Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

BACHELOR PROJECT ASSIGNMENT

Student: **Eldar Iosip**

Study programme: Software Engineering and Management
Specialisation: Software Engineering

Title of Bachelor Project: **Archive and Access Control in the Integration Portal**

Guidelines:

Familiarize yourself with application "Integration portal" created in [1-4], which is based on 3-tier enterprise application architecture implemented using Spring Framework. Implement an access control mechanism for files and folders for the Integration portal. Next, analyze requirements and implement policies and actions related to data archive. Finally, provide a study on files' security in the Integration portal environment.

Expected outputs of the thesis:
1. Analysis, implementation and test of access control mechanism for files and folders.
2. Analysis, implementation and test of data archive.
3. Study and improvements of implementation (if relevant) of files' security.

Bibliography/Sources:

1. STRNAD, Petr. Integrační server pro sdílení a zálohování dat. 2015. Master thesis. ČVUT FEL
2. HAŠLAROVÁ, Kateřina. Front-end pro portál pro sdílení souborů. 2015. Master thesis. ČVUT FEL
3. BLAŽEK, Jiří. Dokončení klientské části integračního portálu. 2015. Bachelor thesis. ČVUT FEL
4. FENGL, Vlastimil. Backend pro integrační portál. 2015. Bachelor thesis. ČVUT FEL

Bachelor Project Supervisor: Ing. Ondřej Macek, Ph.D.

Valid until the end of the winter semester of academic year 2016/2017

[redacted]

doc. Ing. Filip Železný, Ph.D.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-
L.S.

[redacted]

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, October 8, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Bachelor's Thesis

Archive and Access Control in the Integration Portal

Eldar Iosip

Supervisor: Ing. Ondřej Macek Ph.D.

Study programme: Software Engineering and Management

Specialisation: Software Engineering

January 12, 2016

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Macek Ph.D. for his useful advice during the development process and his corrections during the writing of this thesis and also I would like to thank Bc. Radek Ježdík for introducing me to the server side of the project. And last but not least I would like to thank my family for their support.

Declaration

I hereby declare that I completed this thesis without any improper help and I have listed all sources and publications used in compliance with the act “Metodický pokyn č. 1/2009”.

In Prague, January 12, 2016 ....

Abstract

Primary goal of this thesis is to extend current functionality for Integration portal by implementing access control mechanisms and policies related to data archival on the server side of the application. This work aims to describe analysis, design, implementation and testing processes during the development. The last topic provides a study of files' security.

Keywords: bachelor's thesis; java; acl; archival; integration portal; cesnet

Abstrakt

Předmětem této práce je návrh a rozšíření stávající funkcionality Integračního portálu o moduly spravující proces archivace a přístupová práva, včetně analýzy možných bezpečnostních vylepšení při ukládání souborů na fyzická média. V práci jsou prezentovány jednotlivé etapy analýzy, návrhu, implementace a testování. Implementace bude probíhat na serverové části portálu.

Klíčová slova: bakalářská práce; java; přístupová práva; archivace; integrační portál; cesnet

Překlad názvu: Archivace a přístupová práva v rámci Integračního Portálu

Contents

List of Figures

List of Tables

1 Introduction

These days many people use several major storage providers to help them organize their documents and also to get instant access from different electronic devices, such as mobile phones, laptops and PC's, smart televisions etc. In other words, from every device that is capable of browsing web pages.

Unfortunately, the amount of space initially reserved for each user account is limited and possible expansion may incur an extra fee. Because for most customers it is not hard to choose an alternative solution, for instance registering a new account or applying to a different service from another provider, their documents are spread across all these services and this makes them harder to maintain.

The main target of this thesis is a software called "Integration portal" funded under the grant, translated from its original title, "Integration of **CESNET** data services for science teaching groups". It eliminates the problems described earlier and also adds new features, such as data archival, integration with CTU (Czech Technical University) authentication system, labels for files and folders.

Work continues on previous bachelor's and master's theses written by Petr Strnad [1], Kateřina Hašlarová [2], Jiří Blažek [3] and Vlastimil Fengl [4].

Final product is divided into two separate projects, one for server-side code (a codebase for the current thesis) providing an API (Application Programming Interface) for the second project, which contains client-side code (a single-page application). Both projects are under the GIT version control[1].

Expected output of the current thesis is analysis, implementation and testing of an access control mechanism for files and folders stored within a portal, together with policies and actions related to data archival, and an examination of the files' security enhancements of the current system.

Following subsections will introduce each module in more detail.

[1] https://git-scm.com

1.1 Access control module

A typical use-case for the authenticated user is that he wants to share his documents with other users and groups within the system. Assuming that all the files are private by default, said user should create access rules, allowing the selected group of people to initiate the selected group of actions. This rules are customizing the way that user and group members are allowed to manipulate the file or folder (e.g. view, download, upload, modify its metadata, share with other system users and also delete). Created access rules are managed in the system for the whole lifecycle of the file or folder (from creation, followed by a possible shift within the file system structure, ending by its removal or archival processing).

A list of rules which define access to a file or folder is known as Access Control List (ACL). Each rule of this list is described as Access Control Entry (ACE) and holds the different combinations of Access Control Permission (ACP) instances. Within Integration portal, ACL has the same meaning as described below, but also with an extended functionality for user groups and folders.

1.2 Policy module

As mentioned earlier, there is a state of the file, when the archival processing should be activated. This is a particular task which also could have variable pre-processing and post-processing actions, e.g. to notify a user about successful task accomplishment or about an error occurred somewhere in between. The defined use-cases brings us to the Policy module.

Policy itself is a sort of action that is triggered on a specific Node instance and then the predefined rules modify its state. The implementation of each action defines how the node's state will change – if it gets removed, archived or only updated.

1.3 Security

Integration portal may store a big amount of private files and as they are physically stored on external servers they have to be secured from unauthorized access. In this thesis are described possible security enhancements in storing files on the external servers with respect to the currently used frameworks and libraries. The general consideration is that Integration portal is storage-agnostic and the only way to handle the data would be solely through the system's interface. Data stored on the second end have to be unreadable if an attacker gets access to that storage media.

2 Refactoring

This chapter describes the modifications implemented in the system compared to the initial state of the application.

Main goal of all modifications was to preserve the backward compatibility and functionality of the affected parts of the system. To ensure its functionality, by the end of the refactoring it was always tested by the integration tests written earlier by the previous project maintainers [1] [4].

Refactoring process played the main role in familiarization with the system. It helped in discovering a flow through every layer of an application, when particular classes are called and how the exception handling works, and in writing new code.

2.1 Common parent for FileMetadata and Folder classes

In previous implementation there were two separate model classes describing file and folder - FileMetadata and Folder.

That seems to be sufficient for the purpose of the previous system modules, but not for the future ones being implemented - ACL and Policy modules. They require an abstraction over FileMetadata and Folder classes. That's the reason for creating a common parent class, called Node.

Node holds all common attributes and relations, so the concrete classes became lighter in their definitions. It also allows DAO (Data Access Object) to query the parent class in situations when business logic does not care about the concrete instances of the retrieved classes.

The ACL module stores the list of ACE instances inside each node, and the Policy module stores selected policies for further processing.

Obviously, the inheritance strategy should be as fast as possible and it should also avoid unnecessary overhead in queries, due to the main role of the file and folder instances in the system.

2.1.1 Implementation

As an abstraction layer over the database, system uses Java Persistence API specification, shortly called JPA. As the specific implementation of this standard, project is configured with Hibernate ORM (Object-Relational Mapping).

