Advanced System for Qualified Digital Signatures of Documents - SmartSigner

Ricardo Jorge Gonçalves de Almeida

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisor: Prof. Luís Jorge Brás Monteiro Guerra e Silva

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Luís Jorge Brás Monteiro Guerra e Silva
Members of the Committee: Prof. Carlos Nuno da Cruz Ribeiro

June 2018
Acknowledgments

Em primeiro lugar, agradeço o incomensurável e indispensável apoio dos meus pais, Fátima Conceição Pinto Gonçalves de Almeida e Jorge Augusto Silva de Almeida, por me terem educado como o fizeram e sem os quais não teria sido, de todo, possível alcançar esta, ou qualquer outra, meta.

À minha namorada, parceira, amiga, companheira, e tudo o mais, я люблю тебя очень сильно, Юлия Александровна Яньшина. Obrigado por me conseguires alegrar, ou rir por contágio, com essa só tua naturalidade, quando a minha falta de disposição o dita impossível. С тобой все невозможное возможно.

A toda a minha família por serem o meu suporte mais basilar.

Ao Miguel Ventura, sem o qual decisão de mudar de editor de LaTeX teria sido tomada tarde demais e não teria descoberto alguns tesouros do underground.

A toda a equipa da DSI, em particular ao Sérgio Silva, ao David Martinho e ao Luís Cruz, por me terem recebido, por me terem integrado e pela confiança depositada.

Aos meus, que me fazem decidir sair quando menos sensato é mas que, em contrapartida, mantêm a minha sanidade mental.

Por fim, ao meu orientador, Prof. Luís Guerra e Silva, por me ajudar a manter algum rigor na escrita mesmo quando o prazo se avizinha e a realização desta tese se adivinhava impossível.

Obrigado.
Abstract

Paperless workflows are more efficient, economical, environmentally-friendly and traceable. Although in Instituto Superior Técnico academic and administrative management software eliminates the need for paper in many processes, there are still many others that require paper support, most often documents that demand handwritten signatures, such as the registration certificates. In an effort to dematerialize these, the SmartSigner system was developed in-house, allowing for creation and management of Qualified Electronic Signatures in official IST documents, using smart cards, and holding the same legal value as the previously handwritten signatures. In this work, the deployed system is detailed and reviewed for maintenance, and several improvements and corrections to both the SmartSigner Server and its Client Application were delineated and implemented, such as corrections to discovered issues, assimilation with the Citizen Card, development of an application called Back-office to manage the system, improvements on the system deployment method, support for multiple signatures on the same document and recording of all system operations. To prove the adaptability of the improved SmartSigner system, it was integrated into a workflow from DOT, the system that handles IST acquisition requests and other financial matters and that produces purchase order documents necessitating signature. During the developments, some bugs were found, for example, in the Citizen Card, and its description and a solution is presented. The system is also evaluated using performance and usability tests and some conclusions are presented showing, for example, the successful correction of a previously existing major issue acutely affecting the system performance.

Keywords

Paperless workflows, Qualified Electronic Signatures, SmartSigner, Software Development, Integration
Resumo

Processos desmaterializados são mais econômicos, eficientes, rastreáveis e ecológicos. Apesar de, no Instituto Superior Técnico, o software de gestão acadêmico e administrativo eliminar a necessidade do suporte físico em muitos processos, há ainda muitos que o requerem, principalmente documentos que são assinados à mão, como os certificados de matrícula e certidão de curso. Numa tentativa de desmaterialização, o sistema SmartSigner foi desenvolvido internamente, permitindo a criação e gestão de Assinaturas Eletrônicas Qualificadas, utilizando smart cards, contendo o mesmo valor legal que as assinaturas anteriormente manuais. Neste trabalho, o sistema SmartSigner é detalhado e revisto, e vários melhoramentos são implementados para o servidor do SmartSigner e a sua Aplicação Cliente, tais como correções de problemas identificados, integração do Cartão de Cidadão, desenvolvimento de uma aplicação Back-Office para gerir o sistema, melhoramentos no método de atualização para uma nova versão do sistema, suporte para múltiplas assinaturas no mesmo documento e registo de todas as operações no sistema. Para provar a adaptabilidade do melhorado sistema SmartSigner é feita a integração com o sistema DOT, responsável por assuntos financeiros do IST e que gera notas de encomenda que requerem assinatura. Durante os desenvolvimentos, alguns bugs foram encontrados, por exemplo, na biblioteca do Cartão de Cidadão, e a sua descrição e solução são apresentados. O SmartSigner também é avaliado recorrendo a testes de performance e de usabilidade e algumas conclusões são extraídas mostrando, por exemplo, a bem sucedida correção dum grave problema preexistente afetando significativamente o desempenho do sistema.

Palavras Chave

Desmaterialização, Assinaturas Eletrônicas Qualificadas, SmartSigner, Desenvolvimento de Software, Integração.
# Contents

1 Introduction .................................................. 1
   1.1 Goals .................................................. 4
   1.2 Document Outline ...................................... 5

2 Related Work .................................................. 7
   2.1 Digital Signatures ...................................... 9
   2.2 Key Generation ........................................ 12
   2.3 Digital Certificates ................................... 13
   2.4 Smart Card ............................................. 15
   2.5 PDF Signatures ......................................... 16
   2.6 Legislation for digital signatures ....................... 19
   2.7 Technologies ........................................... 20
   2.8 Other Solutions ......................................... 21
      2.8.1 University of Porto, Portugal ....................... 21
      2.8.2 University of Amsterdam, Netherlands .......... 21
      2.8.3 Riga Technhical University, Latvia ............ 21

3 Existing System ................................................. 23
   3.1 Original System Requirements ......................... 25
   3.2 Smart Card Emission ................................... 26
   3.3 The student registration process ....................... 28
   3.4 SmartSigner Integration Goals ......................... 29
   3.5 SmartSigner ............................................ 29
      3.5.1 The Client Application .......................... 31
      3.5.2 The Server (repository) ......................... 31
   3.6 Certifier .............................................. 32
   3.7 The new workflow of the registration process ....... 34
4 New Developments: Corrections and Improvements

4.1 Bugs and errors identification .......................................................... 39

4.2 Improvements to the SmartSigner Client Application ............................. 40
  4.2.1 Refactoring .............................................................................. 40
  4.2.2 PIN-less certificate extraction ..................................................... 41
  4.2.3 Detect card removal ................................................................. 42
  4.2.4 Sign requests sorting per column ............................................... 43
  4.2.5 Sign request refusal ................................................................. 44
  4.2.6 Customizable options on Client ................................................ 44
  4.2.7 Customizable Signature Appearance ......................................... 45
  4.2.8 Auto registering cards ............................................................ 46
  4.2.9 Citizen Card Support .............................................................. 47

4.3 Back-office ...................................................................................... 47
  4.3.1 Filter requests by text ............................................................... 48
  4.3.2 Pagination on API requests that may retrieve many results .......... 49
  4.3.3 User personal signature image ................................................. 49
  4.3.4 Server REST API .................................................................... 49
  4.3.5 DOM changes - live vs explicit ................................................ 50

4.4 Performance issue with sign request upload ........................................ 51

4.5 Java Web Start .............................................................................. 51

4.6 Script for migrating between different database versions ...................... 52

4.7 Server Deployment Process ........................................................... 53

4.8 Multiple Signature Support ........................................................... 54

4.9 Audit log ....................................................................................... 55

4.10 Expired signatures investigation ...................................................... 56

4.11 AMQP with onion protocol .......................................................... 56

5 Case Study - DOT Integration .............................................................. 59

5.1 Purchase Order Document workflow ............................................... 61

5.2 Integration - Requirements ........................................................... 61

5.3 Integration - Implementation .......................................................... 62

6 Evaluation ......................................................................................... 65

6.1 Load Tests ..................................................................................... 67

6.2 User tests ..................................................................................... 68
  6.2.1 Test tasks ................................................................................ 70
  6.2.2 Results .................................................................................... 71
7 Conclusions

7.1 Conclusions ................................................. 75
7.2 Future Work ................................................ 75
List of Figures

2.1 Signed document signature validation process ........................................ 14
3.1 SmartSigner initial high-level architecture ........................................... 26
3.2 UML class diagram of the SmartSigner server application ......................... 30
3.3 SmartSigner Client Application architecture .......................................... 32
3.4 SmartSigner Client Application setup phase fluxogram ............................ 33
3.5 SmartSigner server architecture ......................................................... 34
3.6 Status documents can represent and possible transitions .......................... 35
3.7 Signing and certification process integrated in the SmartSigner system .......... 36
4.1 Revised UML class diagram of the SmartSigner server application .............. 41
4.2 Screenshot of Client Application after smart card is removed .................... 43
4.3 Screenshot of Client Application ....................................................... 45
4.4 Default logo plus text Signature Appearance ........................................ 46
4.5 Back-office interface ............................................................................. 48
5.1 Purchase Order Document workflow integrated with SmartSigner ............... 64
6.1 Stacked histogram representing the ration between the duration before and after the fix . 68
6.2 Histogram representing durations of sign requests upload in Test A ............ 68
6.3 Histogram representing durations of sign requests upload in Test B ............ 69
6.4 Questionnaire results, on a scale of 1 to 5, regarding ease of completion and intuitiveness of the tasks performed ......................................................... 71
6.5 Test users explicit interface preference ................................................ 72
List of Tables

4.1 List of endpoints requiring role User .................................................. 50
4.2 List of endpoints requiring role Administrator ................................. 50
Listings

4.1 Data migration method example for Users ........................................... 52
4.2 Example of a message from the Onion Protocol ................................. 57
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMA</td>
<td>Agência para a Modernização Administrativa</td>
</tr>
<tr>
<td>AMQP</td>
<td>Advanced Message Queue Protocol</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASiC</td>
<td>Associated Signature Containers</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>CC</td>
<td>Citizen Card</td>
</tr>
<tr>
<td>CEGER</td>
<td>Centro de Gestão da Rede Informática do Governo</td>
</tr>
<tr>
<td>CMS</td>
<td>Cryptographic Message Syntax</td>
</tr>
<tr>
<td>CRI</td>
<td>Certificate Request Info</td>
</tr>
<tr>
<td>CRL</td>
<td>Certificate Revocation Lists</td>
</tr>
<tr>
<td>CSR</td>
<td>Certificate Signing Request</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DGES</td>
<td>Direção Geral do Ensino Superior</td>
</tr>
<tr>
<td>DSS</td>
<td>Document Security Store</td>
</tr>
<tr>
<td>eIDAS</td>
<td>Electronic Identification, Authentication and trust Services</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
</tbody>
</table>
HEI  Higher Education Institution
HSM  Hardware Security Module
HTML  Hypertext Markup Language
HTTPS  HTTP Secure
IST  Instituto Superior Técnico
IT  Information Technology
JNI  Java Native Interface
JNLP  Java Network Launch Protocol
JVM  Java Virtual Machine
JWT  JSON Web Token
LTV  Long Term Validation
M-R  Miller-Rabin primality test
OAEP  Optimal Asymmetric Encryption Padding
OCSP  Online Certificate Status Protocol
OID  object Identifiers
OO  Object-Oriented
ORM  Object-Relational Mapping
OS  Operating System
PAdES  PDF Advanced Electronic Signatures
PDF  Portable Document Format
PIN  Personal Identification Number
PKCS  Public Key Cryptography Standards
PKI  Public Key Infrastructure
QRCode  Quick Response Code
RDBMS  Relational Database Management System
REST  Representational State Transfer
RFC  Request For Comments
RT  Request Tracker
SAN  Subject Alternative Names
SCEE Sistema de Certificação Electrónica do Estado
SDK  Standard Development Kit
SHA  Secure Hash Algorithm
SQL  Structured Query Language
SSD  Solid State Drive
SSL  Secure Sockets Layer
TLS  Transport Layer Security
UML  Unified Modeling Language
URL  Uniform Resource Locator
WORA Write once, Run Anywhere
Introduction

Contents

1.1 Goals ................................................................. 4
1.2 Document Outline .................................................. 5
Nowadays, many organizations still rely on paper to execute their workflows, which seems rather archaic, particularly considering the central role that digital technologies have taken in our daily lives. Since most of the information present in these workflows can be easily transformed from paper to a digital resource, most organizations, including Higher Education Institutions, such as Instituto Superior Técnico (IST), have been conducting significant efforts to adopt paperless practices. Indeed, digital transformation is becoming a common buzzword in organizations across the globe, as information technologies are increasingly perceived as the key enabler for achieving the, much sought-after, operational efficiency. The full implementation of paperless information management systems presents various benefits:

- improved business efficiency, since less time is wasted in non-essential tasks;
- higher security on document (legal) validation;
- better process consistency and traceability;
- easier inter-organizational interaction;
- environmentally friendly;
- easier storage and accessibility.

However, physically signed documents cannot be directly transformed into digital resources, since a physically signed or stamped paper document has no legal or authenticity guarantees after being digitized. Therefore, a direct digitization of such documents is not viable since it is trivial for an attacker or ill-intentioned individual to virtually add the image of a signature to a document.

By employing digital signature mechanisms in official documents, organizations expedite the signing process and the validation process, which in physical documents can often be cumbersome, requiring third parties such as notaries. Nevertheless, many organizations are still only able to deal with physical documents, and when a digitally signed document is printed its verifiability features are lost. This is a relevant issue, which needs to be properly addressed, since official documents produced by a given organization will often need to be processed by external organizations (for example, public transportation companies require registration certificates to give students reduced fares).

To tackle the problem of converting from signed or stamped paper documents to digitally signed electronic documents, one needs to understand the purpose of signatures. Why are signatures used? Better yet, why are signatures required? The traditional function of ink signatures is to provide evidence of the provenance of a document (identity) and the intention (will) of an individual with regard to that document. An electronic solution must provide the same guarantees: to give evidence of provenance, the new signatures must be something that only the signing entity or individual could generate; to prove informed consent, the solution must allow the document to be properly reviewed and analyzed, and may
even only enable the generation of a signature if the document was actually open and properly reviewed, for example.

