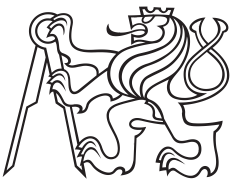
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Application security

Context aware Role Based Access Control (RBAC) in Java EE

Michal Trnka
Open Informatics

December 2014
Supervisor: Mgr. Peter Škopek

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Michal Trnka**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Context-aware Role Based Access Control (RBAC) in Java Enterprise Edition**

Guidelines:

The task is to implement a mechanism into PicketLink project which will extend the current RBAC authorization rules with a new kind of rule, based on security level of the user. The security level will be determined based on authentication method together with user's context. This should be implemented using annotations enforcing security rules and extending the current Expression Language of PicketLink to support security levels. First, it includes the development of mechanisms to obtain the security level from authentication method. Second, it includes development of the architecture, which will let developers use contextual information. It must be possible to check user contexts, such as spatial, environmental, operational and any other type of contexts. Implemented solution will be demonstrated by a case-study involving simple application using extended version of the PicketlLink and will will demonstrate the ability to check user's spatial, environmental and operational context.

Bibliography/Sources:

1. PicketLink reference documentation http://docs.jboss.org/picketlink/2/latest/reference/html/
2. Gustaf Neumann and Mark Strembeck. 2003. An approach to engineer and enforce context constraints in an RBAC environment. In Proceedings of the eighth ACM symposium on Access control models and technologies (SACMAT '03). ACM, New York, NY, USA, 65-79.
3. Lima, J.C.D.; Rocha, C.C.; Augustin, I; Dantas, M. A R, "A Context-Aware Recommendation System to Behavioral Based Authentication in Mobile and Pervasive Environments," Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on , vol., no., pp.312,319, 24-26 Oct. 2011

Diploma Thesis Supervisor: Mgr. Peter Škopek

Valid until the end of the summer semester of academic year 2015/2016

CONFIDENTIAL

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická

L.S.

pro CONFIDENTIAL

Prague, November 4, 2014

Acknowledgement / Declaration

I'd like to thanks at first to my supervisor Peter. He has helped me a lot with the following thesis. Also a big thanks comes to Pedro Igor who provided valuable ideas concerning the thesis. At least but not last I'd like to thank to my family for supporting me during my studies.

I hereby that this Master's thesis is my own work, and it does not contain other people's work without this being stated; and does not contain my previous work without this being stated, and that the bibliography contains all the literature that I have used in writing the thesis, and that all references refer to this bibliography.

In Prague on January 2nd 2015

.....

Abstrakt / Abstract

V moderních aplikacích je použití kontextu nejen důležité, ale často i nevyhnutelné. Použití kontextu při zabezpečování aplikace může přinést znatelné zlepšení jak pro uživatele tak pro vývojáře. Nicméně většina zabezpečení dnešních systémů je vyvinuta bez návaznosti na kontext. V této práci je popsáno jak kontextové prvky mohou být implementovány do stávajících RBAC systémů a především do PicketLinku. Můžeme použít systém levelů zabezpečení, které jsou založeny na kontextu a metodě přihlášení.

Překlad titulu: Zabezpečení aplikací (Autentizace a autorizace pomocí rolí v Javě EE s ohledem na kontext)

In modern application is use of context not only important but also required. Use of context in application security can provide noticeable improvements both on user's and developer's side. However most of the current systems security is developed without context aware elements. In this work is described how context awareness can be implemented in current RBAC systems especially to PicketLink. We can use system of security levels which are based on context and user's authentication method.

Contents /

Tables / Figures

Chapter 1
Introduction

There is significant increase in the usage of the information technologies in few recent decades. Almost every institution uses multiple applications for its needs nowadays, almost every person has a computer and at least one mobile device. Everyone has possibility to be online whenever he wants and so do applications. Applications often need to exchange data in real time and users love to be connected to each other and to have instant access to all information and services provided online.

1.1 Motivation

Users expect every application to fulfill their specific needs and don't hesitate to provide more information about themselves. They create the application's content dynamically with all the data they provide for the applications instantly and want to see personalized output from those applications.

Applications needs to provide data to other applications, by both public and private API. They need to provide data to users. And not to make it that easy users are not equal — some can see just normal data, other can see also some additional data and some super users can often some kind of meta-data.

Securing applications has always been important, with most of them being connected to internet it is even more crucial. There rise two main problems. First is that application is always available not only to normal (in meaning of 'good') users, but also for various attackers. As the internet is already quite old we have currently very powerful tools to tackle those attackers. Second reason is the more users are accessing it the more fine-grained rules are needed to distinguish them.

This problem emerged quite recently and is caused by two major reasons. The first one is legal. Applications today have users from whole world and it brings such questions: Can user from that country legally access that content? Needs to be data from user from given country treated different way? Does the EULA correspond to the laws of the users country? The other reason is caused by users willing to have application customized right for them and not for any fictional average user. And this customization concerns security as well.

There are multiple ways how to achieve personalization of the program and several ways how to create architecture to provide it. Architect, or developer, of the program will have to decide whether to use an existing framework whether to go with RBAC concept, ACL, or choose different one.

I will try to provide deeper analysis of the problem, show possible solutions and present one of them implemented in existing widely used framework.

1.2 Red Hat usage

Red Hat leads information technology development in multiple areas. The company is well-known mainly for its Linux development, both distributions and environment around (for example LibreOffice). But since 2006, when Red Hat bought JBoss group, they are also lead developers for Java EE technologies. One of many their projects is PicketLink – open source project for security and identity management for Java Applications.

In PicketLink was a need for a tool to distinguish different ways by which an user logs in. The basic idea is that: The stronger/more trustworthy authentication method user uses, the higher rights he gets. For example, by logging in with two step authentication an user will be trusted more then someone who uses simple login/password method.

I believe that authentication method is part of the user's context and therefore I proposed that the context information should be also taken into account when deciding how much can the user be trusted. This idea was positively accepted.

1.3 Goal

The task of master's thesis is to implement method for context aware elements for the PicketLink framework, especially connection between user's rights in the application and his context, including his authentication method.

As long as there is currently no other framework providing such functionality a new solution is needed. The solution should be effective, easy to understand, simple and powerful. Also it should be not only testable, but already covered with tests.

The final goal is that the changes presented above will be successfuly developed and merged into PicketLink and will become aviable for broad public.

Chapter 2
Current state of art

In this chapter will be described what is the current state of art. Also there will be discussed current authorization methods with emphasis on RBAC. In next chapter will be explanation what the context is and especially why is so important nowadays. In the end there will be brief introduction in PicketLink and its capabilities. Finally there will be pointed some shortages of the current state and will be suggested possible improvements.

2.1 Authorization

Currently there are many different approaches for access control. Two oldest exclusionary approaches are MAC and DAC. Their roots are in 70s and they can be seen in multiple mutations. [1]

MAC is focused on centralized approach in which access is granted by centralized security authority. For example once you create an document the authority will decide who gets right for it. Similarly if you wanna access an document the authority must agree on it. This approach separates security concerns from content and its creators and therefore allows higher security and that is reason why was used for systems where security was a major concern (e.g. military systems).

DAC leaves decision on content rights on its creator. For example if you create document you can decide who can (or can not in some cases) access it. This approach is very flexible and does not require any other authority in cost of lower security. This approach is normal when the security is not huge concern (universities, some companies) or when the speed of decisions is important (there is no need to wait for authority).

Those two approaches got one common point — they grant the permission directly to user. In systems with a lot of users it can be very difficult to maintain security rules for multiple reasons. The first reason is obvious — it is not easy to keep track on so many users and therefore do not give rights to unwanted users or to forget to some needed users. The second reason is that a lot of users likely produces a lot of content and therefore someone must decide on rights for it (in DAC systems the creator is in charge of that in MAC a security enforcing authority is in charge). The third reason is the most important — it is very hard to keep track on the rules because of the fluctuation of users.

On the other side people in organizations tends to create groups of people with same responsibilities. Those groups can be called roles. Thus we can grant permissions to roles and then assign people into roles. Roles will keep multiple permissions and once set up properly the roles in organization does not change and if so then just very slowly. Therefore when the people change position, leave or hire up we can assign people just to corresponding roles. Also roles can be part of another role and creation multilevel, fine-grained security policy is easier, more maintainable and more readable. This approach is called RBAC. [2]

RBAC stands neutrally between MAC and DAC, or inclines a bit toward MAC and can be used with both of those approaches. If to newly acquired content can grant permissions just some role (e.g. administrator) it acts in role of security enforcing authority and the system is improvement of MAC. But there is also possibility that system can grant permissions directly to user (or in strict RBAC create new role for every new content and grant permission to that group only to creator). Then the creator of the content can decide to which give the rights to access the content, in that case it is improvement of DAC. However in real systems it is combination of both principles (e.g. creator can decide to which roles give permissions, but only administrator can grant roles to users).

Those concepts are abstract and does not show how they should be enforced. They can be used in file systems, in database systems, in programs or even on various business processes out of information technologies.

2.2 Context

When two humans talk to each other they can be very effective in sharing ideas. This is not only because of the richness of the language they use, but also by the common understanding how the world works and ability to interpret implicit situational information. The situational information is often referred as context. Formally context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the users and applications themselves. [3]

Unfortunately that natural understanding of the context does not transfer to computer-human interaction. Requiring users to give a computer all relevant information does not lead to improvement. We do not want to make interaction harder for human and also user can very hardly decide what is relevant and what is not. Better approach is to use the information the computer already has about user. It can be data which can computer acquire during interaction immediately (e.g. time, computer's or user's location, activity of the user, user's identity), data the user provided about himself to computer before and were stored by computer (e.g. history of user's activity, etc) or information provided to any other source reachable by computer (e.g. social networks, user's history in network, etc).

