Large scale secure automatic provisioning of X.509 certificates via web services

César Davide Barbosa Miranda

Master's Degree in Network and Information Systems Engineering
Department of Computer Science
2015

Supervisor
Manuel Eduardo Correia, Assistant Professor, DCC-FCUP
All corrections determined by the jury, and only those, were made.

The President of the Jury,

Porto, _____/_____/_________
Acknowledgements

I would like to thank Professor Manuel Eduardo Correia for his support, advices and revisions in my work to write this thesis.

I also would like to thank my colleagues in HealthySystems, namely Alexandre Augusto, Pedro Faria, Rui Martins, Ricardo Ferreira and João Graça for taking me in and offered their help and expertise in various matters, as also to HealthySystems to take me in and allowing me to develop the work made in this thesis.

I also have to thank my family, friends and my love, Rita for all their support, not only in the developing of this thesis but throughout all my course.
Abstract

The subject of this thesis is the efficient deployment of Public Key Infrastructures (PKI), more specifically, a way to securely and automatically provide and deploy X.509 certificates, using web services as a mean of communication between the Certificate Authority and end-entities.

This is highly relevant since more and more organizations are using PKI services in order to, for example, securely communicate through insecure channels like the Internet, digital signing operations or to provide non-repudiation properties to protocols and information systems.

Although PKI technologies have become pervasive among most institutions, there are still very few well known protocols that support automatic certificate provisioning; the Simple Certificate Enrollment Protocol (SCEP) and the Automatic Certificate Management Environment (ACME). However SCEP suffers from some security flaws and ACME is highly focused on a single PKI application domain.

We examined the problem and tried to understand if it is possible to more generally deploy such an automatic service and to what extent these generalizations can be considered secure. The thesis begins with a constructive research to analyze existing solutions, and from the research made, we devise and implement a fully working solution that we think satisfies our initial goals.

In this thesis we present and describe a fully implementation of an architecture to deploy such a service. Our system relies on the delivery of security elements, such as cryptographic tokens or smartcards containing an administrative certificate, to the PKI end-entities, as a security anchor.

We have also developed a framework architecture for PKI backoffice services, a data model for end-entity interactions with the backoffice PKI services and a secure Remote Procedure Call based API for end-entity client software to use for the automatic
provisioning of X.509 certificates, and other usual PKI operations.
Resumo

O tema desta tese é o desenvolvimento eficiente de uma Infraestrutura de Chave Pública (ICP), mais especificamente uma maneira de aprovisionar certificados X.509 de forma segura e automática, utilizando serviços web como meio de comunicação entre a Autoridade Certificadora e as entidades finais.

Este tema é interessante pois cada vez mais organizações utilizam serviços de ICP para, por exemplo, comunicar seguramente através de canais inseguros como a Internet, operações de assinatura digital ou providenciar propriedades de não-repúdio a protocolos e sistemas de informação.

Apesar das ICPs serem cada vez mais comuns nas instituições, existem poucos protocolos conhecidos que permitem aprovisionar certificados X.509 automaticamente, sendo eles o Simple Certificate Enrollment Protocol (SCEP) e o Automatic Certificate Management Environment (ACME). No entanto o SCEP apresenta falhas de segurança e o ACME é focado em apenas uma área de aplicação das ICPs.

Analisámos o problema e tentámos perceber se é possível implementar o serviço pretendido e tentar perceber se a solução encontrada pode ser considerada segura. Para isso, nesta tese fizemos uma pesquisa construtiva para analisar as soluções existentes, de forma a chegarmos à solução por nós proposta.

Nesta tese apresentamos e descrevemos uma arquitetura para suportar tal serviço. O nosso sistema depende da entrega de elementos de segurança, como tokens criptográficos ou smartcards com um certificado administrativo, às entidades finais, que é utilizado como âncora de segurança.

Para além da solução para o nosso objetivo inicial, também desenvolvemos um conjunto de infraestruturas para suportar uma ICP funcional, mais especificamente uma arquitetura para os serviços de back-office, um modelo de dados para suportar as iterações entre as entidades finais e os serviços de back-office e uma API baseada
em chamadas de procedimentos remotos para o *software* cliente das entidades finais usarem para o aprovisionamento automático de certificados X.509 assim como para outras operações de ICP comuns.
Glossary

PKI  Public Key Infrastructure

3DES  Triple Data Encryption Standard

AES  Advance Encryption Standard

CA  Certificate Authority

RA  Registration Authority

VA  Validation Authority

HSM  Hardware Security Module

CRL  Certificate Revocation List

DN  Distinguished Name

OCSP  Online Certificate Status Protocol

EJBCA  Enterprise Java Beans Certificate Authority

SCVP  Simple Certificate Validation Protocol

CP  Certificate Policy

CPS  Certificate Practices Statement

LDAP  Lightweight Directory Access Protocol

HTTP  Hypertext Transfer Protocol

HTTPS  Secure Hypertext Transfer Protocol

FTP  File Transfer Protocol
VPN  Virtual Private Network
IETF  Internet Engineering Task Force
CMP  Certificate Management Protocol
CMC  Certificate Management Messages Over CMS
PKCS  Public-Key Cryptography Standard
SCEP  Simple Certificate Enrollment Protocol
CRMF  Certificate Request Message Format
RFC  Requests for Comments
OID  Object Identifier
CSR  Certification Signing Request
WWW  World Wide Web
XML  eXtensible Markup Language
SOAP  Simple Object Access Protocol
WSDL  Web Service Definition Language
RPC  Remote Procedure Call
REST  REpresentation State Transfer
AD CS  Active Directory Certificate Services
ACME  Automatic Certificate Management Environment
CLI  Command Line Interface
DCS  Dogtag Certificate System
API  Application Programming Interface
JEE  Java Enterprise Edition
RDBMS  Relational DataBase Management System
JSON  JavaScript Object Notation
UP  University of Porto
# Contents

Abstract iv

Resumo vi

Glossary viii

List of Tables xiv

List of Figures xv

1 Introduction 1

1.1 Motivation .............................................. 3
1.2 Objectives .............................................. 4
1.3 Methodology .............................................. 4
1.4 Assumptions .............................................. 4
1.5 Outline .............................................. 5

2 State of the Art 6

2.1 Cryptography .............................................. 6
2.1.1 Symmetric Cryptography .......................... 7
2.1.2 Asymmetric Cryptography ......................... 7
2.1.3 One Way Hash Functions .......................... 8
2.5.2 EJBCA Web Service API ............................................. 35
2.5.3 Certificate Auto Enrollment Protocols .......................... 35
  2.5.3.1 Simple Certificate Enrollment Protocol (SCEP) ........... 35
  2.5.3.2 Automatic Certificate Management Environment) .... 37

3 Implementation ......................................................... 39
  3.1 Main Idea .......................................................... 39
  3.2 PKI Architecture .................................................. 41
    3.2.1 Backend Database ........................................... 44
    3.2.2 API ............................................................ 47
      3.2.2.1 Login ...................................................... 48
      3.2.2.2 Certificate Request ..................................... 49
      3.2.2.3 Certificate Renew Request ............................... 51
      3.2.2.4 Get Crl .................................................. 51
      3.2.2.5 OCSP Query .............................................. 51
      3.2.2.6 Submit Requests for Approval ........................... 52
      3.2.2.7 Get CAs Chain or Certificates ......................... 52
      3.2.2.8 Get users certificates .................................. 53
    3.2.3 Technologies Used ............................................ 53
    3.2.4 Client Software .............................................. 54

4 Practical Use Case .................................................. 57
  4.1 University of Porto Case Practice ............................... 57
    4.1.1 University of Porto .......................................... 57
      4.1.1.1 U.Porto Card ............................................ 58
  4.2 Setting up EJBCA .................................................. 59
4.3 Preparing the Backend Database ........................................ 62
   4.3.1 End-entities Client Software .................................... 63
      4.3.1.1 Proposed end-entity Client Software Workflow .......... 64

5 Conclusion and Future Work ........................................... 66
   5.1 Implementation limitations ....................................... 66
   5.2 Future Work ..................................................... 67
   5.3 Conclusion ..................................................... 68

A API ............................................................................. 70
   A.1 API Code Organization ............................................. 70
   A.2 API Documentation ................................................ 71

B Client Software Documentation ....................................... 88

References ..................................................................... 95
List of Tables

2.1 HTTP verbs in REST web services .......................... 33
List of Figures

2.1 Example of common PKI . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Example of hierarchical CAs infrastructure . . . . . . . . . . . . . . 11
2.3 General X.509 certificate format . . . . . . . . . . . . . . . . . . . . . 13
2.4 Certificate revocation list block diagram . . . . . . . . . . . . . . . . 15
2.5 DCS Certificates Profiles Web Page . . . . . . . . . . . . . . . . . . . 26
2.6 EJBCA Users Public Web Page . . . . . . . . . . . . . . . . . . . . . . 29
2.7 EJBCA Private Administration Web Page . . . . . . . . . . . . . . . . 30
2.8 Architecture of a typical SOAP-based web service . . . . . . . . . . . 32

3.1 PKI Proposed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.2 Backend Database EER Database Diagram . . . . . . . . . . . . . . . 45
3.3 Login Operation Workflow . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.4 Certificate Request Workflow . . . . . . . . . . . . . . . . . . . . . . . 49

4.1 U.Porto Card . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2 List of EJBCA configuration files . . . . . . . . . . . . . . . . . . . . . 59
4.3 CAs infrastructure created for UP . . . . . . . . . . . . . . . . . . . . . 60
4.4 End-entities proposed Software Workflow Flowchart . . . . . . . . . . 65
Chapter 1

Introduction

Public Key Infrastructures are commonly used and deployed to protect networks, data, critical business systems and users. Whether they are used to authenticate users and systems, to provide access control to systems, or to encrypt data, ensure privacy, to digitally sign documents or emails as a mean to attest their integrity and authenticity, public key operations are nowadays integral to modern operating systems, commercial security products and custom built systems. E-commerce, banking, mobile systems or cloud computing all rely on the use of digital certificate infrastructures to represent and manage the digital identity of users, connected devices, web services, and business applications among others [47].

Public key or asymmetric cryptography, is an essential component for the deployment of PKI systems. It was first discovered by Whitfield Diffie and Martin Hellman, two Stanford mathematicians in 1976 [19]. They were trying to solve two major problems in cryptography: key distribution and identification. Public key cryptography alone, is not able to solve these problems because it does not provide a secure and trusted way to identify the keys owners, neither a way to distribute the public keys among all possible interested end-entities. So, a new technology was necessary to fill this gap in public key cryptography, allowing to identify the keys owners and create a way to efficiently distribute them.

In 1993 Public Key Infrastructure as we know it, began with the establishment of the X.509 certificate standard in RFC 1422. This document standardized the concepts of certification authorities, certificate revocation lists, and certificate trust chains that provided the framework for the more advanced PKI based technologies in use today [19]. Although PKI is now more than 20 years old, only more recently we started
to see public key cryptography and digital signatures as a ubiquitous part of the IT landscape [47]. Two well known systems were developed to fill this gap. One is PKI systems and the other is the Pretty Good Privacy (PGP). Both PKI and PGP systems use public key cryptography, but the systems differ in how they vet and bind public keys to end-entities identities. PGP is based on a web of trust, or peer to peer trust, so, it allows end-entities to choose who they trust. In PKI systems, the trust mechanism is centralized, so, the end-entities rely on a centralized system (Certificate Authority (CA)). If an end-entity trust in a CA, then it trusts in all the end-entities that the CA certified [40].

PKI has evolved, with many standards being created and updated when necessary. The industry has made great progress in creating these PKI standards in the past years, and the PKI solutions vendors are providing increasingly robust products. New PKI technologies and protocols are constantly being developed and there are always new ways to use the technology [21]. However, many standards are immature and still need improvements. Although there are lots of really useful things we can do with PKI, it still has plenty of rough edges and problems that need to be solved [21].

One of the biggest problems in PKI architectures is the first time the Certificate Authority needs to authenticate the end-entity. An end-entity is a application, such as an e-mail client, a web server, a VPN-gateway or a person. They are the entities that will be certified by the PKI. In most cases, human interaction is necessary to ensure that an end-entity is securely and correctly authenticated. In several cases, this verification is done through a collection of ad-hoc mechanisms, which regularly require human interaction. One common end-entity authentication method in PKI is the following: first an end-entity needs to generate a certificate signing request, copy and paste it in the CA web page, prove ownership of the end-entity to be certified by some manual method such as to receive a CA challenge at a (hopefully) administrator-controlled email address corresponding to the requesting end-entity and then, respond to it on the CAs web page. After that, it is also necessary to download the issued certificate and install it in the desired end-entity. This process could be very slow, and almost impossible to deploy it at a large scale infrastructure.

There are two well known protocols that implement automatic end-entities certificate provisioning: Simple Certificate Enrollment Protocol (SCEP) and the Automatic Certificate Management Environment (ACME). However SCEP has known security flaws and ACME is taking its first steps, so lacks field experience, and security flaws may be discovered when the protocol is more used. Also, ACME is only focused in issuing certificates to Web Servers, in order to allow HTTPS connections. The
CHAPTER 1. INTRODUCTION

certificates issued with ACME can only be used for TLS authentication within web servers. In order to better contextualize and understand the relevant real issues behind this problem we have conducted a comprehensive research and analysis of these protocols in order to understand what we can use and what we should avoid. We will better describe these protocols in the State of the art Chapter.

1.1 Motivation

As a part of the second year in the master degree in Network and Information Systems Engineering following the Communication Networks minor at Faculty of Sciences of University of Porto, the students must do a thesis to complete the degree. With the presented options, and the author’s interest in computer networks and security, the author has chosen the challenge provide by the problem of large scale secure automatic provisioning of X.509 certificates via web services.

Our motivation and our objectives consist on deploying a PKI architecture and mechanisms, that allow for a fully automated authentication process and end-entities certificate enrollment. For that, we resort to web services communication mechanisms between the CA and the end-entities to allow end-entities to enroll certificates in a secure and automatic way. We consider that this is highly relevant because PKIs have been present in several companies as a way to secure communications, provide digital signature mechanisms and other areas of application. More and more companies are using it and there are more and more devices in the companies making the IT infrastructures bigger. A good example of these exponential growth is the Internet of Things (IoT). With IoT, a large number of "things" such as electronic devices, software or sensors have been spread to enable these objects to collect and exchange data. And this transmitted data, in some way need to be protected from possible attacks. These bigger and bigger infrastructures have more users, more devices, more applications and all of them needs to securely communicate. As the number of end-entities in PKIs are growing exponentially, we think that the necessary processes to deploy a PKI also needs to be optimized, making the processes more and more easier, faster, scalable, more configurable and automated. This is the problem that we will try in a reasonably new way, mitigate and present a solution that can help achieve the objectives mentioned in the following section.
1.2 Objectives

The main goal of this thesis is to define and implement a solution which allows distributed, flexible, secure and scalable architectures for automated security element provision with X.509 certificates, by the means of web services. We also will try to achieve the following objectives:

- an infrastructure to support a fully functional PKI
- a framework architecture for PKI backoffice services
- a data model for end-entity interactions with the backoffice PKI services

1.3 Methodology

We followed the following methodology as a guideline in the development of this thesis:

Research on the current state of the art on PKI solutions, PKI operations through web services and automatic certificate enrollment protocols.

Definition of an architecture that provides certificate life-cycle operations with authentication/authorization mechanism to ensure PKI reliability.

Implementation of the defined architecture that can be used to fulfill one of the main goals.

Deployment of the implemented architecture in real scenarios in order to use as proof of concept usage.

Analyse the developed research, developed work, architecture and protocols done in the scope of this thesis.

1.4 Assumptions

In the developed solution, we have some assumptions.

One assumption, is that in our developed architecture, the end-entities have deployed secure elements on the end-entity client mechanism. This secure elements should be
generic PKCS#11 devices such as cryptographic USB tokens or smartcards. These
secure elements contain a secure microprocessor, a trusted operating system, a cryp-
tographic engine for execution of cryptographic algorithms and secure storage for per-
sistence of sensitive data and keys. The secure elements should be initially provisioned
with a administrative certificate issued by the PKI administrators. Another technical
issue, is that the middleware that is responsible for the communication between the
end-entities software and the secure elements is not within the scope of this thesis.
So, in our architecture, we only present a solution for the communication between the
server side PKI and the end-entities software.

1.5 Outline

This thesis follows the following structure:

- Chapter 2. State of the Art In this chapter the basics of cryptography, Public
  Key Infrastructure are described. In this chapter, we also present an overview of
  PKI software systems and analyze them. We end the chapter with an overview of
  two well known protocols that provide certificate auto enrollment mechanisms.

- Chapter 3. Implementation In this chapter we explain the infrastructure, and
  the methodology used for creating the prototype.

- Chapter 4. Practical Use Case Here we propose a possible implementation
  of our work with a practical use case.

- Chapter 5. Conclusion and Future Work In this chapter we analyze if
  the thesis goals were reached, summarize the work done and indicating possible
  future developments.
Chapter 2

State of the Art

This chapter contains a good base regarding Cryptography, Public Key Cryptography, Public Key Infrastructure and Services Communication. Furthermore, we also review some software used to provide PKI services, PKI operations through web services and protocols that allow automatic certificate provisioning. This chapter is a result of a literature study and is the basis for the following chapters.