The SmartSigner system was developed at IST to handle sign requests and signatures in electronic documents. The objective of this work is to describe the SmartSigner system, composed of the SmartSigner client application and the SmartSigner back-end server, identify its bugs and faults, expand its functionality, integrate into another system’s workflow, which uses signatures, and evaluate the work done.

1.1 Goals

At the beginning of this work, the SmartSigner system, composed of a Client Application and a Server, was still in its alpha stage despite being already released, which means that there were bugs and features that did not work properly or that were missing. Therefore, it was of paramount importance to the success of this project that the application be maintained, extended and documented. During the project lifecycle, especially in the field of Information Technology (IT), it is common for requirements to change during development or even after deployment. Thus, such changes must be designed and implemented. One example is the need to support the Citizen Card, which was a requirement introduced only after the deployment of the first release of the system. Some other new requirements and corrections were raised by its users, which also should be taken into account. Therefore, all reported bugs had to be corrected and missing functionalities had to be implemented.

The method of deployment of the Client Application also had to be reformed, since it did not allow for easy updates of the application, which would not only prevent the correct operation of the system but could also pose a serious security threat.

Considering that digital signatures clearly present various benefits, there should be an effort to identify and integrate in the system other IST workflows that require signatures. The integration had be documented and the mapping between the system domain and the other system’s domain was be made explicit.

In its alpha version, there was no current practical way to manage the SmartSigner server state. All operations to create, update or delete resources involving domain objects had to be performed by the system administrator, mainly via MySQL commands. Staff without this technical background had to rely on IT to execute simple tasks, such as adding a new queue. A solution that did not have such strict requirements had to be designed and implemented.

The initial version of the system did not support more than one signature in the same document. Since there are documents at IST that require more than one signature, the system had to be changed in order to accommodate for multiple signatures, properly allowing future integration of such workflows.
Likewise, the signature appearance (the visual cue that appears on the signed PDF documents) was static and did not take into account the possible differences that may be required by some processes. For example, despite the digitalization effort, and even if no legal value ensues, some organizations still require signed documents to be sent via fax. In this case, the signature appearance could be an image containing the hand-written signature of the signer. The fax document would therefore look the same, and, if needed, the digitally signed document can be accessed for proof of signature.

All these changes to the system required changes to the domain. Since the initial version of the SmartSigner system already stored a significant amount of important data, and manually adding it was not an option, a strategy to migrate the data to the new version of the system had to be envisioned and implemented.

The SmartSigner is a system that not only handles sensitive information but can also produce results that can be detrimental, such as obtaining a signature on a document with false information. This can be particularly serious for users with higher privileges, like the ones who can change the permissions on a queue. Therefore, all operations that change the state of the system (create, update and delete operations) must be documented for further review and control. Additionally, to mitigate access to sensitive data by unauthorized personnel, all document downloads had to be logged unless otherwise stated.

Digital signatures expire when the associated certificate also expires. Additionally, some signatures that should be valid and legitimate can become invalid if the certificate is revoked after the signature takes place, which is not ideal. An investigation of possible solutions to deal with this problem had to be conducted.

Finally, technical documentation for the application had to be produced, and the system was evaluated in terms of performance and usability.

1.2 Document Outline

This document is structured as follows: Section 2 provides an introduction to concepts and technologies needed to understand the context of the work presented, namely digital signatures; Section 3 describes why the SmartSigner system was created and details its initial state, its components, technologies and a the initially integrated workflow; Section 4 discusses new developments for the SmartSigner system performed in the context of the present work; Section 5 presents a case study of integration with an external workflow, the acquisition workflow of the DOT system, producing a signed purchase order document; Section 6 explains which and how tests were conducted, discussing the results; finally, Section 7 is the conclusion of this work.
Related Work

Contents

2.1 Digital Signatures .................................................. 9
2.2 Key Generation ....................................................... 12
2.3 Digital Certificates ................................................... 13
2.4 Smart Card .......................................................... 15
2.5 PDF Signatures ....................................................... 16
2.6 Legislation for digital signatures ................................. 19
2.7 Technologies ......................................................... 20
2.8 Other Solutions ...................................................... 21
2.1 Digital Signatures

A digital signature is an elegant concept invented by Diffie and Hellman [1], and is a mathematical scheme for demonstrating the authenticity of digital messages or documents, and it is an implementation of an electronic signature, a broader term that refers to any electronic data which intends to serve as a signature. The first implementation of digital signatures was created by Rivest, Shamir and Adleman, known as RSA [2] and combined two main parts: asymmetrical cryptography, a paradigm where a message ciphered with one key can only be efficiently deciphered with the other key (the pair), and a Public Key Infrastructure (PKI) to enable proper and safe public key distribution.

In symmetric-key encryption, the message resulting from encrypting data with a given algorithm and a given key can be decrypted using the same key, even if the algorithm is not the same (the typical case is applying the algorithm steps in the inverse order). As mentioned in the introduction, one of the premises of signatures is provenance. In a practical case scenario, if entity A requests a signed document from entity B to give to entity C, entity C must be able to verify if it was indeed B who created that signature.

In symmetric cryptography, the key is shared by all parties involved. Thus, entity C could not know without a doubt if it was entity A or entity B the origin of the signature. This is where asymmetrical cryptography provides a brilliant advancement: if the procedures for encrypting and decrypting data are different, and if by knowing one of them is still virtually impossible to know its counterpart, entity B could encrypt some data with a secret procedure but make the decryption procedure publicly available, and thus entity C could, for example, decrypt the message delivered by A using B’s publicly available decryption procedure and see if the resultant data is what was expected (for example, a specific document).

In a public key cryptosystem each user places in a public file an encryption procedure $E$. That is, the public file is a directory containing the encryption procedure of each user (and to be then propagated by the PKI). The user keeps secret the details of his corresponding decryption procedure $D$. These procedures have the following four properties [2]:

1. Deciphering the enciphered form of a message $M$ yields $M$. Formally,

$$D(E(M)) = M$$  \hspace{1cm} (2.1)

2. Both $E$ and $D$ are easy to compute.

3. By publicly revealing $E$ the user does not reveal an easy way to compute $D$. This means that in practice only he can decrypt messages encrypted with $E$, or compute $D$ efficiently.
4. If a message $M$ is first deciphered and then enciphered, $M$ is the result. Formally,

$$E(D(M)) = M \quad (2.2)$$

The safety of the whole algorithm relies on the secrecy of one of the procedures. Following the Kerckhoffs’s principle which states that “a cryptosystem should be secure even if everything about the system, except the key, is public knowledge” [3], and Claude Shannon’s maxim stating “one ought to design systems under the assumption that the enemy will immediately gain full familiarity with them” [4], we are left at a status quo where the only component of the procedure which could possibly be kept secret is the private-key component in the asymmetrical cryptography scenario.

To implement signatures the public-key cryptosystem must be implemented with trapdoor one-way permutations [5]. Basically, a function $F$ is one-way if, for a random $x$, given $F(x)$ it is hard to compute $x$. A function $E$ is one-way trapdoor if it is one-way and, in addition, there exists a secret piece of information $D$, called a trapdoor, which represents the inverse function, such that

1. $D(E(x)) = E(D(x)) = x$ (it is a permutation: every message is the ciphered text for some other message and every ciphered text is itself a permissible message); and
2. the knowledge of $D$ enables easy inversion (the trapdoor $D$, in the case of the RSA algorithm, is the private-key: the procedure key of the private procedure).

The idea of Diffie and Hellman has since been formalized, and proof techniques based on complexity theory have been developed. In particular, Yao has initiated the research program of basing cryptographic primitives on general assumptions [6].

To sign text $M$, entity A applies

$$S = D_A(M) \quad (2.3)$$

In equation 2.3, $D_A$ is the private procedure for decrypting $M$, even if it is not encrypted because of the property 1 of trapdoors explained above. Thus, entity B delivers $S$ to entity C and C applies the publicly available procedure in 2.4.

$$M = E_A(S) \quad (2.4)$$

Entity A cannot deny having created $S$, since only he knows the procedure (private-key). Entity C now is in possession of the message $M$ and its signature, $S$. In the RSA algorithm, the methods to encrypt and decrypt messages are 2.5 for a message $M$ and 2.6 for a ciphertext $C$.

$$C \equiv E(M) \equiv M^e \pmod{n} \quad (2.5)$$

$$D(C) \equiv C^d \pmod{n} \quad (2.6)$$
Note that the maximum size, measured in bits, of a message to be ciphered is \( n \). To generate RSA signature keys, first choose two large primes, and compute \( n \)

\[
n = p \cdot q
\]  

(2.7)

You then pick the integer \( d \) to be a large, random integer which is relatively prime to \((p - 1) \cdot (q - 1)\). That is,

\[
gcd(d, (p - 1) \cdot (q - 1)) = 1
\]  

(2.8)

The integer \( e \) is finally computed from \( p, q, \) and \( d \) to be the “multiplicative inverse” of \( d \), modulo \((p - 1) \cdot (q - 1)\). Thus we have

\[
e \cdot d \equiv 1 \pmod{(p - 1) \cdot (q - 1)}
\]  

(2.9)

The RSA algorithm could be used to create primitive signatures, but it was only presented as a proof-of-concept, since plain RSA signatures are not secure, since they are vulnerable to two attacks:

- **reordering attack**: if the size of the message is bigger than \( n \), the message first has to be split, and the signature will also be an array of signatures; an attacker can switch the order of the messages or even drop a message and its corresponding signature;

- **existential forgery attacks**: where an attacker can apply the publicly available procedure \( E_A \) in 2.4 to a random message \( S' \), and later claim that he has both the resulting \( M' \) and the signature \( S' \) [7, p. 150].

These attacks can be prevented by using a hash-then-decrypt scheme, where of directly signing the data intended, one first calculates the hash value of said data and only then applies the signing procedure to the result [8]. One-way hash functions [5], also known as message digest, are functions that can be used to map data of any size to data of a fixed, usually much smaller size. These hash functions must obey to 3 properties [9, Chapter 10]:

1. **Preimage Resistant** It should be hard to find a message with a given hash value.

2. **Collision Resistant** It should be hard to find two messages with the same hash value.

3. **Second Preimage Resistant** Given one message it should be hard to find another message with the same hash value.

By using these functions, not only we can avoid the reordering attack (since the hash is a fixed size, it only needs to be smaller than \( n \)) but also also eliminate the forgery attack [10].

For the purpose of this document, and regarding digital signatures, the final algorithm to generate signatures, RSA combined with a hash function, uses Secure Hash Algorithm (SHA) 2, more specifically the SHA-256 implementation [11].
2.2 Key Generation

As stated in the RSA algorithm specification in Section 2.1, "to generate RSA signature keys, first choose two large primes". However, the procurement of such numbers is not as trivial as it may seem. Finding a prime number is easy enough, but there are some constraints: the prime numbers must not be as small as that would make it rather easy for an attacker to guess it by brute force; since the hash algorithm that will be used afterwards is going to be SHA-256, wish produces a result with 256 bytes, the modulus must be at least the same length: 2048 bits, which is also going to be the key length (for the reasons already mentioned in ??). The modulus $n$ is derived from the multiplication between the two prime numbers, so the sum of number of bits in both prime numbers must be 2048. To accurately represent the number resulting from the multiplication between two numbers with $j$ and $k$ bits, we need a number with $j \times k$ bits, so the prime numbers must be in of 1024 bits each (since the smaller the number, the easier it is to factorize it). For the purpose of this explanation, we will use the 512 bits number as example.

But are there enough primes to choose from? According to Prime Number Theorem [12], the number of primes less than $n$ approximates the function 2.10.

$$\Pi n \sim \frac{n}{\ln (n)} \quad (2.10)$$

This means that, with 512 bits, there are about $\frac{2^{512}}{\ln (2^{512})} \approx 3.778 \times 10^{151}$ prime numbers, making collisions seem virtually impossible.

It only remains to actually find one of such numbers. One seemingly practical solution would be to get them from a list of prime numbers precomputed; this solution would be a serious security flaw since it would considerably reduce the pool of prime numbers to choose from. Just to put it in perspective, approximating the number of 512 bits prime numbers to $10^{150}$, one would need $\frac{10^{150} \times 512}{8 \times 2^{50}} = 5.68 \times 10^{136}$ terabytes\footnote{1 terabyte = $2^{30}$ bytes. This is specified since there is also the 1 terabyte = $10^{15}$ bytes convention.} of information to store them. There is also the concern of the number being made known of by a third party, and both numbers shall be kept secret [13, p. 22]. Therefore, the procurement of prime number must be done ad hoc.

We can get randomness, or at least pseudo-randomness, from the operating system. In 512 bit numbers, the probability of randomly selecting a prime number is $\frac{2^{512}}{\ln (2^{512}) \times 2^{512}} \approx 0.00649$, so the average distance between 2 prime numbers is $\ln (2^{512}) \approx 354.89$. Checking if a number is a prime by factorizing it is computationally expensive (and the reason why RSA implementations can be secure), so another method, based on the Fermat primality test [14] called the Miller-Rabin primality test (M-R) is used instead, which is a probabilistic algorithm [15]. The M-R relies on an equality or set of equalities that hold true for prime values, then checks whether or not they hold for a number that we want to test for
primality. This test can have multiple rounds of checking, and the greater the number of successfully passed rounds, the bigger the probability of the given number being a prime.

The ideas of Michael O. Rabin paper were applied to estimate \( p_{k,t} \), the probability that an odd \( k \)-bit integer that passes \( t \) rounds of M-R testing is actually composite. The probability \( p_{k,t} \) is understood as the ratio of the number of odd composite numbers of a binary length \( k \) that can be expected to pass \( t \) rounds of M-R testing (with randomly generated bases) to the sum of that value and the number of odd prime integers of binary length \( k \). This is equivalent to assuming that candidates selected for testing will be chosen uniformly at random from the entire set of odd \( k \)-bit integers. A probability of \( 2^{-100} \) is included for all prime lengths, since this probability has often been used in the past and may be acceptable for many applications [13, p. 119]. With a 512-bit number, we can achieve this probability with 300 rounds of the test.

