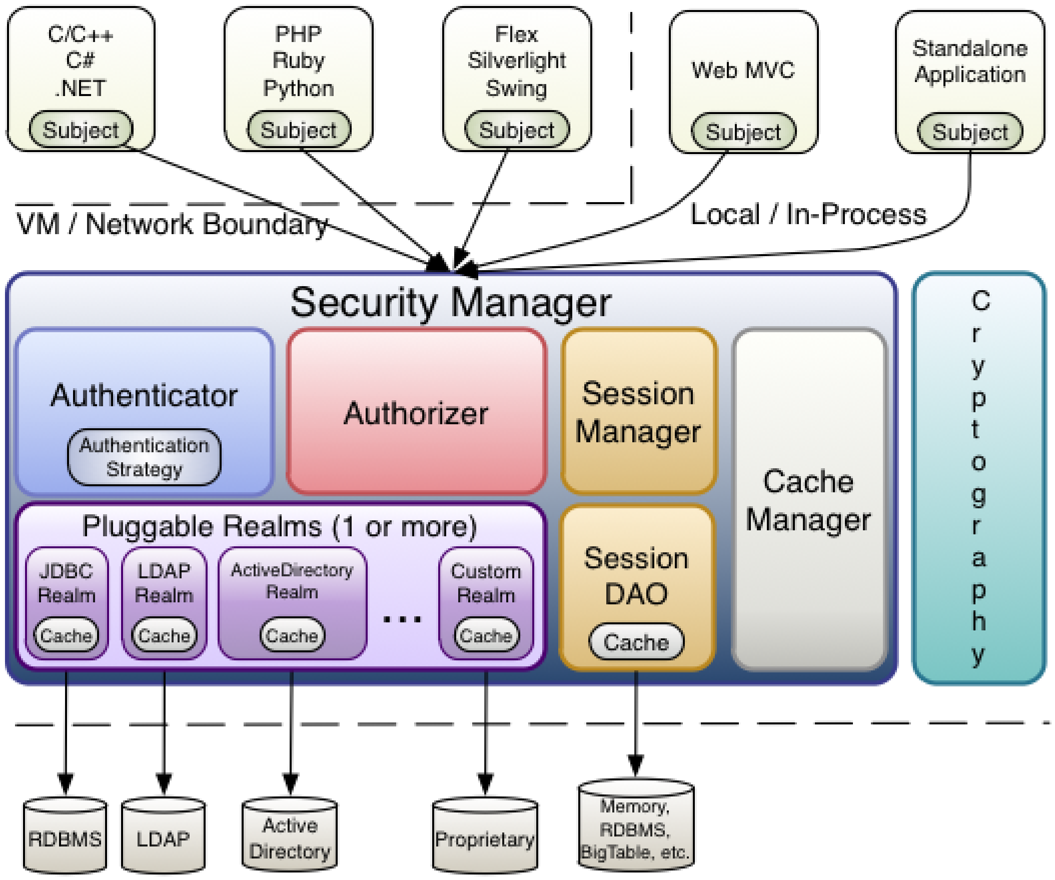

Figure 5 – Apache Shiro's detailed architecture [13]

Figure 5 provides as a detailed perspective of the interaction between all those primary concepts.

Subject

> “A Subject represents state and security operations for a single application user. These operations include authentication (login/logout), authorization (access control), and session access. It is Shiro's primary mechanism for single-user security functionality.” [14]

For Apache Shiro, Subject is just a security term which basically means “the currently executing user”. Subject is the term used for what is frequently called User, but the reason why the term User is not used in Apache Shiro is because User is commonly associated with a human being, when talking about application security, the term Subject can mean a human being, but also a 3rd party process, daemon account, cron job, or anything similar. In other words, Subject simply means “the thing that is currently interacting with the software”, regardless of being a human being or not [15].

SecurityManager

> “A SecurityManager executes all security operations for all Subjects (aka users) across a single application.” [16]

If the Subject represents security operations for the current user, the SecurityManager manages security operations for all users. It is the main core of Shiro’s architecture and acts as a hub for referencing internally nested security components. It is usually felt alone once it is configured, since the Subject provides most of the functionality required by developers [15].

Realms

> “A Realm is a security component that can access application-specific security entities such as users, roles, and permissions to determine authentication and authorization operations.” [17]

A Realm acts as a connector between Shiro and the application’s security data. Shiro can use one or multiple of the Realms which are already configured for an application to perform authentication and/or authorization. These Realms will be used to interact with security-related data when required and complete the authentication / authorization requests [15].

In essence, a Realm can be considered as a DAO for security purposes. It contains the details to connect to data sources and provides to Shiro with the available data associated to the request. It is important to remember that, in order to configure Shiro successfully, at least one Realm must be specified.

Shiro provides some implemented Realms ready to be used and which represent some of the most common security data sources such as LDAP, relational databases (JDBC), text configuration sources (properties and ini files) and others. In case the defaults Realms do not meet the application developer needs, Shiro allows to plug-in any custom Realm implementation which represents the required custom data source [15].

3.2 Configuration

Shiro can be used in any type of Java application, from a simple command-line application to a large enterprise application, because it has been designed to work in any environment. And it is because of the diversity of environments that Shiro supports a number of configuration mechanisms suitable for its own configuration. However, there are mainly two types of configuration supported by Shiro core: Programmatic configuration and INI configuration.

3.2.1 Programmatic Configuration

The simplest way to create a SecurityManager and make it available to the application is to create a DefaultSecurityManager and wire it up in the code presented in listing 1.

```
Realm realm = //instantiate or acquire a Realm instance.  We'll dis-
cuss Realms later.

SecurityManager securityManager = new DefaultSecurityManager(realm);

//Make the SecurityManager instance available to the entire applica-
tion via static memory
SecurityUtils.setSecurityManager(securityManager);
```

Listing 1 - Creation of DefaultSecurityManager [18]

Listing 1 shows how just a few lines of code enable and configure a functional Shiro environment suitable for many applications.

Nevertheless, if custom configuration is needed, Shiro's SecurityManager implementations are essentially a modular object graph of nested security-specific components, thus, it is possible to use the getter and setter methods of any of the components to configure the SecurityManager and any of its internal object graphs.

```
DefaultSecurityManager securityManager = new DefaultSecurityManager(
realm);

SessionDAO sessionDAO = new CustomSessionDAO();

((DefaultSessionManager)securityManager.getSessionManager())
.setSessionDAO(sessionDAO);
```

Listing 2 - Customization of DefaultSecurityManager [18]

Listing 2 presents how to configure the SecurityManager instance in case that custom Session Management is needed, with the use of a custom SessionDAO, setting the SessionDAO directly with the nested SessionManager's setSessionDAO method.

3.2.2 INI Configuration

It is possible to build the SecurityManager object graph with the use of the INI format. Shiro supports this format, and so it ensures a text-based mechanism that works in all environments with almost no 3rd party dependencies. INI suits most applications well, it is simple enough to read and configure, and it encapsulates the configuration logic so that it is possible to modify it anytime without any changes on the source code. Furthermore, it is possible to create the SecurityManager instance from an INI resource path, where the path is configured via prefix keyword and file name. There are three possible prefix options: ‘file:’ for file system, ‘classpath:’ for classpath or ‘url’ for a url path.

```
Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro.ini");

SecurityManager securityManager = factory.getInstance();

SecurityUtils.setSecurityManager(securityManager);
```

Listing 3 - SecurityManager instantiation [18]

If desired, the INI configuration can be built programmatically as well, via the Ini class (see listing 4 below). This class works nearly the same than the Properties class included in JDK, but additionally supports segmentation by section name.

```
Ini ini = new Ini();
//populate the Ini instance as necessary
...

Factory<SecurityManager> factory = new IniSecurityManagerFactory(ini);

SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

Listing 4 - SecurityManager programatic instantiation [18]

The INI file used for Shiro’s configuration is a simple text file and its content is just a list of key/value pairs group within sections, being those keys unique per section only, as opposed to the common used properties files where the keys are unique over the entire configuration.

There are four possible sections in Shiro’s INI file:

- main
- users
- roles
- urls

These four sections are described below.

[main]

The [main] section is where the configuration of the application's SecurityManager instance, and any of its dependencies, occurs. It sounds a complex task to setup object instances through the INI file, where it is only possible to use name/value pairs. However, Shiro uses a mechanism based on convention-over-configuration to enable a simple and effective configuration mechanism.

Defining an object

The example below in listing 5 shows how to define an object through the INI file.