The most important part of the implementation was to correctly decide between different inheritance strategies, considering the ease-of-use with respect to the performance.

As a requirement, there should be a common table in the underlying database so other tables have only one foreign key, e.g. ACE table.

Choice is somewhere in between of the following three strategies: JOINED, TABLE_PER_CLASS and SINGLE_TABLE [5]. Next paragraph describes each strategy and general considerations, either using it for the implementation or not. A MappedSuperclass annotation is not preferable, because it does not create a common table but only injects its attributes into concrete class. The last strategy is the selected one.

JOINED strategy corresponds to the inheritance type, where the parent class and all its subclasses has its own tables in a database. The parent class holds the attributes common to all members of the inheritance, so the uncommon attributes could be found only in subclasses [6]. The disadvantage of this strategy for the purposes of the system is a big amount of **SQL** Join operations when selecting subclasses. It can negatively affect the performance of such an often-used query as selecting files or folders independently.

TABLE_PER_CLASS strategy corresponds to the inheritance type, where parent class and each subclass has its own tables in a database. This approach is different from the previous strategy, because it copies all common attributes from the parent class into every subclass [7]. The disadvantage of this strategy is a big amount of data duplicated across the database tables and the worst performance in queries (using **SQL** Union [8]).

SINGLE_TABLE strategy corresponds to the inheritance type, where only one table is created in a database and it holds all attributes, that all classes - parent and all children - define [9]. To distinguish subclasses, it uses a special column called Discriminator. This column holds the identifier for every subclass, so an ORM could efficiently query them. Due to the fact that FileMetadata and Folder classes have lots of attributes in common, it would be proper to select this strategy.

2.2 The Adjacency List

This section describes the current database schema which holds a virtual file system structure, presented to the end user, and also its advantages and disadvantages in terms of performance and amount of queries targeted to the database. Further section will describe another approach to this problem, comparing it to the current implementation described below.

2.2.1 About

The main principle of the Adjacency List is that each member holds a reference to its parent, where topmost elements are indicated by the parent reference set to NULL [10]. It is fairly simple to represent the model graphically, but from the view of the relational database it could problematic to make it work efficiently without additional algorithms being applied. Table 2.2.1 provides an example of this structure.

ID	Parent
1	NULL
2	1

Table 2.2.1: Adjacency List structure

2.2.2 Hibernate ORM

As the abstract layer over a database, system uses a Hibernate ORM[2] on top of PostgreSQL[3]. Hibernate provides a JPA interface described earlier, so developers don't have to manage the database schema by hand, which makes code simpler to maintain.

For the selected topic it is important to understand the underlying process of retrieving collection elements. Hibernate has many different strategies for Lazy entity loading [11]. These strategies enable efficient retrieval of entities with big collections inside, in our particular case it is a Node entity. The Node holds a collection of its children on the file system – files and folders. In most situations, directories are created to store a big amount of files inside of them and probably other folders and so on. But what if after the file retrieval, system will use only folder's name and other attributes and collections stay unused? In that case, a developer should select an appropriate collection fetching strategy, depending on the defined use-cases [12].

Usually, fetching strategy is set to default in mapping files and a custom strategy is used within a selected transaction [13].

[2] http://hibernate.org/orm/
[3] http://www.postgresql.org/about/

3 Analysis

This chapter is divided into 3 sections, where each of them represents an analysis of each module introduced in 1.1 and 1.2. It defines the requirements and possible restrictions that may occur during the implementation. Subsections may vary depending on the analyzed module.

Implemented modules should not break the existing functionality but rather extend it with the use-cases defined further. There is an assumption that newly created modules would be implemented using the existing project structure and library dependencies for developing and testing.

All the modules logic has to be stored on business and database layers of the system, and behave as a mediator between the Integration portal and the CESNET storage, managing the permissions before the user get access to the physical media. In other words, no special settings related to the ACL and Policy modules and possible security improvements will be made in the CESNET file system.

3.1 ACL module

Further sections introduce the module's problem domain. Starting from the requirement analysis with a detailed explanation of each case, followed by the domain model with an explanation of each class represented inside.

3.1.1 Expected output

ACL module should cover all pre-defined manipulations with the stored files and folders described by the use-cases below, while also preventing unauthorized access. Each use-case should be tested to ensure modules functionality in a possible further refactoring process.

3.1.2 Roles

The following section describes everyone who will use ACL modules functionality, which could also be another system outside of the Integration portal.

Registered user – this is the most general role in the system and it provides user credentials that ACL module could use to reference the ownership to the other registered users. The purpose of the ACL module is to give every registered user a tool to manage their files' access permissions.

Node owner – a concrete role that gives its owner the ability to manage ACL with all its available features. Assumes that it is possible to be a folder owner, but the content of said folder could be owned by different owners with whom the folder was shared.

Granted user – it could be a group or one particular user who has privileges to interact with a node, in other words "ACE target". Depending on the selected ACP, the user could modify a node's state or share it with others. If the target of an ACE is a group, all of its members have the same privileges, excluding those targeted explicitly in another ACE of the same node. To these users apply the most restrictive rules from both sets.

3.1.3 Requirements

Requirement is a textual representation of a condition or capability needed by a user to solve a problem [14]. Each requirement could be a member of either functional or nonfunctional category [15].

For all categories, regardless of their type, a unique identifier exists. It is composed from a combination of letters, describing the module name, and also its numeric order, created for a better referencing in further parts of the thesis.

The content of each requirement should be at least SMART [16].

3.1.3.1 Functional requirements

Functional requirements are written in a format also known as a "user story" [17]. Each story contains a brief description of the selected functionality, available to the target user, optionally expanded by an explanation of its benefit.

Following requirements are not ordered by the priority but rather by the user role it targets.

REQ-A-001 – As a file owner, I want to share the file with a selected user so he could see the file in his "Shared" section.

REQ-A-002 – As a file owner, I want to share the file with a selected group of users, so they can see the file in their "Shared" section.

REQ-A-003 – As a file owner, I want to restrict the permissions for selected user or group.

REQ-A-004 – As a folder owner, I want to restrict the permissions for selected user or group for all of the folder's contents.

REQ-A-005 – As a file owner, I want to restrict the permissions only for one selected user of the group, so I can narrow down selected permissions.

REQ-A-006 – As a folder owner, I want to restrict the permissions only for one selected user of the group, for the folder and its contents, so I can narrow down selected permissions.

REQ-A-007 – As a node owner, I want to restrict permitted users from modifying my own permissions for the node, so others could not restrict my access to the node.

REQ-A-008 – As a granted user, I want to edit permissions for the shared file/folder.

REQ-A-009 – As a granted user, I want to copy file/folder to my own space, without original permissions, so I can create a local copy and become the owner.

REQ-A-010 – As a registered user, I want to view all the files and folders shared with me.

REQ-A-011 – As a registered user, I want to view all the folders shared with me in the system in the first level of the "Shared" folder.

3.1.3.2 Nonfunctional requirements

The following requirements are perceived as system features not visibly available to the end users. They have to be considered in the early phases of the module's architecture and be respected in the implementation phase [18].

REQ-800 – User permissions are more preferable than permissions defined for a group, that the user is member of.

The general purpose of this requirement is to explicitly extend or restrict access only for the selected group members. It may be useful in situations when group member is a student and information in a folder is private to teachers he works with – so folder owner will store an empty ACE for this student and he won't see it in his shared files.

REQ-801 – User cannot assign any permissions to his space root, so other members could subscribe to the updates of his file structure. Space root is always the private place for the currently logged user. Otherwise every newly created file or folder will be shared with some group of users, which may cause an unwanted leak of information.

