|  |  |
| --- | --- |
|  | BOSA signing solution  Technical documentation |

## Table of contents

[Table of contents 2](#_Toc128484102)

[Abbreviations and Acronyms 3](#_Toc128484103)

[Summary 6](#_Toc128484104)

[Introduction 6](#_Toc128484105)

[Deployment environments and main components 6](#_Toc128484106)

[Business overview of signing 8](#_Toc128484107)

[Testing (Mintest) 9](#_Toc128484108)

[Download Server 12](#_Toc128484109)

[The Sign-validation server 13](#_Toc128484110)

[Creation of Token and parameters 16](#_Toc128484111)

[Signature validation 18](#_Toc128484112)

[The BeID Connect software and browser extensions 19](#_Toc128484113)

[CI/CD process 33](#_Toc128484114)

[eSealing 39](#_Toc128484115)

[Future developments 40](#_Toc128484116)

## Abbreviations and Acronyms

|  |  |
| --- | --- |
| Acroform | Adobe Acrobat Forms that contain form fields |
| API | Application Programming Interface |
| BOSA | FOD Beleid en Ondersteuning/SPF Stratégie et Appui |
| BRCA | Belgium Root Certification Authority |
| CI/CD | Continuous Integration (CI) and Continuous Delivery (CD) |
| CLI | Command Line Interface |
| CRL & CRT | Certificate Revocation List |
| CSR | Certificate Signing Request |
| CVE | Common Vulnerabilities and Exposures |
| DSS | Digital Signature Service |
| Eid | electronic ID |
| eIDAS Regulation | Regulation on electronic identification and trust services |
| FPS | Federal Public Service |
| FTS | Federal Trust Services |
| INT | Integration (environment) |
| IdP | Identity Provider is a service that stores and verifies user identity |
| IP | Internet Protocol |
| JSON | JavaScript Object Notation |
| LTA | Long-Term Authentication |
| MS | Microsoft |
| NPM | Node Package Manager |
| OWASP | Open Web Application Security Project |
| PGP/GPG | Key Pretty Good Privacy (PGP)/ GnuPG is a security program used to decrypt, encrypt and authenticate |
| PKI | Public Key Infrastructure |
| PROD | Production (environment) |
| QA | Quality (environment) |
| RFC | Revocation Freshness Constraint |
| SAST | Static Application Security Testing |
| SPA | Single-Page Application |
| Stdin | standard input |
| Stdout | standard output |
| TA | Test and Acceptance (environment) |
| URL | Uniform Resource Locator |
| Xades | XML Advanced Electronic Signatures |
| XML | Extensible Markup Language |
| ZIP | File format used for data compression and archiving |

## Summary

## Introduction

The BOSA signature solution ‘Federal Trust Services’ (FTS) is a web application that allows for processing of files that are ready to be signed by creating a signing session and the output is downloaded in the e-government application.

## Deployment environments and main components

FTS can be deployed through accessing the Openshift containers (<https://paas.gcloud.belgium.be/console/catalog> ), where 4 different environments can be seen:

* TA used by developers to test code that is automatically deployed from commits to “developer” code branch.
* QA used by developers to test with clients’ applications and the code is automatically deployed from commits to “master” code branch.
* INT deployed manually from stable release based on a tag (pre-production).
* PROD deployed manually from stable release based on a tag.

The basic development model is a GIT CI/CD Docker-based automation. The main Docker containers for deployment are:

* Gui-sign: a front-end web application for signing of documents with eIDs. It interacts with the eID card through the browser extension and BeID Connect.
* Sign validation: the back-end application that validates the signatures and timestamps the files
* A browser extension, a mandatory technical bridge between the front-end and the BeID Connect
* BeID Connect, an “OS native” [[1]](#footnote-2) application that interfaces with the underlying OS services to access the smart cards
* Gui-idp
* IDP
* eSealing

Other important components are:

* Mintest: a front-end/back-end web application that allows for testing of the system.
* A download server, where the BeID Connect application and some browser extensions can be downloaded

Gitlab is one of the BOSA code repositories and it has multiple branches. The TA environment tracks the latest version of the “develop” branch, while the QA environment tracks the “master” branch.