```
[main]
myRealm = com.company.shiro.realm.MyRealm
...
```

Listing 5 - Defining an object [18]

The example in listing 5 shows how to instantiate a new object instance of type com.company.shiro.realm.MyRealm and how to relate myRealm name to that object and make it available for future reference and configuration if needed.

Setting object properties

In the following example (listing 6), the lines of configuration are translated into setter methods which are called in application runtime by Apache Shiro. This happens because of the assumption that all objects are Java POJOs and Beans-compatible, through the use of convention as mentioned before in this section.

```
myRealm.connectionTimeout = 30000
myRealm.username = jochoa
```

Listing 6 – Setting primitive values [18]

For this mechanism to work, Shiro uses another Apache project called Apache Commons BeanUtils, which converts the text values from the INI file into the proper primitive types and invoke the setter methods.

Reference values

With Apache Shiro, it is possible to set not only primitive values but also objects.

```
sha256Matcher =
org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
myRealm.credentialsMatcher = $sha256Matcher
```

Listing 7 - Setting object values [18]

As shown in listing 7, the use of a dollar sign ($) enables a mechanism to reference a previously-defined instance. For this “magic” to happen, Shiro takes advantage of BeanUtils once more and it sets the object defined by the name sha256Matcher to the object on the myRealm instance by calling the setter method for that property.

Nested properties

There may be cases where the final object or property to set is not directly a subelement of the main object in hand.

```
securityManager.sessionManager.globalSessionTimeout = 1800000
```

Listing 8 - Setting a nested property [18]

For those cases, Shiro is prepared to understand a multiple dotted notation, and if founded, it will navigate the object graph until it gets the real final object or property to set (see listing 8 for an example).

Collection properties

Lists, Sets and Maps are taken into consideration by Apache Shiro and can be set like any other property.

```
sessionListener1 = com.company.my.SessionListenerImplementation
...
sessionListener2 = com.company.my.other.SessionListenerImplementation
...
securityManager.sessionManager.sessionListeners = $sessionListener1,
$sessionListener2

...
object1 = com.company.some.class
object2 = com.company.another.class
...
anObject = some.class.with.a.Map.property
anObject.mapProperty = key1:$object1, key2:$object2
```

Listing 9 - Setting collections [18]

Listing 9 presents several examples of how to configure collections using the INI file.

Order matters

As shown in listing 9, Apache Shiro uses formats and conventions to easy the use of the INI file for configuration purposes. But it is very important to understand that the order of the lines in INI file matters. Apache Shiro executes each instantiation and assignment in the order it reads them from the [main] section of the INI file, since each line will be translated to setter and getter method calls and then executed.

Overriding instances

Because the order of the INI file lines matters, an object will be overwritten if the same name is used to define a new instance within the same configuration.

```
myRealm = com.company.security.MyRealm
...
myRealm = com.company.security.DatabaseRealm
```

Listing 10 - Overriding an instance [18]

Listing 10 shows how the object myRealm is being instanced twice, being the value of the second instance the one that remains for application purposes. The first instantiation is never used and is considered as if never existed.

[users]

The [users] section allows defining a collection of user accounts. This mechanism is very useful in applications with a small number of user accounts or where user accounts do not need to be created dynamically.

```
[users]
admin = jsmith
simpsons = homer, marge, bart
flinstones = wilma, fred
```

Listing 11 - Defining users [18]

Each line in the user section must follow the conventional format understood by Shiro (listing 11 shows an example of this format):

username = password, roleName1, roleName2, ..., roleNameN

In the previous expression there can be only one value on the left side of the equals sign, and this value would be considered as the username. On the right side of the equals sign there can be multiple comma-separated values. The first value would be taken as the user's password, and it is always required. Any other values after the password are understood as the names of roles assigned to that user and these values are optional. It is possible to encrypt the passwords using any hash algorithm (MD5, Sha1, Sha256, etc.) and use the final result string as the password instead plain-text. Listing 12 shows a simple example of how to use encrypted passwords for user within INI file.

```
[main]
...
sha256Matcher =
org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
iniRealm.credentialsMatcher = $sha256Matcher
...

[users]
user1 = <sha256 encoded password>, role1, role2, ...
```

Listing 12 - Encrypted password for users [18]

In order to use encrypted passwords in INI file, it is required to configure the implicitly created iniRealm in the [main] section. This will indicate Shiro that the passwords are encrypted and which hash algorithm implementation to use (done through the instantiation of the CredentialsMatcher).

[roles]

The [roles] section is where the association of Permissions with the roles is defined. Similar to [users] section, [role] section is useful in applications with a small number of roles or where roles don't need to be created dynamically.

```
[roles]
# 'admin' role has all permissions, indicated by the wildcard '*'
admin = *

# The 'simpsons' role can do anything (*) with any donut:
simpsons = donut:*

# The 'flinstones' role is allowed to 'drive' (action) the footcar
# (type) with license plate 'fredrock' (instance specific id)
flinstones = footcar:drive:fredrock
```

Listing 13 - Different role cases [18]

Each line in this section has to define a role-to-permission(s) key/value mapping through the following conventional format (see listing 13 above):

rolename = permissionDefinition1, permissionDefinition2, ..., permissionDefinitionN

If a role does not require any association to permissions, it will not be needed to list it.

[urls]

This section is described in the Apache Shiro Web Support chapter 5.

4. Apache Shiro Core

4.1 Authentication

Authentication is the process that verifies identities. For users to prove their identities, some identity-related information is needed so that systems can understand and trust that user.

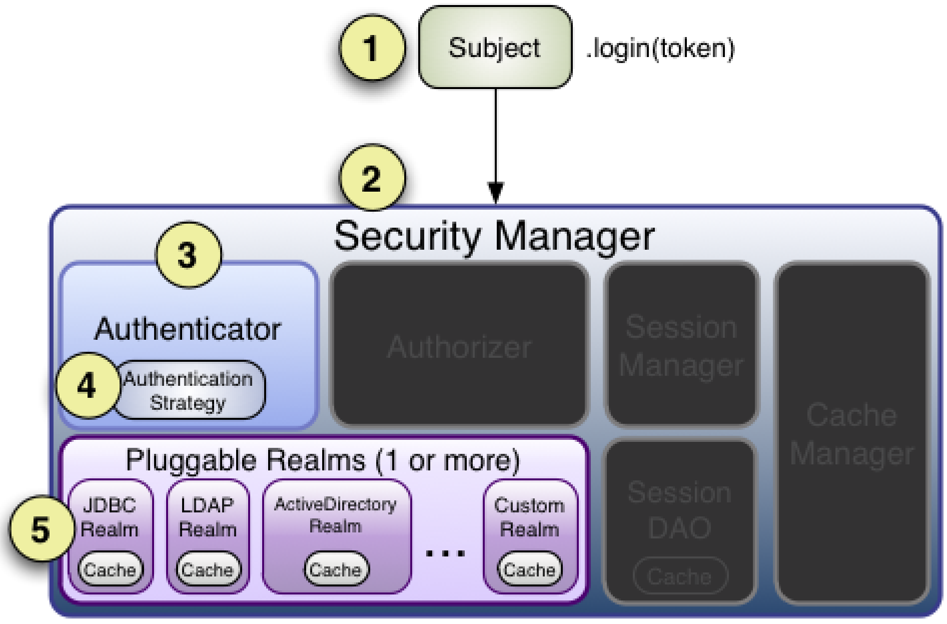

Figure 6 - Apache Shiro's authentication sequence [19]

In Shiro, the process is done by submitting what is called the user's principals and credentials, so that Shiro can match those against what is expected by the application (figure 6 graphically illustrates this process).

Les Hazlewood, one of the founders of Apache Shiro, explains these two terms as follows [19]:

- Principals are attributes that identify a Subject, such as a first name, a username, Social Security Number, etc.

- Credentials are secret values which normally are only known by the Subject itself and which are used to verify that the Subject is the actual real owner of that identity. Credentials can have multiple forms such as passwords, fingerprints or certificates.

One of the most common examples of a principal and credential combination, if not the most common, is the username and password pair. The username is considered as the identity being claimed and the password is then considered as the proof that matches the username or claimed identity. Once the password matches what the application is expecting, Shiro will assume that the user really is who claims to be (out of the assumption that no one else should know what that password is) [19].

4.1.1 Subject's Authentication with Shiro

There are certain steps to take in order to fulfill the process of Subject's authentication using Shiro's API. First, Shiro needs to be provided with both, Subject's principals and credentials. Shiro does not care how the information for authentication is acquired. The process of collecting information from an application end-user is completely separated. An example of how to easily provide Shiro with Subject's information can be done through UsernamePasswordToken class (see listing 14), which supports the most common username/password authentication approach.