REQ-802 – Only the file/folder owner or a user with a relevant right set of permissions has an ability to modify the list of permissions, no matter if they are newly created or modified existing ones. It means that on every such request, system needs to check this condition.

REQ-803 – Moving a file/folder within the same folder should be ignored, so there is no unwanted calculations with the same result.

REQ-804 – If the user is a part of a large amount of groups with different permissions to the node, then the most restrictive permission set should be picked.

3.1.3.3 General considerations

Assuming that every part of the system is behind a security mechanism, it is possible to rely on the existence of the user profile for each request that has been received by the **REST**ful API (excluding the unauthorized attempts) [19].

Each Node should be capable to return the list of all user and group entries having any kind of permissions targeted to its instance.

If the list of ACE on a particular node is empty, it means that only its owner could access and manipulate the node.

To reduce the amount of database queries, and speed up the whole process of retrieving the permissions by the selected Node, there should exist a direct reference to the parent, holding permissions inherited by the node. This strategy allows to jump to the parent faster than by a standard traversal - going upwards by the file system parent reference and so on.

3.1.4 Domain model

From the requirements defined earlier, it is now possible to build a domain model and illustrate the appearing classes with all the relations between them. The ACL module has its own boundary so it is possible to see how the module interacts with other classes, implemented earlier.

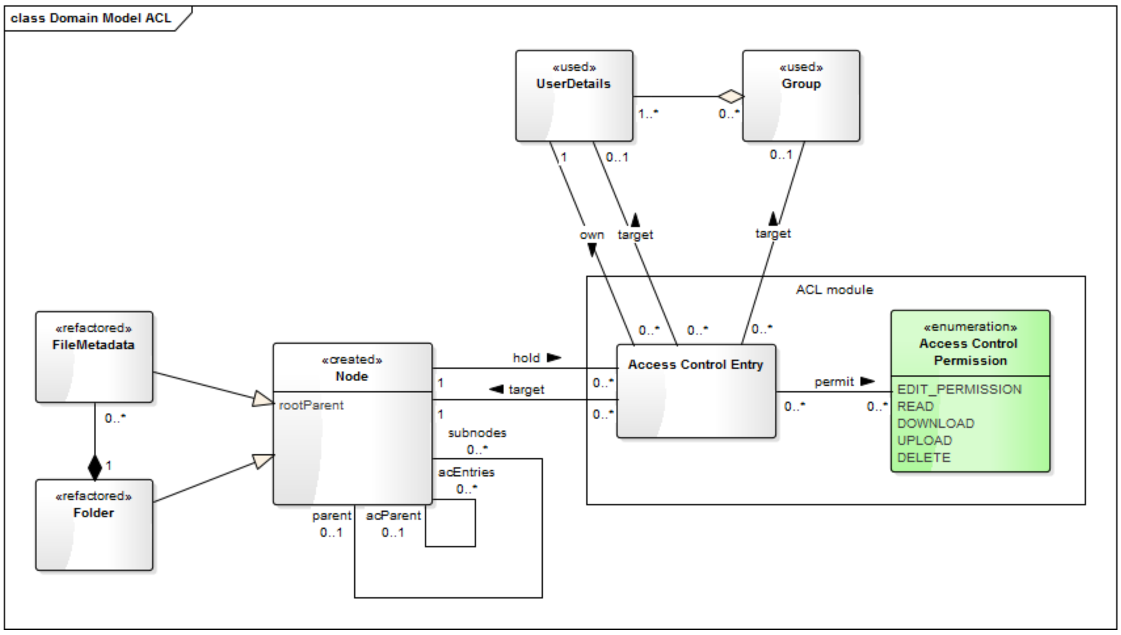

Figure 3.1.1: ACL Domain Model

3.1.4.1 Class explanation

Following subsections describe each class and its important attributes represented above, together with other classes and relationships between them. Some of the classes are taken over from the previous implementation – FileMetadata, Folder, UserDetails and Group. Their attributes were optionally refactored (as described in 2.1) to make them compatible with an ACL module.

3.1.4.1.1 FileMetadata

Holds a detailed information about the file, such as file size, mime type, lifecycle timestamps, reference to the parent folder and others.

3.1.4.1.2 Folder

Container for the FileMetadata and other Folder instances.

3.1.4.1.3 Node

Common class for FileMetadata and Folder classes. It makes it possible to write some common operations that could work with concrete classes without knowing about their type.

For the purposes of the ACL module, Node holds the reference not only to his parent in a file system tree but also to parent, holding the ACE instances. We will take a closer look into this structure in the next chapter.

3.1.4.1.4 UserDetails

Holds the information about the user in the system, such as record about his membership in particular organizational unit and other common information that user can provide.

3.1.4.1.5 Group

Containers, for the particular amount of UserDetails instances. Each group has its own name and an owner who can manage its members.

3.1.4.1.6 Access Control Entry

Holds a set of the Access Control Permission instances. It also targets to the selected Node and UserDetails/Group.

3.1.4.1.7 Access Control Permission

Set of the available permissions, which the user can assign to the ACE instance.

Available options are:

- READ_NODE - view a node from the own space
- WRITE_NODE - create new nodes in folder
- REMOVE_NODE - remove nodes
- EDIT_NODE _PERMISSIONS – allow to modify node's ACL list

3.2 Policy module

Following subsections, with a common structure as provided in ACL module analysis, will introduce policy mechanism inside the Integration portal. Policy and ACL modules do not provide any shared functionality, only common classes for both domain models can be referenced in section 3.1.4.1, where they were already described.

3.2.1 Expected output

Policy module should extend the functionality for the Node instances and operate under the control of CRON (Command Run On), managing the state of files and folders by the rules of the selected policy strategy. Policy module should be extensible in term of creating new policies.

3.2.2 Roles

Basically, the only person who can manage a list of policies for the selected node, is the node owner himself. In future implementations it may be connected with an ACL module, to share a privilege to manage policies with another user in the system.

3.2.2.1 Functional requirements

Functional requirements are written in the same format as it was described earlier in 3.1.3.1. To reflect a possible further connection with an ACL module, role name in each requirement is generalized from "node owner" into "registered user".

REQ-P-001 - As a registered user, I want to create a new policy for the node, so after the defined period of time it will be removed.

REQ-P-002 - As a registered user, I want to update the date of created policy on the selected node.

REQ-P-003 - As a registered user, I want to remove created policy.

REQ-P-004 - As a registered user, I want to show all policies saved with a node.

REQ-P-005 - As a registered user, I want to add multiple policies ordered by the date they need to be executed.

REQ-P-006 - As a registered user, I want to get a notification before any policy start to process, so I always stay informed about all impeding changes to files I own.

3.2.2.2 Nonfunctional requirements

REQ-700 – Count all unsuccessful attempts to process a policy.

REQ-701 – Make a Policy module extendable in terms of new policy types.

REQ-702 – Remove all policies after the node removal.

3.2.3 Domain model

Figure 3.2 presents a domain model for Policy module.