## Diagram Description automatically generatedBusiness overview of signing

Explanation of the steps:

User with eID wants to sign a document

1. The browser calls the e-gov application to sign the documents

2. The e-gov application (FPS) creates a signing session that requests the creation of a Token from the Federal Trust Services

3. If the browser extension is not already installed, the browser is redirected to download the browser extension for the eID software from the BOSA download server

4. In the Federal Trust Services, the following steps are followed:

a. the user reviews the document about to be signed;

b. the user confirms that he/she read the document and then, the user inserts PIN code for eID to sign the document

5. The signed document returns to the user’s browser (optional step)

6. The browser is redirected to the e-gov application, where the signed document is downloaded

## Testing (Mintest)

#### Introduction to Mintest

Mintest has two functions:

1. As a test application for the “Front-end integration”

* It serves as a minimalistic FPS (e-gov) application
* It is where some sanity checks are made to avoid creating impossible requests (E.g. signing an XML with a visible signature etc) but it is also possible to create invalid requests that will be rejected by Sing-validation

1. As a CRL & CRT server for the unit tests. It is important to note that in order to run Sign-validation unit tests in local or TA, Mintest TA must be running.

#### Mintest as a Test application

It is divided in two parts:

1. A list of prepared use case for “single click” testing
2. A more general testcase editor to create custom requests

How does it work?

1. When clicked the “one click” links or the “submit” button Mintest front-end calls its back-end with a URL similar to the following: <http://localhost:8081/sign?json=@o'outDownload'@strue@S'previewDocuments'@sfalse@S'requestDocumentReadConfirm'@strue@S'outFilePath'@s'Xades_JUST.xml'@S'outXsltPath'@s'JUST.xslt'@S'signProfile'@s'MDOC_XADES_LTA'@S'signTimeout'@s9999@S'policy'@s@o'id'@s'http://signinfo.eda.just.fgov.be/SignaturePolicy/pdf/PrivateSeal/BE_Justice_Signature_Policy_PrivateSeal_Hum_v0.12_202204_Fr.pdf'@S'digestAlgorithm'@s'SHA512'@S'description'@s'null'@c@S'inputs'@s@O@o'filePath'@s'test.pdf'@S'xmlEltId'@s'ID1'@c@S@o'filePath'@s'annexeRefs.xml'@S'xmlEltId'@s'ID2'@c@S@o'filePath'@s'nd_test.pdf'@S'xmlEltId'@s'ID3'@c@C@c>
2. When the URL reaches Mintest back-end it will:
   1. Decode the URL
   2. Copy all the input files (here: test.pdf, annexeRefs.xml, nd\_test.pdf) from the URL from its resources to the S3 Configured bucket
   3. Convert all parameters to a proper sign-validation JSON request for the “getTokenForDocument(s)” request

In the example, the request is for:

* a Multi-Document Xades LTA signing
* a custom policy
* a read confirmation
* without document review
* with an output file named : Xades\_JUST.xml

1. Call getTokenForDocument(s) and extract the returned Token
2. Redirect the browser to Gui-Sign with the Token and the “redirect URL” of Mintest. When Gui-Sign has signed the documents, it will redirect the browser to the “redirect URL”
3. Control returns to Mintest through the “redirect URL”
4. It will then delete from S3 bucket:
   1. Input files
   2. Output file

#### Mintest as a CRL & CRT server for Unit tests

For unit testing, custom certificates are created (see below in Test PKI) and in order to cryptographically test them a CRL & CRT repository is needed.

To avoid creating a custom, separate host to store two files, those files are stored in the Mintest TA instance.

Example : <https://mintest.ta.fts.bosa.belgium.be/static/citizen202207.crl>

#### Test PKI

Testing also involves an easy-to-use Java implementation to create a test PKI.

It enables the following possibilities:

* Creation of self-signed Root Certificates
* Creation of chains of sub-Certificates
* Creation of Keystores, CRLs, CSRs

The Sign-validation unit tests depend on certificates created by this PKI, namely the CreateCitizenNonrep class, which will:

* create a Root Certificate
* create a sub-Certificate for Issuing the Citizen Certificate
* create a Citizen Certificate

For “Sign-validation”, it will:

* Create the Keystore with the Citizen Certificate and its chain up to the Root
  + The Keystore will be stored in the project test resources
* Replace the test “Root Trust” in the test keyring with the Root Certificate
  + The test “Root Trust” is located in the main resources

Regarding storage, for the “Mintest” project, it stores in the “static” folder:

* an empty “Issuing CRL”
* an empty “Citizen CRL”

#### Test keys

For test keys to be created, the routine contained in the following link is followed: [https://git-fsf.services.belgium.be/eidas/testpki](https://eur01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgit-fsf.services.belgium.be%2Feidas%2Ftestpki&data=05%7C01%7Cniovi.vlachopoulou%40soprasteria.com%7C9b3948bb35b446d3971908dab29f0688%7C8b87af7d86474dc78df45f69a2011bb5%7C0%7C0%7C638018693351546118%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=F%2FvmGEEnm%2BTEUt%2FoUPAqRQGkmlDdGTXgKjIBJt3d18I%3D&reserved=0)

The chain order consists of three categories of keys, “Root”, “Issuing” and “Leaf” keys.