Day [4] states that computers are not currently enabled to take full advantage of context in human-computer interaction. That was said almost 15 years ago. Nowadays computers can take the advantage of context pretty easily, the problem is the applications are not developed to use it. There is just few applications that are not only aware of context but also adapt their behavior for the user based on context.

Biggest effort in using context for human computer interaction can be seen most probably at search engines currently. They take advantage of user's location, his previous searches and if they have more data available they also use the additional information stated elsewhere (e.g. Google search for information from Gmail and G+). Advantages in using context for search engines can be seen in that users provide less and less relevant information and do not search the right keywords but they get better search results. That shows the way most of application involving human computer interaction will go.

2.3 PicketLink

PicketLink is an application security framework for Java. It provides features for authenticating users, authorizing access to the business methods of application, managing application's users, groups, roles and permissions, plus much more. It provides an easy way to provide security tools to an application. It is able to authenticate, authorize and perform identity management operations using a database or a LDAP identity store. [5]

PicketLink has seven modules and every module can be used separately (except Base):

- Base Module: Authentication API, Authorization and Permission API, Session and Reqeust Based Identity, Http Security API and CDI Integration
- Federation: SAML, OAuth, WS-Trust, OpenID
- IDM: User, Group and Role management, Credential management and validation
- SCIM: SCIM-based management interface for IDM
- REST: Token Authentication, Authorization, JSON Signature and Encryption, JavaScript Libraries
- Social: Social networks integration (e.g. Twitter, Facebook)

Base module is obviously the one where most of the interesting stuff is being done including authorization and authentication. While using PicketLink it is basically required to log in an application, so it can associate real user with user in application and that allows PicketLink to use its authorization tools.

PicketLink can handle two types of authorization architecture. The first one is access control list permissions. It allows any identity entity to get any permission (represented by string) for a single object or instance. The second approach is based on RBAC. It allows users for operations based on their role and group membership. Both methods can be used in multiple ways — by annotating methods and objects or by using PicketLink expression language to describe complex security constrains. PicketLink is based on DeltaSpike project [6] therefore developer can additionally provide own annotations.

2.4 Limitations

Application security can take a big advantage of knowing user's context. It can add additional level of security as well as it can be more comfortable for an user. Application can determine some unusual or suspicious behavior of user based on his context and eventually can change security polices towards him or cause any kind of security warning. Such behavior can include but is not limited to accessing the application from places very far away in very short time frame or browsing system for extended period of time (in term of tens of hours or days) without break.

Users are often forced to use inappropriate authentication methods or whole security steps when it is not needed. Example can be accessing system from inner company network — someone who has right to connect company network (or even has to authenticate to log in) can view a lot of confidential stuff in other network systems so there is no reason to bother user to log in again. Or when users are forced to use strong but complicated authentication method for log in because they can potentially perform some critical action. The right approach is to determine which actions can be done with weaker authentication method and force the user to use the stronger one just when needed.

The previous example shows that authentication method is very important part of context (at least for context-aware security). As far as I know there are no theoretical concepts for using authentication method as part of context and there are no existing security frameworks implementing context awareness elements.

There are currently known powerful, simple and customizable authorization approaches. Frameworks implementing them are also available and widely used. Context awareness concept is well described as well. However nowadays the authorization methods does not include context-awareness elements. They are not neither fully theoretically described nor any existing framework is implementing it.

Chapter 3
Related work

There has been multiple attempts to extend classic role based access control with context-aware elements and make RBAC more fine-grained. These attempts have contained many various ways how to do it. In the following lines they will be introduced to you. All of them are workable and provide a lot of inovative ideas but does not exactly match the needed requirements.

3.1 GRBAC

Generalized Role Based Access Control [7] extends traditional RBAC with object roles and environment roles as addition to classic subject roles. An environment role is based on the current state of context. Object roles allows to classify objects to categories based on security-relevant properties. Environment and object roles can be described hierarchically as traditional subject roles. Policy roles consists of so called transactions and permissions. Transaction consist of an object role, and subject role and an environment role. If one type of the roles is omitted then all roles of that type are allowed for it.

Disadvantage of GRBAC is its exponential increase of rules because of the need to specify policies for all combinations of all three types of roles and therefore it is not suitable for large and complex applications. Also it possesses unnecessary overlapping between environment and object roles roles because is possible to describe certain physical environmental property as object and so add to them object roles instead of environmental roles.

3.2 Enviromental roles

Context roles is concept very similar to GRBAC. It adds another set of roles to RBAC. Those roles can be hierarchically composed and represent current state of environment. Environmental roles are independent on user and stands side by side with subject roles. Policy rules consist of subject roles, environmental roles and permissions. [8–9]

Problem with that approach is when to invoke and revoke environmental roles. Solving the invoking is the easier part, as the role check can be invoked when is the role needed for authorization check. On the other side there is need to determine when revoke the roles because context can change rapidly in some scenarios but determining some roles can be on the other hand time or resource consuming so the right balance needs to be found. Also in large systems there can be huge increase in number of polices though significantly lower then in GRBAC.

3.3 Role assignment based on context

Another approach is to assign roles to user based on context. After authenticating user additional roles are invoked (or revoked) to him based on context. It allows to grant user roles based on context. This approach allows to grant just special roles and in that moment it becomes almost identical to case before. This approach limits number of roles and policy constrains but have can not describe fast changing environment constrains like CPU load. [10]

This approach is developed further into Context-Aware RBAC. [11] It allows roles to be granted based on context and even to check the permission based on instance of object on which is action going to be performed. This functionality is controlled by second layer of authorization architecture. The layer is responsible for granting and revoking roles in the right time as well as managing access to instances of objects.

3.4 xoRBAC

Extended object RBAC is basically traditional RBAC. [12] It adds to security policy context constrains. So it consists of role, context constrains and permission. When permission is checked user need to posses not only the right but also context constrains need to be fulfilled. It has advantage in that it leaves the roles as in traditional RBAC and so transition to this method is very simple in organizations and applications where is RBAC already developed. Disadvantage is that context rules posses no abstraction nor hierarchy given by their aggregation into roles.

There is also proposal for one similar approach. Policy rules are consisted of four elements — permission, role, context and authentication method. [13] This approach inherits most of flaws of GRBAC but is very interesting in that it takes authentication method into account.

3.5 Context based authorization

Those approaches do not have roles but instead place big emphasis on context. First approach is represented by UbiCOSM framerwork. [14] It replaces roles with context and therefore context is used as a foundation for security policy specification and enforcement. Permissions are associated with contexts instead of roles as in normal in RBAC. User has given permission only if he is in the context that has the permission.

Another suggestion is to add another dimension to current security policies. [15] Currently security police determine just whether user have permission. That is done in most of the approaches, including RBAC where roles are just middle element. We can add third dimension which is context so we would get security police with combination of user permission and context. Context is there described more abstractly and complexly, not just with context constrains as in xoRBAC.

Problem with these approaches is that they need somehow compare contexts of users and determine how similar they are and that is something very difficult.

3.6 Activity based security scheme

Security scheme is there represented by three entities — user, activity and object (e.g. instance of object in OOP). Every entity has some credentials. Activity and object have credentials which are required to perform action on object and user has credentials which are little similar to roles or eventually could be aggregated into roles. If an user wants to proceed an activity on given object, he must have required credentials both for activity and object. [16]

That scheme does not contain explicit mention on RBAC nor implements its ideas but is very interesting and with minor tweaks could be used for extended RBAC architecture.

3.7 Level required authorization model

This model extends traditional RBAC by security levels. Every privilege possesses security level. Decision on security policy is not made only by checking if user has corresponding permission, but also if the permission is granted with required security level. [17]

Even this model has nothing to do with context it presents unique technique using some kind of level which can be used in context aware security systems.

Chapter 4
Design

In that chapter is discussed architecture and approach chosen during implementation of the changes into PicketLink

4.1 Requirements

As long as the solution is being integrated into PicketLink there are some limitations. The limitations are those:

- Lightweight — as security is crosscutting problem it should not be hard to use and it must not consume significant amount of resources
- Easy to use — user should be able to use the solution very quickly without any need for creating special classes, methods or configuration. It also should be used similarly to other PicketLink features so the user would learn how to use it rapidly.
- Voluntary use — this is tied to the previous point, but also philosophy of PicketLink is that it allows user to use its features, not forces. If someone wants to use just part of PicketLink he can.
- No changes in current API, only adding the features — this is needed for upward compatibility of PicketLink and it allows easy upgrade of PicketLink version in systems utilizing it.
- Scalable — it should be very easy usable in small applications but it should offer big modularity for big applications.

4.2 Architecture

The main concern is how the context aware elements will be implemented in PicketLink with its current architecture. Adding context or environmental roles is unsuitable for few reasons. The first reason is that roles in PicketLink are handled in complex way. They are handled by complicated mechanism which includes user groups, identity partitions and realms. All tied together closely, including persistence layer. It would make adding and revoking roles difficult process both for implementation and performance. The second reason is that the developer for determining different authentication methods or contexts would have to define special roles. In large scaled applications it could end up declaring tens or hundreds of roles.

As there can not be any change in current role system it is necessary to add some kind of new elements into PicketLink. As the final decision was chosen granting a user security level based on context and on authentication method. Level would describe how the user authentication can be trusted and it would add another possibility for authorization. Level would be represented by comparable object therefore allowing customization. Developers could also determine how many levels they have in application.

Implementation can be divided in two main parts. The first, which is also the most important and most interesting, determines the security level. The second one, which is the one developers will use most of the time, puts the level in use and provides functionalities based on it.

Levels would be determined with `SecurityLevelManager` which would use `AuthenticationLevelResolver` and `ContextLevelResolver` to obtain current level. `ContextLevelResolver` would iterate through all `ContextResolvers` to determine highest context level. The final level would be used in `Identity` class which holds user's identity in PicketLink and provides log in and log out logic.