2.1 Cryptography

Cryptography is the science that study techniques for storing and transmitting data in a particular form so that only those for whom it is intended can read and process it [43]. Since the civilization’s early days, there has been a need for secure communication between two parties. Early on, simple techniques such as invisible inks, tiny pin punctures on selected characters, minute differences between handwritten characters, pencil marks on typewritten characters, grilles which cover most of the message except for a few characters, and so on were used [40]. However, these techniques are not considered cryptography techniques, because they are based on the omission of the message or in hiding secret messages in other messages. These kind of techniques is called steganography [40]. In cryptography, there is no attempt to hide that one message is being passed through a channel. Instead, if a third party intercepts the message, they will not understand the message nor the meaning of it. To achieve this, cryptography uses algorithms in order to cipher and decipher messages. Cryptographic algorithms are commonly classified as symmetric, asymmetric and one way hash functions [40].
2.1.1 Symmetric Cryptography

Symmetric cryptography, also called secret key cryptography is a kind of cryptography where the same key is used for encryption and decryption. So, if two parties want to securely communicate, they need to establish a secret key and share it between them. One of the first symmetric cryptographic techniques was the Caesar Cipher, in which each character in the message is replaced with the character a fixed number of positions above or below it in the alphabet.

Inside of symmetric cryptography, we can sub-divide the algorithms in two different main groups. One is substitution ciphers where each character in the plaintext is substituted for another character in the ciphertext. The other one is transposition ciphers, where the plaintext remains the same, but the order of characters is shuffled around [40].

Symmetric cryptography algorithms present as advantages security and speed relative to asymmetric cryptography operations. However, there are some disadvantages such as the necessity to securely share the key. If an attacker gets hold of a symmetric key, he can read everything encrypted with that compromising key both sides of the conversation. Another problem of symmetric cryptography is scalability. If there are 500 people who want to securely communicate with each other, everyone needs 499 different keys to establish separate and secure communication channels with the others.

Nowadays, the most common algorithms used in the real world are far more advanced and commit to encrypt blocks of data in several rounds with different block and key sizes. The most used symmetric cryptography algorithms are the Triple Data Encryption Standard (3DES), Advance Encryption Standard (AES), International Data Encryption Algorithm (IDEA), Blowfish, CAST-128 and RC5 [50].

2.1.2 Asymmetric Cryptography

Asymmetric cryptography, also known as public key cryptography, uses two different keys instead of one as in symmetric cryptography. In this cryptographic system we have two different keys. A private key, that should remain secret by the owner. The other key, called public key, must be known by the public. Both public and private key form a key pair which are mathematically linked, however, one key can not be inferred from the other key, at least in any reasonable amount of time. This key pair
paradigm will allow for the usual encryption and decryption operations, and add a signature and signature verification functionalities.

In public key cryptography we have advantages such as the needlessness of securely exchanging keys, thus eliminating the key distribution problem and can provide digital signatures that can be repudiated. Although asymmetric cryptography solves the key distribution problem, it is not enough to solve the key management problem. The problem with the key management is that there is no way to assure that a given key belongs to a given user. The user A can publish his public key and identify that key as his key. However, user B can also publish a public key and claim that that public key belongs to A. So a third party, which all users trusts, is necessary to ensure the ownership of a key and a proper identification the key owner. The key management problem can be solved using Public Key Infrastructure, that will be introduced in the following chapter.

Relatively to symmetric cryptography, asymmetric cryptographic operations such as encryption are far slower. There are several public key cryptography algorithms, such as Diffie–Hellman, ElGamal and RSA. Diffie–Hellman key exchange is mostly used for two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. ElGamal algorithm is based on the difficulty of the discrete logarithm problem and was invented by Tahir ElGamal in 1985. Among the above mentioned protocols, the far more popular and used is the RSA algorithm. The RSA so called because of the author’s names, Ron Rivest, Adi Shamir and Leonard Adleman is based on the mathematical problem of prime factorization [40].

2.1.3 One Way Hash Functions

One way hash functions, which are used in various functions such as cryptographic checksums, message digest or fingerprint, are functions that create a fixed sized compressed output from an arbitrary given input and are commonly used to validate a message integrity. Hash functions should be very easy to compute, but almost impossible to reverse the operation. So, given a text x and a hash function f(), it should be very easy and fast to compute f(x), should take a very long time to infer x from f(x). Also, if a single bit from the input is changed, on the average, half of the bits in the output should change [40]. Another property that hash functions must have is the nonexistence of collisions. By collisions, we mean, given an text x and a text y and an hash function f, if we calculate f(x) and f(y), f(x) and f(y) should not be equals
There are some hash algorithms such as MD5 and SHA-1 that were commonly used but now are no longer collision resistant. SHA-2 is a family of functions with different output size, which is currently considered secure and widely used.

2.1.3.1 Digital Signatures

Digital signature is an application example of One Way Hash Functions. In the scope of Public Key Infrastructure, digital signatures are used to ensure that a message was sent by a user, as also to check his authenticity and integrity. For that, it creates a one-way hash of the electronic data to be signed. Then, the private key is used to encrypt the hash and that signed hash is sent to the recipient of the message. When the recipient receives the message, he uses the sender public key to decrypt the encrypted hash, apply the same hash function to the message, and if the hashes are equal, then the recipient knows that the message was sent by the owner of the key, and that the message was not tampered.

2.2 Public Key Infrastructure

Public Key Infrastructure (PKI) is a set of technical mechanisms, procedures and policies that collectively provide a framework, using public key cryptography and digital certificates to ensure the four main security principles: integrity, confidentiality, non-repudiation and access control in communication through insecure channels such as the Internet [47]. There is a spread spectrum of applications and services where a PKI can be applied, as it does not serve a particular purpose rather provides a foundation for several security services and applications.

2.2.1 PKI Structure

Within any PKI, we find different components. However, there is not a rule which determines which are really necessary. The required components for the PKI are the Certificate Authority (CA), digital certificates and end-entities. There are some components that are optional and can be added to the PKI to split the workload between the components. The most common components which are commonly used are a Registration Authority (RA), a Validation Authority (VA) and a Certificate Repository. Bigger infrastructures are advised to have an RA and a VA, so the CA
could delegate the workload to those components avoiding possible problems with the CA workload. Next, we will approach one by one the role of each most common component in a PKI.

Figure 2.1: Example of common PKI

2.2.1.1 Certificate Authority (CA)

A CA is responsible for creating or certifying identities through the cryptographic signing of digital certificates. The CA ensures that an entity with a digital certificate signed by the CA is in fact who he claims to be. For that, the CA carefully checks if the end-entity who is trying to be certified is in fact who he claims to be. This means that the person or server owns the private key corresponding to the given certificate. If there is no trust in the issuing CA, then we cannot trust any of the certificates that they have issued.

There can be three kinds of CAs: a Root CA, a Subordinate CA and a Leaf CA. A Root CA has a self-signed certificate. A subordinate CA or a sub CA is a CA whose certificate is signed by another CA. That CA can be another Sub CA or a Root CA. The leaf CAs are responsible to issue certificates to the PKI end-entities.
There can be different design hierarchies to implement a CA structure. We can have a PKI with only one Root CA, which also is a leaf CA, called a Single/One Tier Hierarchy, or we can have a Root CA with several sub CAs with a two or three tier hierarchy, as we can see in Figure 2.2. When we have a multi-tier hierarchy, we can divide the workload between CAs, as well as end-entities, and if a CA key is compromised, it is easy to recover from it, because only a part of the PKI issued certificates will be compromised.

Figure 2.2: Example of hierarchical CAs infrastructure [21]

When we are building a PKI, we should plan which type of hierarchy is the most appropriate to the case. However, it is recommended to have a multi-tier hierarchy model when the size of the PKI is medium or large.

The multi-tier hierarchical model allows different CAs issuing certificates to different classes of end-entities, that can be a region or a country or company’s department. According to Ned Pyle, ”security increases with the addition of a Tier, and flexibility and scalability increase due to the increased design options. On the other hand, manageability increases as there is a larger number of CAs in the hierarchy to manage. And, of course, the cost goes up” [35]. There is not any recommendation for the limit number of tiers in a PKI structure, however, it is very unusual to have structures with more than three tiers.
2.2.1.2 Hardware Security Modules

Hardware Security Modules (HSMs) are hardware devices that safeguards and manages cryptographic keypairs and allow cryptographic operations to be fully carried out inside the module. HSMs are useful in PKI implementations because, beyond managing private keys safely inside the module, it can and execute cryptographic operations inside the module, then offloads the CAs from heavy cryptographic operations. As the operations are made inside the module, that ensures that the keys never leave the HSM and therefore cannot be exported and are then much more difficult to steal. For example, if an attacker gains access the CA machine, he cannot access the keys, minimizing possible damage [21].

2.2.1.3 Registration Authority

The RA is an optional component of PKI. When the RA is present in a PKI infrastructure, the CA delegates to the RA some responsibilities such as authenticating the end-entities and the information provided by them, and then register entities in the PKI, making the enrollment request to the CA. The CA trusts the RA to identify and authenticate entities according to the CAs policy. There can be one or more RAs connected to each CA in the PKI. Depending on the PKI infrastructure and use mode, the RA can also be responsible for generating the end-entities key pairs. When this happens, there must be a secure way to transfer the private key to the end-entity [47].

2.2.1.4 X.509 Certificates

A digital certificate forms a strong association between an identity and the public/private key pair held by the holder of the identity. The X.509 Certificates are standard digital certificates that are defined in several IETF Standards and it has been updated when necessary. The actual standard is defined in RFC 5280 [4] which defines Internet X.509 Public Key Infrastructure Certificates version 3 and Certificate Revocation List (CRL).

X.509 Certificates are composed by several fields, some being optionals and others required. The required ones are:

Version Specifies the version of the certificate. There are three different versions (v1, v2 and v3).
Figure 2.3: General X.509 certificate format [21]

**Serial Number** This number must be an unique integer for each certificate issued by one CA.

**Signature Algorithm** This field contains the signature algorithm used by the CA to sign the certificate.

**Issuer** The issuer field identifies the entity that has signed and issued the certificate. The issuer field must contain a non-empty distinguished name (DN). The DN can be composed by several fields, that are listed in RFC 1779 [17]. Each DN can have different fields, and it is up to PKI administrator to define which fields should be present to precisely identify the end-entity. There are fields such as CommonName (CN), LocalityName (L), OrganizationName (O), OrganizationalUnitName (OU), CountryName (C) and several others that can be consulted at [17].

**Validity** The certificate validity is the period of time during which the CA guarantees that it will maintain information about the status of the certificate.
Subject The subject field identifies the entity associated with the public key stored in the subject public key field.

Subject Public Key Info This field is used to carry the public key and identify the algorithm with which the key is used.

After the required fields, there are a lot of optional fields that can provide a large amount of additional information. We can have more information about the end-entity such as the Subject Alternative Name, as well as the Issuer in the field Issuer Alternative Name. We can have information about the certificate use constrains in the Key Usage, Extended Key Usage and Basic Constraints fields. We can also put on the certificate, information for the certificate validation through the Certificate Revocation Lists distribution points in the fields CRL Distribution Points and Freshest CRL. Information about the PKI policy terms can also be present in one URL in the Certificate Policies field. We can also define special fields, for fields which there is not a specific field for it. For those special fields an Object Identifier (OID) can be created and added to the certificate. An OID is used as an identification mechanism jointly developed by ISO/IEC and ITU-T for naming any type of object, concept or ”thing” with a global unambiguous name.

2.2.1.5 Validation Authority

A Validation Authority (VA) provides services used to check if a certificate is still valid. The certificate validation is mostly provided through a service for downloading Certificate Revocation Lists (CRLs) and through the Online Certificate Status Protocol (OCSP), however there are other protocols.

Certificate Revocation Lists The X.509 CRLs are defined in RFC 5280 [4] together with X.509 Certificates. CRLs are a list of certificate serial numbers issued by a CA that have been revoked. So, if a certificate’s serial number is present in this list, it means that the certificate should no longer be trusted. CRLs can be downloaded and stored in the end-entity storage, and can be used to validate a certificate when other validation mechanisms are offline. There are ten defined reasons for no longer trusting in a certificate such as key compromise, CA compromise, cessation of operation or privilege withdrawn. When a certificate is revoked a, the reason it was revoked is expressed and then that certificate’s serial number will be present in the next CRL. A diagram of the contents of
a CRL can be viewed in Figure 2.4. The CRLs are generated and published periodically, often at a defined interval. After that interval the CRL is not valid anymore and the CRL issuer should provide new one. When an end-entity wants to perform some operation, they should verify if the certificate is still valid. The CRLs validation format has a lot of problems. For example, if a certificate is revoked and there is still a long time until the next CRL update, it is possible to perform operations with that revoked certificate, because according to the last issued CRL, that certificate is still valid. Other problems with CRLs is the tendency to grow a lot. However, this last problem can be solved with the Delta CRLs, which contain updates to the previously distributed revocation information, rather than all the information that would appear in a complete CRL [4].

Figure 2.4: Certificate revocation list block diagram [21]
Online Certificate Status Protocol is a certificate validation alternative created to specifically address certain problems associated with using CRLs, such as the necessity of frequent downloads and it uses very few network resources because it contains less information than a CRL. With OCSP, the clients need to make a request (normally through http) to the OCSP server with a certificate serial number. The OCSP server must respond to the client if the certificate is still valid, if it was revoked, or if he do not know that certificate status. Currently OCSP is defined in RFC 6960 [36].

Other Validation Protocols Besides CRLs and OCSP, exist some more validation protocols such as the Simple Certificate Validation Protocol (SCVP) defined in the RFC 5055 [11]. SCVP allows a client to delegate certification path construction and certification path validation to a server.

Certification Chains/Path As referred above in the CAs section, CAs can be organized in a hierarchical model where, for example a ROOT CA sings a Leaf CA and this Leaf CA issues certificates to end-entities. When we are validating if a certificate is still valid through CRLs or OCSP, it is not enough to verify if the end-entity certificate is still valid. To really ensure the certificate validity we also need to verify if all the CAs in the chain used to sign the end-entity certificate are still valid. If one of the nodes in the hierarchical chain were revoked, we could no longer trust any of the certificates below that compromised node. So, in order to validate a certificate, we also need to provide a way to get and validate all the certificates in the chain from a certificate until the ROOT CA. Usually, software distributions such as operating systems or browsers, comes with a truststore. These truststores contains a list of common trusted ROOT CAs certificates. If an end-entity certificate’s chain ends in one of those certificates in the truststore, then the systems relies on the certificate.

2.2.1.6 Certificate Policy (CP) and Certificate Practices Statement (CPS)

A CP is a defined ”set of rules that indicates the applicability of a certificate to a particular community and/or class of application with common security requirements”. ”CPs typically fall into two major categories. First, some CPs indicate the applicability of a certificate to a particular community”. ”The second category of typical CPs indicate the applicability of a certificate to a class of application with common security requirements” [3]. CPS is a ”statement of the practices which a certification authority employs in issuing certificates”. ”CPS establishes practices
CHAPTER 2. STATE OF THE ART

concerning lifecycle services in addition to issuance, such as certificate management (including publication and archiving), revocation, and renewal or re-keying” [3].

2.2.1.7 Certificate Repository

The certificates issued by a CA must be accessible to all entities and available for retrieval. For that there must exist a certificate repository that allows certificate fetching through a network service. Over the past few years, the consensus in the information technology industry is that the best technology for certificate repositories is provided by directory systems such as LDAP (Lightweight Directory Access Protocol), although it can be provided by HTTP or FTP as well [9].

2.2.1.8 End-Entities

An end-entity is an application, such as an e-mail client, a web browser, a web server or a VPN-gateway. End-entities are not allowed to issue certificates to other entities, they make up the leaf nodes in the PKI certification tree, so, they are the entities that will be certified by the CAs.

2.2.2 PKI Protocols and Operations

2.2.2.1 Certificate Management Protocols

Certificate Management Protocols define a set of management protocols which defines which operations can be done and how the communication between the CA and the end-entities should be conducted regarding the certificates operations life cycle.

There are a few popular management protocols, some being IETF Standards and others are developed by private companies and are now public IETF drafts. The principal management protocols are:


- Certificate Management Messages Over CMS (CMC), another IETF Standard first defined in the RFC 2797 and obsoleted by RFC 5272 [39] and RFC 6402 [38].
CHAPTER 2. STATE OF THE ART

- Cryptographic Message Syntax Standard (PKCS#7), a non standard developed by the RSA Laboratories and defined in the RFC 2315 [15].

- Simple Certificate Enrollment Protocol (SCEP), another non standard protocol defined in an IETF draft [13]. The SCEP protocol is the one that is most interesting to us because SCEP allows automatic certificate enrollment, which is one of our main goals. SCEP defines a protocol for certificate management and certificate and for CRL queries. However and after more study about SCEP we find that SCEP implementation could be susceptible to a Privilege Escalation Attack. This happens because SCEP does not strongly authenticate certificate requests made by users or devices [46]. SCEP will be studied in more detail in a later section.