```
//Most common scenario of username/password
UsernamePasswordToken token = new UsernamePasswordToken(username,
password);
```

Listing 14 - Providing Subject's information programatically [19]

The official documentation of Shiro defines UsernamePasswordToken as follows:

> A simple username/password authentication token to support the most widely-used authentication mechanism. This class also implements the RememberMeAuthenticationToken interface to support "Remember Me" services across user sessions as well as the HostAuthenticationToken interface to retain the host name or IP address location from where the authentication attempt is occurring.
>
> "Remember Me" authentications are disabled by default, but if the application developer wishes to allow it for a login attempt, it is as easy as to call setRememberMe(true). If the underlying SecurityManager implementation also supports RememberMe services, the user's identity will be remembered across sessions [20].

Listing 15 presents the same example as in listing 14 with “Remember Me” capabilities enabled. This example ensures that Shiro will remember the user identity when it returns to the application later on.

```
//Most common scenario of username/password
UsernamePasswordToken token = new UsernamePasswordToken(username,
password);

//”Remember Me” functionality enabled
token.setRememberMe(true);
```

Listing 15 - Enabling "Remember Me" capabilities programatically [19]

The next step to authenticate a Subject through Shiro, is to submit the collected information (principals and credentials represented as an AuthenticationToken instance). To perform the authentication attempt, it is required to gather the current Subject and call its method login, passing the token with the Subject’s information.

```
Subject currentUser = SecurityUtils.getSubject();
currentUser.login(token);
```

Listing 16 - User authentication programatically [19]

An invocation to the login method always represents an authentication attempt, as seen in listing 16. Finally, the last step is to handle the result of that authentication attempt, which may be a success or a failure. If the call to the login method returns quietly, then the Subject has been successfully authenticated. From this moment, any call to getSubject method in Shiro's SecurityUtils will return an instance of that authenticated Subject.

However, it is possible that the authentication attempt failed, and to help in such a situation, Shiro supplies a full exception hierarchy to catch and indicate exactly what the problem was when authenticating the Subject.

```
try {
    currentUser.login(token);
} catch ( UnknownAccountException uae ) { ...
} catch ( IncorrectCredentialsException ice ) { ...
} catch ( LockedAccountException lae ) { ...
} catch ( ExcessiveAttemptsException eae ) { ...
} ... catch own exception ...
} catch ( AuthenticationException ae ) {
    //unexpected error?
}

//No problems in authentication, continue on as expected…
```

Listing 17 - Shiro's exception hierarchy [19]

Additionally, Shiro's AuthenticationException class can be used for those cases when none of the exception classes in listing 17 handle the needs of the authentication failure as required by the application.

4.1.2 Distinction between “Remembered” and “Authenticated”

As shown in section 4.1.1, Shiro supports "Remember Me" and so it makes a distinction between the notion of remembered Subject and the notion of authenticated Subject.

Hazlewood (2010) defines these two concepts as:

- Remembered Subject is such Subject that has a known identity, but which identity is remembered because it did authenticate successfully in a previous session.

- Authenticated Subject is a Subject that has been successfully authenticated during the current session [19].

Shiro provides with the needed API to know if the Subject is remembered or authenticated.

- A subject is considered remembered if the call to Subject instance’s method isRemembered returns true.

- A subject is considered authenticated if the call to Subject instance’s method isAuthenticated returns true [19].

Even after explaining both concepts, it may still be somewhat unclear where the difference between them lays. Hazlewood uses Amazon.com as an example to illustrate this difference.

Let us say that a user logs in successfully into Amazon.com from and adds some books to the shopping cart. However, the user leaves for a meeting and forgets to log out, and by the time the meeting is over it is late and the user decides to go home and leave the office. The next day, when the user comes back to work, the user realizes that the purchase of the books is incomplete and goes back to Amazon.com. This time, Amazon.com remembers who the user is, greeting the user name and shows the shopping cart with the selected books and some additional recommendations. To Amazon.com, the user is remembered. The user accesses the shopping cart and wants to update the credit card information, in order to finish the order at hand. At this point, even though Amazon.com remembers who the user is, it cannot guarantee that, in fact, the user is who the system remembers (for example a co-worker might be using the same computer). So to perform that kind of sensitive action, such as updating credit card information, Amazon.com will request to the user to login so that it can guarantee the identity of the user. After a successful login, the user will be verified and Amazon.com will then take the user as authenticated [19].

Even though Shiro provides users with a solution for this type of scenarios, it depends on the developers how the workflows and views will respond when Subjects are remembered or authenticated.

4.1.3 Logging Out

Once the Subject does not need any further interaction with the application, a logout action should be performed in order to release all known identifying state.

Shiro’s Subject instance provides a logout method (see listing 18) that will invalidate any existing Session and disassociate any identity information from the Subject.

```
currentUser.logout();
```

Listing 18 – Log out programatically [19]

After a log out, the Subject instance is considered, once again, anonymous again and can login again if wanted.

4.2 Authorization

Authorization, also known as Access Control, is the process of regulating access to resources and control “who can do what” (Figure 7 illustrates this process). Allowing or denying access resources and functions will depend on the roles and permissions assigned to the Subject during authentication.

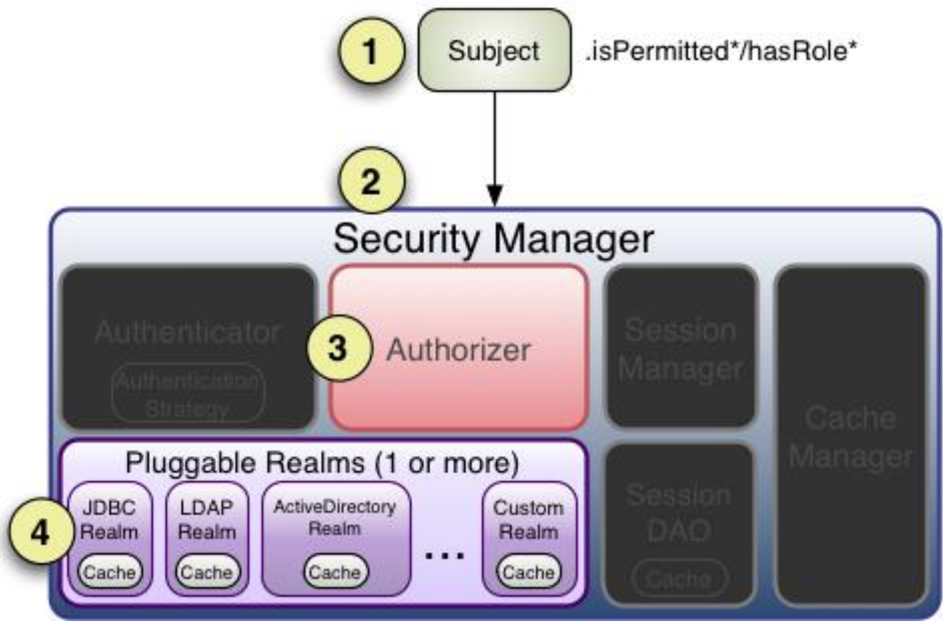

Figure 7 - Apache Shiro's authorization sequence [21]

Authorization in Shiro uses three core elements [21]:

- Permissions
- Roles
- Users

The core elements will be described below.

Permissions

Shiro interprets Permissions as statements that describe behavior and raw functionality in an application, representing explicitly what can be done. Permissions are the lowest-level of granularity within security policies. A well-formed permission statement essentially defines the junction between resources and the possible actions that a Subject can execute against those resources. Probably the most important thing to realize about permissions is that they represent only behavior and they have no notion of who can perform that behavior.

Some examples of permission statements include the following:

- Open a file
- View the '/user/list' web page
- Print documents
- Delete the 'jsmith' user

Shiro provides a powerful and intuitive permission syntax, which is known as the WildcardPermission, to enable easy-to-process permission statements, being human readable at the same time.

```
printer:print
```

Listing 19 – Permissions

Listing 19 example grants a user with the permission to print in a printer (The colon is the special character to delimit the different parts of a permission string).

It is possible to add multiple permissions over the same resource:

```
printer:print
printer:manage
...
```

Listing 20 - Multiple permissions (multiple lines)

The number of parts that can be used is limitless, so each part can contain multiple values. This means that the example in listing 20 can be written as:

```
printer:print, manage
```

Listing 21 - Multiple permissions (comma separated)

Both listing 20 and listing 21 grants the user a unique permission that allows printing and managing a printer.

Additionally, wildcard permissions can be used to model instance-level Access Control Lists. In this case the permissions will have three parts: the domain, the action(s), and the instance(s) to act upon.