The Root key “TestSign Belgium Root” is self-signed, it was used to create two “Issuing keys” :

* “Testsign CitizenCA” with an expiration in 2032
  + This key was used to create the following “Leaf keys”:
    - A Unit test key: “Christian TestLongNames”.
    - Two eSealing keys:
      * “Leaf Sealing Signer”
      * “SAD Leaf Sealing Signer”
    - A Timestamping key: “Timestamp Unit”
* “Expired Test CitizenCA” that expired in 2021.
  + This key was used to create a valid Leaf key:
    - “FredWith IssuingCACertExpired”, a unit test key

A diagram explaining the above chain of keys can be consulted here:

Diagram

Description automatically generated

## Download Server

**URL :** <https://eid.static.bosa.fgov.be/>

The purpose of this server is to download some of the following files:

|  |  |  |
| --- | --- | --- |
| File path | Remark | OS |
| /beidconnect\_2.6<lang>.msi | Where <lang> is one of “\_de”, “\_nl”, “\_fr” and ““ | Windows |
| /beidconnect\_x64\_2.6.msi | Where <lang> is one of “\_de”, “\_nl”, “\_fr” and ““ | Windows |
| /beidconnect\_2.6.dmg |  | MacOS |
| /beidconnect-archive-fedora.rpm |  | Fedora |
| /beidconnect-archive-el.rpm |  | CentOS |
| /beidconnect-archive.deb |  | Debian |
| /BeIDConnect\_Extension\_2.6.dmg | Safari Extension | Mac OS |
| /beidconnect-fx-0.0.11.xpi | Firefox Extension | Multi OS |

## The Sign-validation server

The BOSA sign-validation server includes the Pre-processor, the DSS Library and Rest Service API.

#### Pre-processor

It is based on the DSS WebApp[[2]](#footnote-3), but completely redesigned to be:

* In line with the needed functionalities required for operational behaviour
* Multi-use (not only for eID)
* Multi-client (ways to customise the WebApp towards client needs)
* User-friendly

#### DSS library

It is an open-source software library that contains the DSS classes as-is.

#### Rest Service API

Below the functionality of each endpoint is described:

* Validation Service API

|  |  |
| --- | --- |
| **API call** | **Description** |
| GET /validation/ping | Used to check service availability |
| POST /validation/validateCertificate | Simplified report for validation of certificate |
| POST /validation/validateCertificateFull | Full report for validation of certificate |
| POST /validation/validateCertificates | Simplified report for validation of multiple certificates |
| POST /validation/validateSignature | Simplified report for validation of signature |
| POST /validation/validateSignatureFull | Full report for validation of signature |

* SigningService API

The signing service endpoints are configured using profiles, that are mapped to the DSS parameters. See the following page for details: [3. Sign&Validation Application | 1. Profiles](https://fedictaim.atlassian.net/wiki/spaces/ETS/pages/1824162248/3.+Sign+Validation+Application#1.-Profiles)

|  |  |
| --- | --- |
| **API call** | **Description** |
| GET /signing/ping | Used to check service availability |
| POST /signing/getDataToSign | Returns the Digest (non-forgeable summary of the data) of the document to be signed for use by the signature service |
| POST /signing/getDataToSignMultiple | Same as above, except for multiple documents |
| POST /signing/signDocument | Adds the signature created by the signature service to the document to be signed |
| POST /signing/signDocumentMultiple | Same as above, except for multiple documents |
| POST /signing/extendDocument | Extends the existing signature on a document |
| POST /signing/extendDocumentMultiple | Same as above, except for multiple documents |
| POST /signing/timestampDocument | Adds a timestamp to a document |
| POST /signing/timestampDocumentMultiple | Same as above, except for multiple documents |

* Token API

The Token is passed to the FTS front-end and will be included in in every “Token” call to FTS back-end (Sign validation).

|  |  |
| --- | --- |
| **API call** | **Description** |
| POST /signing/getTokenForDocument | Create a token to sign a single document |
| POST /signing/getTokenForDocuments | Create a token to sign multiple documents |
| POST /signing/getDataToSignForToken | Returns the Digest of the document(s) to be signed for use by the signature service |
| POST /signing/getMetadataForToken | Returns the information about the document(s) to be signed |
| POST /signing/getFileForToken | Returns the file that will be signed. This allows the signer to view what will be signed |
| POST /signing/signDocumentForToken | Adds the signature created by the signature service to the document to be signed |

## Creation of Token and parameters

When the e-government application calls the FTS, it requests the creation of a Token and starts the signing session of the document. The Token functions as a request for authentication by the server and only after the server verifies the signature, it responds to the request.