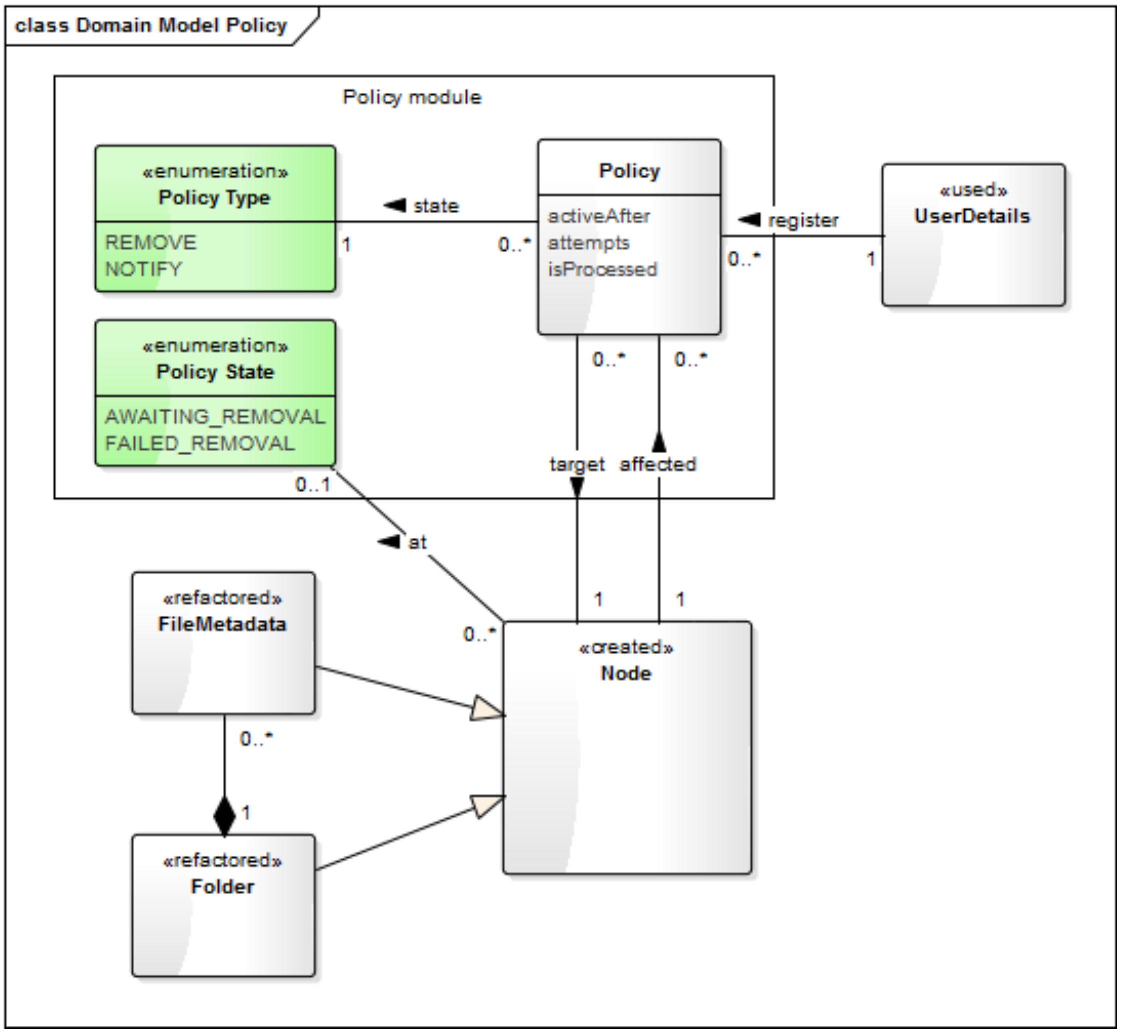

Figure 3.2.1: Policy Domain Model

3.2.3.1 Class explanation

The classes explained earlier in this text could be found starting from 3.1.4.1.1. A new classes created for a Policy module are described further.

3.2.3.1.1 Policy

Holds the policy type and several other attributes, such as *activeAfter* and *attempts*.

- activeAfter - is a date object, describing when the selected policy becomes active and can be dispatched by the system.
- attempts - holds the number of unsuccessful attempts to dispatch the selected policy. Could be used as a counter to indicate that something went wrong and fall back to another policy if possible.

3.2.3.1.2 Policy Type
Set of the available policy types. They define the action that will be triggered, when the activeAfter date will be relevant.

Available options are:

- REMOVE – remove a node recursively when the activeAfter expires.
- NOTIFY – inform a user that next policy will be executed in near future. It makes possible to report upcoming actions, via portal's frontend or by summary email sent to the end user.

3.2.3.1.3 Policy State
Set of the available states obtainable by a node when affected by the policy. This value is stored as an attribute of the Node class so it can be presented to the user, to provide a relevant feedback about its state.

Available states are:

- AWAITING_REMOVAL - it means that node is affected by the Remove policy.
- FAILED_REMOVAL - after a few unsuccessful attempts node could not be removed, therefore it is an indication of failure. Could inform the owner so he could do it manually or report an error. This state may be created when some of the external services are offline.

4 Design

This chapter describes the design process where the selected, early introduced requirements were considered and implemented. As the design process took the largest part of the time, the following text will describe all the initial thoughts from the analysis part, optionally followed by pseudo code snippets implemented in the system.

4.1 ACL module

Having lots of different manipulations inside an ACL tree, sometimes it is hard to describe all attributes that should be set when node moves; therefore, these descriptions are accompanied by pictures visualizing nodes state and its next position.

4.1.1 Terminology

Following list contains all shortcuts invented during the design process illustrated below.

ACL parent – it is a node with a list of ACE instances.

ACL tree – it is a virtual tree (virtual because it doesn't have any representation media it applies on, like file system tree has). There are 2 types of nodes in this tree. First, which defines his own ACE and behave as an ACL parent for its children nodes and second, which inherits ACE from its ACL parent (so its ACE list is empty).

ACL subroot – it is a first type of node described in ACL tree shortcut above.

4.1.2 Calculation of the ACL parent

As it was described in 3.1.4.1.3 and 3.1.3.3 a Node has to know about its ACL parent, placed in upper levels of the ACL tree.

The following pseudo code shows a calculation of the new ACL parent. A detailed explanation could be found in following sections starting from 4.1.3.1.

```
get ACL parent by file system (parent) {
   IF parent is NULL
      return NULL
   IF parent ACL is not empty
      return parent;
   ELSE
      return parents ACL parent
}
```