2.2.2.2 Common PKI Operations and Certificates Life Cycle

In this section we will make a high level abstract review of possible PKI operations, used for management of keys and certificates through their life cycle from creation to retirement. These operations are all supported by Certificate Management Protocols chapter described protocols. We will make a simple overview over the several operations and how they commonly work.

It all begins with an end-entity who wants to be certified by a CA. For that, an end-entity start by generating a pair of public/private keys. In this step we must choose which public key algorithm to use and the key length. We must choose a key length where the amount of time taken to break the key, given current cryptanalytic attacks and processing power, is far too long for practical purposes.

The next step is the creation of a certification signing request (CSR) from the generated key pairs. CSR contains the generated public key and the DN from the requester end-entity. This public key and DN will be present in the certificate issued by the CA. The CSR may be in Certification Request Syntax Specification (PKCS#10) format, a non standard created by RSA Laboratories’s Public-Key Cryptography Standards (PKCS) and defined in the RFC 5967 [45]. PKCS#10 is a binary format for encoding CSRs for use with X.509. The CST can also be in a Certificate Request Message Format (CRMF), which is an IETF Standard which defines a syntax used to convey a request for a certificate and defined in the RFC 4211 [37]. In some PKI implementations, the key generation is made by the RA. The end-entity makes a certification request to the RA, the RA authenticates the end-entity and then generates a key pair and make the
certificate signing request to the CA.

After the RA (we will assume that the entity responsible for this step is an RA) receives the CSR from the end-entity that wants to be certified, the RA must ensure that the end-entities requesting the certification are in fact who they claim to be. For that, the RA must proceed with some or several ways to authenticate the end-entity, which must be defined in the Certificate Policy from the PKI. The authentication techniques used will depend on the business risk model of the organization. If it is a simple organization such as a video renting club, a simple entity authentication process is enough such as a confirmation e-mail sent to the requester mail account. If the case is the issuance of a certificate to use in a passport, or a national ID card a more detailed verification should be required such as personal presence or strong authentication via tokens, smartcards or biometric authentication systems. This is one of the most important steps in all the PKI operations. If the authentication process is not secure and trustworthy enough, any subsequent reliance on the credentials and certificates issued to that identity could be suspect [49].

After the RA authenticates the requesting end-entity, the RA will request to the CA the issuing of a new certificate. The CA will issue a new certificate regarding the class of the requester’s end-entity and the end purpose of the certificate. If the certificate purpose is digital signing, the certificate will have different constraints that a certificate which end would be to provide Secure Socket Layer/Transport Layer Security on web servers. So, according to the end-entity and the certificate’s end the CA will create and sign a new certificate with the CA private key, sending the new certificate to the end-entity who requested it or publishing it into a public directory such as an LDAP based Certificate Repository.

From the moment an end-entity gets a certificate from a CA, it can perform all the certificate related cryptographic operations. However, we will not analyze the possible cryptographic operations, instead, we will analyze operations regarding the certificate life cycle, after the certificate is issued. After a certificate generation, it may go through the following phases:

**Certificate Expiration** As expressed above in the X.509 Certificates section, certificates have a validity time. When that validity expires, certificates should no longer be trusted. One of the reasons why certificates have expiration dates is the amount of elapsed time since the certificate issuance could be enough to break the keys. Another reason could be that the revocation status may no longer be published in the CRLs which make it impossible to know if the certificate was
revoked or simply to make certificate owners pay a renovation fee [21].

Certificate Renewal  The certificate renewal operation is an operation that normally occurs near to the certificate’s expiration date. This operation may be performed in two different ways. One of them is when the same keys, from the prior certificate, are used. The other option is replacing the keys for a new key pair [21]. The most usual and advised is to generate a new key pair. For the same reason why certificates expire, that is the possibility of compromised keys, a new key pair should be created for the new certificate. However, there are cases where it could be admissible to use the same keys. In situations where certificates are issued with a short expiration date. Situations such as temporary workers or visitors on a company. If, for example, a certificate generated with 2048 bit length key pairs with a two months expiration date, if it was necessary to extend the expiration date, the same keys could be used because it is not likely to crack a 2048 bit key pair in that amount of time. This operation is applicable both to end-entities and CAs because the CA keys also have an expiration date.

The certificate renewal operation should be used instead of certificate expiration/new certificate request. That is because there is no necessity of a new authentication process, as the proof of ownership of the private key corresponding to the about to expire certificate is enough to guarantee a strong end-entity authentication.

Certificate Revocation  A certificate may be revoked at any time prior to the expiration date. Unfortunately, mathematical attacks are not the only way to subvert a cryptographic security scheme. Keys may be discovered due to errors in poor software implementations, analysis of hardware key storage. For that, the CAs needs to save the status of all issued certificates and provide ways to revoke the issued affected certificates. When a certificate is revoked, the CA needs to change the status of the certificate to revoked and publish the certificate status in all certificate validation protocols available on the PKI. The certificate revocation is a sensible operation and access should be very restrictive. A wrong certificate revocation will force the issuance of a new certificate, having to repeat all steps for issuing a new certificate unnecessarily. There are ten possible revocation reasons defined the RFC 5280 [4], they being:

- 0 - unspecified
- 1 - keyCompromise
- 2 - CACompromise
There is a special type of revocation type that is certificate revocation reason number 6: certificate hold. In this particular case and this only, the revocation status can be reversed. This special case can be applied in situations where there is no certainty that the certificate has been compromised. As there is no certainty, the certificate is put on hold. When a certificate is on hold, it cannot be trusted. When it is confirmed that the keys were compromised, it is necessary to permanently revoke it with one of the other reasons or we can restore the certificate status to valid again if the suspicion of the certificate compromise does not verify [21]. When a certificate is revoked the users should know the reason why it was revoked and the date when it was revoked as well. Both CRLs and OCSP contain that data.

**Key pair archival and recovery** When the intent of a certificate and respective key pairs is to cypher data, keys archival and recovery operations should be possible. If a private key, for some reason, is lost, the recovery of those lost keys is necessary to recover all data encrypted with a key. If the key is permanently lost, we can consider that all the encrypted information is unreadable, and so, forever lost. For that, keys should be securely archived after being generated. However, when keys are generated in smartcards or cryptographic tokens, and they do not allow export private keys, the keys archival is not possible. In those cases, it is necessary to be very careful with the smartcards or tokens not to lose or damage them, or simply, do not use cryptographic tokens that do not allow export private keys to cypher data [21].

**Cross-Certification** Cross-Certification is a simple entity certification, but instead certifying end-entities, the CA certify another CA outside the current CAs infrastructure. This operation allows that end-entities from a CA can securely communicate with the end-entities from another different CA. But for that, there must be confidence between the CAs, and believe that the entity authentication
by the other CAs are reliable and can be trusted. There can be cross-certification between CAs in one direction or in both directions [21].

2.3 PKI Software Solutions

PKI software solutions are purpose-designed computer software used to implement some of the PKI operations and elements such as CAs and RAs. Now we will make a brief analysis of four different solutions, describing what they offer.

2.3.1 Microsoft Windows Certificate Authority

Since Windows Server 2000, Microsoft offers Certificate Services systems through Active Directory Certificate Services (AD CS). AD CS provides customizable services for issuing and managing public key certificates [41]. AD CS supports a lot of PKI features and components that can be optionally installed using Server Manager, they being:

**CAs** AD CS allows the creation of Root CAs and subordinate CAs, allowing the creation of a hierarchical set of CAs. With the created CAs is possible to issue certificates to users, computers, and services, and to manage certificate validity.

**CA Web enrollment** This feature allows human end-entities to connect to a CA web page to request certificates and retrieve CRLs.

**Online Responder** The Online Responder acts like an OCSP responder. It responds to revocation status requests and answers with a signed response containing the requested certificate status information. CRLs and delta CRLs can also be provided.

**Network Device Enrollment Service** With this feature it is possible to obtain certificates for routers and other network devices, that do not have domain accounts.

**Certificate Enrollment Web Service** The Certificate Enrollment Web Service enables end-entities certificate enrollment through web services using the HTTPS protocol. Together with the Certificate Enrollment Policy Web Service, this
enables policy-based certificate enrollment when an end-entity is not a member of a domain or when a domain member is not connected to the domain.

**Certificate Enrollment Policy Web Service** With this, end-entities can obtain certificates enrollment policy information [20].

This software was not deeply studied, because, beyond a Windows Server License purchase is necessary in order to deploy a PKI service with it, this software is more applied to certify computers in Windows domains networks, so, it does not fit as a solution to our objectives.

### 2.3.2 OpenCa

OpenCA is an Open Source PKI Project and results from a collaborative effort to develop a robust, full-featured and out-of-the-box Certification Authority that implements the most used protocols with full-strength cryptography world-wide. For that OpenCA uses several open source softwares such as OpenLDAP, OpenSSL, Apache Project, Apache mod_ssl. OpenCA can be deployed on Linux, Solaris and MacOS x operating systems. The project development is divided into two main tasks: studying and refining the security scheme that guarantees the best model to be used in a CA and developing software to easily setup and manage a Certification Authority [27]. However the project seems to be a little stopped since the last version of OpeCA with codename "SpecialK" was launched in August 2013.

OpenCa offers five different web interfaces that are used to manage the available components: CA, RA, LDAP and a public interface also called web-gateway. The web interfaces offered are the following [26]:

- **Node** The Node interface manages the database and handles all the export and import functionalities.
- **CA** This interface has all the functions which we need to create certificates and Certificate Revocation Lists (CRLs) and also includes all the functions which we can use to change the configuration via a web interface.
- **RA** In the OpenCA interface is possible to handle several requests such as editing requests, approving requests, creating private keys with smart cards, delete wrong requests and email users.
LDAP  The LDAP interface was implemented to separate the LDAP management completely from the rest of the software. This was done because there are many functions which are really specific for LDAP administrators, with only a few users needing these features.

Pub  In the Public interface we can do all the small things which the users need more specifically the following:

- Generate CSRs for Microsoft Internet Explorer, Mozilla 1.1+, Netscape Communicator and Navigator
- Generate client independent requests and private keys
- Receive PEM-formatted PKCS#10 requests from servers
- Enrolls certificates
- Enrolls CRLs
- Search certificates
- Check certificate status
- Tests user certificates in browsers

Besides OpenCA supports all the necessary components we think that are necessary to our solution, we find some disadvantages such as do not offer OCSP support, CRL updates are manually made and scalability is difficult to achieve with increase of complexity [12]. Due this we decided not to use OpenCA in our solution.

2.3.3 Dogtag Certificate System

Dogtag Certificate System (DCS) is an open source implementation of a software system designed to manage Public Key Infrastructure deployments. DCS has six highly-configurable subsystems, which provide flexibility in designing the PKI. The six subsystems that comprise DCS are [7]:

Certificate Authority  The CA is the subsystem that provides certificate management functionality for issuing, renewing, revoking, and publishing certificates. It is also responsible for creating and publishing CRLs. The DCS CA is a highly configurable set of software components and tools to manage all the certificate life-cycle using standards and services that facilitate the use of public-key cryptography and X.509 version 3 certificates.
**Data Recovery Manager** The Data Recovery Manager (DRM) is an optional subsystem that provides private encryption key storage and retrieval.

**OCSP Manager** The OCSP Manager is an optional subsystem that provides OCSP responder services. However, this manager does not access the CA each time it wants to know a certificate status, but stores the last issued CRL and answers the OCSP request based on the CRL taking from users the load of verifying certificate status in a CRL.

**Registration Authority** This RA is a subsystem that provides local enrollment request verifications. The RA can accept enrollment requests and authenticates them in a local context and then forwards the enrollment request to the designated CA to generate the certificate. The RA can be configured in two different ways. One in which the RA can set up with the appropriate authentication plugin to authenticate the request in an automated fashion and the one where the RA has a local request queue where requests can be stored and reviewed by local RA agents for manual authentication.

**Token Key Service** The Token Key Service (TKS) manages one or more master keys required to set up secure channels directly to the token management system. The privileged operations such as key generation can only be requested on the tokens through a secure channel.

**Token Processing System** The Token Processing System (TPS) provides the registration authority functionality in the token management infrastructure and establishes secure channels between the Enterprise Security Client (ESC) and the back-end subsystems.

Beyond the functionalities described above, DCS also provides a few more tools that help the PKI management.

**Certificate profiles** The DCS provides certificate profiles that are used to configure the content of certificates, the constraints for issuing the certificate, the enrollment method used, and the input and output forms for that enrollment. DCS comes with set of certificate profiles for the most common certificate ends as we can see in Figure 2.5. The administrator can modify and create certificate profiles, but he needs to send them to an agent services page for an agent approve. After the agent approve the certificate profile a dynamically-generated HTML
HSMs and Crypto Accelerators  The DCS supports HSMs and crypto accelerators provided by third-party vendors of PKCS#11-compliant tokens. The server can be configured to use different PKCS#11 modules to generate and store key pairs for all Certificate System subsystems. PKCS#11 hardware devices also provide key backup and recovery features for the information stored on the hardware token.

Enterprise Security Client  The ESC is a cross-platform client for end users to register and manage keys and certificates on smart cards or tokens. This is the final component in the Certificate System token management system, with the TPS and TKS.
CHAPTER 2. STATE OF THE ART

Command Line Interfaces DCS offers two different CLIs. One called pki-server CLI that can be used to manage PKI servers on the local machine. The command does not require authentication, but it should only be accessed by the system administrator. The other CLI is for accessing PKI services remotely given that it is possible to execute some commands without authentication, but another set of commands need authentication through username/password or with a certificate.

REST Web Services PKI provides REST interfaces to allow clients access services on the server. The REST services are implemented in REST style.

2.3.4 EJBCA

EJBCA is a multi-platform PKI Certificate Authority software build with JAVA, licensed under the Lesser General Public License, and is mainly developed by PrimeKey Solutions AB. EJBCA is an acronym for Enterprise Java Beans Certificate Authority. EJBCA is an Enterprise Java Beans (EJBs) application and needs an application server to be executed in. The structure of EJBCA is based on a modular design where packages can be modified and whole components can be moved or added. EJBCA supports a number of certificate management protocols used in a PKI, more specifically SCEP, a subset of CMP, PKCS#7 and PKCS#10. Regarding validation protocols, EJBCA offers certificate validation through CRLs, Delta CRLs generation and OCSP.

EJBCA introduced some concepts to help the PKI administration and management and to facilitate the certificate and user management, their being:

Certificate Profiles In an EJBCA certificateProfile we can define which non user specific content will be present in issued certificates. Here, we can define the available key lengths, the signature algorithm and the type of entities the certificate will suit, so, if the certificate is for a CA or for an end-entity. We can also define X.509v3 extensions which will be present in the issued certificates, such as key usage, extended key usage, basic constraints, key constrains among other extensions. Some extensions can be pre-set with a value, for example CRLDistributionPoint, where the same value is used in all certificates from the same profile. Here is also determined if these certificates will be published and in which publisher. We can link these certificate profiles to CAs, leaving only those CAs issue certificates with a given certificate profile.
**End-entity profiles**  
*End-entity Profiles* define which parts of DN should be registered for various types of end-entities. It defines which parts that are already pre-set, and which can be modified. It also contains other information, that is specific to each individual end-entity. An *End-Entity Profile* can be linked to specific *Certificate Profiles* and CAs so users belonging to a specific *EndEntity Profile* can only get certificates from the specified *Certificate Profile* and CAs.

**Publishers**  
A publisher stores issued certificates to a central location. EJBCA supports LDAP, more specifically OpenDJ and OpenLDAP and Active Directory but it is also possible to create customized plug-ins.

**Crypto Tokens**  
A Crypto Token in EJBCA is where keys are stored. A Crypto Token can be either backed by a software keystore, in a file on the database, or in a HSM PKCS#11 slot.

**Users Public Web Page**  
EJBCA provide a public web page where all users can access and make some operations. These operations are divided in 4 categories: Enroll, Retrieve, Inspect and Miscellaneous as we can see on Figure 2.6. In the Enroll section, users have four options for making certificate requests and get them. The Retrieve section allows users fetch CA Certificates, CAs CRLs and users last certificate. The Inspect section allow users inspect a certificate or PKCS#10 requests and the Miscellaneous section allows users search for user certificates by DN and check certificate’s revocation status.

**Administrators Private Web Page**  
The administration page is only accessible through an administrator certificate login and it is accessible through an HTTPS session. In there we have a GUI that gave administrators several information about all PKI elements such as CAs and allows administrators do all kinds of operations as well. The page is divided in 4 sections: CA Functions, RA Functions, Supervision Functions and System Functions like we can see in Figure 2.7. In CA Functions section the administrator can have an overview about the CAs and Publishers status, activate or deactivate CAs, create CAs, create or edit *Certificate Profiles* and manage Crypto Tokens. In RA Functions section the administrator can add or delete end-entities, as *End-Entities Profiles*.

**Command Line Interfaces (CLI)**  
EJBCA offers Command Line Interfaces which allow administrators directly access the machine where EJBCA is installed. There is a CLI in the proper installation of EJBCA and other, that can be put on any machine and run independently of EJBCA. EJBCA CLIs contains many
functions that can be used in scripts, or come to rescue when the administrator certificate has expired or if we accidentally revoke our administrator privileges for the Administration Web Page.

**Web Service Interface** EJBCA also have a Web Service interface which uses SOAP over HTTPS that allows remote administration and integration with any kind of software.