Text

Description automatically generatedThe ‘getToken’ call contains parameters, that are the requirements from the client. The following image is a snip of the parameters used when testing the signing of a document:

The parameters are the following:

|  |  |  |
| --- | --- | --- |
| Name | ‘name’ | indicates the name of the bucket where the document is stored |
| Password | ‘pwd’ | indicates the password of the bucket |
| Input | ‘in’ | indicates the name of the file stored in the bucket |
| Output | ‘out’ | indicates the name of the new/output file stored in the bucket |
| Profile | ‘prof’ | It is one of the following signature types: XAdES, XaDes.mt, XaDes.mta, PAdES |
| Program xslt | ‘xslt’ | If the document is in an xml format, this program displays it in a more readable and user-friendly way |
| File psp | ‘psp’ | File containing the way the signature is rendered (name, first name, inclusion of picture) |
| Language | ‘lang’ | The language used to sign the document (coming from the e-government application) |
| Signing time | ‘signTimeout’ | The default time between the reviewing of document and the confirmation of signature shall not surpass the signature time indicated. This is by default 120 seconds |
| Download | ‘noDownload’ | If the signed document is downloaded by the user |
| Policy | ‘policyID’ | Inclusion of policies on signature |
| Policy | ‘policyDescription’ | Inclusion of description of policies on signature |
| Policy | ‘policyDigestAlgorithm’ | Inclusion of policies’ algorithm on signature used to secure the policies |
| Confirmation | ‘requestDocumentReadConfirm’ | Checkbox to confirm |

## Signature validation

The cryptographic validity of the signature depends on how secure the algorithms and parameters are when creating the signature and certificates. To ensure that, when validating a signature in a document, the validation is made against a set of rules, the constraints, that dictate how the solution behaves. The constraints are sometimes referred to as a signature validation policy.

Part of the constraints is the Revocation Freshness Constraint (RFC)[[3]](#footnote-4), which applies to certificates and CA certificates. The Revocation Freshness Constraints as defined in ETSI TS 119 172-1 [3][[4]](#footnote-5) shall be used with a maximum value of 0, ensuring that the revocation information is **only** accepted if it has been issued after the best signature time.

* For BRCA4, the RFC is 1 min for OCSP.
* For BRCA3, the RFC is 210 days for the CA certificates.

BRCA3 is still using SHA1. SHA1 is accepted as a signature algorithm for Root and Intermediate certificates, but it is forbidden for Leaf Signing certificates.

Countersignatures of existing signatures can also be validated following the same RFC. The same constraint applies to the validation of timestamps and the revocation itself. The revocation can be either valid or not valid.

In case the result of the validation process returns the error ‘‘Indeterminate - try later” for some of the signatures and for those signatures can be identified as being signed by a BRCA3 certificate, then the validation is re-run with exactly the same rules except that the RFC is changed to 210 days for the CA certificates and the counter-signing CA certificates. These constraints are found in two different .xml files in the policy folder of the project.

## The BeID Connect software and browser extensions

#### High level description

When the user uses the eID smart card for signing, the browser does not provide direct access to the smart card. A browser extension is needed to create the link between the Gui-sign web, a React SPA web application, and the Belgian eID smart card. In this way, the browser can communicate with the eID smart card with the use of the browser extension that utilises the exchange of messages (API) with a native-code host executable running outside of the browser sandbox.

***Signing with the BeID Connect software***

The eID smart card contains a signature certificate that enables the digital signing of documents. When launched, the signature application checks the validity of the certificates and loads the certificate of the trust service provider. The signature is deemed qualified[[5]](#footnote-6) when the certificate comes from trust service providers in accordance with the eIDAS Regulation. The European Commission publishes online a list of these trusted lists, the List of Trusted Lists (LOTL).

Depending on the browser type and the operating system, the details found in the following description may vary. The following links provide a detailed description on the development of the browser extension:

* [Welcome - Chrome Developers](https://developer.chrome.com/docs/extensions/mv3/) (Chrome and Edge)
* [Browser Extensions - Mozilla | MDN](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions) (Firefox)

This is the most updated list of supported browsers and operating systems:

* MS Edge, Google Chrome, and Mozilla Firefox are supported on Windows.
* Google Chrome, Mozilla Firefox, and Safari are supported on MacOS.
* Mozilla Firefox and Google Chrome are supported on a subset of the most used releases of Linux.