```
printer:manage:lp7200
printer:print:epsoncolor
```

Listing 22 - Instance level access control

In listing 22, the first permission defines the behavior to allow the management of the printer with id *lp7200*. The second permission defines the behavior to al-low printing in the printer with id *epsoncolor*. When these permissions are granted to users, they will be allowed to specific behavior on specific instances.

And the wildcard token (*) can be use in any part of a wildcard permission string.

```
*:manage
```

Listing 23 - Wildcard for permissions

If listing 23 example is included to Ini file, any user will be able to use the “manage” action across all domains (not just printers), if granted with such permission.

Roles

Shiro interprets a role as an application unique name integrating a set of behaviors or responsibilities, that is, one or more Permission declarations. Roles are normally assigned to user accounts and so, users are allowed to do those actions attributed to the roles which have been assigned to them.

Shiro supports two types of Roles:

- **Implicit Roles**: Roles which define what set of behaviors the user is allowed to perform (for example a user with role X assigned to it is then allowed to perform behavior A, B and C) [21].

- **Explicit Roles**: Roles which are used as reference for a collection of permissions which then define what type of actions the user is allowed to perform. It is then the application’s job to know what means for a user to have certain role [21].

Users (Subjects)

Subject is the real Shiro’s User concept, which essentially is the “who” of an application. Users (Subjects) can perform actions in the application depending on their associations with roles or direct permissions. It is in the application's data model where it is defined exactly when and how a Subject is allowed to do something or not.

4.2.1 Programmatic Authorization

Interacting directly with the current Subject instance is probably the easiest way to perform authorization and preferred one among developers. A subject instance provides methods to execute role checks, and to enable control access via implicit role names (see listing 24 below).

```
Subject currentUser = SecurityUtils.getSubject();

if (currentUser.hasRole("administrator")) {
    //do some administration
} else {
    //avoid any administration
}
```

Listing 24 - Role check programatically [21]

As an alternative, Shiro’s Subject instance provides methods to assert that the Subject has an expected role before certain logic is executed. The assertion will execute quietly if the Subject has the expected role (see listing 25). If not, an AuthorizationException will be thrown stopping any further action.

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is an administrator and
//therefore allowed to create a user
currentUser.checkRole("administrator");
createUser();
```

Listing 25 - Role assertion check [21]

Shiro implements similar behavior for Permissions to the one explained above for Roles. To check if a Subject is permitted to do a certain action, Subject instance provides the method isPermitted.

```
Permission printPermission = new PrinterPermission("laserjet4400n",
"print");

Subject currentUser = SecurityUtils.getSubject();

if (currentUser.isPermitted(printPermission)) {
    //show the Print button
} else {
    //don't show the button
}
```

Listing 26 - Permission check programatically [21]

Listing 26 shows how to instantiate an instance of Shiro’s Permission interface and pass it to the user to verify if it accepts that permission instance. At the same time, as listing 26 shows, Shiro has the ability to restrict behavior using instance-level access control checks, based on individual data instances. Because this approach can feel a bit too complex for some applications, Shiro allows the use of strings for permission checking (see listing 27).

```
Subject currentUser = SecurityUtils.getSubject();

if (currentUser.isPermitted("printer:print:laserjet4400n")) {
    //show the Print button
} else {
    //don't show the button
}
```

Listing 27 - Permission-as-string check [21]

As with Roles, Shiro’s Subject instance provides methods to assert that the Subject has an expected permission before certain logic is executed. Again, the assertion will execute quietly if the expected permission is supported by the Subject, but if not, an AuthorizationException will be thrown.

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is permitted to create a user
Permission p = new UserPermission("create");
currentUser.checkPermission(p);
createUser();
```

Listing 28 - Permission assertion check [21]

This is the same checking example than listing 28 but using the string permission.

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is permitted to create a user
currentUser.checkPermission("user:create");
createUser ();
```

Listing 29 - Permission-as-string assertion check [21]

Listing 29 reveals a solution for permission checking that is easier for human readability.

4.2.2 Annotation-based Authorization

Besides the Subject API calls, Shiro provides a collection of Java annotations, but it is necessary to enable AOP support in the application prior to the use of these annotations.

The RequiresAuthentication annotation

A method annotated with RequiresAuthentication will only be invoked if it is guaranteed that the Subject is authenticated [22].

```
@RequiresAuthentication
public void updateAccount(Account userAccount) {
    ...
}
```

Listing 30 - RequiresAuthentication annotation [21]

Listing 30 shows an example use of this annotation.

The RequiresGuest annotation

A method annotated with RequiresGuest will only be invoked if the Subject is unknown or anonymous [23].

```
@RequiresGuest
public void signUp(User newUser) {
    ...
}
```

Listing 31 - RequiresGuest annotation [21]

Listing 31 shows an example use of this annotation.

The RequiresPermissions annotation

A method annotated with RequiresPermissions will only be invoked if the Subject is guaranteed with the permission specified as argument in the annotation [24].

```
@RequiresPermissions("account:create")
public void createAccount(Account account) {
    ...
}
```

Listing 32 - RequiresPermissions annotation [21]

Listing 32 shows an example use of this annotation.

The RequiresRoles annotation

A method annotated with RequiresRoles will only be invoked if the Subject is guaranteed with the role specified as an argument in the annotation [25].

```
@RequiresRoles("administrator")
public void deleteUser(User user) {
    ...
}
```

Listing 33 - RequiresRoles annotation [21]

Listing 33 shows an example use of this annotation.

The RequiresUser annotation

A method annotated with RequiresUser will only be invoked if the Subject is guaranteed to have a known identity, either because of being authenticated during the current session or because the application remembers the user from a previous session [26].

```
@RequiresUser
public void updateAccount(Account account) {
    ...
}
```

Listing 34 - RequiresUser annotation [21]

Listing 34 shows an example use of this annotation.

4.3 Realms

“A Realm is essentially a security-specific Data Access Object (DAO)” [17]

As specified in Shiro’s official documentation, a Realm is a component that access security data which is specific to the application such as users, roles or permissions. Once the Realm reaches this data, it translates it into a format that can be understood by Shiro, enabling the possibility to provide with a unique Subject API without dependencies to the amount of data sources used or the security data structure that each of those sources provide.

Usually, a Realm is associated to one data source only, such as file system, relational database or other type of resource. This association requires that, to be able to discover the authorization data, the Realm must be implemented using APIs which are specific to the data source, such as JDBC, File IO and other Data Access APIs.

4.3.1 Configuration

Realms’ configuration is defined and referenced using Shiro’s INI configuration, specifically into [main] section, and configured as part of the securityManager.

```
fooRealm = com.company.foo.Realm
barRealm = com.company.bar.Realm
bazRealm = com.company.baz.Realm

securityManager.realms = $fooRealm, $barRealm, $bazRealm
```

Listing 35 - Realms configuration [27]

For a simple Realm configuration, a definition of one or more Realms is needed, followed by setting them as a collection property on the securityManager object, as seen in listing 35. This configuration allows full control of which realm is used and in which order they will be used.

4.3.2 Credentials Matching

When a Realm is used for Subject's authentication, is its job to make sure that the credentials being submitted by the Subject are a match to the credentials stored in the data store. As specified in Shiro's authentication workflow, if credentials match what the application is expecting then the authentication is considered successful and the system has verified the identity of the user. If not, an AuthenticationException is thrown.

For authentication purposes and due to the fact that credentials matching process is almost identical in all applications, Shiro provides a set of classes (AuthenticatingRealm and subclasses) which support the concept of a CredentialsMatcher to perform credentials comparison. A CredentialsMatcher is an interface that can determine if the credentials provided by a certain AuthenticationToken are a match to a corresponding account's credentials stored in the system.

Listing 36 provides an example of how to set a CredentialMatcher for a Realm:

```
[main]
...
customMatcher = com.company.shiro.realm.CustomCredentialsMatcher
myRealm = com.company.shiro.realm.MyRealm
myRealm.credentialsMatcher = $customMatcher
...
```

Listing 36 - Setting CredentialMatcher for Realm [27]

Shiro uses SimpleCredentialsMatcher class by default for its Realm implementations. The SimpleCredentialsMatcher class verifies that the credentials submitted in the

AuthenticationToken are equals to the stored account credentials by doing a plain direct equality check. It can perform direct quality comparison using most of the traditional byte sources, such as Strings, byte arrays, Files or InputStreams.