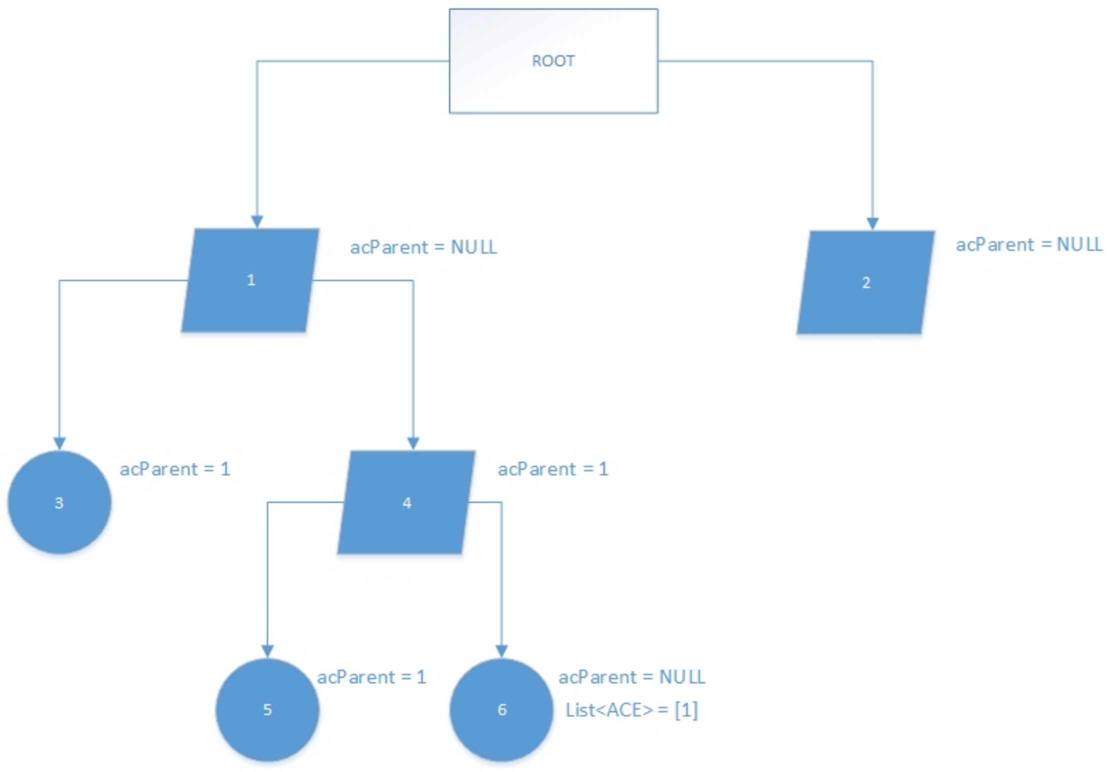

Figure 4.1.1.2: ACL Parent calculation

4.1.3 Possible recalculations of ACL parent

The following cases are illustrated to demonstrate possible use-cases that may occur when using an ACL module’s functionality. All manipulations presume that they are inside the same user space. Extra case when node is copied into another space is described in a last subsection.

4.1.3.1 Node created inside a space root

Whether a created node is an instance of FileMetadata or Folder, neither can have any subnodes on their creation, so there is no need to propagate any ACL inside the node.

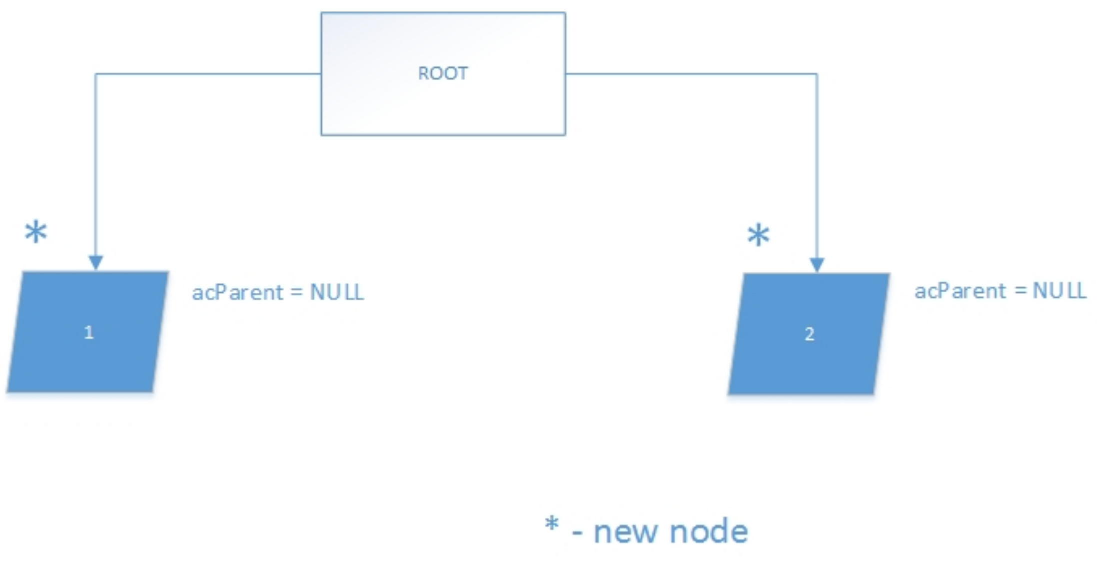

Figure 4.1.2: Node created inside a space root

4.1.3.2 Node created in the file system tree on level deeper than 1

Assuming that space root depth inside the file system tree equals to zero, the computations processed are the most trivial in terms of the ACL module described earlier in 4.1.3.1. In tree depth greater than zero more complicated situations regarding ACL parent recalculations can be found.

Node moved into another folder with inherited ACL

On every new file or folder added into already created node, parent's ACL should be automatically inherited. Assuming that node is placed into another folder affected by a different ACL, the previous ACL should be truncated and replaced by a new one. A reference to the ACL is calculated from the destination file system parent.

```
IF ACL parent of node file system parent is NULL
    return file system parent instance
ELSE
    return ACL parent of file system parent
```

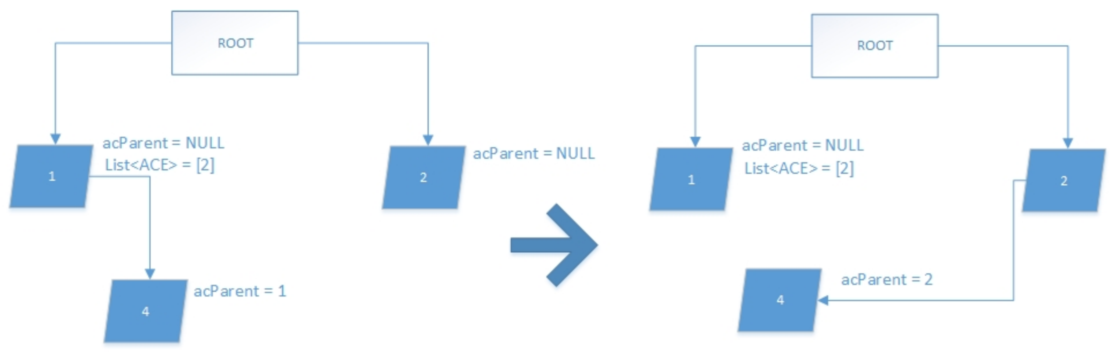

Figure 4.1.3a: Node moved into another folder

Node moved from folder into the space root

This is a special case, because space root can't have any ACL for the whole space, as specified earlier in 3.1.3.2 at REQ-801. So every node moved or copied from the deeper levels of the file system tree becomes private to the current user (ACL is removed when file changes its destination), setting its ACL parent to NULL.

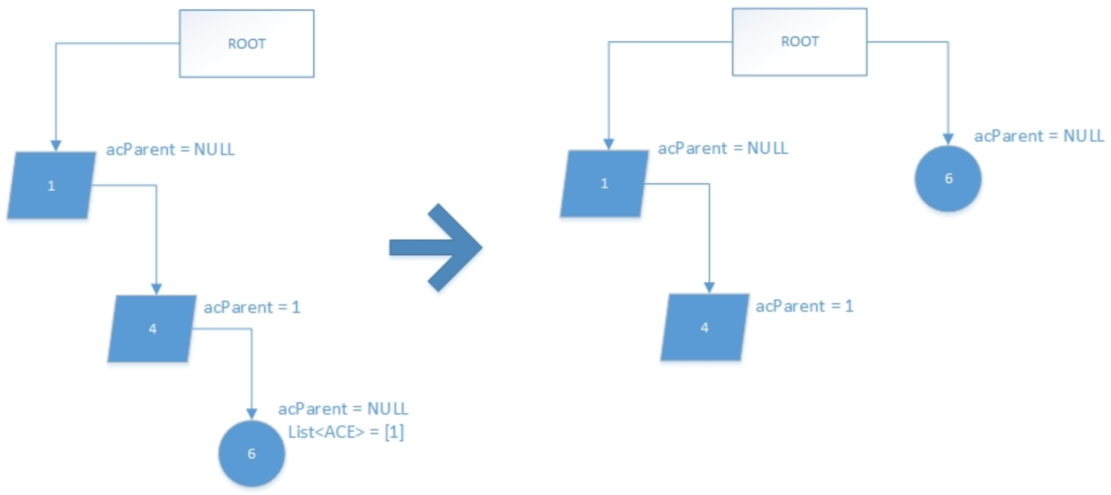

Figure 4.1.3.2b: Node moved from folder into the space root

4.1.4 Creating a new ACE

Creation of a new ACE instance requires the knowledge about the target node and user/group to which the node owner wants to modify access. According to the selected combination of ACP entries, created ACE instance could affect its underlying nodes reducing or extending access control permissions for the target user or group.