The following diagram depicts the flow of the interaction between the Gui-sign web application and the Belgian eID smart card:

Diagram, schematic

Description automatically generated

Explanation

1. The browser extension is loaded and injected in the web application/pages that match those referenced in its configuration file. The configuration file defines the following domains:

* \*.Belgium.be
* \*.fgov.be

Since the Gui-sign application is hosted inside the Belgium.be domain, the browser extension background is loaded and the browser extension page is injected in the Gui-sign application.

1. The injected page provides a class that is the API the Gui-sign will use. The API communicates with the browser extension background, which executes in a kind of sandbox, with messages. Essentially, each API function generates a message that is sent to the browser extension background and waits for a response message.

The browser extension background itself is just a pass-through to a native messaging host application named “BeID Connect”.

1. The BeID Connect application is a native application running as console mode application. The stream it will receive in its stdin is sent by the browser extension background, and BeID Connect will send back through its stdout the result/response to the browser extension background. In turn, this result will be sent back to the API function.
2. As the BeID Connect native messaging host application is running “at operating system level”, it can access all of its resources and use the smart card reader connected to this operating system.

#### Card operation Gui-sign requests

The API provides a route for the request done by Gui-sign to the Belgian eID smart card. Here is the list of the request:

* Version

Return the version of “BeID Connect” (for the Windows administration installation, it returns an extra parameter allowing to install mitigation)

Request :

{

"operation":"VERSION",

"correlationId":"defe3775-b560-401d-5f38-4c83b1cf20f2",

"src":"beidconnect.page",

"origin":"https://sign.int.fts.bosa.belgium.be","tab":377

}

Result :

{

"result":"OK",

"version":"2.7",

"correlationId":"defe3775-b560-401d-5f38-4c83b1cf20f2"

}

* ID

Return a list of detected Belgian eID smart cards.

For each Belgian eID smart card, the following are provided:

* The reader’s name containing the detected smart card
* The reader type (“pinpad” or “standard”)
* The card types

For each Belgian eID smart card, the following information may be requested:

* ID
* Address
* Photo
* idsig
* addrsig
* rrncert
* authcert
* signcert
* cacert
* rootcert

Sample request:

{

"operation":"ID",

"idflags":"77",

"language":"en",

"mac":"0123456789ABCDEF0123456789ABCDEF",

"correlationId":"c2f04f63-3ec9-d35d-e178-42684d0504b6",

"src":"beidconnect.page",

"origin":"https://sign.int.fts.bosa.belgium.be",

"tab":377

}

Result when no card is detected:

{

"Readers":"",

"result":"no\_card",

"correlationId":"c2f04f63-3ec9-d35d-e178-42684d0504b6"

}

Sample request:

{

"operation":"ID",

"idflags":"77",

"language":"en",

"mac":"0123456789ABCDEF0123456789ABCDEF",

"correlationId":"e06d93e6-d6dd-99ed-6f79-f4cbb4dc6870",

"src":"beidconnect.page",

"origin":"https://sign.int.fts.bosa.belgium.be","tab":377

}

Result when a smart card is detected:

{

"Readers":

[

{

"ReaderName":"VASCO DIGIPASS 870 0",

"ReaderType":"pinpad",

"cardType":"BEID",

"id":"AQwwMDAwMDEwMjA2……………………qv0A8SewjElW4CZ",

"photo":"\/9j\/4AAQSkZJRgABAgEBLAEsAAD…….YfyBwP0ArPjGZFHqcV0t…..”,

"idsig":"MGQCMFhhdX………s2blg9BzjLhP8",

"addrsig":"MGUCMQCR……husjrPgB2AwA==",

"rrncert":"MIICgzCCAgmgA…….x8zMzc8=",

"signcert":"MIIDazCCAv……jElW4CZ"

}

],

"result":"OK",

"correlationId":"e06d93e6-d6dd-99ed-6f79-f4cbb4dc6870"

}

* USERCERTS

Return a list of detected Belgian eID smart cards.

For each Belgian eID smart card, the following are provided:

* The reader’s name containing the detected smart card
* The reader type (“pinpad” or “standard”)
* The card types
* Certificates

This function is not used inside the current implementation.

* CERTCHAIN

Return a list of detected Belgian eID smart cards.