In any case, it is more secure to hash any end-user's credentials when storing them in the data store. Shiro provides HashedCredentialsMatcher implementations to support the preferred cryptographic hashing strategies, and Realms can be configured to use implementations instead the SimpleCredentialsMatcher .

```
[main]
...
credentialsMatcher =
org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
myRealm = com.company.shiro.realm.MyRealm
myRealm.credentialsMatcher = $credentialsMatcher
...
```

Listing 37 - Setting CredentialMatcher for Realm [27]

Listing 37 explains how to configure a Realm to use a SHA-256 algorithm CredentialsMatcher. Shiro provides, out-of-the-box, CredentialsMatcher implementations for many other hash algorithms such as SHA-512, MD5 and others.

4.4 Cryptography

Cryptography is the practice of protecting information through methods such as hiding it or converting it into nonsense, and this way keep it safe from undesired access. The methods for applying Cryptography within Java field has being, traditionally, using Java Cryptography Extension (JCE). JCE is an API that provides a framework to easy the implementation of security features within Java applications. It enables features such as encryption, key generation and key agreement, and MAC algorithms.

JCE can be complicated and difficult to use and may require a certain level of expertise (JCE architecture is illustrated in Figure 8. Thus, Shiro implements a Cryptography API which is much easier to understand and use, simplifying JCE concepts, without sacrificing any functionality, since it is still possible to access to more complicated JCE options if needed.

One of the examples that show the complexity in the use of JCE architecture is the Cipher class. The Cipher class is an abstract class, which means that it is needed to use some of the obtuse factory methods provided by JCE to acquire an instance in order to use it. This process can be confusing and it requires type-unsafe string arguments.

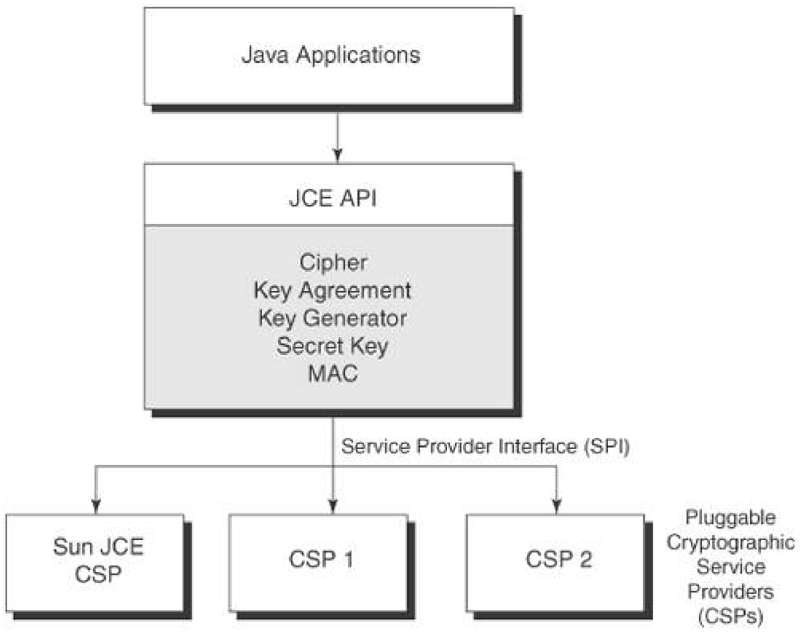

Figure 8 - JCE architectural model and its cryptographic services [28]

Listing 38 shows how to encrypt a simple string with an MD5 algorithm using JCE and getting the final result as a hexadecimal string.

```
String str = "example";
StringBuffer hexString = new StringBuffer();
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] hash = md.digest(str);
for (int i = 0; i < hash.length; i++) {
  if ((0xff & hash[i]) < 0x10) {
    hexString.append("0" + Integer.toHexString((0xFF & hash[i])));
  } else {
    hexString.append(Integer.toHexString(0xFF & hash[i]));
  }
}
```

Listing 38 - MD5 encryption using JCE

Shiro bases its Ciphers and Hashes on a clean object hierarchy, allowing their use by simply instantiation. Listing 39 shows the same case than listing 38, but using Shiro's Cryptography API.

```
String str = "example";
StringBuffer hexString = new MD5Hash(str).toHex();
```

Listing 39 - MD5 encryption using Shiro

Shiro takes Cryptography, a traditionally extremely complex field, and makes it easy for any developer, experienced or not in cryptography, while providing a robust set of cryptography features.

For cryptography, Shiro focuses on two core elements:

- Ciphers
- Hashes

The core elements will be briefly described in the following.

4.4.1 Ciphers

A cipher is an algorithm to perform encryption or decryption. The algorithm usually requires of a piece of information named as key, which directly affects in the encryption of the data. The encryption will vary depending on the key making the decryption extremely difficult without it.

There are different variations of ciphers:

- A Block Cipher is an encrypting method that requires a cryptographic key and the algorithm to use, and which is applied to the whole block of data at once [29, 93].

- A Stream Cipher is an encrypting method that requires a cryptographic key and the algorithm to use, and which is applied to each of the binary digits contained in the data stream independently [29, 93].

- A Symmetric Cipher uses the same (necessarily secret) key to encrypt messages as it does to decrypt messages (see figure 9) [30, 751].

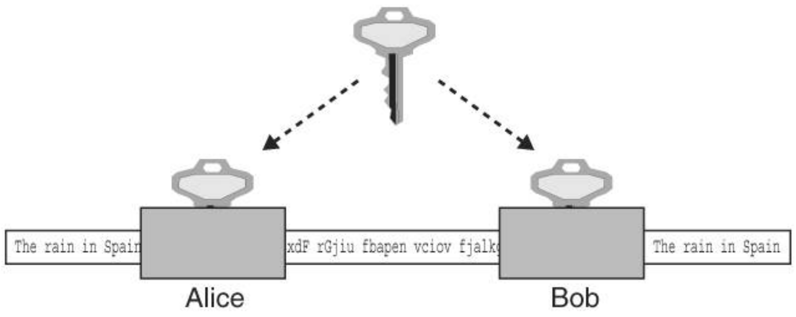

Figure 9 - Encryption using a symmetric cipher [28]

- An Asymmetric Ciphers uses different keys to encrypt and decrypt messages. In case that is not possible to derived one key from the other, then one of the keys can be shared publicly just by creating public/private key pairs (see figure 10) [30, 751].

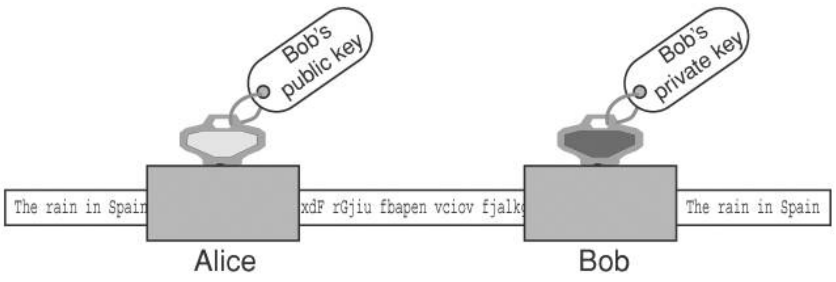

Figure 10 - Encryption using an asymmetric cipher [28]

Unlike JCE, the representation of the different Ciphers within Shiro follows an Object-Oriented class hierarchy, matching the mathematical concepts of each of them, which provides an easy way to extend functionality simply by overriding existing classes. Shiro Ciphers' usability is easier than the ones provide by JCE. With Shiro, an instantiation of a class, with an optional simple configuration with JavaBeans properties, is enough to start using any type of Cipher; meanwhile JCE requires confusing factory methods using type-unsafe string token arguments [31].

Shiro incorporate more secure default settings than JCE does. By default, Shiro Cipher instances will automatically enable the most secure options available as opposed to JCE Cipher instances, which assume defaults by the 'lowest common denominator'. Shiro makes sure that any data is as safe as possible and prevents from accidental security holes [31].

4.4.2 Hashes

A Hash function is a conversion of a given input source into an encoded hash value. It is a one-way irreversible method and it is often used to encrypt sensible data such as passwords or digital fingerprints. The input source is usually called "message" and the output hash value is called “message digest”. Java Development Kit addresses Hashes as Messages Digests [32].

Shiro provides default Hash implementations out-of-the-box, including the most common ones like MD5, SHA1, SHA-256, etc., with type-safe construction methods, instead of being forced to use type-unsafe string factory methods, and additional methods to encode hash data to Hex and Base-64. Furthermore, Shiro's Hash implementations support salts and multiple hash iterations [31].