Besides the features above mentioned, EJBCA also provide a lot of features EJBCA allows the creation of CAs, sub CAs, and a whole type of CAs hierarchy as the creation of sub CAs signed by an external CA, which allows building several infrastructures within a single instance of EJBCA. EJBCA also allows export and import CAs, which make EJBCA quiet portable and scalable. EJBCA supports storing private keys either in software mode in a key store or in a HSM, providing support to the following

EJBCA also has a Logging Tool that supports a customizable log with internal log messages configurable in different languages and the signing of audit logs.

EJBCA supports a distributed infrastructure. We can separate or reproduce some services separated from the central EJBCA installation. We can have an external VA Installation or an external RA as well. With this we can guarantee a higher performance and availability in this crucial PKI service.

We can configure additional security measures with requests / approval mechanisms where he can define that some operations needs an administrator or more approvals in order to complete a request [50] [41] [31] [29] [33].

EJBCA is widely used worldwide, ranging from installations in small companies, for missions critical installations in multinational companies and governments such as France’s Ministry of Defense and Finances, the National Swedish Police Board or Zurich University of Applied Sciences. We can see more successfully deployed PKIs
using EJBCA in here [28].

We think that Dogtag Certificate System and EJBCA are a very complete systems with a lot of features, as well as the further presented APIs.

Between the Dogtag and EJBCA systems, we choose EJBCA to deploy our system because we think that it provides more features. EJBCA can be run on any system with JAVA. We also think that EJBCA is more widely used than Dogtag and have more documentation. Beyond these arguments, EJBCA is a well proved software, being used in many important institutions to support PKIs systems, as we can see here [28].

2.4 Services Communication

In our implementation, we need to enable communication between services. That communication can be made through Web Services, so we made a study about Web Services, what kind of web services exists and their main features.

Web Services has many imprecise and ambiguous definitions. This happens because the term derives from the different existing concepts of web services. The World Wide Web Consortium (W3C), the main international standards organization for the World Wide Web (WWW), describes web services as "a software system designed to support interoperable machine-to-machine interaction over a network" [48]. These interactions usually are made through the Word Wide Web (WWW) typically delivered over Hyper Text Transport Protocol (HTTP). The communication payloads of web services are usually XML documents, however, when performance demands web services can also use binary payloads. A web service is thus a distributed application whose components can be deployed and executed on distinct devices. Web services can be identified in two major classes:

2.4.1 Simple Object Access Protocol (SOAP) Based

SOAP is a communication protocol based on eXtensible Markup Language (XML) messages and those messages is defined in a Web Service Definition Language (WSDL). SOAP web services are independent of any platform, transport protocol, or operating system, but although they are not tied to any particular transport protocol, HTTP is the most popular. They can be run on any platform and written in any language
as long as they can formulate and understand SOAP messages. In SOAP-based web services the information about the server resources are hidden since all requests are made to the same URL [14].

Figure 2.8: Architecture of a typical SOAP-based web service [14]

SOAP web services generally encompass two major types of application interaction patterns [22]:

**Document Style** In the Document interaction style the Web service request takes the form of a complete An XML document that is intended to be processed whole. The XML document can be validated against pre-defined XML schema document. The document approach is easier because it simply relies on XML Schema to describe exactly what the SOAP message looks like while transmission.

**Remote Procedure Call (RPC) Style** RPC indicates that the SOAP message body contains an XML representation of a method or a procedure call with associated input and output parameters to generate XML structures that represent a method call stack.

### 2.4.2 REpresentation State Transfer (REST)

REST or RESTful web services are a kind of software architecture for networks connected through hyperlinks. REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher performing and more maintainable architecture [10]. The most well known applied domains for REST-style is the World Wide Web that is a distributed network of hypermedia. By order words, the RESTful web service is a subset of the Uniform Resource Identifier (URI) which provides a uniform interface semantics to manipulate web resources [14]. The main aspects of REST are:

- Resource Identification through URIs
• Hyperlinks to define relationships between resources and valid state transitions of the service interaction

• Uniform Interface for all resources (HTTP as the Application-level Protocol, observe Table 2.1)

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>Meaning in REST</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Query a resource in web server</td>
</tr>
<tr>
<td>POST</td>
<td>Create a resource in a web server</td>
</tr>
<tr>
<td>PUT</td>
<td>Update an existing resource in a web server</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource in a web server</td>
</tr>
</tbody>
</table>

Table 2.1: HTTP verbs meaning in REST web services

2.5 PKI Services Communication through APIs

In our study, we find some softwares that provide APIs to access some of the functionalities of the whole software. As we refer in the PKI Software Solutions chapter EJBCA and Dogtag offers APIs to access the software’s features. We will now make a brief description of each one.

2.5.1 Dogtag WS API

The DCS offers a REST interface to allow entities to access services on the server. The API was made using the REST web service style. In terms of design, the API interface uses regular HTTP verbs, they being GET for fetches data, POST to create new entries in the namespace and PUT to update fields in the namespace. We can see a resume of all the DCS REST API on this web page [5]. The REST service is implemented as a Java class that implements the resource interface. The REST service class will only exist on the server. A base class called PKIService is provided to simplify the implementation. The API provides a method that can be used to limit the authentication methods for each class of users that can be used to access the REST methods. The API also offers an access control tool where it is possible to define who can access the REST methods. As a client, it is possible to use a normal browser, or a special designed client using programming languages such as Java or Python [6]. DCS provides API calls for managing several objects, their being:
Controller Objects In here is possible to calculate token session key, encrypt data, create random blocks of data and calculate upgraded key set data for token symmetric key changeover.

Certificates Where we can get the list of certificates, get info about a certificate, get a certificate and get the status of a certificate through OCSP response.

Certificate Status In here we can modify and get a certificate status.

Certificate Requests In this one we can get a list of requests and their details and add or modify a request.

Certificate Request Status In this managing object, we can approve or deny requests, and get the status of a request made.

Certificate Profiles In here we can get list of certificate profiles, get certificate profile details, add or modify a profile and delete a profile.

Certificate Revocation List In this one we can get a list of CRLs and their details, get a CRL, add a CRL, modify a CRL and delete a CRL.

CA for OCSP In this managing object, we can get a list of CAs, get CAs details, add or modify a CA and delete a CA.

Keys In here we can get a list of keys, get a key, get a key details and add a key.

Key Requests In the key request management object we can get a list of key requests and their details, add or get a key to/from archival.

Key Request Status With this managing object we can modify a key request status, get key request status and delete a key request.

Users In here we can get the list of users, get users details, add or modify users and delete a user.

System In here we can get some subsystem information such as status, stats, logs and log contents.

Config Using this management object we can and get and modify a lot of configurations. For more information about the API, see [5]
2.5.2 EJBCA Web Service API

Since EJBCA 3.4, EJBCA introduced a JAX-WS 2.0 Web Service Interface, a SOAP Web Services described in a WSDL, used to access EJBCA basic functions. The functionality currently available through the Web Service Interface is documented in the EJBCA Web Service API Reference that can be visited in this javadoc [34]. The Web Services interface requires client certificate authentication from administrators, in the same way as the administrator GUI does. If we want that this functionality to be available in an EJBCA installation, it is necessary to configure a proper file for that during the EJBCA installation. However, for using this web service we need to get all the external libraries that EJBCA provides in order to be able to develop some functionalities using Java. In these libraries are some objects that are used as API calls answers as the possible exception objects that the API can return. There are also libraries such as Bouncy Castle for cryptography operations and Log4J for logging operations. There is a set of access rules for accessing the provided functionalities that can be viewed in here [31].

2.5.3 Certificate Auto Enrollment Protocols

We will now analyze the protocols which contain certificate auto enrollment mechanisms. These protocols allow end-entities request certificates to the CA without, or with a minimal human interaction. We will analyze Simple Certificate Enrollment Protocol and Automatic Certificate Management Environment protocol. With this study we want to understand what are the good parts of these protocols, and if possible, use the same mechanisms as they use. We also want to understand the bad parts of these protocols to avoid the same flaws that are present in these protocols.

2.5.3.1 Simple Certificate Enrollment Protocol (SCEP)

SCEP was developed by VeriSign Inc. for Cisco Systems with the purpose of allowing network administrators to easily enroll network devices for certificates in a scalable and automatic manner.

SCEP supports the following general operations:

- CA and RA public key distribution.
• Certificate enrollment.
• Certificate renewal/update.
• Certificate query.
• CRL query.

We will focus in the certificate enrollment and renewal operations, because they are the ones with the most interest for us. There are three different possible ways to authenticate the end-entity who is requiring a certificate:

1. The first case is when the end-entity already has a certificate issued by the SCEP server, so the server supports certificate renewal, and the request should be signed with that certificate.

2. The second case is when the end-entity has not a certificate issued by the SCEP server, but has credentials from an alternate CA, and when this happens and the PKI policy allows it, the certificate from the alternate CA may be used to sign the request.

3. The third case is when the end-entity does not have any appropriate certificate. The end-entity should create a self-signed certificate and use the same subject DN in the self-signed certificate and in the PKCS#10 request.

When the first and second mechanisms are used, the CA can use the already existent certificates in order to surely authenticate the client and determine the appropriate authorization and allow an automated enrollment using existing end-entity credentials.

When the third mechanism is used, SCEP allows two different authorization mechanisms for the initial enrollment, a manual way where the requester has to wait for an operator to approve a request and a pre-shared secret way where the SCEP server uses a challenge password that must be securely delivered to the end-entity requester and then include that challenge password with the certificate request made to the SCEP server [13].

The problem is that in the automatic way, with the challenge password, end-entities which have a challenge password provided by the SCEP server are able to make a certificate request with the identification data that the end-entity wants.

Because this mechanism does not contain a secure authentication mechanism, it may be possible for an end-entity to take a valid SCEP challenge password, and use it to
obtain a certificate that represents a different end-entity or for example an end-entity with a higher authorization access, or to obtain a different type of certificate than what was intended [42].

2.5.3.2 Automatic Certificate Management Environment)

The Automatic Certificate Management Environment (ACME) is a communication protocol to automate the interactions between a CA and the end-entities, more specifically, web servers. The protocol purpose is to automate the deployment of certificate enrollment processes at a very low cost and it is defined in an IETF draft [2].

ACME protocol was created to serve as a support to the Let’s Encrypt Certificate Authority, an Electronic Frontier Foundation initiative that aims to provide free X.509 certificates for Transport Layer Security encryption (TLS) via an automated process designed to eliminate the current complex processes. The main goal of this initiative is to clear the remaining roadblocks to transition the Web from HTTP to HTTPS, taking to a more secure Web browsing experience. The initiative contributors are Mozilla, Cisco, Akamai, IdenTrust, and researchers at the University of Michigan and will be operated by a new non-profit organization called the Internet Security Research Group [8].

Going back to the ACME protocol. The protocol describes an extensible framework for automating certificate issuance and a domain validation procedure, thereby allowing servers and infrastructural software to obtain certificates without user interaction. As the ACME purpose is to provide certificates to web sites, it performs domain name authentication by the CA to the web server. In domain name authentication or Domain Validation the CA verifies that the requester has effective control of the web server and/or DNS server for the domain, but does not explicitly attempt to verify their real-world identity.

ACME allows a client to request certificate management actions using a set of JSON messages carried over HTTPS. The first step of ACME protocol is the end-entity registration with the ACME server. At ACME, the end-entities are identified by cryptographic key pairs, and then it is possible to add domains to the key pair account. When an end-entity wants to add a new domain to their account, it needs to prove two things: that it holds the private key of the account key pair, and that it has authority over the identifier being claimed, so the domain name.

To prove that it holds the account key pairs, all messages from the end-entity to
CHAPTER 2. STATE OF THE ART

the ACME server are signed with the account private key and validated with the certificate. To prove that it owns the domain being claimed, the server issues the end-entity a set of challenges. These challenges can be different, depending on the case, the ACME server will choose from an extensible set of challenges that are appropriate for the identifier being claimed. The end-entity responds to the server with the challenge’s answers and the ACME server validate the answers. If the challenges are successfully the ACME server will authorize to issue certificates to the pretended domain. The end-entity needs to send a PKCS#10 to the ACME server signed with the private key of the account key pair. The issued certificate is assigned to a URI, which the end-entity can use to fetch the certificate or updated versions of the certificate [2].

In our opinion, this protocol serves for the purpose it was made, more specifically automate web server domains certification. However, in the scope of this thesis objective, this protocol is not enough embracing. By that we mean that we may want to certify different types of end-entities, and the scope of ACME is more directed to certify web servers and the domains they serve.
Chapter 3

Implementation

In this chapter, we describe in detail the main technological components that we have employed in our implementation. We will explain how we have used already existing technologies, and what we have created in order to successfully create a solution for our problem. We will present the main idea who leads to the deployment of our PKI system, and what we need to create to support all the possible PKI operations. We have created an Application Programming Interface (API) in order to enable communication between the end-entities and the backend created, a backend database to save important data to the system functionality such as the end-entities management, and the already existent software we have used to build our solution.

3.1 Main Idea

We will now explain the developed idea, that served as our guide during the implementation of our solution.

As have already been said before, one of the assumptions of this architecture, is that end-entities have in their possession, a cryptographic token or a smartcard to store certificates and protect private keys. The proposed idea is to give end-entities a cryptographic token or smartcard already provisioned with one administrative certificate and keypair.

This certificate, lets call it administrative certificate, will be issued by one special CA from the CAs architecture, lets call him administrative CA. This administrative CA is a CA that only issues administrative certificates. Administrative certificates should
not be used for other ends that not for authentication with our services.

We now have two possible cases. The first case is when we already know which end-entities we will use our services. In this case, we could issue the administrative certificate with the information from each end-entity in the subject DN of the certificate. In case we do not know who our end-entities will be, we will generate a general administrative certificate that does not identify an end-entity, but identify a class of end-entities. This last approach will give us an authentication problem further because we cannot ensure that an end-entity that request a certificate is who it claims to be. So, an additional authentication process should be used to authenticate an end-entity, such as an external certificate from another commonly trusted PKI.

Back to the CAs infrastructure. We would have different administrative CAs, each one of them would issue certificates to identify different classes of end-entities. For example, we would have an administrative CA for administrators end-entities, another for end-entities from one class of end-entity, and other administrative CA for another class of end-entities. A class of users is a set of end-entities with the same role, with access to the same type of certificate profiles and the same set of authorized operations.

So, after provisioning the secure elements with the administrative certificates, these tokens should be delivered to the respective end-entities. We assume that our tokens will be correctly delivered to the end-entities.

With this process, we are able to identify the end-entity and the class the it belongs to.

Now, the administrative certificate will be used by end-entities to authenticate with our services. With this, we ensure that only end-entities to whom we gave secure elements are able to access our protected services.

We provide three different ways to end-entities communicate with our services. One of them, is the API that we created, where all the operations can be made. The other two ways are used to validate certificates, one of them through an OCSP responder and the other is a CRL provider. To access these two validation services there is no need to authenticate end-entities, being open to all the public as is common to all PKI we know.

When an end-entity want to perform some operation, they should invoke the appropriate API call. They will be able to perform a set of operations, according to the administrative certificate we gave them.
That happens because, each class of end-entities (or a set of end-entities with the same administrative certificate) have a set of authorized operations that they can perform. The operations that each class of end-entities can perform are stored in the database created to support the backoffice operations. So, each time an end-entity make an API call, that needs authentication and authorization, we first check if it have permission to do it.

From now one, end-entities can use our API and make requests or, or if they want, program a set of operations in order to automatize a set of common operations.

All processes need to be set up to be automatized. In our case, the initial setup requires some work. Before the process automation it is necessary to create the CAs infrastructure, create the administrative certificates in the secure elements and delivers them to the end-entities. When the end-entities receive the tokens the provisioning process can be completely automated.

### 3.2 PKI Architecture

The main goal of our work is to create a solution to provide an automatic way to provisioning X.509 Certificates via web services, in a large scale and secure mode, however, it was also a goal the creation of a proposal for a full infrastructure to provide most common PKI operations. With this work, we also will provide PKI operations such as certificate validation, certificate revoking and certificate renewing, that we will now present.

We created a set of components to compose an infrastructure to support the PKI and all the operations. Those components were created using virtual machines, with the exception of the HSM, that is a piece of hardware that cannot be virtualized, and the end-entities secure elements for the same reason. We used the Kernel Virtual Machine (KVM) to create the virtual machines that composes our infrastructure, as we can see in Figure 3.1.

- EJBCA as CA
- EJBCA Database
- HSM
- API as RA, VA and Publisher.
CHAPTER 3. IMPLEMENTATION

• Backend Database

• Standard OCSP Responder for standard softwares.

• CRLs provider for standard softwares.

• End-entities with client software and security elements

We now describe the role of each component of the infrastructure.

**EJBCA and HSM** In this infrastructure component we have an installation of the EJBCA software and it will work as the Certificate Authority. This machine is in a private zone, that is not accessible from the Internet. This private zone is protected by a firewall. In the firewall, the EJBCA machine access is limited to connections coming from the API, OCSP responder and the CRL provider, that composes the frontend layer. Beyond the access to the CA being protected by the firewall, all the communications between the CA and the frontend layer are made through HTTPS connections. In the case of the API, the communication between the EJBCA and the API is made using the EJBCA Web Services API, which also encrypts his communication. Beyond this, the only way to access the EJBCA machine, should be in a ”physical” way. Together with the EJBCA machine should be as well an HSM to save the CAs key pairs and to perform cryptographic operations. The purpose of only allowing connections from the frontend layer machines, is to guarantee that the CAs keypairs and the proper CA are more secure.