For each Belgian eID smart card, the following are provided:

* The reader’s name containing the detected smart card
* The reader type (“pinpad” or “standard”)
* The card types
* Certificate Chain
  + Root Certificate
  + Sub Certificate

Sample request:

{

"operation":"CERTCHAIN",

"language":"en",

"mac":"0123456789ABCDEF0123456789ABCDEF",

"cert":"MIIDazCCAvGg…………………+VZ/R3YwM1s31CqiWBpPUGGNjGBIOqqv0A8SewjElW4CZ",

"correlationId":"5bdc3607-8a97-d768-f69c-8774018681e5",

"src":"beidconnect.page",

"origin":"https://sign.int.fts.bosa.belgium.be",

"tab":377

}

Result:

{

"certificateChain":

{

"rootCA":"MIICNDCCAbug……………….AwIBAg7Na6l+U8r4M7VH49\/cw==",

"subCA":

[

"MIIDKTCCAq……………………O+QnM8cnPmg"

]

},

"cardType":"BEID",

"ReaderType":"pinpad",

"result":"OK",

"correlationId":"5bdc3607-8a97-d768-f69c-8774018681e5"

}

* SIGN

It allows crypto signature of a digest/algo for a certificate and it returns the signature.

Sample request:

{

"operation":"SIGN",

"cert":"MIIDazCC…………..IOqqv0A8SewjElW4CZ",

"algo":"SHA256",

"digest":"aKopJG……………hSOqBHV8/BE1zgkfrGTQ0Vk=",

"pin":null,

"language":"en",

"mac":"0123456789ABCDEF0123456789ABCDEF",

"correlationId":"37b9deb4-af00-04b3-a37e-9241329a0471",

"src":"beidconnect.page",

"origin":"https://sign.ta.fts.bosa.belgium.be",

"tab":377

}

Result when the pin entry takes too long:

{

"result":"pin\_timeout",

"correlationId":"37b9deb4-af00-04b3-a37e-9241329a0471"

}

Result when the pin and signature are completed successfully:

{

"signature":"MGQCMC………..EHQp2sOOW6+1S7JKQ4J+u5ACYPZ1RG86",

"result":"OK",

"correlationId":"37b9deb4-af00-04b3-a37e-9241329a0471"

}

* AUTH

It allows crypto authentication of a digest/algo for a certificate and it returns the authentication. The new feature is based on BeID Connect.

#### User authentication

With the new feature, the following step is added after the generation of the signature.

The generated Token with the certification is sent to BE IdP for authentication. The BE IdP requests for authentication at the pin pad reader, where the user inserts their eID smart card pin to authenticate. Any error during the authentication procedure is displayed on the pin pad reader.

The successful authentication redirects to the web application, which in turn redirects to the browser of the user.

#### API flow

Further information on the API flow can be found in the MS Guide: [Native messaging - Microsoft Edge Development | Microsoft Docs](https://docs.microsoft.com/en-us/microsoft-edge/extensions-chromium/developer-guide/native-messaging?tabs=macos)

#### Browser extensions versioning (Google Chrome, Mozilla Firefox and MS Edge)

* Google Chrome, Mozilla Firefox and MS Edge browser extensions have their own versions.

These versions are defined inside the “gulp” script:

*ChromeExtension\ gulpfile.js*

var ChromeVersion = "X.Y.Z";

var EdgeVersion = "X.Y.Z";

var FirefoxVersion = "X.Y.Z";

* For the Safari browser extension, the version is defined inside the makefile (“SafariAppExtension” directory) of the project.

Since a Safari browser extension will be embedded in the “BeID Connect” application, its version must have the same version as the “BeID Connect”.

*SafariAppExtension\makefile*

VERSION = X.Y

#### Browser extensions generation (for Google Chrome, Mozilla Firefox and MS Edge)

The generation of browser extensions for Google Chrome, Mozilla Firefox and MS Edge are done through a Gulp script file.

Two types of browser extension can be generated:

* Debug version:
  + It does not include the limitation of the domains allowed to use the browser extension. This version will primary be used during the development of Gui-Sign since it will probably run at “local host” IP (or a local test domain).
  + It does not “minify” the JavaScript Code to allow Source code debugging in the browser.
* Release version:
  + It does include a list of authorised domains. The following domains are included in the list:
    - https://\*.belgium.be/\*
    - https://\*.fgov.be/\*
  + It does “minify” the JavaScript code to minimise the size.

To generate the Debug version:

1. Go with a command prompt to the “ChromeExtension” directory
2. Launch “gulp”

To generate the Release version:

1. Go with a command prompt to the “ChromeExtension” directory
2. Launch “gulp release”

#### Browser extensions generation (for Safari)

Check the Apple KeyChain setup section before generating the Apple applications.

To make file based generation:

Go to the “SafariAppExtension” directory, start “make”.