Salts are random data which are often used in hash functions as additional input.

```
new MD5Hash("my-password", "my-secret-salt", 1024).toBase64();
```

Listing 40 - Salts and repeated hash iterations

Listing 40 shows how Shiro simplifies and helps using salts and hash iterations, two valuable tools when it comes to hashing data, especially in cases such as user passwords.

5. Apache Shiro Web Support

5.1 Configuration

There are many ways to integrate Shiro into any web application, but possibly the easiest way is to configure a Servlet ContextListener and Filter in the web.xml file, which teaches the application how to read Shiro's INI configuration.

```
<listener>
    <listener-class>
        org.apache.shiro.web.env.EnvironmentLoaderListener
    </listener-class>
</listener>
...
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>
        org.apache.shiro.web.servlet.ShiroFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

Listing 41 - Shiro configuration in web.xml [33]

Listing 41 includes the required code-lines in web.xml file for standard web applications to initialize Shiro.

Shiro’s initialization process has three steps:

1. With the help of the EnvironmentLoaderListener, a Shiro WebEnvironment instance gets initialized and accessible in the ServletContext. This instance contains all configurations needed by Shiro to operate.

2. After initialization, ShiroFilter uses the WebEnvironment to perform all necessary security operations for any filtered request.

3. Finally, the filter-mapping definition ensures that all requests are filtered by the ShiroFilter.

Apache Shiro assumes that the INI Configuration file is located at one of the following two locations, and it uses the one which is found first:

- /WEB-INF/shiro.ini
- shiro.ini file at the root of the classpath.

5.1.1 Web INI Configuration

Besides the standard sections, it is possible to add an additional [url] section in shiro.ini file, which is used for specific web purposes (see listing 42 below).

```
# [main], [users] and [roles] above here
...

[urls]

/index.html = anon
/user/create = anon
/user/** = authc
/admin/** = authc, roles[administrator]
```

Listing 42 - Url section in INI file [33]

Each of the url paths of the application added to the [url] section are automatically referenced to the filter chain specified in the same line. The lines in the [url] section must follow the following format:

Url_Path_Expression = Path_Specific_Filter_Chain

Url Path Expressions

Url path expressions are divided using the equal sign (=) as tokenizer character. The left token is the path expression relative to the web application's context root.

```
/user/** = authc
```

Listing 43 - Url path expression [33]

The expression in listing 43 is understood for Shiro so that any request to the application's path of "/user" or any of its sub paths, as "/user/list", "/user /add", etc., will trigger the authc filter chain.

```
/user/** = authc
/user/list/** = anon
```

Listing 44 - Multiple url path expression [33]

Shiro does evaluate url path expressions following the order in which they are defined and it selects the first match which is found. In listing 44, if an anonymous user intends to reach the url "/user/list" within the application context, the incoming request will never be handled, and this anonymous user will not be allowed to reach that url. The reason is that the "/user/**" pattern matches the incoming request first, and so it will require authentication in order to reach any sub path of the pattern, including the requested "/user/list" url.

Filter Chain Definitions

The right token of a url path expression is a comma-delimited list of filters which are executed for the request that specifically matches that same path, and which are executed in the same order they are defined. This list of filters must follow this format:

filter1[optional_config1], filter2[optional_config2], ..., filterN[optional_configN]

filterN represents the name of a filter bean which must be previously defined within the [main] section. [optional_configN] is an optional string which has a specific meaning in the context of that filter and only for that particular path.

As with url path expressions, the order matters because the filter tokens define a list of filters to use. The comma-delimited filter tokens must be defined in the order that the request has to be handled. Shiro creates some default filter instances in case of web applications, and makes them available in the [main] section automatically. These filters can be configured as any other bean in [main] section and reference in the chain definitions.

```
[main]
...
authc.loginUrl = /login.jsp
...

[urls]
...
/user/** = authc
...
```

Listing 45 - Default filter configuration [33]

It is important to note that listing 45 there is no definition of any class for the FormAuthenticationFilter (authc), but it is already instantiated and available for configuration. If any anonymous user intents to access the path “/user” or any sub path of it, it will be redirected to the page that authc.loginUrl parameter is configured with.

The default filter instances that Shiro provides are:

- anon: Allows immediate access to a path without the need of performing any kind of security checks [34].

- authc: It requires the user to be authenticated to continue with the request. In case the user is not authenticated, this filter redirects the user to the configured loginUrl [35].

- authcBasic: It requires the user to be authenticated to continue with the request. In case the user is not authenticated, it forces the user to login via the HTTP Basic protocol challenge. The user is then allowed to continue on to the requested resource/url once it successfully logs in [36].

- logout: Immediately log out the currently executing subject and then redirect them to a configured redirectUrl. [37].

- noSessionCreation: Disables creating new Sessions during the request [38].

- perms: The user is only given access if it has all the permissions specified by the mapped value. If not, the access is denied [39].

- port: The request must be using the port defined for this filter. If not, the request is redirected to the same url on that specified port [40].

- rest: Translates an HTTP Request's Method into a corresponding action and constructs a permission that will be checked to determine access [41].

- roles: The user is only given access if it has all the roles specified by the mapped value. If not, the access is denied [42].

- ssl: Requires a request to be over SSL. Access is allowed if the request is received on the configured server port and the request is secure [43].

- user: Access is granted to the user only in the case of being a known user. In order for this to happen, the user needs to be authenticated or to be remembered via the 'remember me' feature [44].

5.2 Remember Me Services

Shiro will automatically perform 'rememberMe' services if the AuthenticationToken used in the login process implements the RememberMeAuthenticationToken interface. This interface specifies the isRememberMe method, which tells Shiro to remember, or not, the end-user's identity across sessions.

RememberMe services can be used programmatically with just setting the value to true in any class that supports such configuration (for an example, please check listing 15). Additionally, RememberMe services are also support in Shiro by reading the boolean value of the parameter ‘RememberMe’ which comes from the form or request. In web applications, FormAuthenticationFilter is the default class for the authc filter.

```
[main]
authc.loginUrl = /login.jsp

[urls]
# This line indicates which is the login form page
login.jsp = authc
```

Listing 46 - Setting login form page [33]

Listing 47 contains the required HTML code within the web form configured in listing 46, which must have a checkbox named 'rememberMe'.

```
<form ...>
  Username: <input type="text" name="username"/>
  <br/>
  Password: <input type="password" name="password"/>
  ...
  <input type="checkbox" name="rememberMe" value = "true"/>
  Remember Me?
  ...
</form>
```

Listing 47 - HTML login form [33]

FormAuthenticationFilter looks by default for the parameters ‘username’, ‘password’ and ‘rememberMe’ from within the request parameters.

```
[main]
...
authc.loginUrl = /someOtherLoginPage.jsp
authc.usernameParam = somethingOtherThanUsername
authc.passwordParam = somethingOtherThanPassword
authc.rememberMeParam = somethingOtherThanRememberMe
...
```

Listing 48 - Login form fields custom configuration [33]

If the requested parameters are different than the field names used in the web form, a customize configuration will be needed, as presented in listing 48.

5.3 JSP/GSP Tag Library

Apache Shiro provides a tag library compatible with JSP and GSP technologies, which allows customization of the page output based on the current Subject’s state. With this tool, Shiro enables the capabilities to personalize views based on the current user, through its identity and authorization state. These tags get activated and ready to use by simply adding the following line to the top of the page (or any other customize place page directives are defined):

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
```

Listing 49 - JSP Shiro tag namespace [33]

Listing 49 uses the “shiro” prefix to indicate the shiro tag library namespace, but it is possible to assign any other name as namespace for the tag library. A review of each tag and the functionality they provide will be given in the following.

The guest tag

The guest tag displays its content only in case the current Subject is considered a 'guest', meaning that the Subject that does not have an identity. This happens if the user is neither authenticated nor remembered from a previous site visit [33].

```
<shiro:guest>
    Wrapped content to be displayed!
</shiro:guest>
```

Listing 50 - Guest tag [33]

Listing 50 presents an example of how to use this tag.

The user tag

The user tag displays its content only in case the current Subject is considered a ‘user’, meaning that the Subject has a known identity. This happens either because the user did a successful authentication or because it is remembered from a previous visit via the 'RememberMe' services [33].

```
<shiro:user>
    Wrapped content to be displayed!
</shiro:user>
```

Listing 51 - User tag [33]

Listing 51 presents an example of how to use this tag.