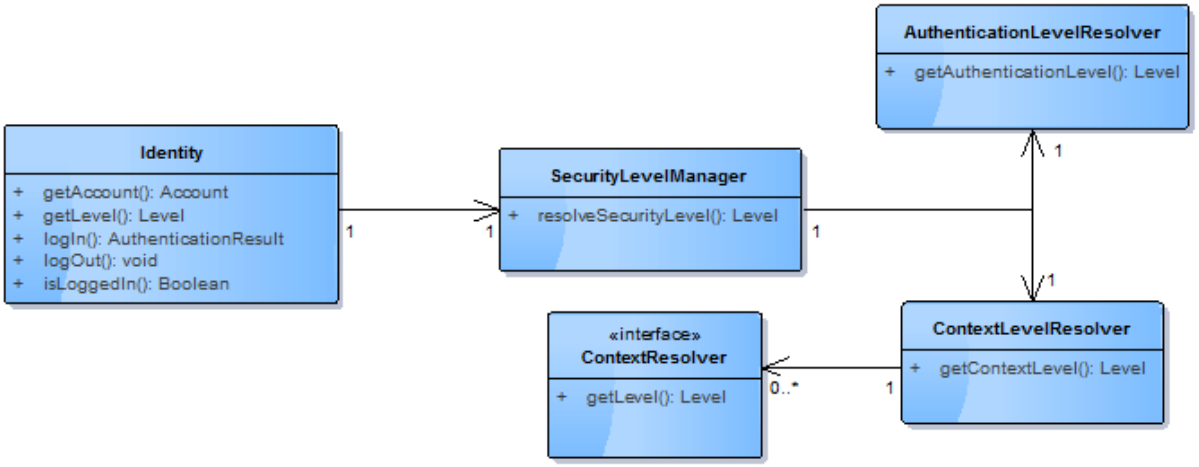


Figure 4.1. Designed architecture of level resolving.

Level is determined during log in in `Identity` bean. In normal cases user's context should not change within his session significantly and therefore determining it once for whole log in should be enough. For applications which expect user's context to change rapidly PicketLink offers option called 'single sign on'. It forces user to sign in for every request. User also should be able to rise his level by logging in with higher level. It can happen when context changes or if user uses another authentication method.

Level should be replaceable and therefore is represented only by interface implementing `Level` interface. It extends `Comparable` interface, because only needed functionality of levels is ability to compare them.

`Level` obtained during log in can be used for authorization rules by predefined annotation or it is be possible to use it in expression language used in PicketLink to describe more complicated rules. Also there should be possibility to obtain instance of current level directly from `Identity` class for needs in application.

`Level` is used in the annotations tied to authentication and authorization. Problem with annotations is that they allow as members only those types: primitive type, String, Class literal, annotation, Enum item, or 1-dimensional arrays of any of the above. From the given options two are suitable — String and Enum of predefined `Levels`. Both of the variants would work very well. First advantage of Enum is readability of such code. Its main advantage over String is the need to be defined before usage and all application `Levels` would be in one place. However as the PicketLink is framework which is used with different applications there would be need to not only define items of Enum but also to define name of Enum to be used with `Levels` in some configuration

file. On the other hand representing `Level` as String in annotations would not need such configuration file and therefore would be much easier to be used. Disadvantage of String is that it has no predefined values and every String can be used to represent a level. This disadvantage could be decreased with usage of Enum of Strings for `Level` representation if application architect wanted such feature. As final decision was chosen String because it does not have any significant disadvantages.

Chapter 5
Implementation

In following sections will be presented how is everything implemented in the PicketLink. Implementation can be divided in two main parts. The first, which is also the most important and most interesting, determines the security level. The second one, which is the one developers will use most of the time, puts the level in use and provides functionalities and it is the part which will developers use.

5.1 Security Level representation and creation

Security level is represented by interface `Level`. It extends another interface `Comparabel<Level>` and therefore ensures that the `Level` instances will be comparable with each other. The interface is shown bellow.

```
1  public interface Level extends Comparable<Level>{
2    @Override
3    int compareTo(Level level);
4  }
```

There is default implementation of `Level`. Note that check whether compared `Level` is instance of `DefaultLevel`.

```
1  public class DefaultLevel implements Level{
2    int value;
3
4    public DefaultLevel(int val){
5      value = val;
6    }
7
8    @Override
9    public int compareTo(Level level) {
10     if(level == null){
11       return 1;
12     }
13     else if(level instanceof DefaultLevel){
14       DefaultLevel defLevel = (DefaultLevel)level;
15       return Integer.compare(value, defLevel.getValue());
16     }else{
17       throw new SecurityLevelsMismatchException("More instances
18         of SecurityLevel in the application");
19     }
20   }
21
22   public int getValue(){
23     return value;
24   }
25
```

```
26      public String toString(){
27        return ""+value;
28      }
29    }
```

Because in the annotations is used String to represent `Level` there is need for a factory which can build levels from given String. This factory is represented by interface `LevelFactory`. It is not possible to have only one `LevelFactory` because developer can provide own implementation of `Level`. Code follows:

```
1   public interface LevelFactory {
2     Level createLevel(String level);
3   }
```

There is provided `DefaultLevelFactory` which created `DefaultLevel` instances. Its code is very simple:

```
1   public class DefaultLevelFactory implements LevelFactory{
2
3     @Override
4     public Level createLevel(String level) {
5       return new DefaultLevel(Integer.parseInt(level));
6     }
7   }
```

As long as there is possibility of more `LevelFactory` instances it needs to be chosen carefully which to use. The best way to accomplish that is to create abstract factory which choose the right `LevelFactory` and the code for choosing `LevelFactory` does not need to repeat over application. `AbstractLevelFactory` is way to go if `Level` instance needs to be created. Code shown bellow:

```
1   @ApplicationScoped
2   public class AbstractLevelFactory {
3
4     @Inject
5     @Any
6     @PicketLink
7     private Instance<LevelFactory> factory;
8
9     Inject
10    private DefaultLevelFactory defFactory;
11
12    public LevelFactory getFactory(){
13      if(factory.isUnsatisfied()){
14        return defFactory;
15      }else{
16        return factory.get();
17      }
18    }
19  }
```

Usage is obvious:

```
1   @Inject
2   AbstractLevelFactory absLevelFactory;
3   ...
4   absLevelFactory.getFactory.createLevel("x")
```

5.2 Determining user's security level

Security level is determined in class `SecurityLevelManager`. This class iterates over all `SecurityLevelResolver` annotated with `@PicketLink` in application and use the highest resolved level. If security level can not be resolved for any reason (authentication method does not have defined level and none of resolvers could resolve the level) then the default level is used.

After implementing first version it was found out that authentication resolvers are same as context resolvers and were merged together. Authentication resolvers are now represented as two predefined out of box `SecurityLevelResolver` and will be described in the end of the section. Developer can provide unlimited number of `SecurityLevelResolvers` to determine level based on context or on a state of application or server (which can be considered as context as well).

5.2.1 Creating SecurityLevelResolver

`SecurityLevelResolver` is represented by the following interface.

```
1  public interface SecurityLevelResolver {
2    Level resolve();
3  }
```

As you can see a resolver has method called resolve and it should return the `Level` it resolved. Resolvers needs to be annotated by annotation `@PicketLink` in order to be injected into `SecurityLevelManager`. Example of usage:

```
1  @PicketLink
2  public class ContextCheck implements SecurityLevelResolver {
3    @Inject
4    DefaultLevelFactory levelFactory;
5
6    @Inject
7    FacesContext fc;
8
9    @Override
10   public Level resolve() {
11     HttpServletRequest request = (HttpServletRequest)
12       fc.getExternalContext().getRequest();
13     if(request.getRemoteAddr().equals("72.57.105.12")){
14       return levelFactory.getFactory.createLevel("3");
15     }
16     return null;
17   }
18 }
```

In the example above you can see resolving security level based on IP address of the request (i.e. of the user). If user's IP is 72.57.105.12 then he is granted level `3`. If the user's IP does not match, then `null` is returned. `null` means that resolver could not resolve security level. Another way to deal with failing level resolve check would be returning default `Level`. Final `Level` resolving mechanism and default security `Level` are described further in this thesis.

5.2.2 Out of box SecurityLevelResolvers

There are two out of box `SecurityLevelResolvers` to provide `Level` determining based on authentication method. Their usage is described further in that chapter and in this section only implementation is shown.

In PicketLink are multiple ways to authenticate user. The first one and the most complex one is to use own `Authenticator` class. The second one is to use default `IDMAuthenticator` in PicketLink and provide own credentials implementation. The last one is to use default `DefaultLoginCredentials` implementation without additional credential and to use it with default `IDMAuthenticator` implementation.

Those resolvers cover usage of own `Authenticator` and own credential. They retrieve annotation `@SecurityLevel(String level)` from credential or `Authenticator` class and then determine `Level` based on value of annotation. Both resolvers return `Level` only if user is logged in because user not logged in can not rise his level by authentication method which was not used yet.

Resolver `SecurityLevelResolver` for resolving `Level` of `Authenticator` works it those steps:

1. Check whether is user logged in. If not return `null`.
2. Check if developer defined own `Authenticator`. If not return `null`.
3. Check if `SecurityLevel` annotation is present. If not return `null`.
4. Retrieve value of `SecurityLevel`.
5. Create `Level` based on the value.
6. Return `Level`.

