Dissertation: Advanced System for Qualified Digital Signatures for Documentes

Ricardo Almeida
ricardo.d.almeida@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2018

Abstract

Process workflows that produce signed documents require digital signatures to become paperless. One example of such workflows is the registration process in Instituto Superior Técnico, generating registration certificates that must be signed by IST staff. In an effort to dematerialize this process, 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 hand-written signatures. However, this solution developed in a short time frame was rather primitive and not without errors. 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, amongst others. 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

1. Introduction

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. 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. By employing digital signature mechanisms in official documents, Instituto Superior Técnico (IST) can not only expedite the signing process but also the validation process, and increase efficiency immensely.

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, identity its bugs and faults, expand its functionality, integrate another system's workflow that uses signatures and evaluate the work done. This document will mention concepts such as Portable Document Format (PDF) signatures, digital certificates, client-server paradigm and sofware development.

1.1. Goals

The SmartSigner system 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 ac-

count.

The SmartSigner system must have this bugs identified and the functionalities missing must be defined and implemented. Additionally, since there is no current practical way to manage the SmartSigner server state, a solution that does not have such strict requirements must be designed and implemented.

2. Existing System

2.1. Original System Requirements

The original goal 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 role of the holder;

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 systems, MacOS, Windows and Linux – however, any solution that requires the development of specific executables for each Operating System (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.

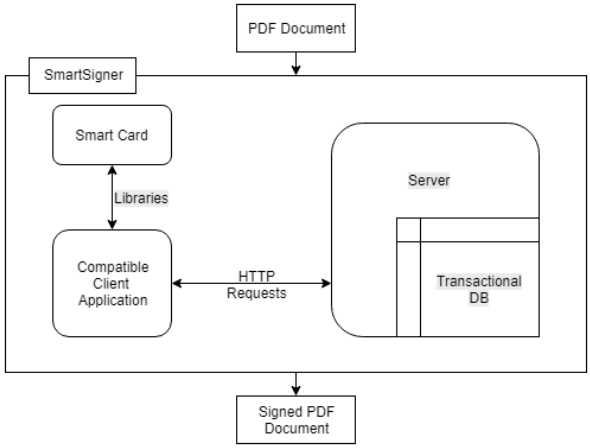

Figure 1: SmartSigner initial high-level architecture

The diagram in Figure 1 depicts a draft of a high-level architecture of the system addressing the previously mentioned requirements.

2.2. SmartSigner

SmartSigner was designed following an Object-Oriented (OO) paradigm, and as such is structured in various object classes.

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 Uniform Resource Locator (URL) which must be called after a document is signed (for the certifier) and the status of the request;

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

User – system users that authenticate using a smart card containing the private key of the corresponding 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 card (token holder) supported by the system; there is cerger card and also a reference to a citizen card;

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

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.

2.3. 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 card readers may need drivers to be used. It 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), to provide authenticity and confidentiality, as shown in Figure 2. 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 Java Virtual Machine (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 pin of the card as user's input to retrieve the certificates. Figure 2 shows the architecture of the Client Application.

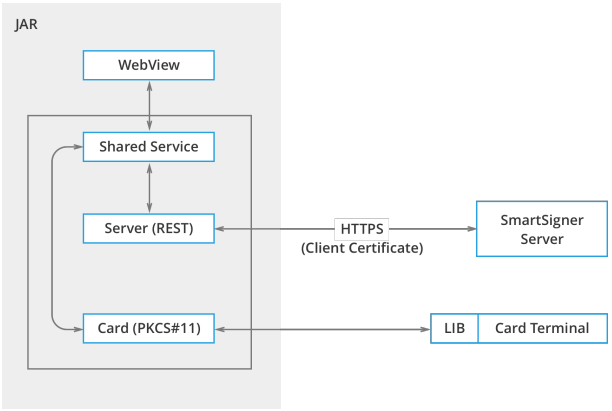

Figure 2: SmartSigner Client Application architecture

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 require not this setup.

2.4. 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 Representational State Transfer (REST) Application Programming Interface (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.

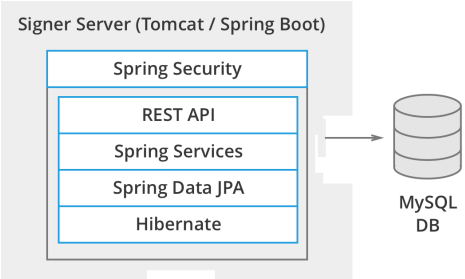

Figure 3: SmartSigner server architecture

2.5. Certifier

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

After a signing request is completed successfully, the digitally signed document is stored in FenixEdu Academic, using the URL configured in the system when the digital document was uploaded, 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 paper version was produced, which has an extra front page, which contains the complete URL of the original digital version, and a Quick Response Code (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. 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.

3. Implementation

3.1. Bugs and errors identification

The original SmartSigner system had some bugs and errors that had to be resolved. The identified erros are the following:

- On launch, the Client Application is rendered with a fixed size. In older systems, or in systems with low resolution, the interface goes offscreen 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.

- 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 $O(1)$.

3.2. Improvements on the SmartSigner Client Application

3.3. Refactoring

After learning the application and studying its code, several potential improvements were identified. The Unified Modeling Language (UML) class diagram shows one in particular: there is a relationship between a queue and a token certificate, associated with a card (token holder), to exhibit a sign permission. That poses a problem: if one card of a particular type is replaced with other, either because the old expired or it was lost, all the sign permissions become meaningless, since there will be no access to the token certificate. This was solved by introducing a new domain object to the system: the SignPermission, which is a 4-tuple containing a User, a Queue, a TokenHolderType and a TokenCertificateType.