The authenticated tag

The authenticated tag displays its content only in case the current Subject has authenticated successfully during the current session. By definition, this tag is more restrictive than the user tag, and it is recommended to use when the user identity needs to be guaranteed [33].

```
<shiro:authenticated>
    Wrapped content to be displayed!
</shiro:authenticated>
```

Listing 52 - Authenticated tag [33]

Listing 52 presents an example of how to use this tag.

The notAuthenticated tag

The notAuthenticated tag displays its content only in case the current Subject has not yet authenticated during the current session [33].

```
<shiro: notAuthenticated>
    Wrapped content to be displayed!
</shiro:notAuthenticated>
```

Listing 53 - NotAuthenticated tag [33]

Listing 53 presents an example of how to use this tag.

The principal tag

The principal tag simply displays the user’s principal (identifying attribute) or a property of the user's principal [33].

```
Hello, <shiro:principal/>, how are you today?

Hello, <shiro:principal property="firstName"/>, how are you
today?
```

Listing 54 - Principal tag [33]

Listing 54 presents an example of how to use this tag.

The hasRole tag

The hasRole tag displays its content only in case the current Subject is assigned with the role defined in the tag [33].

```
<shiro:hasRole name="administrator">
    Wrapped content to be displayed!
</shiro:hasRole>
```

Listing 55 - HasRole tag [33]

Listing 55 presents an example of how to use this tag.

The lacksRole tag

The lacksRole tag displays its content only in case the current Subject is not assigned with the role defined in the tag [33].

```
<shiro:lacksRole name="administrator">
    Wrapped content to be displayed!
</shiro:lacksRole>
```

Listing 56 - LacksRole tag [33]

Listing 56 presents an example of how to use this tag.

The hasAnyRole tag

The hasAnyRole tag displays its content only in case the current Subject is assigned with any of the roles defined in the tag. These roles must be defined as a comma-delimited list of role names [33].

```
<shiro:hasAnyRoles name="developer, project manager, administrator">
    Wrapped content to be displayed!
</shiro:hasAnyRoles>
```

Listing 57 - HasAnyRole tag [33]

Listing 57 presents an example of how to use this tag.

The hasPermission tag

The hasPermission tag displays its content only in case the current Subject is allowed to perform the ability specified by the permission defined in the tag [33].

```
<shiro:hasPermission name="user:create">
    Wrapped content to be displayed!
</shiro:hasPermission>
```

Listing 58 - HasPermission tag [33]

Listing 58 presents an example of how to use this tag.

The lacksPermission tag

The lacksPermission tag displays its content only in case the current Subject is not allowed to perform the ability specified by the permission defined in the tag [33].

```
<shiro:lacksPermission name="user:create">
    Wrapped content to be displayed!
</shiro:lacksPermission>
```

Listing 59 - LacksPermission tag [33]

Listing 59 presents an example of how to use this tag.

6. Prototype for Project Proposal

This section describes the practical use of Apache Shiro to provide a prototype of solution to the problem of how to integrate Apache Shiro in a Java web application, requiring multiple user roles management and authentication through credentials stored in a database, as instantiated in chapter 1, section 3.

6.1 Database Setup and Connection

One of the requirements for the prototype to have is the possibility of authentication through credentials stored in the database. In order to fulfill such a requirement, the application will require a database already prepared for such functionality.

Apache Shiro does not depend on any specific database vendor, since through the use of JDBC (Java Database Connectivity) technology, Shiro will access whichever database configured for such purposes of authentication and authorization.

JDBC and database setups are subjects which are beyond the scope of this thesis and so it is expected that the reader has certain level of knowledge about those subjects.

Listing 60 presents the script used to create the table to store user authentication data.

```
create table users (
  id int not null auto_increment primary key,
  username varchar(255) unique not null,
  first_name varchar(255) not null,
  last_name varchar(255) not null,
  password varchar(255) not null,
  role varchar(255) not null,
  email varchar(255) not null
);
```

Listing 60 - SQL script for user table

In this case, the database vendor chosen is MySQL database (release 5.5) for the only reason that the company uses this same database vendor in all its applications and they already have the expertise for such a technology (please note that the script in listing 60 will possibly fail to work with any other database vendor besides MySQL).

There are several columns created for the 'users' table but it is important to note the column 'role'. This column is the one that will determine which access control each user will have within the application.

6.2 Shiro Integration

In order for the application to read Shiro's INI configuration, it is necessary to configure a Servlet ContextListener and Filter in the web.xml file. This configuration is a way of making the application to understand that Shiro is integrated and should be used when url mapping is required. Listing 61 shows an example of a web.xml configuration.

Once Shiro is enabled, it has its own mechanism to, by default, search for the INI file in certain specific paths of the application, and if found, to configure itself based on the file content.

6.3 Shiro INI Configuration File

Listing 61 contains all Shiro configuration required for authentication and authorization. Either the [users] section or the [roles] section is required, since the user authentication data is stored in a database or the web application does not require a granular control over the actions of the users.

```
[main]
authc.loginUrl = /login.xhtml
roles = org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
roles.unauthorizedUrl=/accessDenied.xhtml

dataSource = org.apache.shiro.jndi.JndiObjectFactory
dataSource.resourceName = java:/comp/env/jdbc/<your-database-name>

sha256Matcher =
org.apache.shiro.authc.credential.HashedCredentialsMatcher
sha256Matcher.hashAlgorithmName=SHA-256

jdbcRealm = org.apache.shiro.realm.jdbc.JdbcRealm
jdbcRealm.permissionsLookupEnabled=false
jdbcRealm.authenticationQuery=select password from users where
username = ?
jdbcRealm.userRolesQuery=select role from users where username = ?
jdbcRealm.dataSource = $dataSource
jdbcRealm.credentialsMatcher = $sha256Matcher

[urls]
/resources/** = anon
/accessDenied.xhtml = anon
/pageNotFound.xhtml = anon
/logout = logout
/login.xhtml = authc
/user/** = roles[ROLE_ADMIN]
/** = authc
```

Listing 61 - INI file configuration for project

Filters Initialization

Shiro automatically detects that the Java application at hand is a Web application and so it will initialize the SecurityManager with a FormAuthenticationFilter. This filter always requires the user to be authenticated to continue with the request, and if not, it redirects the user to the login url configured, forcing the user to log in. Additionally the default loginUrl value is overwritten to point to the expected login url.

There is a second filter initialized, RolesAuthorizationFilter, which allows access if the current user is assigned with the roles specified by the mapped value, denying the access in case the user does not have assigned all the roles specified. If the access is denied, the user will be redirected to the unauthorized url, which value is being overwritten to point to the expected unauthorized url.

JDBCRealm initialization

JDBCRealm is an implementation of the Realm interface that implements authentication support (log-in) operations and authorization (access control) behavior through the use of JDBC calls. After the JDBCRealm is instantiated, there will be several variables which are being overwritten.

The variable permissionsLookupEnabled indicates if the realm needs to execute a query to retrieve permissions information related to roles and users. As indicated at the beginning of this chapter, no permissions are used, so this variable is set as false. AuthenticationQuery is a string variable which reflects the query to be triggered by the realm when user authentication data is required. If left unset, the default query will be used. UserRolesQuery variable will be used when role data is required, and again, if left unset, it will use its own default string value.

A JDBCRealm requires of a datasource which will provide the necessary information about the database where to connect. To help development, Shiro provides a factory implementation intended to be used to look up objects in JNDI. A JDBC datasource can be configured in multiple ways but the configuration of such objects is beyond the scope of this thesis.

To provide an encryption mechanism for the JDBCRealm, so that passwords are encrypted, it is possible to configure a CredentialMatcher object for the realm. Shiro includes multiple options for an encryption algorithm where to choose, including the most common ones, such as MD5 or SHA-256.

Url access via user roles

The last of the sections in the INI file is the [url] section. This section contains different url pattern strings which are mapped to different filter chains, which defines the accessibility of such urls. For most of the web applications there are some urls which do not need of access control since they are always accessible by any user. In Shiro, those urls are mapped to the keyword “anon”. Those urls which will logout the user and redirect to the login url are mapped to the keyword “logout”.

If there is a requirement to get access control depending on the user role, it will be possible to define which roles can have specific access to certain urls, just by mapping those url patterns to the keyword chain “roles[<role-goes-in-here>]”. If the user does not have that specific user, it will not be allowed to access that specific url pattern even though it may be already authenticated in the application. This user may get access to any other url pattern mapped to “authc” keyword or mapped to any specific role he may have.