In implementation all the checks are separated into different private method and if any of the checks fails method throws and exception. If the upper method catches exception it returns `null` otherwise it returns `Level`. Class can be seen bellow:

```
1  @PicketLink
2  public class AuthenticatorLevelResolver implements
3    SecurityLevelResolver{
4
5    @Inject
6    @PicketLink
7    private Instance<Authenticator> authenticatorInstance;
8
9    @Inject
10   @Any
11   private Identity identity;
12
13   @Inject
14   private AbstractLevelFactory abstractFactory;
15
16   protected String resolveLevel(){
17     if(!identity.isLoggedIn()){
18       throw new PicketLinkException();
19     }
20     if(authenticatorInstance.isUnsatisfied()){
21       throw new PicketLinkException();
22     }
23     SecurityLevel a = authenticatorInstance.get().
24       getClass().getAnnotation(SecurityLevel.class);
25     if (a == null) {
```

```
26          throw new PicketLinkException();
27        }
28        return a.value();
29      }
30
31      @Override
32      public Level resolve() {
33        try{
34          String level = resolveLevel();
35          return abstractFactory.getFactory().createLevel(level);
36        }catch(PicketLinkException e){
37          return null;
38        }
39      }
40    }
```

Resolving `Level` based on credentials is similar. `CredentialLevelResolver` injects instead of `Authenticator` instance `DefaultLoginCredentials` directly.

```
1   @Inject
2   DefaultLoginCredentials credentials;
```

However custom credentials must be retrieved from `DefaultLoginCredentails`. This is done following way:

```
1   Object cred = credentials.getCredential();
2   if (cred == null) {
3     throw new PicketLinkException();
4   }
```

The rest of the logic is same as in the `AuthenticatorLevelResolver`.

5.2.3 Retrieving highest security level

User's security level is the security level determined by resolvers or default security `Level`. The highest one is used. Basic obtainment of `Level` is done by iterating through all resolvers and comparing `Level` they return to the current highest one. As the initial highest one is set default `Level`. Method can be seen bellow.

```
1   private Level synchronousResolve() {
2     Level highestLevel = defaultSecurityLevel;
3       for (SecurityLevelResolver resolver : resolverInstances) {
4         Level level = resolver.resolve();
5         if (highestLevel.compareTo(level) < 0) {
6           highestLevel = level;
7         }
8       }
9     return highestLevel;
10  }
```

This method has one disadvantage. `SecurityLevelResolver` can do some time consuming actions when determining context. Operation to determine `Level` might require information from another application. Typical usage can be asking LDAP for user's details or communicating with some network service to determine how long is the user connected to the network etc. As the remote communication is one of the most time consuming action and `SecurityLevelResolvers` does not communicate with each other, and they should not change application state, parallel processing is possible.

Using EJB integrated support for asynchronous processing was clear choice as it is very simple and standardized. `AsynchronousResolverProcessor` carry task of asynchronously processing `SecurityLevelResolver`. For further details see documentation. [18] The code of processor follows.

```
1  @Asynchronous
2  public class AsynchronousResolverProcessor{
3    public Future<Level> processResolver(SecurityLevelResolver resolver) {
4      Future<Level> result = new AsyncResult<Level>(resolver.resolve());
5      return result;
6    }
7  }
```

In `SecurityLevelManager` is another method containing logic for parallel processing. It iterates through all `SecurityLevelResolvers` and retrieve their `Future<Level>` result. Then all `Future<Level>`s are iterated with same logic as in synchronous method. If an unexpected exception occurs then `SecurityLevelManager.synchronousResolve` is called.

```
1  public Level resolveSecurityLevel() {
2    Level highestLevel = defaultSecurityLevel;
3    Set<Future<Level>> set = new HashSet<Future<Level>>();
4
5    for (SecurityLevelResolver resolver : resolverInstances) {
6      set.add(asynchronousResolver.processResolver(resolver));
7    }
8    for (Future<Level> result : set) {
9      Level level = null;
10     try {
11       level = result.get();
12     } catch (Exception e) {
13       return synchronousResolve();
14     }
15     if (highestLevel.compareTo(level)<0) {
16       highestLevel = level;
17     }
18   }
19   return highestLevel;
20 }
```

5.2.4 Default security level

There is set default security `Level` which is used when there is not resolved any higher level. That level can be also understood as minimal level that every user has. Normally it is level corresponding to `1` and represented by class `DefaultSecurityLevel`. However developer can provide different default `Level`. He needs to provide his instance of `Level` in application and annotate it with `@PicketLink @DefaultSecurityLevel` and it will get injected instead of predefined default `Level`.

There are two reason to change the predefined default `Level`. Either the developer decides to change default level from `1` to some other for any reason. Or the developer provides own implementation of `Level` and therefore he must also provide own default `Level`.

5.3 Storing user's security level and invoking security level check

Authentication in PicketLink is done in `Identity` class and once user is authenticated the `Identity` bean holds also Account class representing his identity. Therefore storing security level in that class is obvious choice. Following method was added to `Identity` interface:

```
1  Level getLevel();
```

The best part of the process to place security level determining is during user's log in. In case the method `Identiy.getLevel` is invoked and the user is not logged in then level is determined based only on context and authentication method is not taken into consideration. The simplification of old log in workflow is bellow:

1. `Identity.login` is called
2. Check whether user is logged. If so then `UserAlreadyLoggedInException` is thrown and process terminates.
3. Method `identity.authenticate` is called.
    1. Check whether developer defined own authenticator, if not default one is used.
    2. `Authenticator.authenticate` is called.
    3. If authenticator is in state `Success` then user's account is returned and `Authenticator.postAuthenticate` is called.
    4. If authenticator is not in state `Success` then `null` is returned.
4. If account is `null` then result `Failed` is returned and process terminates.
5. Check if account is valid is performed. If not `LockedAccountException` is thrown.
6. `Identity.handleSuccessfulLoginAttempt` is called and inside this method user's account is stored in `Identity` bean and `LoggedInEvent` is fired.
7. Return result `Success`.

Determining security level should be carried only for user who successfully logged in as it is undesirable to check levels of authentication method for someone who does not pass authentication process. Therefore determining user's level is placed in `Identity.handleSuccessfulLoginAttemp` method mention in point 7 above. This line was added to the method:

```
1  securityLevel = securityLevelManager.resolveSecurityLevel();
```

With security levels in consideration it gives a sense to reauthenticate when user want's to rise his security level. It most of the cases it is done by using different authentication method, but user may choose to relogin when his context changes for one considered more trusted. In order to allow user to rise his security level some changes were needed in login process. The changes are made in points 2 and 3 mentioned above. Changed code is below:

```
if(isLoggedIn()) {
  if(securityLevelManager.resolveSecurityLevel().
    compareTo(securityLevel) <= 0) {
    throw new UserAlreadyLoggedInException();

  }else {
    validatedAccount = authenticate();
    if(validatedAccount != null){
      Property firstdeclaredField =
        getDefaultLoginNameProperty(validatedAccount.getClass());
      Object firstUsername =
        firstdeclaredField.getValue(validatedAccount);
      Property seconddeclaredField =
        getDefaultLoginNameProperty(this.account.getClass());
      Object secondUsername =
        seconddeclaredField.getValue(this.account);
      if(secondUsername == null ||
        !secondUsername.equals(firstUsername)){
        throw new DifferentUserLoggedInExcpetion();
      }
    }
  }
}
else{
  validatedAccount = authenticate();
}
...
```

Two changes in logic of the log in process are made. The first one is check whether new security level is higher then old. If not the user does not have any reason to log in again and exception is thrown. However the change permits user to relog to rise his security level.

The second change seems a bit more complicated but core idea is simple. When is the user trying to rise his level by logging into application again the application must ensure that he was logged in application before and not someone other. I.e. the user must be identical in both cases. The problem is how to obtain username of user. Class representing user's account does not have set fixed username variable. PicketLink allows variability and therefore field annotated with stereotype `IDENTITY_USER_NAME` represents user's username. Method `getDefaultLoginNameProperty` finds the holder of the username value for given class and retrieving it for given instance is easy.

The last modified method is `Identity.unAuthenticate`. Following line was added after logout of user.

```
this.securityLevel = securityLevelManager.resolveSecurityLevel();
```

Because this line is put after deleting user's account from `Identity` class the authentication method is not taken in consideration, same as in method `Identity.hasLevel`.

5.4 Usage

Usage of the `Level` in application is done via using different annotations. In this section is described what annotations are used for it and how they are implemented. Usage can be divided in two parts. The first is how to determine the level and the second one is how to use `Level` for authorization.

5.4.1 Defining user's security level

Developer may develop his own `SecurityLevelResolvers` as stated previously mainly for purposes of determining `Level` from context. However there are two predefined resolvers for determining `Level` based on authentication method.

There is special annotation `@SecurityLevel` used on `Authenticator` class or on credentails. Prepared resolvers extract level ouf the used class and use it for determining final user's `Level`. Annotation is bellow:

```
1  @Qualifier
2  @Target({ TYPE, METHOD, PARAMETER, FIELD })
3  @Retention(RUNTIME)
4  @Documented
5  public @interface SecurityLevel {
6    @Nonbinding
7    String value();
8  }
```

Usage is very simple - developer just annotates the `Authenticator` or credential class with the annotation. If the `Authenticator` or credential is used during login process then out of box resolvers will process the annotation. Usage on `Authenticator` class is simple:

```
1  @SecurityLevel("2")
2  public class CustomAuthenticator extends BaseAuthenticator{
3
4    @Override
5    public void authenticate() {
6      ...
7    }
8  }
```

When such authenticator is used user get `Level` at least 2. Usage with credential is similar:

```
1  @SecurityLevel("3")
2  public class SmsCode{
3    ...
4  }
```

If is such credential used in application then the `Level` 3 is determined using `CredentialLevelResolver`. Usage of credential is following in PicketLink:

```
1  public class SmsLogInController {
2    private String smsCode;
3
4    @Inject
5    DefaultLoginCredentials credentials;
6
```

```
    @Inject
    Identity identity;

    public void logIn() {
      credentials.setCredential(new SmsCode(smsCode));
      identity.login();
    }
    ...
  }
```

5.4.2 Using security level for authorization

Different annotations are used for authorization of methods and classes in PicketLink. Because PicketLink uses for authorization framework DeltaSpike [6] with its own `Interceptor` that entry needs to be added to beans.xml to make any of the PicketLink annotations work:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <interceptors>
    <class>org.apache.deltaspike.security.impl.extension.
      SecurityInterceptor</class>
  </interceptors>
</beans>
```

Basically there are two ways how to use `Level` for authorization of methods. The first one is normal annotation `@RequiresLevel` which defines level that user needs to poses to proceed following method or any method in class if class is annotated. Different annotations can be combined together and in that case all of them needs to be fulfilled. If the developer wants to utilize some more complex rule he can use annotation `@Restrict` which utilizes PicketLink expression language to describe complex authorization constrains.

`@RequiresLevel` annotation contains only one `String` value which represents `Level`. Implementation of the annotation needs to contain another annotation `@SecurityBindingType` to be resolved by DeltaSpike interceptor. They way the interceptor works is slightly complicated but in short DeltaSpike during application deployment add interceptor annotation to all methods which have any annotation with `@SecurityBindingType` and later when the method is being invoked the interceptor takes place before invocation. Implementation of `@RequiresLevel` is bellow:

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Documented
@SecurityBindingType
public @interface RequiresLevel {

  @Nonbinding
  String value() default "";
}
```

Annotation alone can not ensure logic of the authorization and therefore there must be some different method with authorization logic and it needs to be bind with the

annotation. That is done by annotation method returning `Boolean` with annotation which it should secure and annotation `@Secures`. Method has one parameter `InvocationContext`. If method return `False` then `SecurityException` is thrown normally.