3.4. PIN-less certificate extraction

One of the imperfections of the application is that the extraction of public certificates from a card requires the input of the cards PIN number. Since these certificates are, by definition, public, and not protected information, they should be accessed without requiring the owner's secret (in this case, in the form of a PIN). The method implemented to extract the certificates without requiring PIN did so using Public Key Cryptography Standards (PKCS)#11 instructions.

3.5. Detect card removal

Previously, after the user logged in, if the smart card was removed there was no change in the application state, which meant that if the smart card was to be removed, the main menu (containing the list of accessible queues and the sign requests of the selected queue) would still be available, and only fail when attempting to sign a request.

By detecting the removal of the 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.

The way this was achieved was to keep running on the background a simple thread that saves the inserted state (a boolean representing if the card is present) and compares the current state every half a second (which is fast enough to detect it early but not too long so as to disrupt the interaction with the user). This thread will live throughout the Client Application's life cycle. If the current state is different from the saved one, if the card is inserted jump to login interface; if not, jump to waiting for card interface.

3.6. 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 the requests by other fields is important. 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.

3.7. Sign request refusal

It was not always the case that a document contained within a sign request would end up signed in the end, and for a variety of reasons: the sign request was in the wrong queue, the document itself contained false data, or the user that would otherwise sign it would simply refuse for another reason. Therefore, the concept of "refusal" for a sign request was implicit, and when that happened 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:

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

This would obviously take much more time than if that functionality was explicit, and would involve fewer personnel and expedite the process. 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. After confirmation, an API endpoint that was created server-side for purpose is called with the needed arguments (the request id and the reason) and by doing so the sign request changes state.

3.8. Auto registering cards

Initially, it was thought the responsibility of adding the card to a user should be the administrators'. 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 back-office. If this solution is acceptable, when the card is not registered, after trying to log in through the client Application a set of instructions should appear on how to add a card. If not, then the Client Application should implement the functionality to automatically add an unregistered card certificates by

logging in to FenixEdu Academic through a webview.

3.9. Citizen Card Support

To add support for the Citizen Card (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 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 known as cipher suite. This is done in the initial contact, the handshake, where the Client sends:

- The highest SSLprotocol version that it supports;
- The list of cipher suites that it supports, in order of preference;
- other things which are not relevant here.

The Server elects the most secure cipher suite it supports from the the list of cipher suites supported by the Client Application. However, the middleware sends two unsupported algortihms, SHA384withRSA and SHA512withRSA. To circumvent this, a method was defined on the SmartCard object called `getUnsupportedAlgotirhms` 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.

3.10. Back-office

Given that there is no application to handle the back-end of this system, any operation such as adding a queue, user or modifying a queue's permissions must be done by a developer/system admin with 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. For that to happen, 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 files altogether, even if some functionalities (such as get user/me) are shared at the service level.

3.11. Performance issue with sign request upload

As referred in 3.1, 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 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 sign `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 create 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 [1] 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 the queue in session was also modified, since it had the newly created sign request, and the cascade save on the queue side would propagate to all sign requests, one of them being the newly created.

The solution was remove 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.

3.12. 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.

3.13. Server Deployment Process

The systems on FenixEdu Academic all run Linux based operating systems. Therefore, it is only logical to create a shell script for the deployment of the new version. 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, 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 runned;
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.

3.14. Multiple Signature Support

Some workflows may require docuements 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.

This requirement is a little more complex.

Typically speaking, a signature on a document creates a version that cannot be modified. However, PDF documents have a specification to allow for this: only the signed hash of the document contents excluding signature fields that are note filled are considered. PDF signatures can be one of 4 levels (certification level): 0 means simply sign. This signature creates another revision of the document and allows for posterior signatures; level 1 - No changes allowed - this profile locks the document in such a way that any alteration made to it will render the signature invalid - that includes another signatures; level 2 - form filling and signatures - this profile is similar to the level 1 but allows for form filling and 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 adding.

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

1. If the sign request does not allow for multiple signatures, the certification level shall be 1.
2. If the document present 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, but compatibility with previous version is easy to mantain: since it is only one extra boolean field, and the system was not able to do multiple signatures, it shall default to false, and migration is straightforward.

3.15. User tests

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. The results should reflect in modifications to the interface in order to improve usability and reduce errors.

4. DOT integration

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 finantial matters. The integration was performed on the Acquisition workflow.

4.1. 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 though 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 **??**.

5. 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.

5.1. 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 4.1 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 aditional 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.

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 (it 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 wasrefused, 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, the method that returns if the `RefuseSignedPurchaseOrderActivity` 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.

6. Results

6.1. Sign request upload

As mentioned in Section **??**, 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 1 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.
- **Test 2 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 1 on local machine with a HDD:** Adding a request lasted between 142 and 167 milliseconds for the first 40 requests. The 9000 requests took 9 minutes.
- **Test 2 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. The fact that, 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.

7. 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. However, given the importance of this system, everything should run as smooth as possible, and the system was not without its flaws and lack of functionalities. In this work, many corrections that were made were described and many functionalities were developed. 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, even if incomplete, show some important improvements to the system.

This work was important because it enabled the SmartSigner system to be integrated with other workflows that require signatures without changed to SmartSigner. With this in mind, the next steps are to integrate more and more workflows until the need for paper vanishes.

8. Acknowledgements

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, 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 o é mas que, em contraparida, 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.

References

[1] P. T. Fisher and B. D. Murphy. *Spring persistence with Hibernate*. Springer, 2010.