**EJBCA Database** The EJBCA Database, for security and portability reasons, is in a separated from the EJBCA machine. It is protected by the firewall as well.

**API** The API is responsible for receiving communications from the end-entities. All the requests to the API are made through an HTTPS connection to ensure secure and confidential communication.

**Backend Database** The Backend database, that supports all API related operations is on a different machine than API for security and portability reasons.

**Standard OCSP Responder** This machine is a standard OCSP responder, as defined in the EJBCA documentation, more specifically in [30]. This was added to the infrastructure in order to allow standard services such as browsers validate certificates created by our infrastructure.
Figure 3.1: Public Key Infrastructure Proposed
**CRL Provider** The CRLs Provider's act as a bridge between the end-entities and the EJBCA CRL provider. EJBCA provides public pages for the download of CRLs, however, as the EJBCA machine is the protected zone, these pages are not accessible from the Internet, so we had to create this bridge to enable the CRLs download.

**end-entities** The end-entities have a client software to interact with the API. This client software allows the end-entities to invoke the API to perform the desired operations. The end-entities can use the client that we provide in order to automate interactions with our services. Only end-entities with our administrative certificates are able to communicate with our API.

### 3.2.1 Backend Database

The Backend Database was created to support end-entities management, end-entities, permissions, to store certificates and data to make requests to EJBCA. The Figure 3.2 is an EER Diagram of the created database. The Figure 3.2 was created using the MySQL Workbench software.

We will now explain the purpose of each table and how they are related.

**Table CAs** This table is used to store EJBCA CAs data. Data such as the CA *CommonName* and the whole CA DN. It also have a field to store the last CRL retrieved from EJBCA and the date it was retrieved. This information is necessary to make some operations such as certificate requests, CRLs requests or certificate revoke requests.

**Table CertificateProfile** This table’s purpose is to store data of existent certificateProfiles in EJBCA. As the CAs table, the information in this table it is necessary to make EJBCA requests.

**Table EndEntityProfile** In this table, we store information of end-entityProfiles existent in the EJBCA. We also use this table information to make EJBCA requests.

**Table Operations** This table contains all the API operations. The *opName* field contains the name of the operation, that is used to make requests to the API.

**Table LoginSession** This table is for storing information about end-entities session’s information and to store the certificate used by the end-entity to authenticate
CHAPTER 3. IMPLEMENTATION

Figure 3.2: Backend Database EER Database Diagram
with our services. As our login process uses strong authentication using public key certificates, with a challenge-signature method, it is necessary to store the certificate to verify the signed challenge. It also stores the session ID given to end-entities after they conclude the authentication process.

**Table Certificates** In this table will be inserted information about the issued certificates. It also stores the ID of the end-entity whose certificate belongs, the profile ID used to make the request and the dates between the certificate is valid. This could be useful to get information such as which certificates are about to expire or how many certificates an end-entity have.

**Profile** This is a very important table. This table aggregates several necessary information to make requests to EJBCA, they are being a combination between a CA, an EJBCA certificateProfile, an EJBCA end-entityProfile and an EJBCA end-entity. Each of these combinations forms one of our backend database profiles, and each one of these combinations is named in a field called *profileName* which is the profile name. When the end-entities want to make a certificate request, the end-entities must make a request containing a *profileName*. So the *profileName* information could be in the end-entities software or the end-entities can retrieve that information by making an API call to get all the profile names that the end-entity have access. We created this to facilitate how end-entities make a certificate request. In EJBCA, to make a certificate request, we need information such as the issuing CA, an EJBCA certificateProfile, an EJBCA end-entityProfile and an EJBCA end-entity. With this table, we simplify the process for end-entities, that, instead of needing to know the four fields mentioned, they pass to only need to know one field. Beyond aggregate data to make EJBCA certificate requests, these profiles are also related with an access control mechanism. End-entities have access profiles and profiles have permissions to execute operations. So, a profile is also a group of end-entities, and this group of end-entities should have the same role in the PKI. With this we can create a set of classes of end-entities that can perform a set of operations with the most varied set of EJBCA certificateProfiles, EJBCA end-entityProfiles and CAs. This allows a highly configurable and scalable set of rules to end-entities perform operations. However, before we fill this table, we need a lot of planning. It is needed to create all the CAs, all the EJBCA certificateProfiles, EJBCA end-entityProfiles and EJBCA end-entities first, then pass that information to this and the related tables. Finally, we have to assign permissions to the profiles, and define as well, which end-entities have access to which profiles.
CHAPTER 3. IMPLEMENTATION

Users  This table is used to save information about users. In the Figure 3.2, the table Users have a lot of fields from the DN, but these fields can be changed according to a specific case. One specificity of this table is that the username field will not be a common username chosen by the end-entity, but the serialNumber from the subject DN of the administrative certificate used in the authentication process. The serialNumber should be a unique identifier of the security elements delivered to the end-entities.

Groups  This table is used to save which profiles an end-entity with an administrative certificate issued by a determined administrative CA have access to. So, when an end-entity is created in our backend database, it is given a set of permissions that is stored in the Group_Profile table. So, we must have as many groups as administrative CAs.

RenewRequests, Pkcs10Requests and RevokeRequests Tables  These tables will be information about requests made by end-entities. End-entities can make operation requests, and then an administrator with permissions to that, can approve or deny those requests. When the administrator decides to approve or deny a request, this information is also saved in these tables as well who made that decision, when it was made and what the decision was. End-entities cannot do duplicated requests, but they can cancel any requests they made and make a new one, when a decision about the first request was not made yet.

3.2.2 API

PKIs and all the involved components and operations can be a little tricky for unexperienced users. With this work we intend to facilitate the PKI management joining all the public PKI components in one only place. In the developed web services we provide functions to perform PKI operations related to the RA, the VA and the Publisher. Our API allows end-entities to interact with our services. We created API calls to allow all necessary operations. The API follows an RPC style, where we have used URL query strings to pass the API calls parameters to the server and the answers are all made with a defined JSON structures. The only operation that was not implemented, and is very common in several PKIs, is the key recovery/archival. We did not implement a key recovery/archival operation because, as we only use cryptographic tokens to generate the end-entities keypairs, in which the private keys are not exportable, that function was unnecessary. We will now describe the operations we implemented in the
CHAPTER 3. IMPLEMENTATION

API, on the server side, and analyze how we implemented them. All the API calls connections are encrypted using HTTPS connections.

3.2.2.1 Login

The login operation follows the workflow presented in Figure 3.3. The Login operation is necessary to end-entities make all requests that need authentication. The operations that need authentication are identified in the Appendix API Documentation.

Figure 3.3: Login Operation Workflow

We will now explain each step of the operation designed in Figure 3.3:

1. The first step consists on the end-entity makes a login request to the API. The end-entity needs to send the administrative certificate that was given to him with the cryptographic token.

2. When the API receives a login request, it will check if the certificate received is still valid and if was issued by one of the infrastructure administrative CAs. If it was, then we extract the serialNumber field from subject DN. This serialNumber is used as username. Then we check if the user name already exists. If it does not exist, we create a new user in the Users Table, we give access to the Profiles in the Table "Users_Profile", according to the administrative CA of the given certificate. That permissions set is stored in the "Groups_Profile" Table. After the user is created, or if it already existed, then we move to the authentication process. The process consists in challenge-signature mechanism. So, when we receive the certificate, we store it in the database, in the Table "LoginSession", create a random challenge string and send it back to the end-entity.

3 and 4. The user receives the challenge from the API, signs the challenge with his private key corresponding to the sent certificate, and send the signed challenge to the API.
5 After receiving the signed challenge from the end-entity, we get the certificate and the original challenge from the database. We check if the signed challenge matches with the public key in the administrative certificate. If so, we generate a session token, which is a random string, and we store it in the database.

6 Finally the API sends the session token to the end-entity, and then it can use this token to do other operations that requires authentication, such as a certificate request.

After login, end-entities are able to try to perform all operations. We will now analyze how we implemented those operations. We said try to perform because, if the end-entity does not have the necessary permissions to do them, they will receive an error message with an unauthorized operation message.

3.2.2.2 Certificate Request

The Certificate Request operation follows the workflow presented in Figure 3.4. We will now explain each step:

Figure 3.4: Certificate Request Workflow

1 First, the end-entity needs to send to the API a CSR in PKCS#10 format, the session token given after the login, and the profile. The CSR must contain the subject DN from the end-entity, and that subject DN will be used as subject DN in the new issued certificate. This subject DN must have all the DN fields
required in the EJBCA end-entityProfile requested, but, if it does not fulfill those requirements, an error message is returned.

2 The API receives the certificate request alongside with the CSR, the profile and the session token. Before do any operation, the API checks several permissions:

- Checks if the session token is valid and if it was given to the requester end-entity.
- Verifies if the end-entity have access to the profile requested.
- Verifies if the profile requested have permissions to issue certificates.
- Checks if the end-entity already issued any certificate from the profile requested. If so, the request is denied. If the certificate is still valid, the end-entity does not need another certificate with the same profile. If it had one certificate and it was revoked, and a new certificate request is made, a manual verification should be used to mitigate the revoke reason and if the new certificate request is legit. If, for example, an employee no longer works in a company, and if he tries to issue a new certificate, that could be a security problem. In these cases, the end-entities could make a certificate request for an administrator approve. Of course we can find cases where is useful we have an end-entity issue certificates from the same profile, however we followed the one limited approach. In the future, a mechanism that enables multi certificates from the same profile could be implemented.

Only after those verifications are made, the API will make the certificate request to EJBCA. For that the API needs to fetch from the backend database, information related to the profile requested, in order to make the request to EJBCA.

3 EJBCA will analyze if the received certificate request does not have any problem, such as if the subject DN requested fit the EJBCA end-entityProfile, and if so, it will issue a new certificate, sign it and send the it to the API.

4 The API receives the certificate from EJBCA.

5 Before delivering the certificate to the end-entity, the API will store, in the backend database, information about the new issued certificate, such as the validity timestamps, the end-entity who issued it and the subject DN.

6 Finally the API delivers the new issued certificate to the end-entity.
CHAPTER 3. IMPLEMENTATION

3.2.2.3 Certificate Renew Request

The certificate renew request is very similar to the certificate request process. We could have two variants. One of them, wherein the same keys are used and the variant where new keys are used. However, we chose that we always will renew certificates with a new keypair. The renew request should be made close to the certificate validity end. We make some different verifications when a certificate renew request is made. The end-entity needs to send a new CSR where the subject DN must be the same as the old certificate, the profile and the serial number of the certificate that will be renewed. If EJBCA issue a new certificate successfully, the old certificate is revoked and the new certificate is returned to the end-entity.

3.2.2.4 Get Crl

For an end-entity get a CRL, it needs to make a request to the API indicating from each CA it wants to receive the CRL. This operation does not need authentication. So, when the API receives the CRL request, the API check in the backend database, in the CAs table, if the CRL present is still fresh. If the CRL from the database is fresh, then the API sends it to the end-entity. If not, the API requests EJBCA to issue a new one, then retrieve the new CRL and stores it in the backend database. We store the CRLs in the backend database to avoid access EJBCA to fetch a CRL that already was fetched.

3.2.2.5 OCSP Query

We also created an API call for getting the state of a certificate. The end-entity simply needs to send the certificate serial number and issuing CA of the certificate which it wants to know the state. The API receives the request and before asking EJBCA the certificate status, it checks in the backend database if there are some certificate issued with the serial number and CA provided. If not, the API immediately answers to the end-entity that there are no certificate with the specified serial number, so status unknown. We added this function because, in EJBCA, during the installation process, we can choose how EJBCA should handle the unknown certificate status, and the option "Non existing is good" is available. So, we decided to pass the responsibility of answers unknown certificate status to our API, safeguarding any of the options chosen in the EJBCA installation. When the certificate exists we ask EJBCA the state of
the certificate, and if the certificate is valid he just tells the user that the certificate is still valid, but if the certificate was revoked we send the revoked status accompanied by the reason why the certificate was revoked and when it was revoked.

### 3.2.2.6 Submit Requests for Approval

This is a set of operations that allows the creation of requests in order to fulfill some operations. The idea is to permit a non automatic way to do the certificate revoke, issue and renew operations. It is necessary an approval to complete the requests. So, when the end-entities do not have permissions to do the above operations, they could submit a request in order to fulfill them. After, an end-entity with permissions, which we can call an administrator, can accept or deny those requests. The end-entity that made the request can also cancel the request if a decision about the request was not made yet.

So, when a end-entity submit a request for approval, it needs to send all the information that is sent such as if it was a normal request. The data sent is stored in the backend database. The end-entity can see which requests it made and still are waiting for a decision, as well the result of requests that a decision was already made.

The administrator, can retrieve the requests from all users. He can accept or deny those requests. It is saved in the database what the decision was, when it was made and who made it.

So, if a request is denied, the information about the decision is stored in the database and the process is done. But if the decision is positive, we still need to proceed to the execution of the request. So we retrieve the information stored in the backend database, that was sent by the end-entity when it submitted the request. After we proceed to the fulfillment of the operation. However, in cases of issue or renew requests, where the end-entity should receive the certificate, the user end-entity is not waiting to receive the certificate. So, the end-entity, after the decision was made, can get the certificates using the API call `getUserCertificate`.

### 3.2.2.7 Get CAs Chain or Certificates

We provided API calls in order to end-entities get CAs certificates or the whole certification path from a CA. These operations do not need authentication. The end-entities send the request to the API and the API make the request to EJBCA
that answers back to the API and this to the end-entity.

3.2.2.8 Get users certificates

It is possible to get end-entities certificates. We can get end-entity certificates by his subject DN or by a certificate serial number.

3.2.3 Technologies Used

During the implementation of our API and the infrastructure that we create in order to fulfill our objectives, we used a lot of established technologies. We now describe which ones we used, and what we used them for.

EJBCA and EJBCA WebServices As explained in the State of Art chapter, EJBCA is a CA software, built using Java Enterprise Edition (JEE). As also explained above, EJBCA provide JAX-WS 2.0 Web Service Interface that can access the basic functions remotely over client authenticated HTTPS. We used the version 6.2.0.

JBoss The EJBCA software runs on an application server. We chose to use JBoss 7.1.1.GA version because it was the one that is used in the EJBCA installation tutorials.

Apache Tomcat Apache Tomcat is an open source software implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. We used it to provide our API.

MySQL and MySQL Connector MySQL is an open-source relational database management system (RDBMS). MySQL Connector offers a standard database driver connectivity for using MySQL with applications and tools. We used them both in the EJBCA Database and in the Backend Database.

JAVA JAVA is a general-purpose, concurrent, class-based, object-oriented computer programming language. We use JAVA, version 7 to build our API.

Maven Apache Maven is a software project management and comprehension tool. Maven can manage a project’s build, reporting and documentation from a central piece of information. We use Maven to manage the external libraries used and
to deploy the project and build the WAR file, that is deployed in the Tomcat server.

**IntelliJ IDEA** IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software. We use IntelliJ IDEA to program our API and a simple API client used for testing.

**Log4j** Log4j is a Java-based logging utility. We use it to create logs of all performed operations.

**JSON** JavaScript Object Notation (JSON) is a lightweight data-interchange format. All the answers from the API calls as well the exceptions returned via JSON messages.

**Bouncy Castle Crypto APIs** Bouncy Castle is a collection of APIs used in cryptography. It includes APIs for both the Java and the C# programming languages. We used the Bouncy Castle APIs in our API to do cryptographic operations.

**Kernel Virtual Machine (KVM)** KVM is a full virtualization solution for Linux. It consists of a loadable kernel module that provides the core virtualization infrastructure and a processor module. We used the KVM virtualization technologies to create our infrastructure.

### 3.2.4 Client Software

The API is not a friendly interface. So a client software is useful to interact with the API, simplifying those interactions. We have created a client software to interact with the API. This client software allows the end-entities invoke the API in order to perform the desired operations. With this client software, end-entities can easily automate interactions with our services. We used this client software to test if the API was answering as expected to the various requests. The client was created using JAVA and we used the following technologies to help to the client development:

- **Apache HttpComponents** to help us make HTTPS connections to the API.
- **Bouncy Castle Crypto APIs** to make cryptographic operations.
- **Maven** To manage the external libraries.
• **JSON** to decode the messages received from the API.