For the most cases, when ACE is created inside the file system tree (in levels between 1 and n, where n is the current maximum depth of the file system tree), the current node copies whole ACL from its referenced ACL parent as presented in Figure 4.1.4.

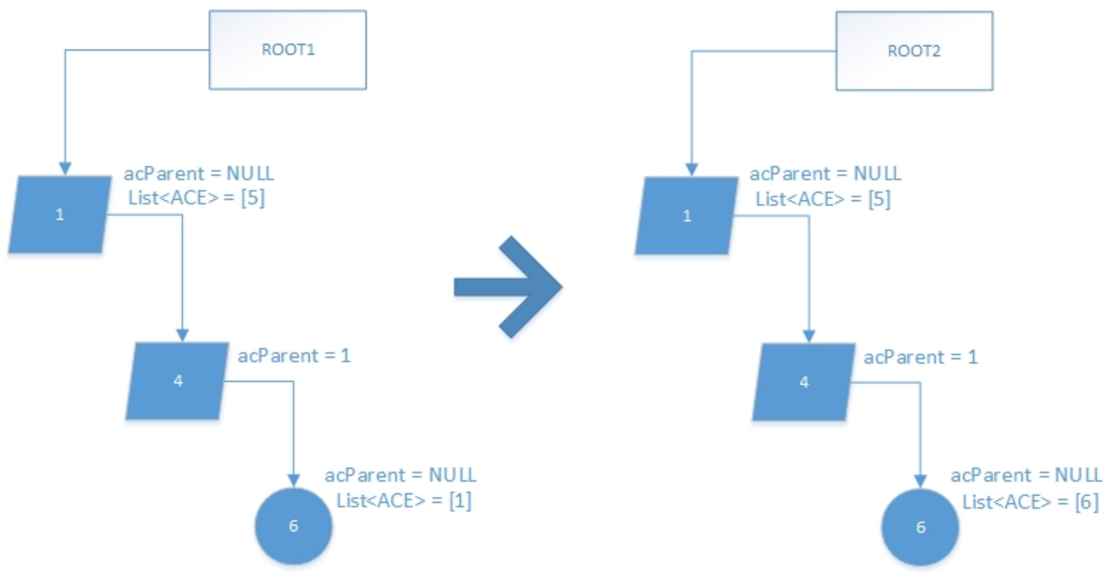

Figure 4.1.4: Creating a new ACE

This approach was intended to solve a problem with a big amount of requests targeted to a database, caused by a wide spread of ACE entries in the ACL tree where each node affected by the ACL holds a list of its ACE instances. Problem occurs at the moment when system wants to know if selected user has access to the requested node. On this type of request, it will traverse an ACL tree upwards to find all ACP for user and also for groups he is member of. This is an extremely demanding process assuming how frequent the ACL querying is. So the approach with the ACE copy was selected for the implementation.

The downside of the selected solution are possible duplicates and overhead on ACE update, traversing downwards the file system tree, but the read operation becomes faster, assuming that most of the queries will read, rather than write.

4.1.5 Updating an existing ACE

The updating process of an ACE is the same as on its creation, depending only on the stored ACL parent. The only optimization of this case, that if node has no any ACE records after the update, ACL tree under that node get recursively recalculated, removing unnecessary ACL subroots (Fig. 4.1.5), to boost performance on future read requests.

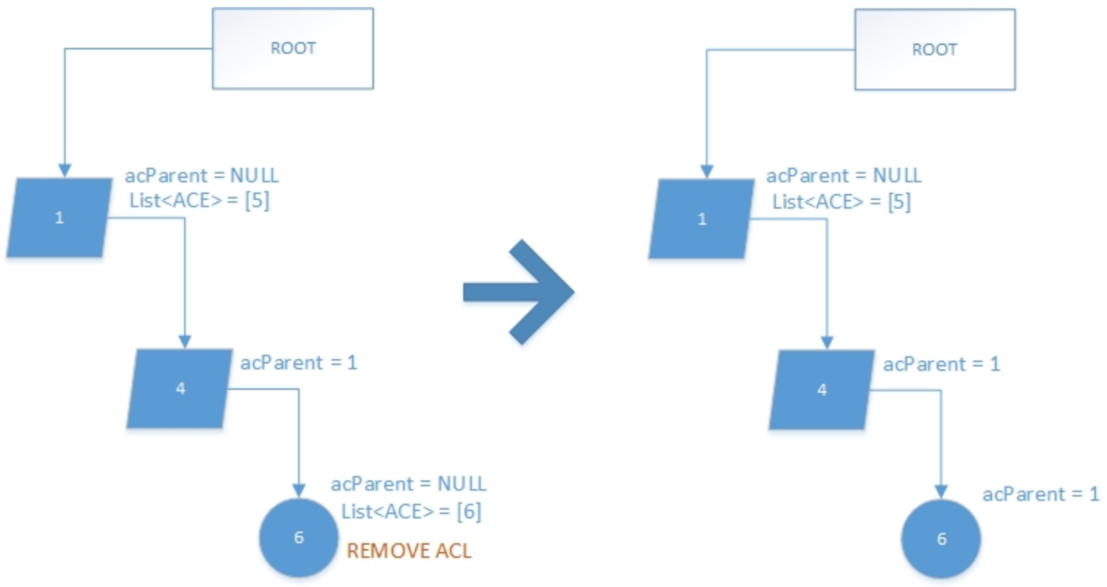

Figure 4.1.5: Updating an existing ACE

4.1.6 Copying a Node from other space

When current user wants to copy node into his own space, a strategy that will be used for an ACL copy must be selected. User can choose whether to keep original ACE entries or remove them. First strategy requires a recursive traversal with possible merge of each ACE from the original source with a new ACL parent. The second only requires the knowledge of the destination folder. In both cases, newly created node inherits an ACL from the upper nodes, by process already described in 4.1.3.2.

Implicit action for copying is to ignore original ACL, to keep the meaning of the operation clean and eliminate an ambiguous result.

4.1.7 Removing a Node

In general ACL is stored as a collection of ACE instances inside a Node and by the enabled cascade should be removed recursively as well as the Node instance. This step should manage the ORM (Object-Relational Mapping) with an appropriate setting by itself.

4.1.8 Conclusion

In the simplest case of the permission retrieval is a θ(1) and the worst is θ(k), where *k* is the amount of levels having nodes with a custom ACL. The worst case scenario is θ(n) under the assumption that each folder modifies the set of the allowed users and groups, so it become a new subroot from the ACL point of view. An optimization of the worst case scenario was described in 4.1.4.

4.2 Policy module

This section describes how the Policy functionality was implemented. Because it was a second module from the list of the expected output and refactoring was already done, the system was already prepared for the new module.