In securing method for `@RequiresLevel` was used different approach. Because when authorization fails on insufficient `Level` it means that user actually have chance to do something about it. When security check for appropriate `Level` fails then completely new `Exception` is thrown. The thrown exception is `InsufficientSecurityLevelException`. At user's exception handling process can be that special exception caught and appropriate steps can be taken to inform user what happened and how he can correct current situation. Implementation of the exception is bellow:

```
1  public class InsufficientSecurityLevelException extends
2    PicketLinkException{
3
4    private static final long serialVersionUID = 101667538978437287L;
5
6    private Level level;
7
8    public InsufficientSecurityLevelException(Level level, String msg){
9      super(msg);
10     this.level = level;
11   }
12
13   public Level getLevel(){
14     return level;
15   }
16 }
```

Implementation of securing method is very simple, mainly because moving securing logic to `AuthorizationUtil` class. This is done because same check is performed for expression language and code in one place is easier to maintain (as is stated in don't repeat yourself principle). `AuthorizationUtil` contains only static methods because for the checks from expression language in PicketLink are required stateless classes without any injections. Method itself is not complicated:

```
1  public static boolean hasLevel(Identity identity, Level level){
2    return (level.compareTo(identity.getLevel()) <= 0);
3  }
```

In securing method by itself is annotation obtained from `InvocationContext` (this is done by private method which existed in PicketLink before security level modification) then the value from annotation is acquired and corresponding `Level` is created with help of `AbstractLevelFactory`. Finally the `Level` is compared to current `Level` stored in `Identity` class. The code follows:

```
1  @Secures
2  @RequiresLevel
3  public boolean hasLevel(InvocationContext invocationContext){
4    RequiresLevel requireslevel = getAnnotation(invocationContext,
5      RequiresLevel.class);
6
7    String level = requireslevel.value();
```

```
    Level requiredLevel = abstractFactory.getFactory().createLevel(level);
    if (!AuthorizationUtil.hasLevel(identity, requiredLevel)){
      throw new InsufficientSecurityLevelException(requiredLevel,
        "Exception text");
    }

    return true;
  }
```

Usage of the `@RequiresLevel` annotation is following:

```
  @RequiresLevel("3")
  public void makeOrder(Order o){...}
```

Simple security constrains can be comfortably described by the annotation above, but for complex rules there is expression language. Usage with the PicketLink expression language is a bit different. `@Restrict` annotation and its securing method in PicketLink already exist. It was required to add function to expression language for using `Level` in expression language. Therefore it is added function `hasLevel` to expression language. It was done by adding this line to `PicketLinkFunctionMapper`:

```
  addFunction("hasLevel", ELFunctionMethods.class.
    getMethod("hasLevel",String.class));
```

Also it was necessary to add to evaluation context `LevelFactory`. Evaluation context is used during evaluating function. Functions have access only to evaluation context and can not use normal injections or access global context. `LevelFactory` is added into evaluation context in class `ELProcessor` that way:

```
  @Inject
  private Instance<AbstractLevelFactory> abstractLevelFactoryInstance;
  ...
  evaluationContext.setLevelFactory(this.
    abstractLevelFactoryInstance.get().getFactory());
```

Also evaluation context had to be changed to hold `LevelFactory` instance. It was done adding private member to it and creating setter and getter method for it. That piece of code was added to `ELEvaluationContext` class:

```
  private LevelFactory levelFactory;

  ...

  void setLevelFactory(LevelFactory factory){
    levelFactory = factory;
  }

  LevelFactory getLevelFactory(){
    return levelFactory;
  }
```

The method in `ELFunctionMethod` class is responsible for all logic of the function. With all changes above done it is simple to make it work now. It obtains evaluation context, retrieves `Identity` and `LevelFactory` out of it and use `AuthorizationUtil` for `Level` check. Code follow:

```
1  public static boolean hasLevel(String level){
2    ELEvaluationContext evaluationContext = ELEvaluationContext.get();
3    Identity identity = evaluationContext.getIdentity();
4    return AuthorizationUtil.hasLevel(identity,
5      evaluationContext.getLevelFactory().createLevel(level));
6  }
```

By all the changes mentioned previously to PicketLink expression language is possible to write code such as this:

```
1  @Restrict("#{hasLevel('2')}")
2  public void dummyMethod(){...}
```

Previous example shows very simple usage of `@Restrict` annotation which is equal to annotation `@RequiresLevel`. An example of more complex rule is here:

```
1  @Restrict("#{hasLevel('2') or hasRole('manager')}")
2  public void dummyMethod(){...}
```

Chapter 6
Verification

Verification is very important part of the every work. It is not only necessary verify implementation. Whether it does not contain any major (or minor) bugs and to provide regression tests. It is also very important to verify design and how the code will be usable for its targeted user. Someone can even verify whether the given task was fulfilled.

Verification of implementation is something that is often skipped or made poorly, mainly because of lack of the time, money or will. Sometimes even programmers consider themselves error-free. But verification of implementation is not only for them and to test code they coded. It can often represent regression tests which ensure that once coded functionality is not changed or broken by accident during further development. Best way to ensure correct implementation are integration and unit tests. Integration tests are task of architect of the system or specialized tester. As long as this work is only part of the project where are not integration tests included explicitly this work does not cover them. Completely different story are unit tests. It is widely believed that it is duty of every good programmer to cover at least critical parts of code with unit tests.

Verification of the design is much harder and it covers wide variety of aspects. Good designed code exhibits above average performance, stability and integration with other application from functional point of view. Well designed application also allows users to use it easily and effectively. However measuring of those attributes is often very hard and therefore automatic testing is not suitable. For the work there are two chosen approaches to test the design. One is done by author of the thesis and it is to develop demo application which will prove easy to use design. The second one is to let the users use it and provide feedback. As long as the project to which is the work committed is open source it is not problem for anyone to use the application, view source code or provide directly feedback. The third way the design is ensured is by Red Hat quality assurance department. Because PicketLink is developed under Red Hat they take care about integration and stress tests in specialized department.

Verification of the objective is somehow hardest. It needs to be evaluated by human and in the best case by someone who did not developed it by himself. That can be done by various people. Often as verification of the objective is taken acceptance of the program by client (or someone other who assigned the task).

6.1 Unit tests

Unit tests are tests which purpose is to test the smallest possible fragment of code, called unit. In object oriented programming the element is class and the smallest segment of code is method. Therefore unit tests covers normally a method in class and rarely whole class. Code well covered by unit tests is much less prone to failures both due to errors during application creating as well as its maintenance and further enhancement.

6.1.1 Design of tests

Single unit test ensures one functionality of one method. It covers only logic included in the method and nothing more. That ensures that elementary functionality of method is valid and code is functional on the lowest level. However unit tests does not fully ensure correct cooperation between methods and especially their synchronization and access to shared resources as memory, file system or database, etc. Also unit tests are not meant to test larger parts of programs (like modules) and their integration in application.

There are many tests for the changes stated in previous chapter. Those classes (or changes in them done in that assignment) are covered by tests:

- `AbstractIdentity` — new log in logic.
- `DefaultLevel`
- `DefaultSecuritylevelManager` — synchronous and asynchronous processing of `Level`.
- `AuthenticatorResolver`
- `CredentialResolver`

Every class is covered by one test case. Only exception is `Abstractidentity` because of the limitations of the testing framework and the explanation will follow in next sections. Almost never it happens that tested class does not call another methods in same class or different class. As the unit tests test only one method or class it is required to provide some dummy code instead of the real one from another class. That way we can control all inputs into tested method or class and we can be sure that error did not propagate from called methods. Those dummy methods or classes are called stubs or mocks.

Authorization part of the code is not covered by unit tests. That is due to its strong connection with DeltaSpike framework and therefore testing of those parts would require integration tests. Also exceptions are not tested because they hold no logic and can be viewed as information holders (not it creators or processors).

6.1.2 Used technologies

For unit tests as testing framework is used JUnit. [19]. It was already used in the project and another test frameworks does not provide any significant benefits. JUnit has plenty of features and only part of them is actually used for testing of security levels. The used features includes

- Grouping tests in test cases.
- Predefined assertions.
- Test runners, in that case `Parameterized` test runner.
- Ability to define pre-test and post-test methods.

JUnit is tool for testing normal Java application. When it comes to Java EE with dependency injections to private members normally JUnit can not help with testing those classes. There are two ways how to make JUnit work even in environment with dependency injections in CDI environment. The first one is to code the classes that they use method parameters injections. The second method is to use or develop framework which will process injections using reflection which allows access to private members.