If the test implies that the picked number is not a prime number, the picked number is added by 2 (resulting in next odd number) and the tests are repeated. A prime number is expected to be found in about \( \ln \left( \frac{2^{512}}{2} \right) \approx 177 \), which is the average distance between prime numbers divided by 2 since we only check for odd numbers.

### 2.3 Digital Certificates

We already know how the algorithm works, and how the keys are obtained. However, there is still a problem to be solved: how to distribute the public key? Let’s suppose entity A signs (the hash of) a document \( D \) and sends it to entity B. For B to verify the validity of the signature, he needs to now A’s public key, \( K^{-1}_A \). The way he retrieves this information is to be considered cautiously, lest we end up with a security flaw: an attacker H could sign document \( D' \) with his own private key, send it to B, and make B believe that the document was signed by A if he can intercept the public key that B has of A and switch it with his own public key, \( K^{-1}_H \). To prevent this type of attacks, a PKI is needed to disseminate the aforementioned public keys and digital certificates were created.

A digital certificate is an electronic document used to prove ownership of the public key. A Certificate Authority (CA), which is an entity that issues certificates, uses his own private key to sign those certificates, such that its public key can be validated by the requester. The function of the CA is to create the certificates for all the parties involved, except those above it in the chain of trust, which is an ordered list of certificates, containing CA Certificates [16] and the final certificate that was used to sign the data.

For example, given entities A, B and C, A issues B’s certificate, and B issues C’s certificate. A is designated the root CA because its trusted by default and is self signed. In order for another entity to trust a document signed by C and containing C’s certificate, by verifying that the certificates chain from C to B and then to A, which this entity trusts (because it is the root CA), trust in C’s certificate ensues.
These certificates follow a standard designated by X.509 [17]. Digital certificates are verified using a chain of trust. The trust anchor for the digital certificate is the root certificate authority. For the validation of the certificate to succeed, the signature’s associated certificate must be part of a certificate chain trust containing a CA already recognized by the operating system (such as VeriSign\(^2\)), no additional steps are required. Otherwise, the root CA can be manually added to the system’s trust store.

Certificates have an expiration date, being usually valid for a couple of years. There are situations which require a certificate to be revoked: for example, if the private key associated with the public key is stolen from an entity, the signatures henceforth generated with it can no longer be trusted, since the thief could now use the key to generate valid and apparently legitimate signatures. To accommodate this possibility, Certificate Revocation Lists (CRL) [17] are included in the certificate, containing the list of revoked certificates. Those are maintained by the CA which issued the corresponding certificate. These lists can be constantly changing, and since the distribution of these lists is a problem within itself, a protocol known as Online Certificate Status Protocol (OCSP) [18] was created, to enable real-time certificate status checking.

It is important to know that a document signed with the private key, the pair of the public key contained in a certificate, is only considered valid if both the signature and the certificate are valid. Since a signed document must contain the public certificate of the signer, to check if the signature is valid, one can compute the hash of the document, and then compare it to the result from executing the public decryption procedure described in the certificate. If they don’t produce the same result: the hash, as shown in Figure 2.1, the signature is considered invalid.

![Figure 2.1: Signed document signature validation process](http://www.verisign.com)

\(^2\)http://www.verisign.com
For the signature to be valid, the certificate associated must also be validated. To verify if a certificate is valid and or trusted, the system must first check if it is not yet expired by comparing the current date with the certificates validity stored in the field “Not After”. Then, attempt to establish a chain of trust to a trusted CA, by referring to the field containing the certificate issuer (CA that created the certificate) until a certificate that is trusted by the system is reached. Finally, the status of the certificate status must be checked using CRL or, preferably, OCSP.

There is still one missing link to make the chain of trust secure. How can a CA trust that the entity asking for its public key to be issued a certificate is in fact in possession of the associated private key? To prove key ownership, the certificate requester must send a Certificate Signing Request (CSR) to the CA. The most common format is the Public Key Cryptography Standards (PKCS)#10 specification [19], and the process is the following:

1. A Certificate Request Info (CRI) value containing a subject distinguished name, a subject public key and optionally more attributes are constructed by an entity requesting the certification;

2. The CRI is signed with the subject’s private key;

3. These two components, along with a signature algorithm identifier, are sent to the CA, which may check the legitimacy of the information in the request regarding the subject and verify if the signature matches the public key contained therein.

2.4 Smart Card

As mentioned, the security of signature algorithms relies on the ability to keep secret the private key of each signer. To achieve so, the key must not ever be copied, and knowledge of it must only involve the absolutely required people and/or systems to generate signatures. Ideally, it should only exist in one place, and should never be accessed by an outside system, hence the utilization of a Hardware Security Module (HSM) usually the best way to enforce these restrictions. An HSM is a physical computing device that safeguards and manages digital keys for strong authentication and provides cryptographic functionalities.

One of the existing and portable, yet secure, HSM is the Smart card. Smart cards are pocket-sized cards embedding integrated circuits containing a tamper-resistant security system capable of providing security services, authentication or running cryptographic algorithms [20]. Due to its reduced size, and consequently the small computational power, most smart cards run a simplified version of Java that is faster and utilizes less memory than version desktops use.

The smart cards mentioned in this document are all contact smart cards, since they provide extra security and hold both the public and private keys, with enough computing power to generate signatures.
within the card, preventing exposure of the private key as intended, since the private key never leaves the card; instead, the system that intends to use it must send the data to be signed and what algorithm to use, for example, and the smart card returns the result if supported. PKCS#11 refers to the programming interface to create and manipulate cryptographic tokens. As long as the proper libraries are provided from the smart card manufacturer, one can communicate with the smart card using this protocol.

2.5 PDF Signatures

Portable Document Format (PDF), a file format developed in the 1990s to present documents, including text formatting and images, in a manner independent of application software, hardware, and operating systems, also supports digital signatures. A digital signature may be used to authenticate the identity of a user and the document’s contents. It stores information about the signer and the state of the document when it was signed. The signature may be purely mathematical, such as a public/private-key encrypted document digest, or it may be a biometric form of identification, such as a handwritten signature, fingerprint, or retinal scan [21]. The type that interests us is the digital signature.

Digital signatures in ISO 32000-1 [21] currently support two activities: adding a digital signature to a document and later checking that signature for validity. Revocation information is a signed attribute, which means that the signing software must capture the revocation information before signing. A similar requirement applies to the chain of certificates. The signing software must capture and validate the certificate’s chain before signing the document.

Signatures shall be created by computing a digest of the data (or part of the data) in a document, and storing the digest in the document. To verify the signature, the digest shall be re-computed and compared with the one stored in the document. Differences in the digest values indicate that modifications have been made since the document was signed.

A PDF is a structured format and an entry is one of its fields. A message digest shall be computed over a range of bytes in the file, that shall be indicated by the ByteRange entry in the signature dictionary. This range should be the entire file, including the signature dictionary but excluding the signature value itself (the Contents entry). This is done to allow for digital signatures; otherwise, by adding the signature the document hash would change, and the signature could never be validated, since the message digest would include the signature itself.

A PDF document may contain the following standard types of signatures:

- One or more approval signatures. These signatures appear in signature form fields. Signature fields represent digital signatures and optional data for authenticating the name of the signer and the document’s contents;

- At most one certification signature. This signature, if present, shall be the first to be included,
and dictates what can be subsequently done with the document, e.g., if additional signatures are allowed. This will be important later on, when discussing multiple signatures in one document.

The PDF signatures referenced in this document will be PKCS#7 signatures [21, Ch. 12.8.3.3], which are used to digitally sign, digest, authenticate or encrypt arbitrary message content. The PKCS#7 object shall conform to Cryptographic Message Syntax (CMS), the IETF’s standard for cryptographically protected messages [22], and to the specification PKCS#7 as defined in the Request For Comments (RFC) 2315 [23].

The SubFilter entry specifies the syntax of the encryption dictionary contents. It allows interoperability between handlers; that is, a document can be decrypted by a handler other than the preferred one (the Filter entry) if they both support the format specified by SubFilter [21, Ch. 7.6.1]. There are two subfilters possible for these signatures:

- **adbe.pkcs7.detached**: The original signed message digest over the document's byte range shall be incorporated as the normal PKCS#7 SignedData field. No data shall be encapsulated in the PKCS#7 SignedData field.

- **adbe.pkcs7.sha1**: The SHA1 digest of the document's byte range shall be encapsulated in the PKCS#7 SignedData field with ContentInfo of type Data. The digest of that SignedData shall be incorporated as the normal PKCS#7 digest.

In 2017, CWI Amsterdam and Google announced they had performed a collision attack against SHA-1, publishing two dissimilar PDF files which produced the same SHA-1 hash [24]. It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file. Therefore, the *adbe.pkcs7.sha1* subfilter shall not be used.

The PKCS#7 object should contain the following information:

- Time stamp information as an unsigned attribute: the timestamp token shall conform to and shall be computed and embedded into the PKCS#7 object as described in Appendix A of RFC 3161 to define [25];

- Revocation information as a signed attribute: proof that the certificates were not expired nor revoked at time of signature. This attribute may include all the revocation information that is necessary to carry out revocation checks for the signer’s certificate and its issuer certificates. Since revocation information is a signed attribute, it must be obtained before the computation of the digital signature;

- At minimum, it shall include the signer’s X.509 signing certificate. This certificate shall be used to verify the signature value in Contents. Ideally, the whole certificate chain is included.
We now know how a signature can be included in a PDF file and how it can be validated. The only missing step is to explain the actual process. As stated in Section 2.1, the algorithm used is the RSA combined with SHA-256 as defined in PKCS#1: RSA Cryptography Specifications Version 2.1 [26], as the process for PDF signatures is:

1. Calculate the hash of Byte Range of the document excluding the signature field;
2. Padding the hash obtained using Optimal Asymmetric Encryption Padding (OAEP). The padding protects against two attacks [27]:
   - Since exponentiation is homomorphic, RSA-based protocols are traditionally protected against chosen-ciphertext attacks [28], where an attacker with two signatures of the same entity can claim he has a third signature that was generated by the signer, because of the equation 2.11.
   
   \[
   RSA(x) \times RSA(y) = RSA(x \times y) \pmod{n} \tag{2.11}
   \]
   Adding a padding (or redundancy) into \(x\) and \(y\) we can make equation 2.11 not be true.
   - Encryption padding is also necessary to avoid dictionary attacks [29]: by adding a random string to the encrypted message, the re-encryption of very short messages (such as yes or no) does not allow for building-up dictionaries.

   Padding works by, after hashing on step 1, concatenating the message digest with predefined-strings. Substitution and permutation can be used as well;
3. Convert the result of the padded hash into a number. This result is treated as an octet string and converted to a nonnegative integer using the function OS2IP as described in RFC 3447 [26, Ch. 4.2];
4. Calculate the modular exponentiation using the private exponent (private-key) and the modulus \(n\), as defined in 2.1. The number obtained is the signature;
5. Convert the signature, a nonnegative integer, into an octet string, using the reverse of the previous function, I2OSP [26, Ch. 4.1].

This signature is also sent to a timestamp server to be signed and also included in the PDF signed document.

It is important to mention that PDF Advanced Electronic Signatures (PAdES) is a set of restrictions and extensions to PDF and ISO 32000-1 making it suitable for Advanced Electronic Signature [30]. One important benefit from PAdES is that electronically signed documents can remain valid for long periods, even if underlying cryptographic algorithms are broken.
2.6 Legislation for digital signatures

Digital signatures are an implementation of advanced electronic signatures. According to EU regulation No 910/2014 [31], referred to as the Electronic Identification, Authentication and trust Services (eIDAS) Regulation, an advanced electronic signature is an electronic signature which meets the following requirements:

1. it is uniquely linked to the signatory;

2. it is capable of identifying the signatory;

3. it is created using electronic signature creation data that the signatory can, with a high level of confidence, use under his sole control; and

4. it is linked to the data signed therewith in such a way that any subsequent change in the data is detectable.

All of the requirements in this list are met by digital signatures such as PAdES [30]. PAdES specifies precise profiles making it compliant with the European eIDAS regulation. An electronic signature that has been created in compliance with eIDAS has the same legal value as a handwritten signature.

Qualified electronic signatures are capable of identifying the signatory remotely, using electronic identification means, for which prior to the issuance of the qualified certificate, a physical presence of the natural person or of an authorized representative of the legal person was ensured. If a body governed by public law serves as the CA to the certificates that will be used by the signers, we can elevate the signatures to qualified. ‘Electronic time stamp’ means data in electronic form which binds other data in electronic form to a particular time establishing evidence that the latter data existed at that time. This regulation states that an electronic signature which contains a qualified time stamp (i.e. distributed by the appropriate government entity), holds the same legal value as a handwritten signature would have, effectively making it a viable replacement for old-fashioned signatures.

For the purposes of this document, and when referring to a digital signature, one can read advanced qualified electronic signature.

One important note is that a signed document can only be validated in its digital format. Therefore, any printed document that previously contained a digital signature loses its legal value after being printed or suffering any other transformation. On the other hand, a digitally signed document can be digitally copied and retain its signature properties, since it becomes virtually indistinguishable from the original (the sequence of bytes is exactly the same for the copy and the original).
2.7 Technologies

All the technologies and techniques mentioned in this document are briefly explained in this section.

Java is an Object-Oriented (OO) programming language that runs on a Java Virtual Machine (JVM). It is concurrent, class-based and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers “Write once, Run Anywhere (WORA)”, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

iText is a library for Java used to create, read and modify to generate and read PDF documents. The version referenced in this document is version 2.1.7, which is the last to have a General Public License (GPL), a free, copyleft license for software and other kinds of works.

Representational State Transfer (REST) Web services are a way to provide interoperability between different systems over the Internet, providing compatibility between different systems communicating.

Spring Framework provides a comprehensive programming and configuration model for enterprise-level Java-based applications, featuring dependency injection, aspect-oriented programming paradigms, and others.

AngularJS, Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), Javascript are languages and tools for web development, specially helpful to develop interfaces and for front-end development.

JavaFX is a software platform for creating Graphical User Interface (GUI) for applications in Java that can run in a wide variety of devices.

Quick Response Code (QRCode) is a type of matrix bytecode used in this context to translate an image to a web Uniform Resource Locator (URL).