As seen in listing 61, the url pattern “/user/**” is mapped to “roles[ROLE_ADMIN]”. This will produce that only users with the role ROLE_ADMIN are able to access to all url pages located within that url path.

7. Project Results and Future Development

Due to the limited scope of the thesis and the company needs, this project was left as a full prototype project. In the future, the company will decide whether to move forward with the project or cancel it. The company situation is changing and they need to use their efforts and resources in other projects which require more attention at the moment.

However, the integration of Apache Shiro into a Java application following the requirements initially asked was completed, and is fully working now. The main requirements were developed and the application is now able to authenticate users through the use of data stored in a database combined with the management of multiple user roles.

This study has helped to better understand the wide range of possibilities that Apache Shiro offers as a security framework for Java applications. This is reflected in the fact that the company is considering using the prototype developed during this thesis as guidelines to implement security capabilities in their current and future applications.

A few ideas were presented in relation to the prototype developed in this study. The company is considering the possibility to replace INI file configuration to a full Java integrated configuration. This means that Apache Shiro would be fully integrated and configured through Java technology, avoiding any type of XML or INI files. Furthermore, the company may consider the option to fully replace any local authorization mechanism for a cloud system. This would mean that instead of managing user authentication data using a database, a cloud system would be used, avoiding any need of local resources.

8. Conclusions

Web applications improve business processes and offer expanded and competitive opportunities providing the complex and rich experience that users demand. However, production software is continuously under attack and only an effort to produce more stable and reliable applications will prevent attackers and bring users the confident feeling that they are protected from exploitation. Developers should be responsible for security, just as they are responsible for functionality and quality. Furthermore, vulnerabilities in custom applications need to be discovered and resolved in the most efficient manner possible.

Apache Shiro is a flexible security framework for Java applications that supports the four cornerstones of application security: authentication, authorization, enterprise session management, and cryptography. It is in use at all types of organizations, from big government to tiny applications, and has an extremely active community and well-documented codebase.

The goal of this thesis was a prototype integration of Apache Shiro within a Java web application enabling user management features such as multiple user role management and user authentication through data stored in a database. Such a prototype was developed successfully thanks to the extreme flexibility and easily adaptation of Apache Shiro to any kind of Java application. With just a minimum configuration setup, Shiro provides not only database authentication mechanisms and user role control access, but an extensive additional stack of security and user management capabilities.

Additionally, the study and integration of Apache Shiro has brought up discussion about the current design patterns and the future of security within Java applications which will reflect in more reliable and secure applications.

References

1. Scott Oaks (2001), Java Security, 2nd Edition, O'Reilly Media

2. David Flanagan (1999), Java in a nutshell, 3rd Edition, O'Reilly Media

3. Gail Zemanek Bayse (2004), A Security Checklist for Web Application Design, SANS Institute
   http://www.sans.org/reading-room/whitepapers/securecode/security-checklist-web-application-design-1389
   [Access date: February 2014]

4. OWASP Top Ten Project, Top 10 2013/Risk, Open Web Application Security Project (OWASP)
   https://www.owasp.org/index.php/Top_10_2013-Risk
   [Access date: February 2014]

5. OWASP Top Ten Project, Top 10 2013, Open Web Application Security Project (OWASP)
   https://www.owasp.org/index.php/Top_10_2013
   [Access date: February 2014]

6. OWASP Top Ten Project, Introduction, Open Web Application Security Project (OWASP)
   https://www.owasp.org/index.php/Main_Page
   [Access date: February 2014]

7. OWASP Top Ten Project, Top 10 2013/ProjectMethodology, Open Web Application Security Project (OWASP)
   https://www.owasp.org/index.php/Top_10_2013/ProjectMethodology
   [Access date: February 2014]

8. HDIV: Java Web Application Security Framework
   http://hdiv.org/
   [Access date: February 2014]

9. Spring Security
http://projects.spring.io/spring-security/
[Access date: February 2014]

10. Java Authentication and Authorization Service
http://www.oracle.com/technetwork/java/javase/jaas/index.html
[Access date: February 2014]

11. JGuard
http://www.jguard.net/
[Access date: February 2014]

12. Apache Shiro Website, Introduction (2014)
http://shiro.apache.org/introduction.html
[Access date: February 2014]

13. Apache Shiro Website, Architecture (2014)
http://shiro.apache.org/architecture.html
[Access date: February 2014]

14. Apache Shiro API Documentation, Subject Interface
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/subject/Subject.html
[Access date: April 2014]

15. Application Security With Apache Shiro
http://www.infoq.com/articles/apache-shiro
[Access date: April 2014]

16. Apache Shiro API Documentation, SecurityManager Interface
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/mgt/SecurityManager.html
[Access date: April 2014]

17. Apache Shiro API Documentation, Realm Interface
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/realm/Realm.htm
l
[Access date: April 2014]

18. Apache Shiro Website, Configuration (2014)
http://shiro.apache.org/configuration.html
[Access date: April 2014]

19. Apache Shiro Website, Authentication (2014)
http://shiro.apache.org/authentication.html
[Access date: April 2014]

20. Apache Shiro API Documentation, UsernamePasswordToken Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/authc/Usernam
ePasswordToken.html
[Access date: April 2014]

21. Apache Shiro Website, Authorization (2014)
http://shiro.apache.org/authorization.html
[Access date: May 2014]

22. Apache Shiro API Documentation, RequiresAuthentication Annotation Type
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authz/annotation/
RequiresGuest.html
[Access date: June 2014]

23. Apache Shiro API Documentation, RequiresGuest Annotation Type
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authz/annotation/
RequiresGuest.html
[Access date: June 2014]

24. Apache Shiro API Documentation, RequiresPermissions Annotation Type
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authz/annotation/RequiresPermissions.html
[Access date: June 2014]

25. Apache Shiro API Documentation, RequiresRoles Annotation Type
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authz/annotation/RequiresRoles.html
[Access date: June 2014]

26. Apache Shiro API Documentation, RequiresUser Annotation Type
http://shiro.apache.org/static/current/apidocs/org/apache/shiro/authz/annotation/RequiresUser.html
[Access date: June 2014]

27. Apache Shiro Website, Realms (2014)
http://shiro.apache.org/realm.html
[Access date: June 2014]

28. Mikalai Zaikin (2007), Sun Certified Enterprise Architect for Java EE 5 Study Guide, Revision 0.3, Chapter 08
http://java.boot.by/scea5-guide/ch08s02.html
[Access date: June 2014]

29. April J. Wells (2007), Grid Application Systems Design, 1st Edition, CRC Press

30. Peter J. Ashenden (2008), The Designer's Guide to VHDL, 3rd Edition, Morgan Kaufmann

31. Apache Shiro Website, Cryptography Features (2014)
http://shiro.apache.org/cryptography-features.html
[Access date: June 2014]

32. Apache Shiro Website, Terminology (2014)
http://shiro.apache.org/terminology.html
[Access date: June 2014]

33. Apache Shiro Website, Web Support (2014)
http://shiro.apache.org/web.html
[Access date: June 2014]

34. Apache Shiro API Documentation, AnonymousFilter Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authc/AnonymousFilter.html
[Access date: June 2014]

35. Apache Shiro API Documentation, FormAuthenticationFilter Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authc/FormAuthenticationFilter.html
[Access date: June 2014]

36. Apache Shiro API Documentation, BasicHttpAuthenticationFilter Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authc/BasicHttpAuthenticationFilter.html
[Access date: June 2014]

37. Apache Shiro API Documentation, LogoutFilter Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authc/LogoutFilter.html
[Access date: June 2014]

38. Apache Shiro API Documentation, NoSessionCreationFilter Class
https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/session/NoSessionCreationFilter.html
[Access date: June 2014]

39. Apache Shiro API Documentation, PermissionsAuthorizationFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authz/PermissionsAuthorizationFilter.html
[Access date: June 2014]

40. Apache Shiro API Documentation, PortFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authz/PortFilter.html
[Access date: June 2014]

41. Apache Shiro API Documentation, HttpMethodPermissionFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authz/HttpMethodPermissionFilter.html
[Access date: June 2014]

42. Apache Shiro API Documentation, RolesAuthorizationFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authz/RolesAuthorizationFilter.html
[Access date: June 2014]

43. Apache Shiro API Documentation, SslFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authz/SslFilter.html
[Access date: June 2014]

44. Apache Shiro API Documentation, UserFilter Class https://shiro.apache.org/static/current/apidocs/org/apache/shiro/web/filter/authc/UserFilter.html
[Access date: June 2014]