In the implemented client software, we created an Interface with several methods, one for each of the API possible operations. These methods take arguments, that are mostly system paths to files or strings. For example, for the login operation, we need to provide the administrative certificate path and the private key path in order to fulfill the operation. The API calls needs other arguments, but they are extracted from the provided files, or are treated inside the client. The certificates and cryptographic keypairs should be inside the cryptographic tokens, but, as we did not treat that part of our proposed infrastructure in this thesis, we used normal operating systems to save certificates and cryptographic keypairs. This client should be adapted in order to communicate with the secure elements. The available functions are described in the Appendix Client Software Documentation. We need to set the client truststore. We can chose the system truststore, or create a new one and set the new truststore in the client. With this approach, we do not need to add CAs to the system truststore, which can be an advantage. Anyway, we got to have the ROOT CA certificate in the selected truststore. It is very simple to use this client software in order to program a set of instructions to automatically get all the certificates an end-entity have access. The following code snippet is an example of how simple is to get all certificates an end-entity have access:

```java
APITalker apiTalker = ClientFunctions.getInstance();
apiTalker.login(administrativeCertificatePath, privateKeyPath);

JSONObject jsonAnswer =
    apiTalker.getAvailableProfiles(administrativeCertificatePath);
String type = jsonAnswer.getString("type");

JSONArray availableProfiles = (JSONArray) jsonAnswer.get(type);

for (int i = 0 ; i < availableProfiles.length(); i++){
    apiTalker.getNewCertificate(administrativeCertificatePath, csrPath,
    availableProfiles.getString(i), destinyPath );
}
```

Of course that this is a simplistic approach, where we do not take into account possible exceptions that may occur. In the following code snippet we check if a certificate with a given serial number is valid:

```java
```
APITalker apiTalker = ClientFunctions.getInstance();
JSONArray jsonAnswer =
    apiTalker.checkCertificateValidity(serialNumber, issuerCA);

JSONArray jsonFinalAnswer = null;

try{
    jsonFinalAnswer = (JSONArray) jsonAnswer.get("ocspAnswer");
} catch (JSONException e){
    System.out.println(jsonAnswer.getString("type"));
    System.out.println(jsonAnswer.getString("code"));
    System.out.println(jsonAnswer.getString("description"));
    System.exit(1);
}

It is possible to simplify the implemented methods. For example, we could adapt the \texttt{checkCertificateValidity} method, and turn it into an boolean method, hiding even more the complexity of dealing with the JSON structures.
Chapter 4

Practical Use Case

In this chapter we present a possible implementation of a PKI using our proposed architecture applied to the context of University of Porto (UP). We also present the relevant work necessary to set up the API.

4.1 University of Porto Case Practice

First, we will introduce the University of Porto and some infrastructures that UP have which we can integrate in our proposed PKI architecture.

4.1.1 University of Porto

The University of Porto is a Portuguese public university located in Porto, Portugal. It was founded in 22 March 1911 and currently is the second largest Portuguese university in terms of number of students. Currently UP houses 32000 students, 2400 teachers and researchers along with 1,600 administrative staff. UP is currently formed by 14 faculties and a business school offering a variety of courses, covering the whole range of study areas and all levels of higher education [24]. Despite its size and number of members, UP does not own a PKI. It only introduced digital signature operations in 2013/14, only in the processes of international mobility students using the Portuguese Citizen Card [23]. However the Portuguese Citizen Card does not authenticate the role that the signer have inside the UP organization.
4.1.1.1 U.Porto Card

The U. Porto Card, was introduced in 2008 and results from a partnership between the University and Banco Santander Totta. It is a smartcard integrating different technologies, contact and contactless, which currently gathers several functionalities:

Identification It identifies students and contributors as members of the U. Porto academic community.

Access Control Allows access to internal services within the various organic units like library loans or university’s buildings access control.

Third Party Services Access to other services provided by third parties, under special terms.

The smartcard was developed by Gemalto and belongs to Optelio Contactless D32 R5 smartcards family. Beyond the capabilities harnessed by UP, the smartcard also features a cryptographic coprocessor for public key encryption and Dynamic Data Authentication (DDA), contains a 32k EEPROM memory available for applications and data. The smartcard also has some applets such as PayPass M/Chip4 developed by Mastercard or VSDC2.7.1 developed by Visa among others, but the one that interests us is Classic IAS V3 (GemSAFE) which is a PKCS#11 PKI application that allows cryptographic operations such as digital signature, strong authentication and certificate storage. This applet follows the PKCS#15 Standard and all PKIs that uses PKCS#11 APIs can be used with this applet [44].

Concluding, the UP Card is capable of generating cryptographic key pairs and storing certificates. In our proposed architecture, the clients should have a cryptographic token or smartcard and the UP Card is ideal for that role. So the idea, in this practical
use case, is to use the UP Card as the end-entity cryptographic token. However, in our work, we did not develop the cryptographic provider for the UP smartcard.

### 4.2 Setting up EJBCA

We will now explain all the steps to deploy a complete installation of our architecture.

As said in the State of the Art Chapter, EJBCA is a software that runs on an application server. We choose JBoss 7.1.1. EJBCA also needs a relational database management system and we choose MySQL for that role. First, we need to configure JBoss to properly interact with MySQL. After that, we follow two different sets of instructions, that complements each other, to complete the installation of EJBCA. Those instructions are available here [32] and here [18]. In the EJBCA installation, we need to configure a set of files in the EJBCA conf/ directory, where we can make various configurations, such as the initial CA, the default algorithms to use, configure the access to the database and some of passwords to be used in the service, but we will not enter into details in this part. The configuration files available, that we can edit to configure EJBCA can be viewed in Figure 4.2

Figure 4.2: List of EJBCA configuration files

![List of EJBCA configuration files](image)

After the installation is concluded, we need to use the superadmin user to access EJBCA administration page. The superadmin user is an EJBCA user created during the EJBCA installation that have access to the EJBCA administration Web page. To access the EJBCA administration page, it is necessary to import the superadmin certificate to the browser where we will access the administration Web page. The
superadmin certificate and private key are generated during the installation process in a keystore protected with a password defined in one of the EJBCA configuration files. Once we have access to the EJBCA administration Web page, we need to create and configure the following components:

**CA Infrastructure** The first thing we need to create is the CA infrastructure. We decided to create a three tier hierarchal CA infrastructure in this particular practical use case. In the Figure 4.3 we can see the proposed infrastructure. We have a single ROOT CA, called UPROOTCA. The ROOT CA will sign two sub-CAs. One of them, will sign the administrative CAs, and the other one will sign the operational CAs. The operational CAs are the CAs responsible to issue the operational certificates. So we created eight issuing CAs. Four of them are administrative CAs for students, staff, teachers and administrators. The other four CAs, also for students, staff, teachers and administrators, but for the operational certificates.

![Figure 4.3: CAs infrastructure created for UP](image)

When we are creating a CA in EJBCA, we can define several configurations
such as the signing algorithm, the subject DN, validity, if is a ROOT CA or if is sub-CA. We can also define CRL specific data such as the default CRL issuer, the CRL expiration period or the default OCSP service location.

**CertificateProfiles** After creating all the CAs, we also need to create the EJBCA *certificateProfiles*. We already have some pre-defined *certificateProfiles* that we can use as templates for new *certificateProfiles*. The templates available are, for example, *Enduser*, *RootCA* or *SubCA*. We created our *certificateProfiles* based on the *Enduser certificateProfile*. We created eight different certificate profiles. Four of them for the administrative certificates and the other four are for the operational certificates. In here, we can also define the certificate Key Usage, Extended Key Usage and other X.509v3 certificate extensions. We decided to create *certificateProfiles* that enable us to perform digital signature operations. We created four different *certificateProfiles* for each of the end-entities classes, because we can predefine fields such as available CAs, what can prevent, for example, that a student issue a certificate from the teachers CA.

**End-EntityProfiles** Posteriorly, it is also necessary to create the *end-entityProfiles*. In this case, we do not have any templates to facilitate our jobs. But after the creation of the first *end-entityProfile*, we can use it as a template to create new profiles. In this step, we can define which DN fields will be present in the further issued certificates. In this case, we though the following fields would be recommended to undoubtedly identify a person inside and outside the UP infrastructure:

- **CN** That would be the name of the entity.
- **O** This field should always be UP.
- **OU** The Organizational Unit fields would identify the organic unit or faculty in UP.
- **L** The location of the entity.
- **ST** To identify the state of the end-entity.
- **C** The country of the end-entity
- **emailAddress** For the UP institutional email address.
- **UID** The unique identifier would be used for the unique personal number in UP
- **Title** This field would identify the role of the entity inside the UP infrastructure.
• **serialNumber** The smartcard RFID.

We defined eight different end-entity Profiles, where we predefined some fields in DN, such as UP as Organization for all profiles. We defined the Title field according to the end-entity role, so student, staff, teacher or administrator. We can also define the CAs and certificate Profiles, the end-entity Profiles can use, define the along with some other configurations.

**EJBCA End-Entities** EJBCA have end-entities and our Backend Database have end-entities. If we used these two end-entities management mechanisms, it would lead to a duplication of end-entities. However, both are necessary, because, for example, to make a certificate request to EJBCA, it is necessary to use an EJBCA end-entity make the request. The easiest solution would be at the same time as we create an end-entity in our Backend Database, we create an equal end-entity in EJBCA, but this would lead to a duplication of end-entities in the two platforms. A solution that was thought and successfully implemented was the creation of a single end-entity in EJBCA and associate this end-entity with only one CA in EJBCA. Thus, all the certificates issued by one CA would belong to the same EJBCA end-entity, and in the Backend Database we could see which certificates belong to each real end-entity, since we safeguard that information in the Backend Database. With this approach, we cannot use some EJBCA WS functions, such as the revoke user function where all the certificates from an end-entity are revoked. However, we can emulate the majority of functions we cannot perform using our backend database. In the PKI planning, it is necessary to decide what options is the best to the given case.

this operation still can be done using the Backend Database, where we would need to fetch all the certificates from an end-entity and then revoke them one by one. With this, we finish the necessary EJBCA configurations.

### 4.3 Preparing the Backend Database

After configure EJBCA, we also have to fill some tables in the Backend Database, as said in the Backend Database section. Actually, this step is manually made, but in the future, we can make a simple script that reads the relevant information from the EJBCA database and then fill the Backend Database. The tables that are need to fill and with what are:
- **CAs** With all the CAs that we created.

- **CertificateProfile** With all the EJBCA certificateProfiles created in EJBCA.

- **end-entity Profiles** With all the EJBCA end-entityProfiles created in EJBCA.

- **Profile** With the profiles that we think that would be necessary. As said before, this Profile Table is a junction between a CA, an EJBCA certificate profile, an EJBCA end-entityProfile and an EJBCA end-entity. We created eight different profiles according to the end-entity role, four for the administrative CAs and four for the operational CAs.

- **Operations** To be filled with all the operations that the API provides.

- **ProfileOperations** We need to fill this table to give Profiles access to operations. In our case we give all the privileges to administrators. Then the students, teachers and staff would have the same set of permissions. They would have access to issue the first certificate for each Profile they have access.

- **Groups** We need to fill this table with what we call groups. A group is associated with an administrative CA and defines a class of end-entities.

- **Group_Profiles** In this table, we set which Profiles, the end-entities from a group has access to. It will be based on this table that the Users_Profiles Table will be filled, when a new end-entity is created.

After filling all the described Tables, the Backend Database is ready, and the service is able to start. All the other tables will be filled as soon as the end-entities start to interact with the API.

### 4.3.1 End-entities Client Software

We can use the implemented client software to implement a new client software where we can define a workflow that can work in several manners. We could implement a client software where end-entities had to manually make request for certificates, or we can make a client software implementation where end-entities only need to insert the UP smartcard and press a button and the card would be provisioned with the certificates the end-entity has access to. The PKI administrator has to decide how he wants to implement the client. We provide an API and architecture that
are customizable, scalable, with customizable access control, allows certificate auto enrollment and a client that he can use to implement automatic operations.

4.3.1.1 Proposed end-entity Client Software Workflow

We propose an end-entity client software workflow to securely and automatically provision the UP smartcards with certificates. The proposed solution, can be seen in Figure 4.4.

The first and second step is the Login with our API and then get the end-entity available profiles. Then it is checked if the end-entity already have in their possession and the certificates matching the available profiles and if they are still valid. If the end-entity does not have certificates that matches the available profiles, or the owned certificates are invalid a certificate request is made. In this solution we allow automatic certificate provisioning, but, if for some reason, the certificate request be synchronously rejected by the API, we automatically submit a certificate request for an administrator approval. However, before submit such request, it first checks if there are equivalent active requests, and if so, it does not submit another request to avoid duplicate requests. Before make any certificate request, the client software also checks if the end-entity have valid certificates in the certificate repository, corresponding to the wanted profile, which may result from a certificate request previously submitted and was already approved, avoiding unnecessary requests. A similar mechanism exist to certificate renewals. When this process ends, the end-entity is able to perform operations with the owned certificates.
Figure 4.4: End-entities proposed Software Workflow Flowchart
Chapter 5

Conclusion and Future Work

In this thesis, we made a literature study on the matters of cryptography and PKI, more specifically in a way to enable a secure and automatic way to issue X.509 certificates. We have studied the basics of cryptography, PKI, PKI software solutions, services communication, PKI services through web services and certificate auto enrollment protocols. After the literature study, we took advantage of study made and we collected the good ideas from the solutions and protocols we presented in the State of the Art chapter. We also used already existing software like EJBCA and the EJBCA Web Services API to construct our architecture as explained in Implementation chapter.

We think that our certificate auto enrollment protocol, relatively to SCEP and ACME is more configurable, scalable and embracing, allowing to apply our solution to very different PKI application ends. As we have the API, we can build different client software approaches in order to achieve different PKI implementations. We can have certificates auto enrollment, asynchronously certificate enrollment, and provide a large specter of certificates to large scale groups of end-entities.

5.1 Implementation limitations

In the solution that we developed, we did not take into account any problems like server load balancing and high availability, however, these problems should be taken into account if we want this to be more than a prototype.

The security of our solution depends one the correct delivery of the initialized secure
elements. After that process is complete, the process is quite automatic, secure and simple. However the necessity to physically send the token, for example to a new client, turns the process a bit slow to some business models, due to geographic distances.

5.2 Future Work

The proposed architecture is a highly scalable, configurable and relatively simple to understand. There are some points where we think that can be improved in the future.

Extend the API We would like to extend the API to create more operational functions. The end-entities management is a point which needs some improvement. The Dogtag Certificate System has a good end-entities functionalities deployed and we would like to improve our API to enable such operations.

Secure elements re-configuration With this, we mean change the administrative certificate in the cryptographic token already in use. If we want to change the class of end-entities to another class of end entities to receive different certificates and permissions, we have two options. Or we manually set those permissions in the Backend Database and then the end-entity would have access to the new profiles, or we could replace the administrative certificate in the cryptographic token. The end-entity, in our backend database would be the same, since the cryptographic token was not replaced. We only need to add procedures to clear the old permissions in the backend database. With this, we are trying to avoid the need to physically change the cryptographic token, when we want to change the end-entity class of an already existing end-entity. This operation can be achieved with the current existing API operations. All we need is to define how this operation should be done.

Creation of a graphical user interface (GUI) Although is not strictly necessary, a GUI in a standalone application or in a Web page would help the PKI management. The GUI should be built to facilitate data visualization, wherein it would be more useful for an administrator than to a simple end-entity. It would help to better visualize, for example, the end-entity requests, and could select several requests to approve or deny at the same time.

Improve messages security In the future, if we conclude that the actual messages exchange between the API and the end-entities is not secure enough, then we will
then create an extra security mechanism. If an attacker, somehow could steal the session token that we provide, we could impersonate the stolen end-entity. So, if before we send a message to the API, we create an hash of the message, sign it with the administrative certificate correspondent private key and send this signed hash together with the original message, we avoid that an attacker could impersonate the end entity.

5.3 Conclusion

In this thesis, we addressed the problem of providing X.509 certificates in an automatic and secure way in large scale infrastructures using web services. We presented a solution, that solves the proposed problem and we included it in a PKI infrastructure in order to present a full PKI solution with a possibility of end-entities certificates auto enrollment. For that, we used several already existing softwares like EJBCA and built our solution on top of that existing and proved softwares. We consider that our solution is secure, scalable, highly customizable and simple to understand and deploy, after the initial configuration. The initial configuration that is necessary in order to deploy our solution is a bit more complex and should be done by an experienced user, both in PKI and the used software in our solution, to guarantee that the infrastructure is correctly implemented. After that, we think that a person without such PKI knowledge can understand how to work with our solution, so making PKI more desirable to entities without the knowledge about how to implement and manage PKIs. However, we must admit that our solution is not perfect. Security flaws may exist, and our proposed solution is not ideal to very large scales, or geographically distant and disperse end-entities due the need to deliver the secure elements. The need to initially provide cryptographic tokens to the end-entities are possible, but the work load necessary to deploy such service in a very large scale is very difficult. In small, medium and minor big infrastructures we think that a solution is quite suitable to deploy a quality PKI service. Although we think that the software is capable of supporting such a large scale service we can not ignore that the physical limitations turns the implementation not impossible but unappetizing. SCEP still a better solution to really large scale infrastructures, but we think that the protocol should be improved in order to correct the identified flaws. ACME is and the Let’s Encrypt Initiative is a very good solution, that we think that will revolutionize the way we surf the Internet turning it really more secure. However, as we have already said, it only covers a bit of possible PKI end application. We consider that we reached all the objectives proposed in the beginning
of this thesis, providing a secure and automatic way to issue certificates for entities, an infrastructure to support a fully functional PKI, a framework architecture for PKI backoffice services and a data model for end-entity interactions with the backoffice PKI services.
Appendix A

API

A.1 API Code Organization

In order to organize the code we divided the code in five different Java packages and several Classes. We divided the code in packages according with which infrastructure elements the code will interact. Inside each package, we also divided the code according to which PKI common operations the code will deal. The code is organized in the following order packages and classes:

frontend In the frontend package we have the Java servlet responsible to listen all the requests from users and administrators. It also will have several classes that will treat the users requests. These classes are organized according to the PKI operations:

MainServlet The MainServlet is a Java Servlet responsible for receiving all requests from end-entities. Then, according the requested operation, it forwards the request to one of the other classes from this package after make some verifications.