Since we will be dealing with digital signatures, and consequently public-key cryptosystems, it may be helpful to understand what PKCS#11 is. PKCS#11 is one of the Public-Key Cryptography Standards and it defines a platform-independent Application Programming Interface (API) to cryptographic tokens, such as smart cards, and, for example, allows access to its X.509 certificates. The manual for this API utilized for this project is the PKCS#11 Cryptographic Token Interface Base Specification Version 2.40 provided by oasis.

MySQL is an open-source Relational Database Management System (RDBMS), and is based on Structured Query Language (SQL), a domain-specific language used in programming and designed for managing data held in a RDBMS, or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data where there are relations between different entities/variables of the data.

---

2.8 Other Solutions

2.8.1 University of Porto, Portugal

The Universidade do Porto, U. Porto, has already integrated digital signatures⁴. They have two HSMs running SignServer, an open-source solution to digital signatures, with qualified certificates that are used to sign documents that will then appear as signed by “Universidade do Porto”. This solution differs from SmartSigner in a major way since there is no guarantee of who actually sign the document, and there can be no delegation of responsibility or functions, which in our case is solved with different smart cards. For example, if there is document that must be signed by the responsible faculty for a given course, we can ask the qualified organism to issue a new card for that role, and give it to the corresponding authority. With SmartSigner it is also possible to sign internal documents and have them validated between different IST centers. The lack of portability of this solution compared to SmartSigner, which distributes a Client Application, is also a factor to consider.

2.8.2 University of Amsterdam, Netherlands

The University of Amsterdam was one of the Higher Education Institution (HEI) pioneers in implementing digital signatures in Europe. This was done recurring to meetings between representatives of national universities, and each had to implement digital signatures as they saw fit as long as they followed the standards and security requirements explained in section 2.6. These meetings followed the Scrum methodology and contained staff from national universities, using it also as peer pressure to incite the continuous development of everyone involved [32].

2.8.3 Riga Technical University, Latvia

Riga Technical University, the largest technological university in the Baltic States⁵, also has implemented a solution for digitally signed documents but in a different way: instead of the more conventional PAdES, they used the Associated Signature Containers (ASiC) [33] for eIDAS which produces a ZIP⁶ file containing both the data and the metadata (including signatures) as opposed to modifying the document itself (such as PDF signed documents) [34]. The main advantage of this approach is greater freedom on what documents/files can be signed, whereas the main disadvantage is the added complexity of the documents in various fronts: instead of a document, we end up with a ZIP format folder (container), and the validation of the signature(s) either require(s) extra steps, specific software ore, in the case of this

⁴https://sigarra.up.pt/up/pt/noticias_geral.ver_noticia?p_nr=24070
⁵http://fsd.rtu.lv/university/?lang=en
⁶Archive file format that supports lossless data compression.
University, a website where one can upload the ZIP file to check if it is valid. A digitally signed PDF document can be validated using the official PDF reader, Adobe Reader.

The generation (or validation, if so desired) can be done by using the provided DigiDoc3 client called Latvian ID-software⁷. The Latvian Public Administration Services Portal also supported digital signatures by making available a webpage explaining what they are, what purpose they serve and how to make use of them, including using the national ID card⁸.

⁷https://www.id.ee/?id=37268
3
Existing System

Contents

3.1 Original System Requirements ......................... 25
3.2 Smart Card Emission ...................................... 26
3.3 The student registration process .......................... 28
3.4 SmartSigner Integration Goals ............................. 29
3.5 SmartSigner .................................................. 29
3.6 Certifier ..................................................... 32
3.7 The new workflow of the registration process .......... 34
This chapter describes the original SmartSigner system, why it was born, which cards are used with it and the workflow that originated the necessity of said system.

### 3.1 Original System Requirements

The original goal underlying the development of the SmartSigner system was to replace the hand-written signatures required on the declarations of enrollment produced during the student registration process. Thousands of these documents had to be printed and be physically signed, and thus the system that integrated digital signatures in this process was developed.

The original system, however functional, was rather primitive, and for a good reason: it was developed in a short time: the development began on June the 3rd, 2016 and was released on September the 11th of the same year, in effectively 3 months and 8 days. Even so, the system was envisioned with a modular design, that would enable its easy maintenance and future expansion. The requirements presented below reflect that philosophy:

1. The system must be generic, divided into queues, sign requests, users, permissions - the concepts should not be tightly coupled with those of FenixEdu Academic or any other system;

2. The system shall be able to produce valid and legitimate signed PDF documents – the PDF format is supported in most operating systems, such as Android, MacOS, iOS, Windows and Linux and is the favored document type on many institutions, including IST;

3. The signatures shall be produced resorting to a smart card - issued by the government as required for qualified electronic signatures and that describes the function of the card holder with regard to IST;

4. The system shall be divided into a local Client Application and the server – since the Client Application is going to communicate with the smart card, whose manufacturer must provide the libraries for, using for example a web-app would be extremely more complicated;

5. The communication between the Client Application and the server must be stateless, namely through HTTP requests - access to sockets is limited and controlled, and can be forbidden altogether on some systems, but HTTP requests should be allowed, even on non-administrator accounts;

6. The Client Application must be compatible with the three most used Operating System (OS), MacOS, Windows and Linux – any solution that requires the development of specific executables for each OS must be avoided at all cost;
7. The server should connect to a persistent transactional database - to avoid faulty states when an operation is not fully completed or fails after changes were already committed;

8. Each user shall authenticate to the system using its own, individual, smart card.

![Figure 3.1: SmartSigner initial high-level architecture](image)

The diagram in Figure 3.1 depicts a draft of a high-level architecture of the system addressing the previously mentioned requirements.

### 3.2 Smart Card Emission

The digital documents are to be signed with a smart card, according to the requirements described in section 3.1. One of the possible smart cards that can be legally used for signing documents with is the Citizen Card (CC). However, in IST’s case and in many other situations the authority responsible for signing each type of official document is a person who holds a specific function (position) in the university. This represents a problem for an external entity that needs to verify or validate a signature. The problem lies in the difficulty to correlate the person who signed a document using his CC with its function at the organization: such information may not be publicly accessible. In addition, the person may no longer hold that function at the time of verification, which makes it even more difficult because the third party entity needs to discover whether the function claimed at the time of signature was indeed correct. Finally, not all system users own a CC (e.g. faculty from a foreign country most often only possess an identification document from their country of origin).
The public organizations in Portugal must follow government regulations, and, in our concrete case, since IST is a public school, its employees are also public workers. Therefore, delegating the responsibility of stating which employee has which office function to a governmental entity seems intuitive and practical. The creation of smart cards containing that information is conducted by the Centro de Gestão da Rede Informática do Governo (CEGER), the governmental office responsible for certification. At request, CEGER produces smart cards for some employees of IST, with certificates created and managed by Sistema de Certificação Electrónica do Estado (SCEE), a hierarchy of trust that handles the electronic security of the government and strong authentication of Public Administration organisms to the citizens. These certificates follow the X.509 standard, as referred in Section 2.1 and it is imperative that they contain information about the person that the smart card is being assigned to, and the position of that employee in the organization.

In X.509v1, this data could not be properly represented, but X.509v3, an extension of the certificate, contains a field to hold the Subject Alternative Names (SAN). Since simply stating the function of the card owner has no legal value, given that there is no backed legislation (and these signatures must hold the same legal value as the traditional ink signature), an object Identifiers (OID) was created with that purpose (2.16.620.1.1.2.2.0.2.1), holding legal value. The SAN also contains the holder’s email. The aspects and purposes of how and when the certificates contained in these smart cards can be used are explained in an official document issued by CEGER, entitled Declaration of Practices of Certification [35], with the registered OID 2.16.620.1.1.2.3.1.2 and made publicly available, containing the guidelines that developers and consultants must follow to properly implement a system that correctly and legitimately uses the issued cards.

The smart cards contain three certificates: one for authentication and key agreement to prove one’s identity to a remote computer, one for signature with its key usage as non-repudiation, and one for ciphering which can be used to protect email messages. Every certificate should include usage of signature algorithm SHA-256 with RSA encryption, as mentioned in the end of section 2.1. The maximum validity of the certificates is 5 years, although the currently emitted cards have a validity of three years. The size of the RSA keys is 2048 bits. The keys for authentication and ciphering are generated in cryptographic hardware modules according to the Federal Information Processing Standard (FIPS) 140-2 level 3, a standard that specifies the security requirements of a security system protecting sensitive information [36]. Signing private keys should be handled with special care, since they can be used to legally sign documents, and as such are generated inside the card itself, meeting the requirements for secure signature creation devices, which follow the eIDAS minimal requirements for that purpose, and has a Common Criteria’s Evaluation Assurance Level 4+, an international standard for computer security certification. All this information can be found in [35].

\[1\text{http://www.ceger.gov.pt}\]
3.3 The student registration process

IST is the leading engineering school in Portugal and one of the leading engineering schools in Europe, with more than 12,000 enrolled students almost 2,000 faculty members and researchers. IST’s academic and administrative management software, FenixEdu Academic, has been developed in-house for more than 13 years and is the result of the school’s efforts to eliminate the need for paper and more efficiently maintain academic information, such as student, teacher and staff information, class schedules, allocated classrooms, exam schedules, etc. Aligned with this long-term effort, it was decided to improve an important, yet still inefficient, workflow: the student registration process.

The student registration process starts after the National Contest for Admissions to Higher Education is completed. The admission list, containing all students that were admitted to the BSc degrees of IST on a given year, is provided by Direcção Geral do Ensino Superior (DGES), which is the governmental office responsible for the admissions to all public universities. IST imports the list of admitted students into FenixEdu Academic system, classifying these students as new candidates and creating new accounts (username/password pairs) for them, which will subsequently enable them to access all computer resources in the school, namely the FenixEdu platform.

On the day of registration, candidates must physically come to the school, where they receive a sheet containing the account information, with their username and password, which is then used to access FenixEdu Academic system. The candidate must then complete its registration process, at the beginning of which he is invited to introduce his Citizen Card, enabling the system to collect the candidate’s personal information stored in the card (e.g. ID number, photo, name, address, etc). If no Citizen Card is provided, the information that would otherwise be automatically collected must be manually introduced by the candidate. The registration process proceeds and the candidate is then required to fill several additional online forms containing varied information mostly required for statistical purposes. After that, the candidate registration is processed by the FenixEdu Academic system, and he effectively becomes a student of IST. The system then issues an identification sheet, containing all the student's relevant information, that must be signed by the student after checking that all the information therein is correct.

In order to complete the registration process, some sheets needed to be printed:

- The student’s schedule;
- The tutor info;
- The tuition payment info, which consists of an ATM reference and the value in debt;
- Eight registration certificates that must contain an embossed stamp and must be signed by a member of the staff of the academic office of IST
Form required by banks, to be filled and signed by the student.

Apart from the necessity of printing all these documents, some need to be ink signed and stamped and physically delivered to the student: a process that overall is inefficient and expensive.

Outside the campus, many organizations also require a signed version of a registration certificate document. Some examples are the public transportation services (to offer reduced fares to students), and health insurance companies. If the eight registration certificates received by the student are not enough, the student may need to request additional ones. These new ones need to be printed as well, ink signed and stamped by someone in the academic office. In the year of 2016, IST had 1714 new registrations, therefore in the absence of a digital solution IST staff would have to print, hand-sign and emboss 8 sheets of paper per student, amounting to 13 712 documents in, roughly, one week.

### 3.4 SmartSigner Integration Goals

The SmartSigner system approach tackles resource waste in the student registration process, while eliminating the need for paper. All printed sheets that do not require a signature can be easily removed from the process by making use of the FenixEdu Academic system, e-mail or even SMS, leaving only the documents requiring signatures and stamps to be dealt with. To enable their transformation to digital format as well, a model was defined that replicates the system's entities involved in the process. This model is generic enough to enable its use in other workflow processes that may also require electronic signatures.

In the case study presented, the registration certificate had to be ink signed by a member of IST with a proper authority, who asserts that the student is indeed a student of the school. SmartSigner allows staff members with such authority to digitally sign the registration certificate document, using a smart card emitted by the government which asserts that the owner of that smart card is, for example, a staff member of IST.

### 3.5 SmartSigner

SmartSigner was designed following an OO paradigm, and as such is structured in various object classes, as illustrated in Figure 3.2.

**Document** - class aggregating all the information associated with each document that may be required by the application;

**SignRequest** – relation between a Document and a Queue, containing the URL which must be called after a Document is signed (for the certifier) and the status of the request;
Figure 3.2: UML class diagram of the SmartSigner server application

Queue - logical grouping of sign requests, created by an administrator;

User – system users that authenticate using a smart card containing the public key certificate that was previously extracted and stored in the server;

TokenHolder – class representing card objects and the relation with a User;

TokenCertificate – contains X509 certificate, a TokenType and the certificate hash, which can be used to compare between certificates or find one in specific;

TokenHolderType – type of smart card (token holder) supported by the system, such as CEGER card or Citizen Card;

TokenType – type of certificate present in the token holders. There is authentication, signing and cipher.

Note that in the Unified Modeling Language (UML) diagram the TokenHolderType and the TokenType are not depicted (for simplicity reasons). TokenHolder has one TokenHolderType and a TokenCertificate has one TokenCertificateType.

A signed document is the result of three conditions:

1. the existence of a document to be signed;

2. an authority with the responsibility of signing the document;

3. a mechanism to generate the signature.

Completing the analogy, the paper document will be a computer file, the authority will be a user with authorization to sign that document, and the mechanism will be a token holder, in this case a smart card.

After generating a digital document to be signed, the originating system (the FenixEdu Academic in this particular case) will deposit the document in the proper queue of the SmartSigner system. The
SmartSigner system segregates the documents to be signed by queues. The documents on each queue can only be signed by a predefined group of users, which should be the ones legally authorized to sign the kind of documents that will be deposited in the queue, and have a sign permission configured in the system to do so.