Also JUnit does not provide any mocking capabilities by itself. Creating mock classes can be done manualy for plain JUnit tests but it requires a lot of writing of dummy code. Therefore it was decided to use mocking framework. It had not only create mocks

but also inject mocks into tested class. Mockito [20] was already used in the project and it was decided to go with it. It provides both mocking or stubbing and also injecting of the mocks into tested class.

There is one problem which can not Mockito help to solve. It is unable to create annotated mocks. Therefore testing annotated credential or `Authenticator` is impossible with mocks. Fortunately security levels do not take into account logic inside `Authenticator` or credentials during determining `Level` and creating mocks manually is not hard as long as it concerns only two additional stubs.

6.1.3 Test cases

There are five basic test cases testing different classes. Those test cases contains roughly thouse test scenarior:

1. `AbstractIdentity`
   - Test basic correct log in.
   - Test logout.
   - Test use case when different use log in.
   - Test logging in second time with different `Level`.
2. `DefaultLevel`
   - Test `Level` comparing with values lower, same and higher.
   - Test `Level` comparing with another implementation of `Level`.
3. `DefaultSecuritylevelManager`
   - Test retrieving highest level from multiple different `Levels`.
   - Test synchronous retrieving.
   - Test asynchronous retrieving.
4. `AuthenticatorResolver`
   - Test correct resolving of `Level`.
   - Test missing annotation.
   - Test when user is not logged in.
   - Test when `Authenticator` is not defined.
5. `CredentialResolver`
   - Test correct resolving of `Level`.
   - Test missing annotation.
   - Test when user is not logged in.
   - Test when credential is not defined.

Those test cases should cover most of use cases of which can occur in the application. They contain all exception states or states where they should perform nothing. The correct functionality is tested as well. Especially tests concerning `Identity` are made preciously because that method is called in every application very often and it is core class of the PicketLink and its wrong functionality could disable whole framework.

6.1.4 Implementation of tests

Implementation of the test is in this section described in the same order as are test cases listed above. All of the previously stated test cases are held together in one class.

Only exception is `Identity` test case where it was impossible due to way how JUnit handles automatic parameters inputs. JUnit parameterized runner uses predefined parameters of the test for the whole test case and it can not define input parameters for single test. Therefore tests for `Identity` are separated in two distinct test cases which extends abstract test with common methods.

Abstract test case contains `Identity` class represented by `DefaultIdentity` implementation. Also in abstract test case are mocks for all dependencies of `Identity` and creation of `Authenticator` mock. In the example bellow you can see how mock injection is done. There are created mocks at first and after their declaration is object which should be mocks injected to. Finally all the mocks are initialized in `setUp` method. Method `setAuthenticator` is overloaded so it can provide easy usage with great variability. Code:

```
1  public abstract class IdentityTestCase {
2
3    @Mock
4    protected Instance<Authenticator> authenticator;
5
6    ...
7
8    @Mock
9    protected SecurityLevelManager securityLevelManager;
10
11   @InjectMocks
12   DefaultIdentity identity;
13
14   @Before
15   public void setUp(){
16     MockitoAnnotations.initMocks(this);
17   }
18
19   protected void setAuthenticator(){
20     setAuthenticator(true);
21   }
22
23   protected void setAuthenticator(boolean success){
24     setAuthenticator(success, null);
25   }
26
27   protected void setAuthenticator(boolean success, Account acc){
28     when(authenticator.isUnsatisfied()).thenReturn(false);
29     Authenticator au = getAuthenticatorMock(success, acc);
30     when(authenticator.get()).thenReturn(au);
31   }
32
33   protected Authenticator getAuthenticatorMock(boolean status,
34     Account acc){
35     Authenticator auth = mock(Authenticator.class);
36     if(status){
```

```
37          when(auth.getStatus()).thenReturn(Authenticator.
38            AuthenticationStatus.SUCCESS);
39        }else{
40          when(auth.getStatus()).thenReturn(Authenticator.
41            AuthenticationStatus.FAILURE);
42        }
43        if(acc == null){
44          acc = mock(Account.class);
45          when(acc.isEnabled()).thenReturn(true);
46        }
47        when(auth.getAccount()).thenReturn(acc);
48        return auth;
49      }
50    }
```

Class `BasicTestCase` contains three of the four tests for the `Identity` class. First test is basic test which proves whether `login` method works. There is set mock for `SecurityLevelManager` which returns `Level` when asked. Also authenticator is made basic - it always succeeds and makes mock `Account`. After the `login` method on `Identity` is `Level` stored in `Identity` compared to expected `Level` to ensure it was resolved. If you wonder why `Level` with another value then '1' (which is default) is not used it is because there is no need for it. As long as the `SecurityLevelManager` is mock it would not return the default one in case of any unexpected event.

```
1   @Test
2   public void basicLoginTest(){
3     when(securityLevelManager.resolveSecurityLevel())
4       .thenReturn(new DefaultLevel(1));
5     setAuthenticator();
6     identity.login();
7     assertTrue(identity.getLevel().compareTo(new DefaultLevel(1))==0);
8   }
```

The second test ensures correct behavior of `logout` method. First part of the test is same as in the `basicLoginTest`. It adds execution of `logout` method after successful `login` and then compares if the newly obtained `Level` corresponds to the `Level` which returns `SecurityLevelManager` on its second call. If you are curious why there can not be only that test instead of the previous one it is to determine where the errors occurs. If the `basicLoginTest` succeeds and `logoutTest` not it is sure that `logout` failed. If both tests fails it is sure that `login` method failed.

```
1   @Test
2   public void logoutTest(){
3     when(securityLevelManager.resolveSecurityLevel())
4       .thenReturn(new DefaultLevel(2),new DefaultLevel(1));
5     setAuthenticator();
6     identity.login();
7     assertTrue(identity.getLevel().compareTo(new DefaultLevel(2))==0);
8
9     identity.logout();
10    assertTrue(identity.getLevel().compareTo(new DefaultLevel(1))==0);
11  }
```

The last test covers scenario when user is trying to rise his `Level` but logs in as different user. In the test user 'Joe' logs in at first and then user 'Roe' tries to rise the level. `DifferentUserLoggedInExcpetion` is expected to be thrown:

```
1  @Test(expected = DifferentUserLoggedInExcpetion.class)
2  public void differentUserLogsInTest(){
3    when(securityLevelManager.resolveSecurityLevel())
4      .thenReturn(new DefaultLevel(1),new DefaultLevel(2));
5    Account acc = new User("Joe");
6    setAuthenticator(true,acc);
7    identity.login();
8    assertTrue(identity.getLevel().compareTo(new DefaultLevel(1))==0);
9    acc = new User("Roe");
10   setAuthenticator(true,acc);
11   identity.login();
12 }
```

The last test case is in separated class because of usage of the parameters for the test. Parameters are declared as collection of `Object` arrays and values of each array are inserted into constructor of the showcase. Class needs to be run with `Parametrized` runner. Usage of the JUnit parameters:

```
1  @RunWith(Parameterized.class)
2  public class DifferentLogInLevelsTestCase extends IdentityTestCase{
3
4    @Parameters
5    public static Collection<Object[]> data() {
6      return Arrays.asList(new Object[][] {
7      { 1 , 1 , false}, { 2, 1 , false}, { 1, 2 , true},
8      });
9    }
10
11   private int firstLevel;
12   private int secondLevel;
13   private boolean result;
14
15   public DifferentLogInLevelsTestCase(int first, int second
16     , boolean res) {
17     firstLevel = first;
18     secondLevel = second;
19     result = res;
20   }
21 }
```

Test itself tries to log same user twice. From the given parameters is clear that once without trying to rise level and once with same level. That should cause `UserAlreadyLoggedInException` which is caught. If no exception is thrown then `fail` is raised. Last set of arguments tries successful rise of `Level`. Test method:

```
1  @Test
2  public void userAlreadyLoggedInTest(){
3    when(securityLevelManager.resolveSecurityLevel())
4      .thenReturn(new DefaultLevel(firstLevel),
5      new DefaultLevel(secondLevel));
6    Account acc = new User("Joe");
```

```
    setAuthenticator(true,acc);
    identity.login();
    assertTrue(identity.getLevel().compareTo(
      new DefaultLevel(firstLevel))==0);

    if(result){
      identity.login();
      assertTrue(identity.getLevel().compareTo(
      new DefaultLevel(secondLevel))==0);
    }else{
      try{
        identity.login();
        fail("There were expected expection which did not happen");
      }catch(UserAlreadyLoggedInException e){}
    }
  }
```

`DefaultLevel` test case use parameters as well. It compares two levels in three cases — first one is smaller, they are equal, second one is higher. The test case tests whether `DefaultLevel` throws `SecurityLevelsMismatchException` when is compared to another implementation of `Level`. Those two tests:

```
  @Test
  public void testCompare(){
    Level one = new DefaultLevel(firstLevel);
    Level two = new DefaultLevel(secondLevel);
    assertTrue(one.compareTo(two)==result);
  }

  @Test(expected = SecurityLevelsMismatchException.class)
  public void testMismatch(){
    Level one = new DefaultLevel(firstLevel);
    Level two = mock(Level.class);
    one.compareTo(two);
  }
```

DefaultSecuritylevelManager is tested for synchronous and asynchronous resolving of the `Level`. First of all mocks needs to be initialized:

```
  @Mock
  private Instance<Level> levelInstance;
  @Mock
  private Instance<SecurityLevelResolver> resolverInstances;
  @Mock
  private AsynchronousResolverProcessor asynchronousResolver;
  @InjectMocks
  private DefaultSecurityLevelManager levelManager;
  @Mock
  Iterator<SecurityLevelResolver> iterator;
```