All Apple applications must be sent to Apple for the notarisation process:

1. Start “make notarise”.
   1. This will send the “BeID Connect” install package to Apple for notarisation.

xcrun notarytool submit $(DMGFILE) --keychain-profile "AC\_PASSWORD" --wait

* 1. The command line result returns a “ResquestUUID” that you can use to poll the notarisation process.
  2. The result of the notarisation will be sent by email (or it can be polled)

Once the notarisation process succeeds, the installation package can be used.

During installation of “BeIDConnect\_Extension\_X.Y.dmg”, the MacOS will check this notarisation online. This entails that the MacOS must be online.

To allow offline installation, this notarisation can be embedded in the installation package itself:

1. Start “make staple”.

xcrun stapler staple BeIDConnect\_Extension\_X.Y.dmg

This will generate the BeIDConnect\_Extension\_X.Y.dmg packages that can be uploaded to the BOSA website.

More Information about notarizing apps can be found at the Apple web site ([Notarizing macOS software before distribution | Apple Developer Documentation](https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution)) (depreciated: [Notarizing apps when developing with Xcode 12 and earlier | Apple Developer Documentation](https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/customizing_the_notarization_workflow/notarizing_apps_when_developing_with_xcode_12_and_earlier))

#### BeID Connect versioning

A new “BeID Connect” release updates the version in the following files in the repository:

*common\general.h*

#define BEIDCONNECT\_VERSION "X.Y"

*macos\makefile*

VERSION = X.Y

*macos\BeIDConnect.xcodeproj\project.pbxproj*

MARKETING\_VERSION = X.Y;

*SafariAppExtension\makefile*

VERSION = X.Y

*SafariAppExtension\BeIDConnect.xcodeproj\project.pbxproj*

MARKETING\_VERSION = X.Y;

*win32\Makefile*

MAJOR\_VERSION=X

MINOR\_VERSION=Y

*win32\VS\_2019\Resource.rc*

FILEVERSION X,Y,0,0

PRODUCTVERSION X,Y,0,0

VALUE "FileVersion", "X.Y.0.0"

VALUE "ProductVersion", "X.Y.0.0"

#### BeID Connect generation for Windows

To make a file-based generation:

1. Go to the “win32” directory
2. Start “make”

This will generate all the installation MSI packages that can be uploaded to the BOSA website.

#### BeID Connect generation for MacOS

Check the Apple KeyChain setup section before generating the Apple applications.

To make file-based generation:

Go to the “macos” directory, start “make”.

All Apple applications must be sent to Apple for the notarisation process.

1. Start “make notarise”.
   1. This will send the “BeID Connect” install package to Apple for notarisation.

xcrun notarytool submit $(DMGFILE) --keychain-profile "AC\_PASSWORD" --wait

* 1. The command line result returns a “ResquestUUID” which you can use to poll the notarisation process.
  2. The result of the notarisation will be sent by email (or it can be polled)

Once the notarization succeeds, the installation package can be used.

During installation of “beidconnect\_X.Y.dmg,”, the MacOS will check this notarisation online. This entails that the MacOS must be online.

To allow offline installation, this notarisation can be embedded in the installation package itself:

1. Start “make staple”.

xcrun stapler staple beidconnect\_X.Y.dmg

This will generate the beidconnect\_X.Y.dmg packages that can be uploaded to the BOSA website.

More Information about notarizing apps can be found at the Apple web site ([Notarizing macOS software before distribution | Apple Developer Documentation](https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution)) (depreciated: [Notarizing apps when developing with Xcode 12 and earlier | Apple Developer Documentation](https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/customizing_the_notarization_workflow/notarizing_apps_when_developing_with_xcode_12_and_earlier))

#### BeID Connect generation for Linux

The generation of the compilation Linux applications is done in the CI/CD pipeline. The tool needed to generate the package is the Docker. The Signature of the package will be done with a PGP Key that is also needed.

The following steps are required to create the BeID Connect packages for the various supported Linux distributions:

1. Get the latest files:
   1. Get the last release folders for “reprepro”, “rpmrep”, “eid-archive-beidconnect”, and “common” as well as the Dockerfile used for this generation.
   2. Copy them in a local work folder. Alongside the “common” folder, there should also be a “release\_vX.XX” folder containing the three other folders.
      1. Either from the last person to create a release
      2. Or from the Gitlab repository (if created, need for a link update - in development)
   3. Download the artifacts from the [beidconnect Gitlab pipeline](https://git-fsf.services.belgium.be/eidas/beidconnect/-/pipelines) related to the version of the release (commit id).
   4. Extract those ZIP files locally and verify that you have every supported distribution inside the resulting “products” folder.
2. Prepare the “rpmrep” folder:
   1. Inside the “rpmrep” folder, clean-up every “.rpm” file and every “repodata” folders (only keeping the file structure).