### 3.5.1 The Client Application

The Client Application is a standalone Java application using the JavaFX graphics package for the interface design. This component is responsible for generating document signatures. Being a Java application, it does not require installation; however, the terminals (card readers) may need drivers installed to be used. The Client Application communicates with the smart card via the terminals (card readers) to execute the cryptographic algorithms necessary to generate a valid signature using the libraries provided by the card vendor (bit4ipki.dll for the CEGER card, pteid.dll for the citizen card in Windows; the libraries vary by operating system and architecture). The libraries are loaded using java.lang.Runtime.loadLibrary() and its methods are called using reflection (the ability of Java of identifying and invoking previously unknown class attributes and methods at runtime). Users authenticate themselves with the system by sending the authentication certificate present in their registered card via HTTP Secure (HTTPS), and if the server recognizes it, a Secure Sockets Layer (SSL) context is established between the two, using Transport Layer Security (TLS) v1.2, to provide authenticity and confidentiality, as shown in Figure 3.3. The Client Application can also be used by the administrators to extract the public key certificates of a new smart card, necessary to add it to the system, by starting the Java application with a specific flag (-Dadmin). This flag is a JVM argument and, if present, starts the application in the extraction mode; otherwise starts the Client Application normally, by trying to login and connect to the SmartSigner server. Extraction mode currently requires the Personal Identification Number (PIN) of the card as user’s input to retrieve the certificates. Figure 3.3 shows the architecture of the Client Application.

After launching the application, there is a setup phase where the user selects the terminal, the card type and then introduces the pin and tries to login or extracts the certificate. After successfully logging in the smart card, a preferences.cfg file is created so further utilizations of this application do not require this setup. This initialization process is represented in Figure 3.4.

### 3.5.2 The Server (repository)

The Server is the component responsible for maintaining the state of the SmartSigner system (users, token holders, queues, etc.) and orchestrating the document workflow. It makes extensive use of the Spring Framework due to the benefits it provides, such as exposing its REST API as defined in the
controllers, using Hibernate to manage objects in session and connecting to a MySQL database for persistence. The Client Application communicates with this server to retrieve information about the resources and to make operations. This server also stores the actual documents, so that they can be signed or downloaded. Within the signing workflow, the server both validates if the signature matches the signer certificate (and the sign permission corresponds), and if the signature is in fact valid. The repository where documents are stored could be in a different machine other than the one where the Server is hosted, in which case is the Server’s responsibility to act as a proxy between the SmartSigner’s Client Application and such repositories.

### 3.6 Certifier

The certifier is a component from the FenixEdu system where digitally signed documents are available.

In Section 2.6 it is mentioned that the migration from a digital version of the signed document to a physical one is not trivial, since the signature validity is lost on the transformation. The adoption of paperless workflows has been slow, and most often only within the organizations. While many organizations already accept incoming (external) documents through email, website interfaces or web services, this is still not the norm, at least in Portugal. Therefore, the problem of communicating digitally signed
documents with organizations that would not accept them in digital form, most often because they do not possess the technological means to do so, was a pressing issue that needed to be addressed.

After a signing request is completed successfully, the produced digitally signed document is stored in FenixEdu Academic, using the URL configured in the system when the digital document was uploaded (callback URL), and can be referenced through an identifier. In order to enable easy access to the digital document for organizations without the ability to process and store it in digital format, a print ready digital document was produced, which has an extra front page containing the complete URL of the original digital version, and a QRCode to quickly enable access to that same URL, effectively acting as an access code to the digital document.

The receiver of a digitally signed document now has two options for accessing its original, digital version: either by manually introducing the link in a browser, or by scanning the QRCode using a regular smartphone or webcam. The different document states and possible transitions between them are illustrated in Figure 3.6, showing that the printed signed document cannot be directly validated. Clearly, there is no way to validate a signed document in its printed format, and the same can only be used to point to the digitally signed document, which differs from the real life scenario where hand signed paper documents can be!verified.

Ideally, the Certifier would be managed by a governmental organization that stores the signed document, so external entities can verify that the paper version of the signed document points to a valid signature. Originally, it was intended for Agência para a Modernização Administrativa (AMA) to host
a system with that exact function. However, for reasons unbeknownst to us, that never happened and the FenixEdu team ended up developing the Certifier module, since it was deemed necessary for the practical implementation of SmartSigner, for the reasons stated above.

3.7 The new workflow of the registration process

Upon the creation of the student in the FenixEdu Academic system, a registration certificate in the form of a PDF file is automatically generated and added to a SmartSigner queue. This enables academic office staff, allocated to that queue (meaning they have sign permission), to sign the document using the Client Application. The document is then transmitted to the Server and validated by it, and then sent back to FenixEdu Academic system again that routes the signed document to the Certifier, the module explained in Section 3.6. The Certifier validates if the signature is legitimate, and stores it for later retrieval by entities requesting proof of validity.

After all steps are successfully executed, the signed document is stored in the student’s document repository, enabling him to access it whenever needed. Previously, additional certificates would be requested in FenixEdu Academic, and when ready (days later) the student would receive a notification to pick them up in the Academic Office. Now, the student automatically receives the signed version in his e-mail inbox, and can also access it on his document repository at any time without needing to request it formally. Figure 3.7 shows the signing and certification process after the registration process is complete.

There are various and clear benefits to the new process. First, at no point is printing required, and the 8 registration certificates that needed to be printed in the previous, physical, workflow could be eliminated. There is no need for physical transportation of documents, which was cumbersome.
Figure 3.6: Status documents can represent and possible transitions

for IST staff. The document signature can be made in bulk (more than one at a time), and is much faster than hand-signatures and requires no physical presence of the staff of the academic office during the registration process, since the SmartSigner application can be run on any location with Internet access, on any system that can run Java applications. Additional copies of, for example, the registration certificate do not require additional requests, since a copied digitally signed document maintains its authenticity properties between copies, which does not happen in paper documents.
Figure 3.7: Signing and certification process integrated in the SmartSigner system
New Developments: Corrections and Improvements

Contents

4.1 Bugs and errors identification ........................................... 39
4.2 Improvements to the SmartSigner Client Application .................. 40
4.3 Back-office ............................................................. 47
4.4 Performance issue with sign request upload .......................... 51
4.5 Java Web Start ........................................................ 51
4.6 Script for migrating between different database versions .......... 52
4.7 Server Deployment Process .............................................. 53
4.8 Multiple Signature Support ............................................ 54
4.9 Audit log .................................................................. 55
4.10 Expired signatures investigation ....................................... 56
4.11 AMQP with onion protocol ............................................. 56
4.1 Bugs and errors identification

The original SmartSigner system had some bugs and errors that had to be resolved. To be able to do that, a developer has to identify them first. This is done in mostly two ways: querying the users of the system for any functional complaints they may have and studying and testing the system locally. FenixEdu has its own implementation of a Request Tracker (RT), a ticket-tracking system used to coordinate tasks and manage requests among a community of users. This tool allowed for identification of the following bugs and/or errors:

- On launch, the Client Application is rendered with a fixed size. In older systems, or in systems with low resolution, the interface goes off-screen and there is no practical way to resize it since the mouse does not reach the edges;

- When a sign request is created for a document that already existed on the system, and that may even have already been signed, the system should recognize that situation. This can happen during the registration process, when the signed document for some reason does not reach the Certifier (Internet failure). This is not as much of a bug as it is a system limitation, but should be resolved nonetheless.

Studying the code and testing the system locally allowed for further detection of undesirable behavior such as:

- If the smart card is removed from the reader during the life-cycle of the Client Application, the application seemingly continued to work, but failed later, only when trying to sign a document;

- When an error occurs during the libraries loading process on the Client Application, due to incompatible OS version or some other reason, there is no feedback to the user, and the application simply fails to startup. The resulting exception must be caught and a modal (pop-up) with the error should be presented to the user;

- If the signed document was valid, and was signed by someone with sign permission on that given queue, but it was different from the original document, the Server would still accept the signature, making it possible for an attacker aware of this fact, and with access to a computer with the Client Application, to make the Server accept a document that was not on the sign request originally by switching the document that would be otherwise signed with the Client Application;

- On the Client Application, the queues that were shown to the user were all queues that the user had access to. However, since the user could have more than one card (token holder), the queues listed should only be those where the inserted card could be used to effectively sign documents in such queues;
• If a user had two cards of the same type, Server would only deem as valid documents that were signed with the first smart card that was added to the Server;

• When signature failed for any reason, an error modal would be shown, instead of updating the document icon on list of sign requests selected and that served that very same purpose;

• The number of sign requests shown should depend on profile. Since the signer can only batch sign the requests shown in a given page, some users could need more of them to appear to expedite the signing process, e.g., in the registration process (imagine having to sign thousands of documents, 50 at a time: the extra user input needed could very well be circumvented);

• The creation of token holders sometimes fails;

• The upload of documents to create sign requests seems to scale with the uptime of the Server, which should not happen, evidenced by gradually increasing of upload sign request operation time via FenixEdu Academic; this operation should ideally be \( O(1) \).

### 4.2 Improvements to the SmartSigner Client Application

#### 4.2.1 Refactoring

After analyzing the code of the SmartSigner Client Application and the Server, several potential improvements were identified. The UML class diagram presented in Figure 3.2 shows one in particular: there is a relationship between a queue and a token certificate, associated with a token holder (smart card), to exhibit a sign permission. That poses a problem: if one card of a particular type is replaced with another, either because the old one expired or it was lost, all the sign permissions become useless, since access the token certificate that was contained within the lost card, and that was associated with the sign permission, would no longer be possible.

To avoid the above issue, a sign permission must become a relationship between a user, a queue, a token holder type (smart card type) and a token type (certificate type). The token certificate should not have to be specified, since most smart cards only contain one key pair for digital qualified signatures, including CEGER card, but there is no guarantee that future smart cards will uphold the same characteristic; keeping it this way is more versatile and future-proof.

After entering the Client Application, a user would be presented with all queues in which a permission to sign was granted. Following the sign permission modification, the user should only see the queues in which, for that user, the card used to log in has sign permission associated. This was achieved by introducing a new domain object into the system: the SignPermission, which is a 4-tuple containing a User, a Queue, a TokenHolderType and a TokenCertificateType, and is represented in Figure 40.
4.2.2 PIN-less certificate extraction

A nuisance of the Client Application is that the extraction of public certificates from a card requires the introduction of the smart card PIN. Since these certificates are, by definition, public, as opposed to protected information, they should be accessed without requiring the owner’s secret (in this case, in the form of a PIN). Upon further analysis of this issue, it was noted that the certificates are being extracted with a `java.security.KeyStore`, which is initialized with a `PasswordProtection` that receives the PIN as a character array argument. The typical way to retrieve public certificates through a KeyStore is by using `null` as the PIN, or an empty string converted to a character array; this method proved unsuccessful for the card in question and with the libraries provided, making the KeyStore an invalid candidate for a solution.

Generally, a solution in which the developer can have fine control over what is happening is preferred, and to have such control one needs to go towards a level of less abstraction. The libraries used allowed the developer to treat the smart card as a KeyStore, but, as declared in Section 2.4, the smart card is also compatible with the PKCS#11 protocol, allowing for the much sought-after control.

To communicate with a smart card via PKCS#11 methods, one needs to open a session in the smart card. Since we only want to retrieve the public certificates, the insertion of a PIN should not be required, since no write operations will be performed, and thus a public read-only session is the valid option. `CK_OpenSession()`, a PKCS#11 method, requires as arguments the slot of the terminal, the flags, and the type of session and returns a number that identifies the session opened in the smart card. The slot is already obtained in the application after the user selects the terminal and card type; the flag can be `CK_SERIAL_SESSION` since we can retrieve the certificates sequentially and the session is a `CK_RO_PUBLIC_SESSION` for the reasons previously mentioned.

To retrieve objects from the smart card, the method used is `C_FindObjects()`, but first a search must be initialized in the card using `C_FindObjectsInit()`, which receives the session number as an argument, and a template acting as search parameters, which is a `CK_ATTRIBUTE` array. `CK_ATTRIBUTE`
has three fields: the first is the type of object, necessary so the translation can happen between C structs (the libraries are native code, in this case C) and Java objects (Java uses Java Native Interface (JNI), a programming framework to call methods on said library using reflection); the second is the type of data in which constants defined by the PKCS#11 are used, and the final is a pointer containing the actual object we want to send or retrieve. The search template will contain the type `CKA_TOKEN` as true, since the certificate is a token.

This procedure would retrieve all certificates, including the certificate chain, which is not wanted for in this scenario: the Server only stores the actual certificated. To circumvent this problem, a search should be made for private keys first, by specifying the `CKA_CLASS` as `CKO_PRIVATE_KEY`. Knowing that the private key object and public key object have the same `CKA_ID`, we can use this search to retrieve the IDs and then do another search for the `CKO_CERTIFICATES` for each of the IDs obtained. The certificate itself is retrieved as a byte array in the `CKA_VALUE` of `CK_ATTRIBUTE`, and by using the Java's Certificate Factory the translation to an X.509 certificate object is trivial, along with the `CKA_LABEL` to properly distinguish between types of certificates (authentication, signing, and cipher). In case of the CC, the provided library PTEID is used and has a special method for that purpose that does not require a PIN.

### 4.2.3 Detect card removal

Previously, in the SmartSigner Client Application, the removal of the smart card from the card reader during the application life-cycle would not be detected. In the event of such, the main menu, containing the list of accessible queues and the sign requests of the selected queue, would still be available, and the Client Application would only fail when attempting to sign a request. It actually would not fail if, for example, a different queue was selected on the Client Application triggering an API invocation to the Server, despite the authentication method being done using the smart card due to HTTPS protocol agreeing on a key for communication instead of asking for a challenge to be authenticated by the card every communication trip between the Server and the Client. This flaw violates the fail-fast principle: the process approach to fault isolation advocates that the process software module be fail-fast, meaning it should either function correctly or it should detect the fault, signal failure and stop operating. Processes are made fail-fast by defensive programming. They check all their inputs, intermediate results, outputs and data structures as a matter of course. If any error is detected, they signal a failure and stop [37, p 10].

By detecting the removal of the smart card as it happens, one can prevent any faults by returning to the login screen and wait for the insertion of the card again, thereby turning an error into a feature and avoid an application stoppage altogether.