Test concerning asynchronous level resolving contains three different mocks of `SecurityLevelResolvers`.

```
  @Test
  public void asynchronousLevelResolveTest()
    throws InterruptedException, ExecutionException{
```

```
4     when(levelInstance.isUnsatisfied()).thenReturn(true);
5     SecurityLevelResolver one = mock(SecurityLevelResolver.class);
6     SecurityLevelResolver two = mock(SecurityLevelResolver.class);
7     SecurityLevelResolver three = mock(SecurityLevelResolver.class);
```

As the resolving is done asynchronously there is need to mock `AsynchronousResolver` to return `Future` class containing the `Level`. This is done following way:

```
8     Future<Level> futureOne = (Future<Level>) mock(Future.class);
9     when(futureOne.get()).thenReturn(new DefaultLevel(2));
10    when(asynchronousResolver.processResolver(one)).thenReturn(futureOne);
```

Also there is need to mock collection. This is done by mocking iterator which should return `True` for all result and be finished with `False` and mocking method `next` to return items from collection in order we want:

```
11    when(iterator.hasNext()).thenReturn(true,true,true,false);
12    when(iterator.next()).thenReturn(one,two,three);
13    when(resolverInstances.iterator()).thenReturn(iterator);
```

Final part of the test just resolve `Level` and compare it to the expected one:

```
14    levelManager.init();
15    assertTrue(levelManager.resolveSecurityLevel()
16      .compareTo(new DefaultLevel(3))==0);
17  }
```

Synchronous test is a little bit different. Initialization of `SecurityLevelResolvers` is a bit different because they return directly the `Level`. Three `SecurityLevelResolvers` are made altogether for `Levels` '2', '3' and '1'. There is shown initialization of test and logic for creation one of `SecurityLevelResolvers`:

```
1   @Test
2   public void synchronousLevelResolveTest()
3     throws InterruptedException, ExecutionException{
4     when(levelInstance.isUnsatisfied()).thenReturn(true);
5     SecurityLevelResolver one = mock(SecurityLevelResolver.class);
6     when(one.resolve()).thenReturn(new DefaultLevel(2));
```

Then asynchronous resolving needs to throw an exception. This is done by mocking `Future` object which will throw an ExecutionException:

```
7     Future<Level> fut = (Future<Level>)mock(Future.class);
8     when(fut.get()).thenThrow(ExecutionException.class);
9     when(asynchronousResolver.processResolver(one)).thenReturn(fut);
```

The last part is similar to asynchronous resolve test. Its only difference lays in two iterations over collection and therefore iterator needs to be mocked a bit different way:

```
10    when(iterator.hasNext()).thenReturn(true,true,true,false
11      ,true,true,true,false);
12    when(iterator.next()).thenReturn(one,two,three,one,two,three);
13    when(resolverInstances.iterator()).thenReturn(iterator);
14
15    levelManager.init();
16    assertTrue(levelManager.resolveSecurityLevel()
17      .compareTo(new DefaultLevel(3))==0);
18  }
```

Last two test cases are for `AuthenticatorResolver` and `CredentialResolver`. As both of them are almost identical there will be shown implementation of `AuthenticatorResolver` tests and then described difference in `CredentialResolver` tests. To test correct functionality of the `AuthenticatorResolver` there needs to be created stub of the `Authenticator`. And because Mockito can not create annotated classes it was needed to create one manualy. Required classes do nothing or return null and are not interesting and therefore are omitted in the code preview:

```
1  @SecurityLevel("2")
2  class testAuthenticator implements Authenticator{
3  @Override
4  public void authenticate() {
5  ...
6  }
```

Initiation of mocks for test is following:

```
1  @Mock
2  Identity identity;
3
4  @Mock
5  AbstractLevelFactory abstractFactory;
6
7  @Mock
8  private Instance<Authenticator> authenticatorInstance;
9
10 @InjectMocks
11 AuthenticatorLevelResolver resolver;
```

Testing correct behaviour of the `AuthenticatorResolver` requires to simulate user logged in, returning previously created `testAuthenticator` from instance, mocking abstract factory and `LevelFactory` and finally checks whether `AuthenticatorResolver` called `LevelFactory` once with correct parameter:

```
12 @Test
13 public void successTest(){
14   when(identity.isLoggedIn()).thenReturn(true);
15   when(authenticatorInstance.isUnsatisfied()).thenReturn(false);
16   when(authenticatorInstance.get()).thenReturn(new testAuthenticator());
17   LevelFactory factory = mock(LevelFactory.class);
18   when(abstractFactory.getFactory()).thenReturn(factory);
19   resolver.resolve();
20   verify(factory,times(1)).createLevel("2");
21 }
```

Another test verifies whether `AuthenticatorResolver` resolves only if an user is logged in:

```
22 @Test
23 public void userNotLoggedInTest(){
24 when(identity.isLoggedIn()).thenReturn(false);
25 assertTrue(resolver.resolve()==null);
26 }
```

Very similar test to check behavior when developer did not specify own `Authenticator`:

```
27   @Test
28   public void unsatisfiedAuthenticator(){
29     when(identity.isLoggedIn()).thenReturn(true);
30     when(authenticatorInstance.isUnsatisfied()).thenReturn(true);
31     assertTrue(resolver.resolve()==null);
32   }
```

And last test checks whether developer did not use `SecurityLevel` annotation on `Authenticator` if the `AuthenticatorResolver` returns null:

```
33   @Test
34   public void missingAnnotationTest(){
35   when(identity.isLoggedIn()).thenReturn(true);
36   when(authenticatorInstance.isUnsatisfied()).thenReturn(false);
37   when(authenticatorInstance.get()).thenReturn(mock(Authenticator.class));
38   assertTrue(resolver.resolve()==null);
39   }
```

`CredentialResolver` is different only in that it checks credential stored in `DefaultLoginCredentials` with dummy `Credentials` class. describe tests by itself - nezapomnout na popis toho ze mockito neumi vytvaret oanotovany tridy a nutnost inner trid a parametrized runned jde jen pro jeden set vstupu

6.2 Demo application

Creating of demo application serves two purposes. It is one of very few methods how to prove that design of framework is capable of being used in real applications. Framework developer can verify on demo application if the framework allows all its functionality to be used and if so then if it is easy to use it. Developer also might find some integration bugs or parts of framework which are very hard to be effectively used in real application.

Demo application also provides nice overview of functionality both for new developers using it and for current developers as well. New developers deciding which framework to use will take a loot on demo application in first to see how it is used. A lot of them may take a look at demo application even before reading documentation. Developers who are decided already to use framework will welcome showcase as something showing them best practices and the way how framework was developed to be used.

6.2.1 Design

The first thing to clarify when developing showcase is what to show. In the designed showcase was needed to show usage of declaring security level through different `Authenticators` and usage of credentials. That can be understood as environmental context. Also it was needed to show usage of `SecurityLevelResolvers` based on user's spatial context.

As demo application was chosen simple e-shop with books. It distinguish two operational levels and default one. The default one is used for browsing e-shop. The second security level for normal authentication. This level allows user to see his previous orders. The third level lets user make orders and change personal information. To obtain basic operational right user needs just to log in using login/pwd. But for getting permissions for making orders and changing personal information user has do additional security checks. User can rise his level using SMS code or he can define trusted IP

level	rights	obtained by
one	browse e-shop	default
two	view order history	login/pwd authentication
three	place orders change shipping address change trusted IP address	use SMS code to verify user log in from trusted IP address

Table 6.1. Permissions and corresponding security levels in demo application.

addresses in personal information. Logging from trusted IP addresses will grant user second security level directly without SMS code.

As it is only simple showcase application it has only predefined set of items in shop and supports only single user. The simplification helps to keep it as simple as possible without dealing with item management and with user management as well. Such design also does not need any database or any other persistence tool. Application should also show how to handle various exceptions. It is required that application will distinguish between user not logged in and user that is logged in but not having higher security level.

6.2.2 Implemenation

Showcase uses JSF [21] framework for view layer. It provides the easiest usage. It contains many of predefined view components and thus speed up development of application. In showcase is not needed high performance nor any complex design or custom components therefore downsizes of JSF are not big concern. Application is written in Java EE and besides PicketLink it does not use any other frameworks.

Though application does not have any complex logic or user management it is separated in two layers. The first bottom layer provides data for the application. It populates application with predefined books and provides list of all books. It also manages orders — it populates default orders and allows to make an order and get the list of all done orders. Last of its duties is to hold user's settings.

Second top layer collects inputs from user and process them for bottom layer. In such simple application JSF pages get data directly from bottom data layer when they need to be displayed. Controllers for input data are used for changing user settings, two authenticators and mainly for storing actual shopping basket. Application consists only from main page with all functionality of e-shop and three minor pages used for exception handling, order payment and sms code authentication.

Exception handling is closely related to view technology used. For JSF used in application it was needed to define own `ExceptionHandlerFactory` and also own `ExceptionHandler`. Only `InsufficientSecurityLevelException`s are handled different way as only the security level is concern of demo application. Required `Level` instance is retrieved from exception and corresponding to it user is redirected to main page with error message telling him to log in or user is redirected to SMS verification page.

For normal log in is used `SimpleAuthenticator` with hard coded username and password. It has defined `Level` '2'. To show usage of security levels with credential two step authentication with SMS is done by special `SMScredential` which have defined `Level` '3'. `Authenticator` had to be implemented also for SMS log in but it does not have defined any `Level`. SMS code is also hard coded into corresponding `Authenticator`.

Last interesting part of the show case is `SecurityLevelResolver` implementation for determining user's IP address. It checks whether user is logged in and if so then it retrieves IP address from the request came. If it is same as the defined trusted IP address then it grant `Level` '3'.