4.2.1 CRON service

System defines a service for all CRON tasks that may occur in portal's flow. Each method of the service is annotated by the annotation called Scheduled. It takes a string parameter with a CRON expression defining when the annotated method should be executed [20].

For the purposes of Policy module, CRON task is scheduled to run every day at 6:00 am, so the user can get a notification before another task removes archived files.

A notation is "0 6 * * * ?".

Following diagram (Fig. 4.2.1) illustrates a process that CRON service initiates.

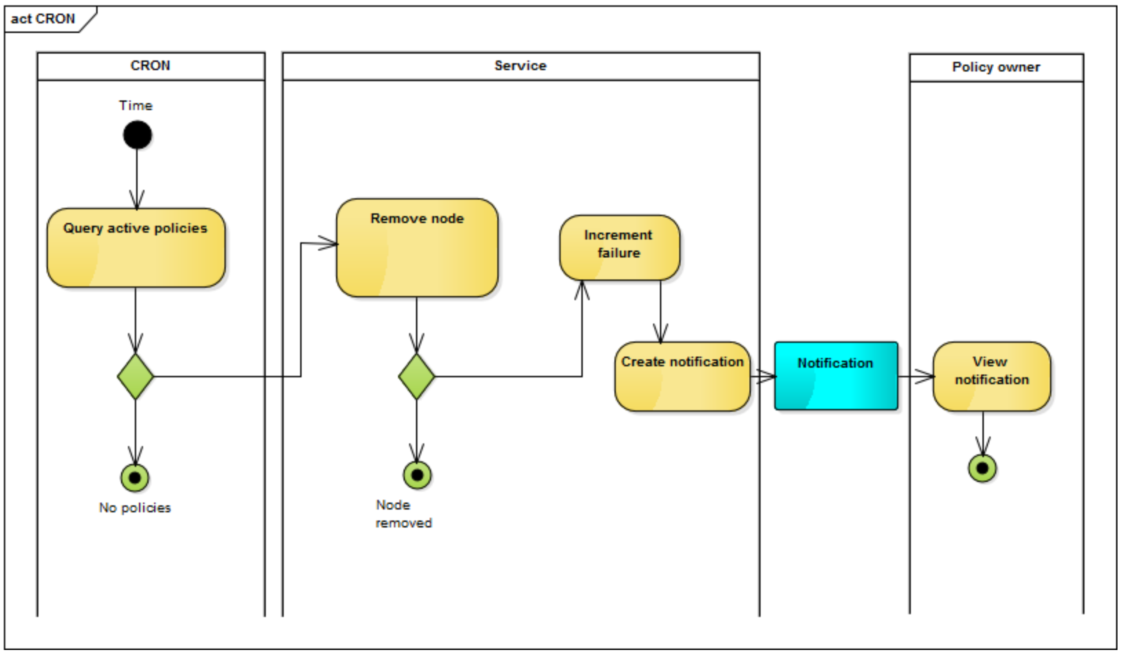

Figure 4.2.1: CRON service

4.2.2 Target node

Each policy irrespective of its type has to be targeted on a particular Node instance and hold a reference to its owner. The Node can have various policies ordered by date in an ascending order.

4.2.3 Node policy state

After CRON finishes its task, any node with DELETE policy will indicate its state as AWAITING_REMOVAL and any node with NOTIFY policy will change its policy state into AWAITING_NOTIFICATION, so other system modules, e.g. Notification module, will pull the nodes by this state and notify their owners about upcoming event.

Task that will remove the nodes will run last, and no other policies or states will be set.

4.2.4 API

As most parts of the system, policy module has its own role in implemented API. It defines **CRUD** controller actions in *NodeController* class. It uses node as a resource and policy as module operation on it. Each action has the same URL but different HTTP method. E.g., if user wants to add a new policy, he sends a request using the POST method, and when he wants to delete he uses the DELETE method. It makes API URL clear and easy to understand.

5 Testing

During the introduction period it was very important to understand how each system component works and behaves. Already written tests helped understand what to expect from the output of each Service method or Controller response being tested.

In the current project, tests are divided by their target application layer – Controller and Service.

5.1 Controller

Controller tests target the API of the Integration portal and check if backend returns expected response (including http status code and content passed in its body) from the various requests that may be passed from the frontend application. As frontend application uses a JSON format for communication between its API level and backend's API endpoints, Unit tests have to understand the provided response and parse it, to be able to check selected major attributes, that confirm the response is valid. For that case Spring has a method *jsonPath* that tests if the selected JSON attribute matches an expected value.

5.2 Service

Service tests target specific service methods, and thus simplifies the whole process of finding possible errors, because of a smaller amount of code being executed. It may be useful in situations when a module has complicated business logic under one API call and requires many dependencies and mock instances to return a valid response. It is also very important to test how service methods cooperate with each other – they may do correct computations and serve valid responses on their own, but when some sort of aggregation logic with additional input parameters is needed, the methods may break.

5.3 Mock objects

Tests are based on mock objects that are defined statically in XML files. They have a simple structure and are used during database setup. Springockito[4] library provides a small extension that simplifies the process of creating mock objects and also provides several methods for managing the way mocks get reinitialized after the tests completion (e.g. after each method execution, after testing class finished its tests). A mocked file structure that handles different manipulations with an ACL was constructed for the purposes of testing the ACL module.

5.4 Additional tests

From the very beginning of the implementation, testing was handled manually. Manual testing helped discover every layer of the application step by step using a debugger tool, integrated in **IDE**.

[4] https://github.com/springockito/springockito

For manual testing a REST client called Paw[5] was selected, that helps organize various API calls into different groups and projects. For every tested request a JSON object is constructed, derived from the representation objects stored on backend, and with an appropriate method it sends a request to server. Response is parsed and provided to the tester. Advantage of this solution is that every call is stored within a project, it could be exported and it is possible to emulate a real usage of the API.

For automated testing an existing solution was used, as described in previous sections. ACL module has more complicated tests because it affects several nodes at once, so every changed state needs to be checked.

5.5 Results

ACL module was covered at 70% by automated tests and 40% by manual requests on API stored in a separate project (10 unit tests and 15 manual).

Policy module was covered at 80% by automated tests (5 unit tests).

12 unit tests that cover creation and manipulation of files and folders were also added. Correct assignment of an ACL parent is also checked during the test.

All calculations reflect a part of implemented tests against the amount of defined requirements.

[5] https://luckymarmot.com/paw

6 File security

Nowadays, the biggest part of the information is transported and stored on a variety of storage media in its digital form, so the data that may contain sensitive information should be properly secured. Primary target of the following text is to analyze current design of file exchange between Integration portal and CESNET storage and find additional libraries that could provide useful tools for securing files, stored on a physical media.

6.1 Terminology

Following terms will be often used in further text, as they are part of the Cryptography basics.

- Plaintext – plain text data in a readable format.
- Ciphertext – result of the encryption, something that appears to be meaningless.
- Encryption – process of translating plaintext into ciphertext.
- Decryption – process of transforming ciphertext back to plaintext.
- Key – secured parameter of the encryption algorithm.

6.2 Current transfer mechanism

As it was introduced earlier in 0, the Integration portal uses a storage media provided by CESNET. The communication between portal and storage server is handled via SFTP (Secure File Transfer Protocol) which enables secure network transfer between two points and also file system management capabilities (e.g. resuming interrupted file transfers, navigating within remote directories etc.) [21].