This was achieved by running, on the background, a simple thread that initially saves the smart card insertion state, a boolean representing if the card is correctly inserted in the reader, and comparing the current state every half a second, which is fast enough to detect removal early but not so slow as to
disrupt the interaction with the user. This thread will live throughout the Client Application’s life-cycle, and if the current state is different from the saved one one of two actions is triggered: if the card is inserted the Client Application jumps to the login interface; otherwise, an interface stating the message “please, insert the correct smart card in the specified card reader” is presented to the user., as shown in Figure 4.2.

![Figure 4.2: Screenshot of Client Application after smart card is removed](image)

### 4.2.4 Sign requests sorting per column

Although most of the time the user may want to see the list of sign requests ordered by creation or modification date, depending on the status, the functionality of ordering these requests by other fields is important for easier sorting and browsing. Hence, every column header should serve as a button to set the sorting parameters. A click on a column header that was already selected switches the direction (ascending or descending) of the sorting.

Since the application uses Spring Data JPA, each database query is defined as easily as creating a method with a properly formatted name\(^1\). Previously, the method to retrieve the sign requests from a given queue with a specified status was `findByQueueAndCreatorAndStatusInOrderByModifiedDescCreatedDesc()`. Since the method name does not change after compilation, sorting by another fields was not possible. However, these methods can include a `Pageable`\(^2\), that can include the following parameters: start (the offset of the results), page size (the number of results) and sort objects, each containing the sort direction and the respective field. The method is replaced by one using a more generic approach, with the

---


\(^{2}\) [https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html)
signature findByQueueAndStatusIn(Pageable page, SignRequest.Status status), allowing sorting by different criteria.

4.2.5 Sign request refusal

It was not always the case that a document contained within a sign request would end up being signed, and for a variety of reasons: the sign request could be in the wrong queue, the document itself could contain false or inaccurate data, or the user that would otherwise sign it could simply refuse doing so for any another reason. Therefore, the concept of “refusal” for a sign request was implicit, and in such event a system administrator would be contacted, and would then manually access the MySQL database and change the status of the sign request with the following statement:

```sql
UPDATE sign_request SET status = 1 WHERE sign_request.id = [id];
```

This would obviously take much less time if that functionality was explicit, would involve fewer personnel and expedite the process. The sign request can have three states:

- **PENDING** – when the sign request is already created and associated with a queue and has a document associated with it that can be signed;

- **SIGNED** – the document contained is signed and its signature was validated by the Server. A sign request in this state also contains the original unsigned version of the document;

- **REFUSED** – the sign request was refused. A sign request in this state has a (descriptive) reason as to why there was a refusal (e.g. the document was in the wrong queue). Requests in this state can always be consulted, including download of the document associated.

Only sign requests in the **PENDING** state can be refused. This change was integrated in the Client Application in a rather simple way: a button next to the “sign” button, which only appears if at least one request is selected, opens a modal where the user can specify the reason for the refusal, as shown in Figure 4.3. After confirmation, an API endpoint that was created server-side for this purpose is called with the needed arguments (the request id and the reason of refusal), and by doing so the sign request changes state.

4.2.6 Customizable options on Client

As stated in section 4.1, some properties should depend on profile. There are two files that are used by the Client Application: the preferences file, and the application properties file. The purpose of each one is slightly different: the first exists to spare the user of selecting the terminal and card type every
time, and the application properties is a readable file that can easily be edited, and the natural choice to define the variables we want to customize, such as the page size.

Therefore, the page size will be read from this file on the client application initialization, and can be overwritten by the user.

### 4.2.7 Customizable Signature Appearance

In the Goals subsection 1.1, it was mentioned the signature appearance (the visual cue that appears on the signed PDF documents) is static and does not take into account the possible differences that may be required by some processes.

Originally, the SmartSigner Client Application created a document that showed a symbol expressing the state of the signature validation (Invalid, Unknown or Valid), a header stating who signed (or certified) the document and the respective email and the role, organization and date under that. This was achieved using the iText 2.1.7 library which uses different layers. However, the multiple-layer model for digital signature appearances in PDF was deprecated and is no longer part of the ISO standard for PDF ISO 32000 [21]. The reasoning is simple: by making a dynamic image present in layer 3 regarding state, it could mislead people that would trust the image presented as symbolizing the signature state: an attacker could send a document with an invalid signature but with the signature appearance containing the image of a valid signature, making it seem valid to the end user. Also, one of the main purposes of digital signatures is to prevent modifications to the document; having dynamic content is against this principle.
The iText library didn’t have support to generate signatures using only the layer 2. Therefore, some modifications to the class that stamped the signature appearance, `PdfSignatureAppearanceOSP`, was necessary. The method that created the signature appearance created an array of templates representing the layers, and this was slightly changed to include the logo, the text desired and a background, if so specified, in a single layer; these three components were customizable, and the new default signature appearance is represented in Figure 4.4.

![Default logo plus text Signature Appearance](image)

**Figure 4.4: Default logo plus text Signature Appearance**

The Client Application, instead of a simple Sign button now contains a drop-down list with the Signature Profiles, each one generating a different signature appearance. Those are: invisible, logo only, text only, logo plus text and personal signature image, as explained in Section 4.3.3.

### 4.2.8 Auto registering cards

Initially, it was conceived that it is a SmartSigner system Administrator’s responsibility of adding a smart card to a user, yet since the smart card is personal, and not transmissible, and the authentication method used by the FenixEdu Academic system, called Técnico Id, is being used by virtually all internal applications, the process of adding a new card can be improved in one of the following ways:

- The first time a user tries to log in, if the smart card is not recognized by the SmartSigner Server, but the authentication certificate is a valid CEGER issued certificate, then the user could log in to the Técnico Id platform and the smart card certificates could be automatically uploaded to the Server; or

- the FenixEdu platform, or the Back-office, explained in the section 4.3 could allow the user to add its smart card certificates to the Server. The certificates can be extracted using the Client Application.

The functionality that allows a user to associate cards by extracting them and then uploading them is implemented. However, there is a necessity to instruct the user on how to extract the certificates to posteriorly send to the Server via the back-office. If this solution is acceptable, after trying to unsuccessfully log in through the client Application a set of instructions will appear on how to add a smart card. Otherwise, then the Client Application should implement the functionality to automatically add an unregistered card certificates by logging in to FenixEdu Academic through a web-view.
4.2.9 Citizen Card Support

To add support for the CC, the new PTEID Standard Development Kit (SDK) must be used. Since the application uses a Strategy pattern, and there is an abstract class representing smart card objects, and there is already a class representing the CC, the prototypes in the super class should be defined, such as signing, establishing an SSL context or extracting certificates.

The method to extract certificates does not need to be overridden, since the one implemented in Section 4.2.2 worked as intended. However, a bug in the provided middleware for the CC was discovered.

To establish an SSL connection, the Client Application and the Server need to agree on which algorithms to use for authentication, encryption and key generation, which is formally known as cipher suite. This is done in the initial contact, the handshake, where the Client sends:

• The highest SSL protocol version that it supports;

• The list of cipher suites that it supports, in order of preference;

• Other information which is not relevant here.

The Server elects the most secure cipher suite it supports from the list of cipher suites supported by the Client Application. However, the middleware sends two unsupported algorithms, SHA384withRSA and SHA512withRSA. To circumvent this, a method was defined on the SmartCard object called getUnsupportedAlgorithm that returns the list of comma separated unsupported algorithms, which CitizenCard class implements. With that, we set the Java security property jdk.tls.disabledAlgorithms and prevent the Client Application from advertising a cipher suite that is not supported by the underlying hardware.

4.3 Back-office

Given that there is no practical to handle the back-end state of the SmartSigner system, any operation, such as adding a queue, a user or modifying a queue’s permissions must be done by a developer/system admin with access to the production database and using MySQL commands, which is impractical, inefficient and prone to errors. A back-office must be developed as a web-app that will communicate with the already existing SmartSigner Server. All system operations, such as managing users, queues, token holders, and permissions will be done via the back-office component. To enable these functionalities, the served API Restful interface should be extended in order to handle the new operations to the state, since the currently implemented REST API allows mostly for read operations, such as queues and requests, and always has the card being used in context, and this API should be completely independent from the back-office API. Therefore, the controllers should be implemented in
different controllers altogether, even if some functionalities (such as get user/me) are shared at the service level, since the authentication method is different for the Client Application (which is done resorting to a smart card) and the back-office, which will use a user-password combination.

The access to the back-office must obviously require some form of authentication, but cards should not be used as in the Client Application given that the system’s administrators may not even own a card; any form of secure authentication is sufficient. Since this module will be developed in IST, the login can be integrated with the already in-place system, which is a combination of user and password to access FenixEdu. In SmartSigner back-office, a simple combination of username and password will be used; a Single-Sign On (SSO) strategy using JasigCAS\(^3\), which is already used for the FenixEdu platform authentication.

The Back-office is represented in Figure 4.5

![Figure 4.5: Back-office interface](image)

4.3.1 Filter requests by text

In the SmartSigner system, some queues can contain thousands of sign requests, and even with the functionality to sort the requests by different fields, it is still really difficult to, for example, find a specific request. Therefore, the functionality of searching for a request by the title should be implemented,

\(^3\)https://wiki.jasig.org/display/CAS/Home
using a text box as input from the user. For the back-end, the Spring JPA helps us again: the keyword “containing” behaves in a similar way to the keyword “like” on MySQL, so only results containing the text introduced are returned. To match results ignoring letter case, the "ignoreCase" keyword is already implemented in Spring.

### 4.3.2 Pagination on API requests that may retrieve many results

Making the sign requests page-able is as simple as sending the offset and the page size as parameters when invoking the API, as described in Section 4.2.4. Resorting to some easy arithmetics, the page starts with 1 and the offset with 0, representing the first page. Changing to the next page requires adding the offset by page size, and subtracting retrieves the previous page. It is not necessary to check for impossible values (such as page number -1) because the buttons for previous and next page will be disabled if the page is either 1 or the value for the last page. In the event that the list of sign requests gets shorter after the API invocation, due to requests being signed or refused elsewhere, the Server will return the correct values for the last page and those will replace the state on the front-end.

### 4.3.3 User personal signature image

When signing a PDF document, it is possible to add a visual signature appearance. To further allow for customization, each user can set its personal signature appearance, it being an image of, for example, his handwritten signature. Both the Client Application and the Back-office interface will contain a button to upload the image file, which will be linked to the user. To apply it in a PDF document, this is done prior to signing with the Client Application, as explained in Section 4.2.7.

### 4.3.4 Server REST API

In the SmartSigner system, the users can have two roles:

- Regular users, that will upload, review, sign, refuse or download documents;
- Administrators that manage the system’s state, by adding users and token holders (cards and certificates), creating, removing and editing queues and managing permissions.

This is used for access control regarding the API.

Table 4.1 and 4.2 specify the list of endpoints to be accessed by users and administrators of the system.
Table 4.1: List of endpoints requiring role User

<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/queues</td>
<td>Returns Page of queues with permissions</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/{queue}</td>
<td>Returns queue with permissions</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/{queue}/requests</td>
<td>Returns the list of requests</td>
</tr>
<tr>
<td>POST</td>
<td>/queues/{queue}/requests</td>
<td>Creates a sign request on a queue</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/{queue}/download</td>
<td>Downloads the documents of the requests in arguments as a zip file</td>
</tr>
<tr>
<td>GET</td>
<td>/requests/{request}/download</td>
<td>Downloads the sign request document for signature</td>
</tr>
<tr>
<td>POST</td>
<td>/requests/{request}/refuse</td>
<td>Refuses the signature of a sign request</td>
</tr>
<tr>
<td>GET</td>
<td>/users/me</td>
<td>Returns the user currently authenticated</td>
</tr>
<tr>
<td>POST</td>
<td>/users/me/tokenholders</td>
<td>Adds a token holder to the user logged in</td>
</tr>
<tr>
<td>GET</td>
<td>/users/me/signature</td>
<td>Downloads the user personal signature image</td>
</tr>
<tr>
<td>POST</td>
<td>/users/me/signature</td>
<td>Uploads the user personal signature image</td>
</tr>
</tbody>
</table>

Table 4.2: List of endpoints requiring role Administrator

<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/queues</td>
<td>Creates a queue on the system</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/defaultparams</td>
<td>Returns the default queue parameters</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/enforcedparams</td>
<td>Returns the parameters that can’t be changed</td>
</tr>
<tr>
<td>PUT</td>
<td>/queues/{queue}</td>
<td>Edits variables of a queue</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/{queue}/info</td>
<td>Information and statistics about the queues</td>
</tr>
<tr>
<td>GET</td>
<td>/queues/{queue}/permissions</td>
<td>Permissions associated with a queue</td>
</tr>
<tr>
<td>POST</td>
<td>/queues/{queue}/permissions</td>
<td>Adds permission to queue</td>
</tr>
<tr>
<td>PUT</td>
<td>/queues/{queue}/permissions</td>
<td>Edits the permission users have on queue</td>
</tr>
<tr>
<td>POST</td>
<td>/queues/{queue}/signers</td>
<td>Adds a signer permission to queue</td>
</tr>
<tr>
<td>DELETE</td>
<td>/queues/{queue}/signers/signpermission</td>
<td>Removes user’s signing permission</td>
</tr>
<tr>
<td>GET</td>
<td>/users</td>
<td>Users resource</td>
</tr>
<tr>
<td>POST</td>
<td>/users</td>
<td>Adds user to system</td>
</tr>
<tr>
<td>GET</td>
<td>/users/{user}</td>
<td>Information of User</td>
</tr>
<tr>
<td>PUT</td>
<td>/users/{user}</td>
<td>Edits user fields</td>
</tr>
<tr>
<td>GET</td>
<td>/users/{user}/permissions</td>
<td>List of user’s permissions</td>
</tr>
<tr>
<td>GET</td>
<td>/users/{user}/tokenholders</td>
<td>List of user’s registered token holders</td>
</tr>
<tr>
<td>POST</td>
<td>/users/{user}/tokenholders</td>
<td>Add token holder to user</td>
</tr>
<tr>
<td>GET</td>
<td>/tokenholders/{tokenholder}</td>
<td>Returns token holder detailed info</td>
</tr>
<tr>
<td>DELETE</td>
<td>/tokenholders/{tokenholder}/signers</td>
<td>Removes token holder from user</td>
</tr>
<tr>
<td>PATCH</td>
<td>/tokenholders/{tokenholder}</td>
<td>Changes the status of a token holder</td>
</tr>
<tr>
<td>GET</td>
<td>/tokenholdertypes</td>
<td>Returns list of supported token holder types</td>
</tr>
<tr>
<td>GET</td>
<td>/tokentypes</td>
<td>Returns list of supported certificate types</td>
</tr>
</tbody>
</table>

4.3.5 DOM changes - live vs explicit

The Back-Office has possibly two different interactions regarding editing resources. 