6.2.3 Usage

Requirements for the showcase is Maven 3.0 [22] or better and Java 6.0 (JDK 1.6) or better. Demo application is designed to run on JBoss Enterprise Application Platform 6 [23] or WildFly. [24]

To deploy application an application server needs to be started at first. Also path to the maven needs to be added to system paths. For deploying the application following command needs to be used in directory with demo application:

- For EAP: mvn clean package jboss-as:deploy
- For Wildfly: mvn -Pwildfly clean package wildfly:deploy

This deploys application to server. Showcase is then available on address: `http://localhost:8080/picketlink-levels-complex/home.jsf`. To undeploy application from the application server this command needs to be used:

- For EAP: mvn jboss-as:undeploy
- For Wildfly: mvn -Pwildfly wildfly:undeploy

Application has only limited in memory storage with history of the orders and therefore restarting application resets state of it.

6.3 Acceptance by community

One of the ways how to measure whether the assignment was completed correctly is acceptance by someone who will use the application. In case of the open source projects it can be also acceptance of the contribution into project, good feedback by community and its usage.

Developing something for open source project can be very challenging. Unless there is clear feature request form community leaders then any commit can be rejected. First step when contributing any new idea is to verify basic ideas of the future contribution with project community. For the project stated in the thesis there have been created design document[1]). There is shown what is purpose of this work, what it would bring to PicketLink and simple proof of concept. Also there can be seen an discussion under document showing clarification of ideas. As the community leader liked the proposed feature it was encouraged to work further on it. That showed that the idea of security levels is not wrong and should be developed further.

Development and implementation of the project continued well. Open source community is great in that the people involved in can provide great feedback. There have been multiple online meetings with project leader Pedro Igor who provided highly appreciated feedback. The feedback to security levels was taken in consideration and final version of the code was developed. Pull request into master branch[2]) was accepted and that act can be understood as proof of usability and considerable added value into PicketLink project.

[1]) `https://developer.jboss.org/docs/DOC-52801`
[2]) `https://github.com/picketlink/picketlink/pull/423`

There were only minor changes in the code from pull request made by Pedro Igor for final version accepted into PicketLink. He removed asynchronous processing shown in this work because he did not want any more dependencies in PicketLink (the proposed asynchronous solution used enterprise beans). He proposed that if the security levels will be liked by developers and synchronous processing will be limiting to use tools that provides plain Java.

As the code was accepted to community in November and release of PicketLink takes around half of the year those changes did not make it to any final release, however those changes were included in candidate release PicketLink 2.7.0 CR3 release on December 23rd 2014. However as it was released very recently there is no feedback from developers using it. Showcase of the application is included in official PicketLink showcases since November 2014[1]).

Appreciation of contribution is shown in email send by project leader Pedro Igor. Text of the email is found in appendix A.

[1]) `https://github.com/jboss-developer/jboss-picketlink-quickstarts/pull/27`

Chapter 7
Conclusion

Purpose of the work was to extend current PicketLink framework with context aware elements, especially tied to authentication method.

In the work are described different ways how to implement context aware role based authorization control. However none of those existing methods could be used for given task because it was not possible to develop new implementation of RBAC with context aware elements but it was needed to add context aware elements to existing framework without changing its functionality.

As completely new way to deal with context aware elements was chosen context representation by security levels. Security levels are based on context and are very easily integrated into PicketLink and actually could be used for extending any RBAC system. Proposed solution uses another security element of levels to define how the user can be trusted based on his context. One of its main advantages is that it does not change framework to which it is integrated and security levels by itself provide big customization.

Security levels were implemented into PicketLink successfully. Also automatized unit tests were developed and submitted to project. As proof of successful implementation can be taken community appreciation of the work and its including into the master branch of project and its released in candidate release version of PicketLink.

As the security levels were released already there is huge expectation of feedback from real usage. Feedback from developers using security levels in PicketLink may provide great ideas how to improve it in future versions. Unfortunately the release was made very recently and there is no feedback yet.

7.1 Future work

During development of the context aware elements it came out that authentication method is part of context. This finding is pretty unique and it could be worked on more. There is not much research about authentication as part of context and its role in context.

One direction of future research can be trying to implement security levels in other authorization architectures. Concept of security levels seems to be very transferable and it could enrich another architectures. Using it with DAC, MAC, access control lists or many others could greatly improve those concepts.

Second direction of future research is to extend context abilities of the security levels. Possible research would involve not only one dimensional levels but some multidimensional representation of levels. It would require tools for level comparing and determining their similarity. For example for mobile application there is so many context properties that their description in one dimension is very hard and limiting.

Another direction of future research is closely tied to previous statement. Now is the highest level from all used used but it would be worth to make also research considering

some more complex logic. For example if final level could be some kind of multiplication of levels or levels would have not only their value but also their weight in final level.

References

[1] Ravi Sandhu. *Access control: The neglected frontier*. In: *Information Security and Privacy*. 1996. 219–227.

[2] M. Hitchens, and V. Varadharajan. Design and specification of role based access control policies. *Software, IEE Proceedings* -. 2000, 147 (4), 117-129. DOI 10.1049/ip-sen:20000792.

[3] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. *Towards a better understanding of context and context-awareness*. In: *Handheld and ubiquitous computing*. 1999. 304–307.

[4] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*. 2001, 5 (1), 4–7.

[5] *PicketLink Reference Documentation*. `https://docs.jboss.org/picketlink/2/latest/reference/html-single/`.

[6] *DeltaSpike Security Module*. `http://deltaspike.apache.org/documentation/security.html`.

[7] Matthew J Moyer, and M Abamad. *Generalized role-based access control*. In: *Distributed Computing Systems, 2001. 21st International Conference on.*. 2001. 391–398.

[8] Michael J Covington, Wende Long, Srividhya Srinivasan, Anind K Dev, Mustaque Ahamad, and Gregory D Abowd. *Securing context-aware applications using environment roles*. In: *Proceedings of the sixth ACM symposium on Access control models and technologies*. 2001. 10–20.

[9] Seon-Ho Park, Young-Ju Han, and Tai-Myoung Chung. *Context-role based access control for context-aware application*. 2006.

[10] Goran Sladić, Branko Milosavljević, and Zora Konjović. Context-sensitive access control model for business processes. *Computer Science and Information Systems/ComSIS*. 2013, 10 (3), 939–972.

[11] Devdatta Kulkarni, and Anand Tripathi. *Context-aware role-based access control in pervasive computing systems*. In: *Proceedings of the 13th ACM symposium on Access control models and technologies*. 2008. 113–122.

[12] Gustaf Neumann, and Mark Strembeck. *An approach to engineer and enforce context constraints in an RBAC environment*. In: *Proceedings of the eighth ACM symposium on Access control models and technologies*. 2003. 65–79.

[13] Ghita Kouadri Mostéfaoui, and Patrick Brézillon. *A generic framework for context-based distributed authorizations*. 2003.

[14] A Corrad, Rebecca Montanari, and Daniela Tibaldi. *Context-based access control management in ubiquitous environments*. In: *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*. 2004. 253–260.

[15] Joao Carlos D Lima, Cristiano C Rocha, Iara Augustin, and Mário AR Dantas. *A Context-Aware Recommendation System to Behavioral Based Authentication in Mobile and Pervasive Environments.* In: *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on.* 2011. 312–319.

[16] Le Xuan Hung, J Hassan, AS Riaz, SMK Raazi, Y Weiwei, Ngo Trong Canh, Phan Tran Ho Truc, Sungyoung Lee, Heejo Lee, Yuseung Son, and others. *Activity-based security scheme for ubiquitous environments.* In: *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International.* 2008. 475–481.

[17] Zhang Wendong, and Zhang Kaiji. *A role-based workflow access control model.* In: *Education Technology and Computer Science, 2009. ETCS'09. First International Workshop on.* 2009. 1136–1139.

[18] *Asynchronous Method Invocation - The Jave EE 6 Tutorial.*
`http://docs.oracle.com/javaee/6/tutorial/doc/gkkqg.html`.

[19] *JUnit - about.*
`http://www.junit.org`.

[20] *mockito - simpler & better mocking.*
`http://www.mockito.org`.

[21] *Mojarra JavaServer Faces.*
`https://javaserverfaces.java.net/`.

[22] *Maven -Welcome to Apache Maven.*
`http://maven.apache.org/`.

[23] *Red Hat Jboss Enterprise Application Platform.*
`www.redhat.com/en/technologies/jboss-middleware/application-platform`.

[24] *Wildfly homepage.*
`http://www.wildfly.org/`.

Appendix A
Letter of recognition

Hi Michal,

Would like to thank you for your contribution and all the hard work to improve PicketLink. The work that you performed in order to support Security Levels[1]) was an important addition to the project's requirements and capabilities.

It fits nicely in purpose of the PicketLink and it helps to move PickeLink forward. I appreciate your good coding skill, test coverage, the quickstart you provided to showcase the new feature and also all the time you spent writing the documentation.

I hope you can help us with more ideas and code !

Best regards.

Pedro Igor

[1]) `http://picketlink.org/news/2014/12/23/Release-2/`

Appendix B
List of abbreviations

OOP	Object oriented programming
MAC	Mandatory access control
DAC	Discretionary access control
RBAC	Role based access control
GRBAC	Generalized role based access control
xoRBAC	Extended object role based access control
EJB	Enterprise java bean
CDI	Context and dependency injection
SMS	Short message service
IP	Internet protocol. Sometimes IP is used for IP address as well
JSF	Java Server Faces

Appendix C
Content of CD

SourceCodes	folder with source codes
—showcase.tar	demo application
—PullRequest.patch	Pull request in git .patch file
—PullRequest.diff	Pull request in git .diff file
—picketlink-2.7.0.CR3.zip	PicketLink release with included pull request
Thesis	folder with thesis
—thesis.pdf	readable file of thesis
—thesis.zip	source codes of thesis written in TEX
ToolsToRunCode	applications needed to run the code
—apache-maven-3.2.5-bin.zip	maven for building the code
—wildfly-8.2.0.Final.zip	wildfly application server to run code on