FeOperator In this class we defined methods to deal with the end-entities login process.

FeIssuer In this class there are defined methods to deal with the certificates issuance and renewing.

FeRevoker This class deals with certificate revoke operations.
FeValidator  FeValidator class contains methods to check the certificate’s status through OCSP and CRLs.

FePublisher  In this class, we deal with the Publisher common operations.

dbfrontend  In this package there are classes that interact with the backend database. The classes of other packages do not directly access the database. We have the following classes:

DbCommons  In this class there are the necessary methods to connect to the database and some common methods with queries used by several operations.

DbCertRequest  In this class there are methods that deal with new certificates issuance.

DbFeCertRenewal  In this class there are methods that deal with certificate renewals.

DbFeCertRevoke  In this class we deal with certificate revoke operations.

DbFeLogin  In this class we deal with end-entities session informations.

DbFeGetCRL  In this class there are methods to deal with the CRL operations

backend  In this package there are classes that communicate with the CA, so, the EJBCA software. All the communication pass through a SSL/TLS connection that uses strong authentication. The classes in this package use the EJBCA Web Services Interface provided by EJBCA.

utilities  In this packages we define some objects in order to save end-entities requests. We also have a class with a lot of methods that are commonly used.

exceptions  For code organization reasons, we created a package to put all the created exceptions.

A.2 API Documentation

The functionality currently available in our API is documented in this Appendix. The format and the parameters of the requests that can be made to the server, the format of the server answers and the error messages, are specified in this document.
login1
Description Login with challenge-signature verification step 1.
Type HTTPS POST
Roles Any User
Arguments
'operation': 'login1'
'certificate': certificatePEMString

Answer
JSON {
  'status': 'OK',
  'timestamp': timestamp,  // answer timestamp
  'OK': JSON {
    'type': 'challenge',
    'challenge': JSON {
      'challenge': StringChalenge, // String
      'loginID': loginID          // String
    }
  }
}

Error Answer
JSON {
  'status': 'Exception',
  'timestamp': timestamp,  // answer timestamp
  'Exception': JSON {
    'type': ErrorType,       // String
    'code': ErrorCode,       // String
    'description': errorDescription  // String
  }
}

login2
Description Login with challenge-signature verification step 2.
Type HTTPS POST
Roles Any User
Arguments
'operation': 'login2'
'signedChallenge': signedChallenge
'loginID': loginID

Answer
JSON {
  'status': 'OK',
  'timestamp': timestamp, // timestamp
  'OK': JSON {
    'type': 'sessionID',
    'sessionID': sessionToken // String
  }
}

Error Answer
JSON {
  'status': 'Exception',
  'timestamp': timestamp, // timestamp
  'Exception': JSON {
    'type': ErrorType,       // String
    'code': ErrorCode,       // String
    'description': errorDescription  // String
  }
}
### logout

**Description**
Exit Session

**Type**
HTTPS POST

**Roles**
Any User

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'logout'</td>
</tr>
<tr>
<td>'sessionID'</td>
<td>sessionID</td>
</tr>
<tr>
<td>'user'</td>
<td>AdministrativeCertificateSerialNumberInDN</td>
</tr>
</tbody>
</table>

**Answer**

```json
JSON {
    'status': 'OK',
    'timestamp': timestamp,
    'OK': JSON{
        'type': 'ack',
        'ack': 'succeeded'
    }
}
```

**Error Answer**

```json
JSON {
    'status': 'Exception',
    'timestamp': timestamp,
    'Exception': JSON{
        'type': ErrorType
    }
}
```

### ocspQuery

**Description**
Query the state of a certificate

**Type**
HTTPS GET

**Roles**
Any User

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'ocspQuery'</td>
</tr>
<tr>
<td>'serialNumber'</td>
<td>serialNumber</td>
</tr>
<tr>
<td>'issuerCA'</td>
<td>issuerCA</td>
</tr>
</tbody>
</table>

**Answer**

```json
JSON {
    'status': 'OK',
    'timestamp': timestamp,
    'OK': JSON{
        'status': {'good' | 'revoked'},
        'reason': reason,
        'date': revokeDate
    }
}
```

**Error Answer**

```json
JSON {
    'status': 'Exception',
    'timestamp': timestamp,
    'Exception': JSON{
        'type': ErrorType
    }
}
```
### getCrl

**Description**
Get the last CRL from the specified CA

**Type**
HTTPS GET

**Roles**
Any User

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'getCrl'</td>
</tr>
<tr>
<td>'ca'</td>
<td>CertificateAuthorityCNinDN</td>
</tr>
</tbody>
</table>

**Answer**

```json
JSON {
  'status': 'OK',
  'timestamp': timestamp, //timestamp
  'OK': JSON{
    'type': 'base64Crl',
    'base64Crl': base64Crl //base64String
  }
}
```

**Error Answer**

```json
JSON {
  'status': 'Exception',
  'timestamp': timestamp, //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```

### revokeCertificateRequest

**Description**
Revoke an certificate with the specified serial Number

**Type**
HTTPS POST

**Roles**
Authenticated User with this operation privelegies

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'revokeCertificateRequest'</td>
</tr>
<tr>
<td>'user'</td>
<td>AdministrativeCertificateSerialNumberInDN</td>
</tr>
<tr>
<td>'sessionID'</td>
<td>sessionToken</td>
</tr>
<tr>
<td>'profile'</td>
<td>profile</td>
</tr>
<tr>
<td>'serialNumber'</td>
<td>serialNumber</td>
</tr>
<tr>
<td>'issurCA'</td>
<td>issuerCACN</td>
</tr>
<tr>
<td>'revokeReason'</td>
<td>revokeReason</td>
</tr>
</tbody>
</table>

**Answer**

```json
JSON {
  'status': 'OK',
  'timestamp': timestamp, //timestamp
  'OK': JSON{
    'type': 'ack',
    'ack': 'suceeded'
  }
}
```

**Error Answer**

```json
JSON {
  'status': 'Exception',
  'timestamp': timestamp, //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```
### revokeCertificateRequestSubmission

**Description**
Save a request for approval to revoke a certificate with the specified serial number

**Type**
HTTPS POST

**Roles**
Any authenticated user

**Arguments**
- `operation`: 'revokeCertificateRequestSubmission'
- `user`: AdministrativeCertificateSerialNumberInDN // String
- `sessionID`: sessionToken // String
- `serialNumber`: serialNumber // String
- `issuerCA`: issuerCACN // String
- `revokeReason`: revokeReason // String

**Answer**

```json
{  
  'status': 'OK',  
  'timestamp': timestamp,  
  'OK': JSON{
    'type': 'ack',  
    'ack': 'succeeded'
  }
}
```

**Error Answer**

```json
{  
  'status': 'Exception',  
  'timestamp': timestamp,  
  'Exception': JSON{
    'type': ErrorType // String  
    'code': ErrorCode // String  
    'description': errorDescription // String
  }
}
```

### getRevokeCertificateRequests

**Description**
Gets all revoke certificate requests made by given user

**Type**
HTTPS GET

**Roles**
Any authenticated user

**Arguments**
- `operation`: 'getRevokeCertificateRequests'
- `user`: AdministrativeCertificateSerialNumberInDN // String
- `sessionID`: sessionToken // String

**Answer**

```json
{  
  'status': 'OK',  
  'timestamp': timestamp,  
  'OK': JSON{
    'type': 'revokeRequests',  
    'revokeRequests': [{
      'reason': reason,  
      'issuerCA': issuerCA,  
      'user': user,  
      'serialNumber': serialNumber,  
      'idRequest': idRequest}
    }, {...}]
}
```

**Error Answer**

```json
{  
  'status': 'Exception',  
  'timestamp': timestamp,  
  'Exception': JSON{
    'type': ErrorType // String  
    'code': ErrorCode // String  
    'description': errorDescription // String
  }
}
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Type</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAllRevokeCertificateRequests</td>
<td>Gets all revoke certificates requests made by all users</td>
<td>HTTPS GET</td>
<td>Authenticated User with this operation privlegies</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>'operation' : 'getAllRevokeCertificateRequests'</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'user' : AdministrativeCertificateSerialNumberInDN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'sessionID' : sessionToken</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'profile' : profile</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Answer</strong></td>
<td>JSON {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'status' : 'OK',</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'OK': {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'type' : 'revokeRequests',</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'revokeRequests' : [</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{'reason': reason,'issuerCA': issuerCA,'user': user,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'serialNumber': serialNumber,'idRequest': idRequest,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Error Answer</strong></td>
<td>JSON {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'status' : 'Exception',</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'Exception' : JSON{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'type' : ErrorType</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'code' : ErrorCode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'description' : errorDescription</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| approveRevokeCertificateRequest | Approve the revokeRequest, made by someuser, with the given ID if a decision wasn't made yet | HTTPS POST | Authenticated User with this operation privlegies |
| **Arguments** | 'operation' : 'approveRevokeCertificateRequest'                                |            |                                               |
|               | 'user' : AdministrativeCertificateSerialNumberInDN                           |            |                                               |
|               | 'sessionID' : sessionToken                                                    |            |                                               |
|               | 'idRequest' : idRequest                                                       |            |                                               |
|               | 'profile' : profile                                                          |            |                                               |
| **Answer**     | JSON {                                                                       |            |                                               |
|               |   'status' : 'OK',                                                           |            |                                               |
|               |   'timestamp' : timestamp,                                                  |            |                                               |
|               |   'OK': {                                                                  |            |                                               |
|               |     'type' : 'ack',                                                          |            |                                               |
|               |     'ack' : 'succeeded'                                                      |            |                                               |
|               | }                                                                           |            |                                               |
| **Error Answer**| JSON {                                                                       |            |                                               |
|               |   'status' : 'Exception',                                                   |            |                                               |
|               |   'timestamp' : timestamp,                                                  |            |                                               |
|               |   'Exception' : JSON{                                                       |            |                                               |
|               |     'type' : ErrorType                                                       |            |                                               |
|               |     'code' : ErrorCode                                                       |            |                                               |
|               |     'description' : errorDescription                                        |            |                                               |
|               | }                                                                           |            |                                               |
### denyRevokeCertificateRequest

**Description**
Deny the revokeRequest, made by some user, with the given ID if a decision wasn't made yet

**Type**
HTTPS POST

**Roles**
Authenticated User with this operation privelegies

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'cancelRevokeCertificateRequest'</td>
<td>String</td>
</tr>
<tr>
<td>'user'</td>
<td>AdministrativeCertificateSerialInDN</td>
<td>//String</td>
</tr>
<tr>
<td>'sessionID'</td>
<td>sessionToken</td>
<td>//String</td>
</tr>
<tr>
<td>'idRequest'</td>
<td>idRequest</td>
<td>//String</td>
</tr>
<tr>
<td>'profile'</td>
<td>profile</td>
<td>//String</td>
</tr>
</tbody>
</table>

**Answer**

```
JSON {
    'status' : 'OK',
    'timestamp' : timestamp,                 //timestamp
    'OK' : JSON{
        'type' : 'ack',
        'ack' : 'succeeded'
    }
}
```

**Error Answer**

```
JSON {
    'status' : 'Exception',
    'timestamp' : timestamp,                                   // timestamp
    'Exception' : JSON{
        'type' : ErrorType                                //String
        'code' : ErrorCode                               //String
        'description' : errorDescription      //String
    }
}
```

### cancelRevokeCertificateRequest

**Description**
Deny the revokeRequest, made by same user, with the given ID if a decision wasn't made yet

**Type**
HTTPS POST

**Roles**
Any authenticated user

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation'</td>
<td>'cancelOwnRevokeCertificateRequest'</td>
<td>String</td>
</tr>
<tr>
<td>'user'</td>
<td>AdministrativeCertificateSerialInDN</td>
<td>//String</td>
</tr>
<tr>
<td>'sessionID'</td>
<td>sessionToken</td>
<td>//String</td>
</tr>
<tr>
<td>'idRequest'</td>
<td>idRequest</td>
<td>//String</td>
</tr>
</tbody>
</table>

**Answer**

```
JSON {
    'status' : 'OK',
    'timestamp' : timestamp,                 //timestamp
    'OK' : JSON{
        'type' : 'ack',
        'ack' : 'succeeded'
    }
}
```

**Error Answer**

```
JSON {
    'status' : 'Exception',
    'timestamp' : timestamp,                                   // timestamp
    'Exception' : JSON{
        'type' : ErrorType                                //String
        'code' : ErrorCode                               //String
        'description' : errorDescription      //String
    }
}
```
### pkcs10Request

**Description**
Create a new certificate from the given csr for the profile given

**Type**
HTTPS POST

**Roles**
Authenticated User with this operation privelegies

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation' : 'pkcs10Request'</td>
</tr>
<tr>
<td>'user' : AdministrativeCertificateSerialNumberInDN //String</td>
</tr>
<tr>
<td>'sessionID' : sessionToken //String</td>
</tr>
<tr>
<td>'pkcs10' : pkcs10 //Base64 Csr String with Headers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>'profile' : profile //String</td>
</tr>
</tbody>
</table>

**Answer**

```json
JSON (  
  'status' : 'OK',  
  'timestamp' : timestamp, //timestamp  
  'OK' : JSON({  
    'type' : 'certificate',  
    'certificate' : certificate //Base64 Csr String without Headers  
  })  
)
```

**Error Answer**

```json
JSON (  
  'status' : 'Exception',  
  'timestamp' : timestamp, //timestamp  
  'Exception' : JSON({  
    'type' : ErrorType //String  
    'code' : ErrorCode //String  
    'description' : errorDescription //String  
  })  
)
```

### pkcs10RequestSubmission

**Description**
Save a request for an pkcs10 Request with the specified profile and user

**Type**
HTTPS POST

**Roles**
Any authenticated user

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>'operation' : 'pkcs10RequestSubmission'</td>
</tr>
<tr>
<td>'user' : AdministrativeCertificateSerialNumberInDN //String</td>
</tr>
<tr>
<td>'sessionID' : sessionToken //String</td>
</tr>
<tr>
<td>'pkcs10' : pkcs10 //Base64CsrString</td>
</tr>
<tr>
<td>'profile' : profile //String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Answer</th>
</tr>
</thead>
</table>
| JSON (  
  'status' : 'OK',  
  'timestamp' : timestamp, //timestamp  
  'OK' : JSON({  
    'type' : 'ack',  
    'ack' : 'suceeded'  
  })  
)
```

<table>
<thead>
<tr>
<th>Error Answer</th>
</tr>
</thead>
</table>
| JSON (  
  'status' : 'Exception',  
  'timestamp' : timestamp, //timestamp  
  'Exception' : JSON({  
    'type' : ErrorType //String  
    'code' : ErrorCode //String  
    'description' : errorDescription //String  
  })  
)
```
### getPkcs10RequestSubmissions

**Description**
Get all pkcs10 requests made by given user

**Type**
HTTPS GET

**Roles**
Any authenticated user

**Arguments**
- `operation`: `getPkcs10RequestSubmissions`
- `user`: AdministrativeCertificateSerialNumberInDN //String
- `sessionID`: sessionToken //String

**Answer**
JSON
```
{
  'status': 'OK',
  'timestamp': timestamp,  //timestamp
  'OK': {
    'type': 'Pkcs10Requests',
    'Pkcs10Requests': [Pkcs10Request{'profile':profile,'csr':csr,'user':user, 'idRequest': idRequest},Pkcs10Request{....}]
  }
}
```

**Error Answer**
JSON
```
{
  'status': 'Exception',
  'timestamp': timestamp,  //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```

### getAllPkcs10RequestSubmissions

**Description**
Get all pkcs10 requests made by all users

**Type**
HTTPS GET

**Roles**
Authenticated User with this operation privileges

**Arguments**
- `operation`: `getAllPkcs10RequestSubmissions`
- `user`: AdministrativeCertificateSerialNumberInDN //String
- `sessionID`: sessionToken //String
- `profile`: profile //String

**Answer**
JSON
```
{
  'status': 'OK',
  'timestamp': timestamp,  //timestamp
  'OK': {
    'type': 'Pkcs10Requests',
    'Pkcs10Requests': [Pkcs10Request{'profile':profile,'csr':csr,'user':user, 'idRequest': idRequest},Pkcs10Request{....}]
  }
}
```