In the Manage Users interface, any change made is immediately reflected on the Server. There is a watcher that compares the Server state with the one the user can edit on the front-end, and any change detected results in the modifications being sent to the Server in real-time. If any error occurs, a message is presented to the user and the state is reversed.

In the Manage Queues interface, the user edits the resources on the interface and, when finished, clicks on a button to explicitly persist the modifications on the Server.

There should exist one workflow to avoid unnecessary confusion to the users. However, since there was no decision on what workflow to implement due to divergence of opinions between FenixEdu devel-
opers, it was left for the tests to determine which one to use.

### 4.4 Performance issue with sign request upload

As referred in 4.1, there was a problem with the upload of sign requests: the time it took to receive a response from the Server after the API invocation with a document seemed to increase with Server uptime. This was observed by a developer who empirically noticed that, in 2017, the batch that uploaded all student certificates of enrollment took significantly longer than in 2016.

By carefully analyzing the code, the bug was discovered. There was a circular dependency caused by the service that created the new sign request.

The `SignRequest` object has a field representing its relationship with the `SignQueue`, and the `SignQueue` has a list of sign requests. The API endpoint to create a sign request would instantiate it but then also add it to the list of requests of the queue in question. The catch here is that the queue has a cascade save on the list of requests, meaning that any modification to the queue to be committed would also trigger a cascade save to every sign request on that queue. For that, the objects would be locked in the database. However, the sign request object also has a cascade type save in the queue direction.

The Hibernate, an Object-Relational Mapping (ORM) Java EE framework intended to translate between relational databases and objects in an OO development environment [38] that manages the objects in the Spring session. What happened was, when attempting to save the sign request, the cascade annotation would propagate the transaction to the queue but, since the queue in session was also modified (it had the newly created sign request), and the cascade save on the queue side would propagate to all sign requests, including the newly created, creating a vicious cycle until a timeout was reached and the transaction could finally be committed.

The solution relied on removing the line of code that added the sign request to the queue explicitly. The proper way to save the request is saving it directly, and rely on Hibernate cascade save to add it to the queue.

### 4.5 Java Web Start

Regarding the deployment process of the Client Application, this consisted of copying and distributing the executable Jar file to all systems that may use it. This distribution process, or lack thereof, really hinders future updates, and therefore should be changed. Since the Client Application was developed in Java, a possibly viable solution is to utilize Java Web Start, a framework developed by Sun Microsystems that enables users to start a Java application directly from the Internet, using a web browser or a Java Network Launch Protocol (JNLP) file, which is a file specifying where the application is being served.
To implement this, a web server hosted in a domain owned by IST is used to serve the Java Client Application, which must be signed, and this way the update process consists in only changing the served Jar file and all users starting the application would download the new version instead, as the protocol specifies.

This process works in the following way: the Server will also contain a signed JNLP file specifying the name of the Client Application Jar, the URL of the website, and some other properties. A button on the back-office will trigger the protocol, downloading the version of the Client Application in the Server, checking the signature of the Jar file, and running it automatically if all certificates are valid and security checks are successfully passed. In this manner, updating the Client Application just requires changing the version available on the Server.

### 4.6 Script for migrating between different database versions

The new class factorization will cause changes to the objects, and consequently the database representation. It was quickly found that the modifications made the new version incompatible with the old database. Since the old data must be maintained, a proper way of migrating the old data to the new version must be implemented. There are two mains ways to do this: either import the data running the new version connected to the database, and making the proper mapping of objects, or run the old version, which can map the database to objects, and somehow translate to the new objects, for example, in the form of MySQL INSERT commands.

To migrate the data from the old version to the new, the following strategy was employed:

1. Load a dump into a database;
2. Run the old version of the database with an added activity to generate the script consisting mainly of MySQL INSERT commands;
3. Run the aforementioned activity;
4. Deploy the new version of the SmartSigner Server and run it on an empty database;
5. Run the script in the database the Server connects to.

The script is generated by iterating through every table on the database and producing INSERT commands for each resource, generating a new ID for each because the ID changed from being a number to a string, and keeping a HashMap for mapping between new and old resources for the relations table, as the next example illustrates.
Listing 4.1: Data migration method example for Users

```java
private void migrateUsers() {
    Iterator<User> a = userRepository.findAll().iterator();
    while (a.hasNext()) {
        User u = a.next();
        String id = IdGenerator.generateSecureId(8);
        idHashMap.put(new Key("user", u.getId()), id);
        String command =
            MessageFormat.format("INSERT INTO user (id, name, username) VALUES
            ({0}', '{1}', '{2}');\n", id,
            u.getName(),
            u.getUsername());
        addToSql(command);

        for (String role : u.getRoles()) {
            command = MessageFormat.format("INSERT INTO user_roles (user, roles) VALUES
            ('{0}', '{1}');\n", id, role);
            addToSql(command);
        }

        logger.debug("COMMAND: " + command);
    }
}
```

4.7 Server Deployment Process

FenixEdu Academic is run on Linux-based operating systems. Therefore, it is only logical to create a shell script for the deployment of the new version on such systems. The script works in the following way:

1. The Client Application is compiled, generating a executable Java file;
2. The Client Application executable is signed and copied to the Server resources folder, to be made available for Java Web Start;
3. The Server is compiled and an executable is generated;
4. The Server executable is copied to the virtual machine where it runs;
5. If necessary, the script for data migration is run;
6. The running version of the Server is replaced with the new one. This way, both the Server and the Client Application are updated at the same time.

### 4.8 Multiple Signature Support

Some workflows may require documents with multiple signatures, but currently the application only supports one signature per document (and the sign request can only have three states, signed, refused or pending). Support for multiple signatures on a single document and validation should be included, and therefore the status of a sign request must be extended.

When a document is signed, a new version is created which cannot be modified without invalidating the signature. However, PDF documents have a specification to allow for this: the signature can be made to not cover the whole document, but only its contents, allowing for further modification of, for example, signature fields. PDF signatures can have 4 levels of certification:

- **Level 0** - simply sign. This signature creates another revision of the document and allows for posterior modifications, such as new signatures. This will not generate a certification signature;
- **Level 1** - No changes allowed - this profile locks the document in such a way that any modification made will render the signature invalid - that includes the addition of other signatures;
- **Level 2** - form filling and signatures - this profile is similar to the level 1 but allows for form filling and the addition of another signatures, them being within an already created field or creating a signature field *ad hoc*;
- **Level 3** - this profile is less restrictive than profile 2 because it also allows for page addition.

We now know that the type of signature generated by the Client Application must depend on the document supporting multiple signatures or not. In the same queue one could have sign requests for single signature documents or multiple signature ones, so the parameter must be defined at the request level, in the moment of sign request creation, or at the Client level, prior to generating the signature. Since the document received can have a specific formatting, and can also already be signed, doing so prior to the time of signature could lead to an incorrect result. The most viable option becomes defining that parameter when the sign request is created, which is either the back-office interface to upload documents to a given queue or in the API that is used to automatically add a sign request (and currently used by the FenixEdu Academic system).

The need to support level 3 was not identified as a requirement for three reasons: doing so would increase the complexity of the referred System, and no document was found that could possibly have another page added after it was signed. Ideally, documents that only need to be signed once should be
locked immediately after signature, hence in this case we would use certification level 1. For multiple signatures, since level 3 was already excluded, we could use either level 0 (and simply creating a new revision) or use level 2 (allowing for form filling). Signed documents can only have at most one signature with a certification level greater than 0, therefore in the Client Application we must check not only if the sign request allows for multiple signatures but also if there already is a signature present in it. By following the PDF standards, if a signed document has a certified signature it must be the first one. Combining all this information, we now reached at a solution which only requires a binary parameter in the sign request, and it works in the following manner:

1. If the sign request does not allow for multiple signatures, the certification level shall be 1.

2. If the document associated with the sign request has no signature the certification level shall be 2.

3. Otherwise, the certification level shall be 0.

The domain does change regarding the sign request object since a new boolean field representing if the document should enable multiple signatures, but compatibility with previous version is easy to maintain: since it is only one extra boolean field, and the system was previously not able to support multiple signatures, its value shall default to false, and migration is straightforward.

4.9 Audit log

One of the main concerns raised regarding the SmartSigner system, which handles sensitive information, was the lack of a mechanism to properly log the operations that change the database state. Envers\(^4\) is a tool that can be used to audit tables and is compatible with the Spring Framework, and has a concept of revision: one transaction is one revision. As the revisions are global, having a revision number, it is possible to query for various entities at that revision, retrieving a (partial) view of the database at that revision. The objects to be audited must be annotated with \(\text{@Audited}\), which will be scanned by the compiler; the database will have one extra table for each audited entity, one for the revision history and one for the objects changed per revision. The tool also stores the user who performed the operation, and timestamps every revision.

Although the integration with such an audit system hinders performance, because one write operation for updating a resource on the database becomes four writes (one for the resource in question, one for the revision table with the authenticated user and a timestamp, one for the table containing the resources changed by revision, and one for the table that stores the previous state of said resource), this was deemed an inevitable drawback, and the integration took place.

\(^4\)http://hibernate.org/orm/envers/
## 4.10 Expired signatures investigation

As mentioned in Section 2.3, after a certificate expires, the signatures issued with it also lose their validity, which does not happen with hand-signatures. Investigation about possible ways to circumvent this problem was conducted, in the attempt of avoiding expiration of otherwise valid signatures in digitally signed documents to become invalid.

As mentioned in Section 2.5, one of the added benefits of the PAdES is the Long Term Validation (LTV). This works by attaching a Document Security Store (DSS) to the signed PDF document, and adding another timestamp prior to the expiration date of the timestamp server. If the timestamp is valid, you can trust in the revocation evidences and check that the signing certificate and certification chain was valid and not revoked at the time that the document was signed. The global expiration time is limited by the timestamp certificate validity. When a timestamp is close to expire, another one is added, keeping the signature valid.

To integrate this change into the SmartSigner system, the new version of iText would be needed. However, the new version is protected under a proprietary license, making it not free, and the budget for it’s acquisition was not yet granted to the FenixEdu team.

## 4.11 AMQP with onion protocol

In the current day, the system relies on the HTTP protocol to handle communication between its components. However, this poses a potential problem: if the network is unavailable between any two modules, or extremely slow, then either the message is lost or it becomes the responsibility of each module to retry later (or any other fault-tolerant strategy). Furthermore, since all communication must be protected, modules may need to authenticate themselves to one another. The envisioned solution for this problem is to implement Advanced Message Queue Protocol (AMQP). This is a protocol in which a central module (henceforth called Message Server) handles the communication in the following way: if A needs to send a message to B, it will construct a formatted message containing all the relevant information plus the recipient, and send it to the Message Server. The Message Server will then redirect the message to the appropriate recipient and, in case of failure, it will follow the retrial policy predefined for the recipient module.

Utilizing this strategy provides various benefits:

- The necessity of authentication between modules no longer exists: to provide a secure internal network, only authentication to the Message Server is required. For example, let’s suppose the authentication between components is done resorting to symmetric key. If there are n modules, the addition of another one requires n new symmetric keys. With this scenario, only one extra key
is required (shared between the Message Server and the added module);

- If there is a module that uses replication, the other modules may have to be made aware of that. If that responsibility relies on the load-balancer, he may drop the previous requests. The server persists them, making sure to deliver at the correct node when available.

In its primitive version, there is a difficulty in replacing the SmartSigner sign request workflow with this system: the path is hardcoded in each module's code, as seen in (Figure containing the matriculation certificate). Since the concept of Message is abstract, the Message Server does not have the information of where to redirect the message next, and each module, when receiving a message, also may not know which workflow it originated from, and therefore can't redirect it properly if that is the case. There are two ways to circumvent this inconvenience: either add a tag that the Message Server reads and triggers the correct workflow, or making all information self-contained on the Message itself and each module reads where to redirect it after processed. The latter will have the name of Onion Protocol.

Following is an example of a Message in the Onion Protocol format.

**Listing 4.2: Example of a message from the Onion Protocol**

```json
{
  message: [message1],
  nextModule: [module2],
  payload:
    {
      message: [message2],
      nextModule: [module3],
      payload:
        {
          message: [message3]
        }
    }
}
```

When a module receives this Message, it processes the message contained by executing a method to be defined. After that, if there is a `nextModule` present, it sends the payload to it, replacing the message (in this case, `message2`) with the result of the execution of the method if process so specifies.

Although this feature was completely implemented, the integration of SmartSigner and the other systems with it did not take place for infrastructural reasons.
5

Case Study - DOT Integration

Contents

5.1 Purchase Order Document workflow ................................. 61
5.2 Integration - Requirements ............................................. 61
5.3 Integration - Implementation ......................................... 62
This chapter details a proof-of-concept integration of the SmartSigner with an existing workflow system that produces documents which must be signed. Before this integration, such documents had to be printed and hand-signed. The workflow that was chosen for this proof-of-concept was DOT, a system developed at IST that handles acquisition processes, purchase order documents and other financial matters. The integration was performed on the Acquisition workflow.

5.1 Purchase Order Document workflow

The workflow chosen to integrate into SmartSigner was the Acquisition workflow, because in its duration a document is produced, printed, signed, returned, scanned and then sent to the supplier via email. This makes for a perfect candidate, since some of this steps can be entirely eliminated by the integration.