As it was mentioned in 1.3, Integration portal architecture allows to switch between different storage providers that may provide different capabilities in term of data encryption and may allow direct access to the files from their own interface that could be vulnerable to attacks. Further text will consider these situations and analyze possible solutions that could be implemented in specified context.

6.3 Types of encryption

There are two types of encryption: symmetric and asymmetric.

Symmetric encryption uses only one key (which is private) for encryption and decryption, and is often used in situations when there is only one endpoint, e.g. when files are encrypted and decrypted on the same machine (not counting combination solutions when used with asymmetric encryption). Comparing to the asymmetric encryption, symmetric encryption is much faster and it may have an advantage in database systems where performance is an issue [22].

Asymmetric type uses a pair of keys for its encryption activity, where first key is called public and second is private. Comparing to the symmetric encryption, asymmetric is often described as being more secure by not requiring a secure channel for initial exchange [23].

For both types of encryption, security depends only on keeping the private key private [24].

6.4 General assumptions

As the primary content of each user space is a collection of files, they can be in various sizes and formats. Every file on its creation should be encrypted with a random key. That key should never be derived from the user's password. If so, with an appropriate amount of iterations, a long re-encrypting process may be required, considering the possibility of the user storing gigabytes of data in his space.

6.5 Java libraries

Following sections contain the information about available libraries and extensions that could be used in implementation.

6.5.1 Java Cryptography Extension

As the system is written in Java, it has a big amount of security components related to authentication, secure communication, cryptography and others [25]. One of them is JCE (Java Cryptography Extension) which provides an API for the implementation of wide range of security providers [26]. For historic reasons, default key size in standard installation equals to 128 bit, in situations when a 256-bit key needed, it could be extended by a couple of jar files [27]. Added dependency creates a new step in deployment process that may be inappropriate for some environments.

6.5.2 Bouncy Castle Cryptography Library

It is very powerful, open-source cryptography library with a various implemented generators and can be used as a provider for JCE [28]. It also provides a light-weight API that is useful in situations when the basic configuration meets the requirements and added abstraction level could simplify the whole implementation. For the 256-bit key it is a prerequisite to install JCE from 6.5.1.

6.5.3 Java Simplified Encryption (jasypt)

It is a Java library which allows the developer to add basic encryption capabilities to project with minimum effort [29]. Could be used on top of Bouncy Castle library introduced in 6.5.2. It provides a support for binary encryption and integrates with Spring and Hibernate.

6.6 Concept

For the current implementation it would be correct to obscure property files so they don't get accidentally stored under version control or viewed by an authorized person. Such an occurrence may allow direct access to CESNET storage and lead to the information leak.

Privacy of the stored information should also be considered, and an appropriate encryption algorithm selected, that will secure data on the storage media with respect to the frequency with which files get encrypted and decrypted.

Libraries that were mentioned in 6.5.2 and 6.5.3 provide a different level of abstraction, but both work with byte arrays, so file encryption is possible. The fact that file encryption and decryption should be used when file leaves and approaches the Integration portal should be considered during the implementation – big files should be processed efficiently, without unreasonable memory usage that may occur when working with streams and byte arrays.

7 Conclusion

During the development of several additional features – ACL and Policy modules, provided in thesis guideline, the refactoring process provided a large advantage. It helped to get in touch with the system faster than by traversing code, that was previously maintained by several developers [1] [4].

To be compatible with previous implementations, a custom algorithm was chosen for the ACL module that can provide basic ACL functionality. As there are two types of users in the system – standard users and group members, there was a non-functional requirement that user's ACL is more valuable comparing to his group ACL. It was successfully implemented and validated by several integration tests in 5.5.

The Policy module was implemented to provide different management capabilities for the end user. Module was written as an add-on to a file and folder instance. It should be easily extended with additional policies that may provide additional functionality. The module was successfully tested with unit tests.

The code was written in accordance to the project's semantics and every created method is commented with an appropriate description.

For future development, the analysis part could help with further optimizations. Provided requirements may be extended for additional user roles in the system and access to the files may be categorized by them.

8 Bibliography

[1] P. Strnad, "Integrační portál pro sdílení a zálohování dat," ČVUT FEL, 2015.

[2] K. Hašlarová, "Front-end pro portál pro sdílení souborů," ČVUT FEL, 2015.

[3] J. Blažek, "Dokončení klientské části Integračního portálu," ČVUT FEL, 2015.

[4] V. Fengl, ČVUT FEL, 2015.

[5] "37.2 Entity Inheritance," [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm.

[6] "37.2.4.3 The Joined Subclass Strategy," [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00681.

[7] "37.2.4.2 The Table per Concrete Class Strategy," [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00680.

[8] "7.4. Combining Queries," [Online]. Available: http://www.postgresql.org/docs/9.0/static/queries-union.html.

[9] "37.2.4.1 The Single Table per Class Hierarchy Strategy," [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00679.

[10] T. Cormen, "Representations of graphs," in *Introduction to Algorithms*, 3rd Edition ed.

[11] "Lazy loading," [Online]. Available: https://en.wikipedia.org/wiki/Lazy_loading.

[12] "20.1. Fetching strategies," [Online]. Available: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch20.html#performance-fetching.

[13] "Improving performance," [Online]. Available: https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html#performance-fetching-custom.

[14] "Software requirements," [Online]. Available: https://en.wikipedia.org/wiki/Software_requirements.

[15] "Differentiating between Functional and Nonfunctional Requirements," [Online]. Available: http://searchsoftwarequality.techtarget.com/answer/Differentiating-between-Functional-and-Nonfunctional-Requirements.

[16] B. K. Mike Mannion, "SMART Requirements," April 1995. [Online]. Available: http://www.win.tue.nl/~wstomv/edu/2ip30/references/smart-requirements.pdf.

[17] S. W. Ambler, "User Stories: An Agile Introduction," [Online]. Available: http://www.agilemodeling.com/artifacts/userStory.htm.

[18] "Non-functional requirement," [Online]. Available: https://en.wikipedia.org/wiki/Non-functional_requirement.

[19] "Representational state transfer," [Online]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer.

[20] "33.4.2 The @Scheduled Annotation," [Online]. Available: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html#scheduling-annotation-support-scheduled.

[21] "What is SFTP, and how do I use it to transfer files?," 12 11 2015. [Online]. Available: https://kb.iu.edu/d/akqg.

[22] "Symmetric & Asymmetric Database Encryption," [Online]. Available: https://en.wikipedia.org/wiki/Database_encryption.

[23] "Asymmetric Encryption," [Online]. Available: https://support.microsoft.com/en-us/kb/246071.

[24] "Public-key cryptography," [Online]. Available: https://en.wikipedia.org/wiki/Public-key_cryptography.

[25] "Java SE Security," [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.

[26] Oracle, "3 Basic Security Architecture," [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html.

[27] Oracle, "Import Limits on Cryptographic Algorithms," [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html

[28] "The Bouncy Castle Crypto Package For Java," [Online]. Available: https://github.com/bcgit/bc-java.

[29] "Java Simplified Encryption," [Online]. Available: http://www.jasypt.org/index.html.

9 Acronyms

CESNET	Czech Education and Scientific NETwork
ACL	Access Control List
ACE	Access Control Entry
ACP	Access Control Permission
SQL	Structured Query Language
REST	Representational State Transfer
API	Application Programming Interface
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
CRUD	Create, Read, Update, Delete

Appendix A

Package Structure

Path	About
portal/src/main/java/cz/cvut/fel/integracniportal/*	Portal's main source
portal/src/test/java/cz/cvut/fel/integracniportal/*	Mocks and Service unit tests.
thesis.pdf	Thesis PDF file