**Error Answer**
JSON
```
{
  'status': 'Exception',
  'timestamp': timestamp,  //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```
<table>
<thead>
<tr>
<th><strong>aprovePkcs10RequestSubmission</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Approve the certificate issuing, made by some user, with the given ID if a decision wasn't made yet</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>HTTPS POST</td>
</tr>
<tr>
<td><strong>Roles</strong></td>
<td>Authenticated User with this operation privileges</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'operation' : 'aprovePkcs10RequestSubmission'</td>
</tr>
<tr>
<td></td>
<td>'user' : AdministrativeCertificateSerialNumberInDN</td>
</tr>
<tr>
<td></td>
<td>'sessionID' : sessionToken</td>
</tr>
<tr>
<td></td>
<td>'idRequest' : idRequest</td>
</tr>
<tr>
<td></td>
<td>'profile' : profile</td>
</tr>
<tr>
<td><strong>Answer</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'status' : 'OK',</td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
</tr>
<tr>
<td></td>
<td>'OK' : JSON (</td>
</tr>
<tr>
<td></td>
<td>'type' : 'ack',</td>
</tr>
<tr>
<td></td>
<td>'ack' : 'succeeded'</td>
</tr>
<tr>
<td></td>
<td>)</td>
</tr>
<tr>
<td><strong>Error Answer</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'status' : 'Exception',</td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
</tr>
<tr>
<td></td>
<td>'Exception' : JSON (</td>
</tr>
<tr>
<td></td>
<td>'type' : ErrorType</td>
</tr>
<tr>
<td></td>
<td>'code' : ErrorCode</td>
</tr>
<tr>
<td></td>
<td>'description' : errorDescription</td>
</tr>
<tr>
<td></td>
<td>)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>denyPkcs10RequestSubmission</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Deny the pkcs10 request, made by some user, with the given ID if a decision wasn't made yet</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>HTTPS POST</td>
</tr>
<tr>
<td><strong>Roles</strong></td>
<td>Any authenticated user</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'operation' : 'cancelPkcs10RequestSubmission'</td>
</tr>
<tr>
<td></td>
<td>'user' : AdministrativeCertificateSerialNumberInDN</td>
</tr>
<tr>
<td></td>
<td>'sessionID' : sessionToken</td>
</tr>
<tr>
<td></td>
<td>'idRequest' : idRequest</td>
</tr>
<tr>
<td><strong>Answer</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'status' : 'OK',</td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
</tr>
<tr>
<td></td>
<td>'OK' : JSON (</td>
</tr>
<tr>
<td></td>
<td>'type' : 'ack',</td>
</tr>
<tr>
<td></td>
<td>'ack' : 'succeeded'</td>
</tr>
<tr>
<td></td>
<td>)</td>
</tr>
<tr>
<td><strong>Error Answer</strong></td>
<td>JSON (</td>
</tr>
<tr>
<td></td>
<td>'status' : 'Exception',</td>
</tr>
<tr>
<td></td>
<td>'timestamp' : timestamp,</td>
</tr>
<tr>
<td></td>
<td>'Exception' : JSON (</td>
</tr>
<tr>
<td></td>
<td>'type' : ErrorType</td>
</tr>
<tr>
<td></td>
<td>'code' : ErrorCode</td>
</tr>
<tr>
<td></td>
<td>'description' : errorDescription</td>
</tr>
<tr>
<td></td>
<td>)</td>
</tr>
</tbody>
</table>
# cancelPkcs10RequestSubmission

**Description**
Deny the pkcs10 request, made by same user, with the given ID if a decision wasn't made yet

**Type**
HTTPS POST

**Roles**
Any authenticated user

**Arguments**
- `operation`:  'cancelOwnRevokeCertificateRequest'
- `user`: AdministrativeCertificateSerialNumberInDN
- `sessionID`: sessionToken
- `idRequest`: idRequest

**Answer**
JSON (```
'operation': 'cancelOwnRevokeCertificateRequest',
'user': AdministrativeCertificateSerialNumberInDN,
'sessionID': sessionToken,
'idRequest': idRequest
''
)
```

**Error Answer**
JSON (```
'operation': 'cancelOwnRevokeCertificateRequest',
'user': AdministrativeCertificateSerialNumberInDN,
'sessionID': sessionToken,
'idRequest': idRequest
''
)
```

# certificateRenewalRequest

**Description**
Create a new certificate from the given csr if the public keys match with the old certificate

**Type**
HTTPS POST

**Roles**
Authenticated User with this operation privelegies

**Arguments**
- `operation`: 'pkcs10Request'
- `user`: AdministrativeCertificateSerialNumberInDN
- `sessionID`: sessionToken
- `pkcs10`: pkcs10
- Headers: `profile`, `oldCertSN`

**Answer**
JSON (```
'operation': 'pkcs10Request',
'user': AdministrativeCertificateSerialNumberInDN,
'sessionID': sessionToken,
'pkcs10': pkcs10
''
)
```

**Error Answer**
JSON (```
'operation': 'pkcs10Request',
'user': AdministrativeCertificateSerialNumberInDN,
'sessionID': sessionToken,
'pkcs10': pkcs10
''
)
```
### certificateRenewalRequestSubmission

**Description**  
Save a request for an certificate renewal Request with the specified profile and user

**Type**  
HTTPS POST

**Roles**  
Any authenticated user

**Arguments**

- `operation`: certificateRenewalRequestSubmission
- `user`: AdministrativeCertificateSerialNumberInDN
- `sessionID`: sessionToken
- `pkcs10`: pkcs10
- `profile`: profile
- `oldCertSN`: oldCertificateSerialNumber

**Answer**

```json
{  
  'status': 'OK',  
  'timestamp': timestamp,  
  'OK': JSON{  
    'type': 'ack',  
    'ack': 'succeeded'
  }
}
```

**Error Answer**

```json
{  
  'status': 'Exception',  
  'timestamp': timestamp,  
  'Exception': JSON{  
    'type': ErrorType,  
    'code': ErrorCode,  
    'description': errorDescription
  }
}
```

### getcertificateRenewalRequestSubmissions

**Description**  
Gets all certificate renewal requests made by given user

**Type**  
HTTPS GET

**Roles**  
Any authenticated user

**Arguments**

- `operation`: getPkcs10RequestSubmissions
- `user`: AdministrativeCertificateSerialNumberInDN
- `sessionID`: sessionToken

**Answer**

```json
{  
  'status': 'OK',  
  'timestamp': timestamp,  
  'OK': {  
    'type': 'CertRenewRequests',  
    'CertRenewRequests': [CertRenewRequest{profile=profile, csr=csr, user=user, idRequest= idRequest, oldCertSn=oldCertSn}, Pkcs10Request{...}]
  }
}
```

**Error Answer**

```json
{  
  'status': 'Exception',  
  'timestamp': timestamp,  
  'Exception': JSON{  
    'type': ErrorType,  
    'code': ErrorCode,  
    'description': errorDescription
  }
}
```
### getAllCertificateRenewalRequestSubmissions

**Description**
Ggets all certificate renewal requests made by all users

**Type**
HTTPS GET

**Roles**
Authenticated User with this operation privelegies

**Arguments**
```
operation: 'getAllCertRenewRequests'
user: AdministrativeCertificateSerialNumberInDN //String
sessionID: sessionToken //String
profile: profile //String
```

**Answer**
```
JSON {
  'status': 'OK',
  'timestamp': timestamp, //timestamp
  'OK': {
    'type': 'CertRenewRequests',
    'CertRenewRequests': [CertRenewRequest{'profile'=profile,'csr'=csr,'user'=user,
      'idRequest'= idRequest, 'oldCertSn'=oldCertSn},Pkcs10Request{....}]
  }
}
```

**Error Answer**
```
JSON {
  'status': 'Exception',
  'timestamp': timestamp, //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```

### aproveCertificateRenewalRequest

**Description**
Aprove the renewal request , made by some user, with the given ID if a decision wasn't made yet

**Type**
HTTPS POST

**Roles**
Authenticated User with this operation privelegies

**Arguments**
```
operation: 'aproveCertificateRenewalRequest'
user: AdministrativeCertificateSerialNumberInDN //String
sessionID: sessionToken //String
idRequest: idRequest //String
profile: profile //String
```

**Answer**
```
JSON {
  'status': 'OK',
  'timestamp': timestamp, //timestamp
  'OK': JSON{
    'type': 'ack',
    'ack': 'suceeded'
  }
}
```

**Error Answer**
```
JSON {
  'status': 'Exception',
  'timestamp': timestamp, //timestamp
  'Exception': JSON{
    'type': ErrorType //String
    'code': ErrorCode //String
    'description': errorDescription //String
  }
}
```
<table>
<thead>
<tr>
<th>denyCertificateRenewalRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>Roles</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Answer</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Error Answer</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>cancelCertificateRenewalRequest</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>Roles</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Answer</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Error Answer</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### getCertificateFromSerialNumber

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieve if exist the certificate with the specified serial number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>HTTPS GET</td>
</tr>
<tr>
<td>Roles</td>
<td>Any user</td>
</tr>
<tr>
<td>Arguments</td>
<td>'operation': 'getCertificateFromSerialNumber'</td>
</tr>
<tr>
<td></td>
<td>'certificateSN': certificateSN</td>
</tr>
</tbody>
</table>

**Answer**

```
JSON {
  'status': 'OK',
  'timestamp': timestamp,
  'OK': JSON{
    'type': 'certificate',
    'certificate': certificate
  }
}
```

**Error Answer**

```
JSON {
  'status': 'Exception',
  'timestamp': timestamp,
  'Exception': JSON{
    'type': 'ErrorType',
    'code': 'ErrorCode',
    'description': 'errorDescription'
  }
}
```

### getCertificatesFromDN

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieve if exist the certificates with the specified DN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>HTTPS GET</td>
</tr>
<tr>
<td>Roles</td>
<td>Any user</td>
</tr>
<tr>
<td>Arguments</td>
<td>'operation': 'getCertificatesFromDN'</td>
</tr>
<tr>
<td></td>
<td>'DN': DN</td>
</tr>
</tbody>
</table>

**Answer**

```
JSON {
  'status': 'OK',
  'timestamp': timestamp,
  'OK': JSON{
    'type': 'certificates',
    'certificate': [certificate, certificate, ...]
  }
}
```

**Error Answer**

```
JSON {
  'status': 'Exception',
  'timestamp': timestamp,
  'Exception': JSON{
    'type': 'ErrorType',
    'code': 'ErrorCode',
    'description': 'errorDescription'
  }
}
```
### getCAChain

**Description** | Retrieve if specified CA exists his certificate chain
---|---
**Type** | HTTPS GET

**Roles** | Any user

**Arguments** | 
- `'operation': 'getCAChain'`
- `'CADN': CADN` //String

**Answer** | JSON
- `'status': 'OK'`
- `'timestamp': timestamp`
- `'OK': JSON{
  - `'type': 'certificates'`
  - `'certificate's': [certificate, certificate, ... ]` //List withBase64 Certificate String without Headers
}

**Error Answer** | JSON
- `'status': 'Exception'`
- `'timestamp': timestamp` // timestamp
- `'Exception': JSON{
  - `'type': 'ErrorType'` //String
  - `'code': ErrorCode` //String
  - `'description': errorDescription` //String
}

### getCACertificate

**Description** | Retrieve if specified CA exists his certificate chain
---|---
**Type** | HTTPS GET

**Roles** | Any user

**Arguments** | 
- `'operation': 'getCACertificate'`
- `'CADN': CADN` //String

**Answer** | JSON
- `'status': 'OK'`
- `'timestamp': timestamp`
- `'OK': JSON{
  - `'type': 'certificate'`
  - `'certificate': certificate` //Base64 Certificate String without Headers
}

**Error Answer** | JSON
- `'status': 'Exception'`
- `'timestamp': timestamp` // timestamp
- `'Exception': JSON{
  - `'type': 'ErrorType'` //String
  - `'code': ErrorCode` //String
  - `'description': errorDescription` //String
}
<table>
<thead>
<tr>
<th>getAvailableProfiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Roles</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
<tr>
<td>Answer</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Error Answer</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Client Software Documentation

The functionality currently available through our developed Client software are documented in this Appendix. The Parameters and their format are here described. The check the possible API answers it is necessary to see the Appendix API Documentation. The returned answers are unencapsulated of the first JSON object.

Table B.1: Login operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>privateKeyPath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated login2 API JSON answer</td>
</tr>
</tbody>
</table>

Table B.2: Logout operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>logout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated logout API JSON answer</td>
</tr>
</tbody>
</table>

Table B.3: Get available profiles operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getAvailableProfiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated getAvailableProfiles API JSON answer</td>
</tr>
</tbody>
</table>
Table B.4: Check certificate validity operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>checkCertificateValidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>CertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>ocspQuery</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.5: Check certificate validity operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>checkCertificateValidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>serialNumber issuerCA</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>ocspQuery</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.6: Get last CRL operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getLastCRL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>CA         destinyPath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>getCrl</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.7: Get new certificate operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getNewCertificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath csrPath profile destinyPath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>pkcs10Request</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.8: Submit a certificate request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>submitCertificateRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath csrPath profile destinyPath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>pkcs10RequestSubmission</em> API JSON answer</td>
</tr>
</tbody>
</table>
Table B.9: Get pending certificate requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getPendingCertificateRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>getPkcs10RequestSubmissions</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.10: Get all pending certificate requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getAllPendingCertificateRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>getAllPkcs10RequestSubmissions</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.11: Approve certificate request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>approveCertificateRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>idRequest</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>approvePkcs10RequestSubmission</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.12: Deny certificate request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>denyCertificateRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>idRequest</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>denyPkcs10RequestSubmission</em> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.13: Cancel certificate request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>cancelCertificateRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>idRequest</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <em>cancelPkcs10RequestSubmission</em> API JSON answer</td>
</tr>
</tbody>
</table>
Table B.14: Revoke a certificate operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>revokeCertificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>profile</td>
</tr>
<tr>
<td></td>
<td>serialNumber</td>
</tr>
<tr>
<td></td>
<td>CA</td>
</tr>
<tr>
<td></td>
<td>reason</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <code>revokeCertificateRequest</code> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.15: Submit a revoke certificate request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>submitCertificateRevokeRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>profile</td>
</tr>
<tr>
<td></td>
<td>serialNumber</td>
</tr>
<tr>
<td></td>
<td>CA</td>
</tr>
<tr>
<td></td>
<td>reason</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <code>revokeCertificateRequestSubmission</code> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.16: Get pending revoke certificate requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getPendingCertificateRevokeRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>csrPath</td>
</tr>
<tr>
<td></td>
<td>profile</td>
</tr>
<tr>
<td></td>
<td>destinyPath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <code>getRevokeCertificateRequests</code> API JSON answer</td>
</tr>
</tbody>
</table>

Table B.17: Get all pending certificate requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getAllPendingCertificateRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated <code>getAllRevokeCertificateRequests</code> API JSON answer</td>
</tr>
</tbody>
</table>
### Table B.18: Approve certificate revoke request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>approveCertificateRevokeRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>idRequest</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>Decapsulated approveRevokeCertificateRequest API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.19: Deny certificate revoke request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>denyCertificateRevokeRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>idRequest</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>Decapsulated denyRevokeCertificateRequest API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.20: Cancel certificate revoke request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>cancelCertificateRevokeRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>idRequest</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>Decapsulated cancelRevokeCertificateRequest API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.21: Renew certificate operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>renewCertificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td></td>
<td>csrPath</td>
</tr>
<tr>
<td></td>
<td>profile</td>
</tr>
<tr>
<td></td>
<td>destinyPath</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>Decapsulated certificateRenewalRequest API JSON answer</td>
</tr>
</tbody>
</table>
### Table B.22: Submit a certificate renew request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>submitCertificateRenewRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated certificateRenewalRequestSubmission API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.23: Get pending certificate renew requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getPendingCertificateRenewRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated getCertificateRenewalRequestSubmissions API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.24: Get all pending certificate renew requests operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getAllPendingCertificateRenewRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated getAllCertificateRenewalRequestSubmissions API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.25: Approve certificate renew request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>approveCertificateRenewRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated approveCertificateRenewalRequest API JSON answer</td>
</tr>
</tbody>
</table>

### Table B.26: Deny certificate renew request operation

<table>
<thead>
<tr>
<th>Method Name</th>
<th>denyCertificateRenewRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>administrativeCertificatePath</td>
</tr>
<tr>
<td>Returns</td>
<td>Decapsulated denyCertificateRenewalRequest API JSON answer</td>
</tr>
</tbody>
</table>
**Table B.27: Get certificate with serial number operation**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getCertificateWithSerialNumber</th>
</tr>
</thead>
</table>
| **Parameters**       | serialNumber  
                        | destinyPath  |
| **Returns**          | Decapsulated `getCertificateFromSerialNumber` API JSON answer |

**Table B.28: Get all certificate with DN operation**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getCertificatesWithSubjectDN</th>
</tr>
</thead>
</table>
| **Parameters**       | subjectDN  
                        | destinyPath  |
| **Returns**          | Decapsulated `getCertificatesFromDN` API JSON answer |

**Table B.29: Get certificate chain operation**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getCAChain</th>
</tr>
</thead>
</table>
| **Parameters**       | CASubjectDN  
                        | destinyPath  |
| **Returns**          | Decapsulated `getCAChain` API JSON answer |

**Table B.30: Get CA certificate operation**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>getCACertificate</th>
</tr>
</thead>
</table>
| **Parameters**       | CASubjectDN  
                        | destinyPath  |
| **Returns**          | Decapsulated `getCACertificate` API JSON answer |
References


REFERENCES