In order to integrate SmartSigner with any other system’s workflow, first one needs to understand the workflow itself. Processes on DOT, as in all workflow systems, move through a sequence of states. In the workflow of interest, when a regular acquisition reaches a certain state, a document is produced that needs be signed. Thich is the state 6 of a regular acquisition process in the DOT system. We proceed to describe the relevant steps of the workflow.

When a regular acquisition process reaches the state 6, the user belonging to the purchase team staff generates a PDF document that becomes available on the user’s interface and that must be printed and signed. After this, another activity is made available in the system, for that particular process, that enables the user to confirm that the purchase order was sent to the supplier. After receiving the physically signed document back, the member of the purchase team staff proceeds to execute that activity and sends the signed document to the supplier, via email or fax, confirming the purchase order, in whichever order preferred.

One disadvantage of this old workflow is that the signed document is not attached to the process, unless the user manually scans it and adds it. Even then, the digital version is not considered a signed document for any intents and purposes, as discussed in Section 2.6.

5.2 Integration - Requirements

The integration of the purchase workflow with the SmartSigner system must be as seamless as possible, which means keeping the workflow interfaces as identical as possible; every step that can be automated should be. The following list describes the requirements of the process integration with SmartSigner:
1. The activity that creates the purchase order document will automatically upload it to the SmartSigner Server, creating a sign request on the corresponding queue;

2. When the document is signed, it should be automatically uploaded from the SmartSigner into the DOT system;

3. The activity that enables the user to confirm that the purchase order was sent to the supplier must become available only after the document is signed and uploaded into the DOT system;

4. If the sign request is refused in SmartSigner, the DOT system should log that event and erase the original purchase order document;

5. Since the purchase order document is now a purely electronic document, which should be sent by email to the supplier, it makes sense to automate the process. Therefore, after the purchase order is received on Dot to an interface to send the email to the supplier, with a predefined template, and with the signed purchase order document automatically included will be made available;

6. Authentication between the DOT system and the SmartSigner server must be done with a JSON Web Token (JWT) token and a shared symmetric key.

### 5.3 Integration - Implementation

The DOT system is based on Java, and it is structured into modules. The main module includes the others via Maven. The module that handles the activities mentioned in Section 5.1 is the expenditures module.

In order to address the requirements 2, 3 and 5, a field reflecting the signing status of purchase order object. To simplify further developments and add semantic meaning, that field should be an Enum set, with the following possible values:

- **CREATED** - The document was only created;
- **PENDING** - the document was successfully uploaded to SmartSigner Server and is pending signature;
- **SIGNED** - The document was signed and received;
- **REFUSED** - The document was refused.

This is the only domain change to the system.

Additionally, some properties need to be defined. The DOT system has an application properties file, where the following properties should be added:
• `smartsigner.enable` - A boolean to enable or disable SmartSigner integration;

• `smartsigner.queue` - The id of the queue where the purchase order documents will be upload to;

• `smartsigner.jwt.secret` - The symmetric key used to sign the JWT token;

• `smartsigner.signatureField` - The signature field name on the document to be signed;

• `smartsigner.url` - The URL of the SmartSigner server;

• `smartsigner.addRequestEndpoint` - The path to be concatenated to the URL for creating sign requests.

The DOT system uses an automatic logger to log activities. Therefore, the new events to be logged should be coded as activities, which are the “add signed purchase order” and “refuse purchase order”.

The bulk of the work is to change the mentioned activities behavior. For the activity that creates a purchase order, which is formally designated by `CreateAcquisitionPurchaseOrderDocument`, after the purchase order document is created, an additional method will be called that will perform the following operations:

1. Create and sign a JWT token containing the username of the user that is logged in;

2. Generate a nonce, which is another JWT containing the document id and sign it;

3. Invoke the SmartSigner Server API with the following arguments: the queue, the filename of the document, the title of the sign request, a flag for allowing or not multiple signatures, a description, the signature field name, the document file, and the callback to be invoked when the document changes status (signing or refusal) in the SmartSigner. The callback will be concatenated with the nonce, e.g., `dot.tecnico.ulisboa.pt/mission/[id]/sign?nonce=[nonce]`;

Then, the SmartSigner Server API endpoint for sign request upload must be called, and after receiving confirmation the purchase order document changes its status to `PENDING`.

For the SmartSigner Server to interact with DOT system, the latter must define an API to be called when the document is signed or refused. That endpoint will receive, as arguments, the process which it belongs to, the file (if it is signed), the reason (if it was refused) and the nonce as a path variable. Since the SmartSigner Server invokes the callback with the default arguments, i.e., the file if the sign request document was signed or the reason if it was refused, the purpose of this last argument is to be an argument representing the process id for the DOT system. If this endpoint is called with a file, then the `AddSignedPurchaseOrderActivity` is triggered, otherwise the `RefuseSignedPurchaseOrderActivity` is triggered instead.
For the requirement 3 described in Section 5.2, the method that returns if the SendPurchaseOrderToSupplierActivity is enabled needs to meet one more condition to return true: the purchase order document must have a SIGNED status.

For the requirement 5 to be implemented, the activity that confirms that the purchase order was sent, instead of only moving to state 7, now redirects to the send email interface, that the DOT system already has, and populates the email fields, such as the supplier email, the body and subject (which uses a predefined template but can be changed) and adds the signed purchase order document as an attachment to spare extra work to the user.

The following diagram in Figure 5.1 depicts process derived from the integration between the purchase order workflow and the SmartSigner system.

**Figure 5.1:** Purchase Order Document workflow integrated with SmartSigner
6

Evaluation

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Load Tests</td>
<td>67</td>
</tr>
<tr>
<td>6.2 User tests</td>
<td>68</td>
</tr>
</tbody>
</table>
User tests must be performed in order to identify difficult or non-intuitive workflows or interfaces in the existing SmartSigner Client Application and Back-office. Also, the dilemma raised in Section 4.3.5 should be resolved by comparing the metrics obtained from tasks done in both interfaces, namely correctness of the task and user explicit preference.

The tests are described in this Chapter 6.

6.1 Load Tests

As mentioned in Section 4.4, there was a problem with the sign request upload. Two tests were conducted to verify the issue and prove that it was resolved. The first created 9000 documents through a POST invocation to the API endpoint /queue/queue/requests, with the Server and the database running on the same PC, containing a Hard Disk Drive (HDD). The second test was uploading the same number of requests, but the second time on the Server hosted on the IST network on a machine with an Solid State Drive (SSD).

Prior to fixing the issue, the results were:

- **Test A on local machine with a HDD**: Adding a request lasted between 14 to 18 seconds for the first 40 requests. The test was not finished for the 9000 requests since the problem was self-evident, and the estimated completion time would be 2200 minutes.

- **Test B on network machine with an SSD**: Adding a request lasted between 1.21 and 1.34 seconds for the first 40 requests. The 9000 requests took 231 minutes.

After the fix, the results were:

- **Test A on local machine with a HDD**: Adding a request lasted between 42 and 67 milliseconds for the first 40 requests. The 9000 requests took 9 minutes.

- **Test B on network machine with an SSD**: Adding a request lasted between 93 ms to 133 ms for the first 40 requests. The 9000 requests took 21 minutes.

By comparing the data, it becomes clear that there was a major improvement, represented by the stacked histogram in Figure 6.3. After the fix, the test executed in a similar time frame on the local machine with a HDD and on the network machine with an SSD, contrary to the tests executed before the fix, suggests that the bottleneck that was in the database commit operation was virtually eliminated.

The reduction in execution time was even more significant. On the local machine with a HDD there a reduction of almost 99% on the worst case scenario, and a reduction of 89% on the network machine with an SSD, proving irrefutably that the problem was, indeed, solved.
6.2 User tests

In order to properly create tests, one needs to define the profiles of the users, which are four different profiles as shown in the following list.

- **Administrator** - system administrator staff;
- **Operator** - users strongly used to administrative software; members of the management office;
- **Batch Signer** - users that handle bureaucracy and generally do not have many computer skills;
- **Occasional Signer** - faculty of various backgrounds.

The goals of the user tests, segregated by profiles, are as follow:

- **Administrator**
- Check if interfaces are easy to use and used correctly, such as add user, add card, manage permissions;
- Determine which interface is better for edit operations, either the one where the changes trigger a live update or the one where the user must explicitly save the state, as explained in Section 4.3.5.

• Operator
  - Check if user can add a document to a given queue without explaining any of the interfaces.

• Batch Signer
  - Check if selection of multiple documents is intuitive enough, the tester signs one by one or midway the signer realizes that more than one request can be selected;
  - Check if user can correctly verify if a sign request has been signed.

• Occasional Signer
  - Check if the user can sign a document with ease, since these users may use the application only twice per year;
  - Check if the user can refuse a sign request.

The tasks to be performed by the users will reflect these goals. After their completion, the users must fill a questionnaire to evaluate the ease of performing each defined task, and an additional question for the administrator to pick between the live interface or the one where the changes must be made explicit, as explained in Section 4.3.5.
6.2.1 Test tasks

For the Administrator profile, the tasks performed were the following:

1. Adicionar uma fila com o nome Orçamentos que permita a criação manual de pedidos por utilizadores. O resto dos parâmetros são irrelevantes para esta tarefa.

2. Adicionar um utilizador com o nome Ricardo e username ist12345.

3. Dar permissões de acesso ao utilizador Ricardo para que ele consiga inserir pedidos e ver todos os pedidos que estão na fila “Orçamentos” anteriormente criada.

4. O utilizador Sérgio perdeu o cartão e, por isso, é necessário inativar-lhe o cartão.

5. Na interface de Gerir Filas, adicionar permissão de leitura de 3 utilizadores existentes (ist1234, ist1111, ist4321) à fila “Pagamentos”;


7. Adicionar um cartão ao utilizador com o nome Ricardo (ist12345). Para tal, utilize o ficheiro zip de nome “cartao_cidadao_certificados.zip” que se encontra no desktop.

For half of the participants, the order of the 5th and 6th tasks was switched, to avoid bias towards the first interface, since in the questionnaire the user will choose which interface is preferred.

For the Operator profile, the tasks performed were the following:

1. Adicionar o documento “pagamentos-abril.pdf” que tem no desktop à fila “Pagamentos” com o nome “Relatório de pagamentos de Abril”.

2. Fazer download do documento “pagamentos-janeiro.pdf” que já foi assinado. Verificar a assinatura.


For the Occasional Signer profile, the tasks performed were the following:

1. Entrar na aplicação SmartSigner utilizando o cartão CEGER providenciado, com o pin 2798, escolhendo o servidor https://kirk.ist.ul.pt:8443.

2. Assinar o documento que contém a pauta da cadeira de Engenharia de Software na aplicação SmartSigner.

For the Batch Signer profile, the tasks performed were the following:

1. Entrar na aplicação SmartSigner utilizando o cartão CEGER providenciado, com o pin 2798, escolhendo o servidor https://kirk.ist.utl.pt:8443.


3. Recusar um documento qualquer na mesma fila dando como justificação “Tarefa teste”.

6.2.2 Results

The tests were conducted in a desktop computer with Windows installed. The database was reset between each test. The tests were recorded, and the final state of the database was saved as well. The test involved 14 participants: 7 administrators, 3 batch signers, 2 occasional signers and 2 operators.

To measure correctness, the database state was compared with the intended state. Errors were detected in only 3 tests: one administrator did not add a read permission on task 3 and two administrators did not correctly add the three access permissions in the queues interface on task 5. The first can be explained by confusion, since the question is made in such a way that the user had to infer that “seeing” requests in a given queue meant a read permission and “adding” requests to a queue represented a write permission. The other two forgot to press the “save” button after adding the permissions, making the live interface a seemingly better option.

Figure 6.4 shows the number of questions by score regarding the user questionnaire, and some conclusions can be extracted: overall, the user satisfaction was achieved. The only two scores below 4 out of 5 are regarding the task to be done in the explicit save interface. The final proof needed that the live interface is the better one is manifested through the users preference, as made evident in Figure 6.5.

![Figure 6.4: Questionnaire results, on a scale of 1 to 5, regarding ease of completion and intuitiveness of the tasks performed](image)
Figure 6.5: Test users explicit interface preference
Conclusions

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Conclusions</td>
<td>75</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>75</td>
</tr>
</tbody>
</table>
7.1 Conclusions

The deployment of the SmartSigner system and its integration with the FenixEdu platform was arguably a step forward for IST, made evident by the improvements the students registration workflow underwent. Albeit, such a system must be properly maintained and nurtured so the extent of the benefits keeps only growing. In this work, many corrections that were made were described and many functionalities were developed: the bugs were identified, the system domain went through an inspection and posteriorly corrected for the sign permissions, the Client Application does not require the PIN to extract public certificates anymore, the removal of the smart card during the Client Application life-cycle is not a fatal error, the sign requests are sortable and can be refused, the Client Application has more customizable options expressed in a properties file, the signature appearance of the signed PDF documents is now customizable as well, the user can register a new smart card without involvement from IT, the Citizen Card is completely supported, a back-office webapp was developed to manage the system state, the user can now add his personal signature to the documents, a major bug that was hurting the system performance was identified and corrected, the Java Web Start now allows for proper Client Application distribution, a script was created to ease the release of new SmartSigner Server versions, the system now allows for multiple signatures in one document, all operations to the database and documents downloads are logged and a protocol for message exchange, although not integrated yet into the system, was designed and implemented.

The integration of SmartSigner with another system, DOT, proved the utility of the SmartSigner system and the correct choice of its conceptual design. The results show some important improvements to the system, reveal the ease of use of the back-office interfaces and helped settle a disagreement arisen during the development regarding live modifications versus explicit save operation.

7.2 Future Work

Although there was some extensive work put into the developments described in this work, there is still much to be done. The SmartSigner System shall make full use of the PAdES specification to properly enable LTV signatures in PDF documents, the AMQP must be integrated with the deployed version of the SmartSigner system to prevent lost messages and enabling tracing of a message in a given workflow, the functionality allowing the user to register an owned smart card in the System using only the Client Application could be a “quality of life” improvement and integration with remote repositories is also to be developed, since the documents are currently being stored in the same machine where the Server runs.
Bibliography