[b. Create a new folder in the file structure for the new distribution to be supported. This step is **ONLY** required in case of a new RPM-based distribution to be added to the supported list.]

1. Copy the new “.rpm” files (from the “products” folder) in their corresponding folder in “rpmrep”.
2. Prepare the “reprepo” folder:
   1. Inside the “reprepo/incoming” folder, clean-up every file.
   2. Copy the new Debian files (.deb, .ddeb, .buildinfo, …) in the “reprepo/incoming” folder.
3. Prepare the Dockerfile:
   1. Inside the Dockerfile, change the “ARG” line to match the name of your local folder containing “reprepro”, “rpmrep”, “eid-archive-beidconnect” (if you followed step one, this should be “release\_vX.XX”).

The following steps 5 and 6 are **ONLY** required in case of a new Debian-based distribution to be added to the supported list. If that is not the case, continue from step 7

1. Prepare the Debian Docker:
   1. Inside the Dockerfile, comment the “FROM Ubuntu” line and uncomment the “FROM Debian” one.
   2. Build the docker image using the Dockerfile (docker build . -t release-builder-debian).
   3. Run the image inside a container and access the container CLI.
2. Inside the Debian Docker:
   1. In the “eid-archive-beidconnect” folder:
      1. run the "dch -i" command to add to the changelog.
      2. update the “eid-archive-beidconnect/debian/postinst” file to support the new distribution.
      3. run the “dpkg-buildpackage -rfakeroot -uc -us” command to create the new eid-archive-beidconnect package.
   2. In the “reprepro/conf” folder:
      1. Update both the “distributions” and “incoming” files to recognize new versions.
   3. On the host machine (outside docker):
      1. Run the “docker cp [container]:release\eid-archive-beidconnect .\new\_release\” command to copy the new archive package.
      2. Inside the Dockerfile, comment the “FROM Debian” line and uncomment the “FROM Ubuntu” one.
3. Prepare the Ubuntu Docker:
   1. Build the docker image using the Dockerfile (docker build . -t release-builder-ubuntu).
   2. Run the image inside a container and access the container’s CLI.
   3. Run the “bash” command, then run the “shopt -s globstar” command.
4. Inside the Ubuntu Docker, for the Debian based packages:
   1. Run the “cd reprepro” command to move inside the correct folder.
   2. Run the “reprepro processincoming incoming” command to build the packages for TA.
   3. When TA tests are done, run the “reprepro copysrc <destination-distribution> <source-distribution> <source-package>” command for each distribution.
      1. E.g : “reprepro copysrc bullseye ta/bullseye beidconnect”
5. Inside the Ubuntu Docker, for the RPM-based packages:
   1. Run the “cd rpmrep” command to move inside the correct folder.
   2. Run the “rpmsign --resign --key-id 0x69FA2D05 \*\*/\*.rpm” command to resign the various “.rpm” files. If the GPG key changed, use the new key-id.
   3. For every folder in the “rpmrep” file structure, run the “createrepo\_c <folder>” command.

E.g: “createrepo\_c fedora/35”

1. On the host machine (outside docker):
   1. Run the “docker cp [container]:release .\release” command to copy the file locally on your machine.
   2. ZIP the “release” folder and send it to the person responsible of uploading the files on the server.

#### Tools setup for the BeID Connect generation

* **Visual studio 2022**

Used to build, debug, and test:

* + C++
  + SDK
  + Python application
* **NodeJS 16.15.0 LTS**

Used by:

* + Gui-sign
  + BeID Connect browser extension
* **gulp-cli**

Used for generating all browser extensions

* **Visual Code**

- Extension Pack for Java

○ Used by: Sign & validation

○ Install JDK-17.0.3.1

- Maven for Java

○ Used by: Sign & validation

- Spring Boot Extension Pack

○ Used by: Sign & validation

- Lombok Annotations Support for VS Code

○ Used by: Sign & validation

* **MinIO**

Used by Sign & validation as storage

* + Create dir
  + Download MinIO
* **Boost**

Used for BeidConnect.exe generation

* + Create a "3P" folder inside the beidconnect folder
  + Copy Boost inside the "3P" folder
* **WiX Toolset v3.11.2**

Used for creating the BeID Connect MSI

* .**NET Framework 3.5 SP1**

Used by WiX Toolset v3.11.2

## CI/CD process

#### Development principles

The CI/CD covers multiple components, that represent different pipelines:

* Gui-sign
* Sign-validation
* Gui-idp
* IDP
* eSealing
* BeID Connect

The development principles in this section are applicable to all of the above components, with the exception of BeIDconnect, which follows a different process detailed in the next section.

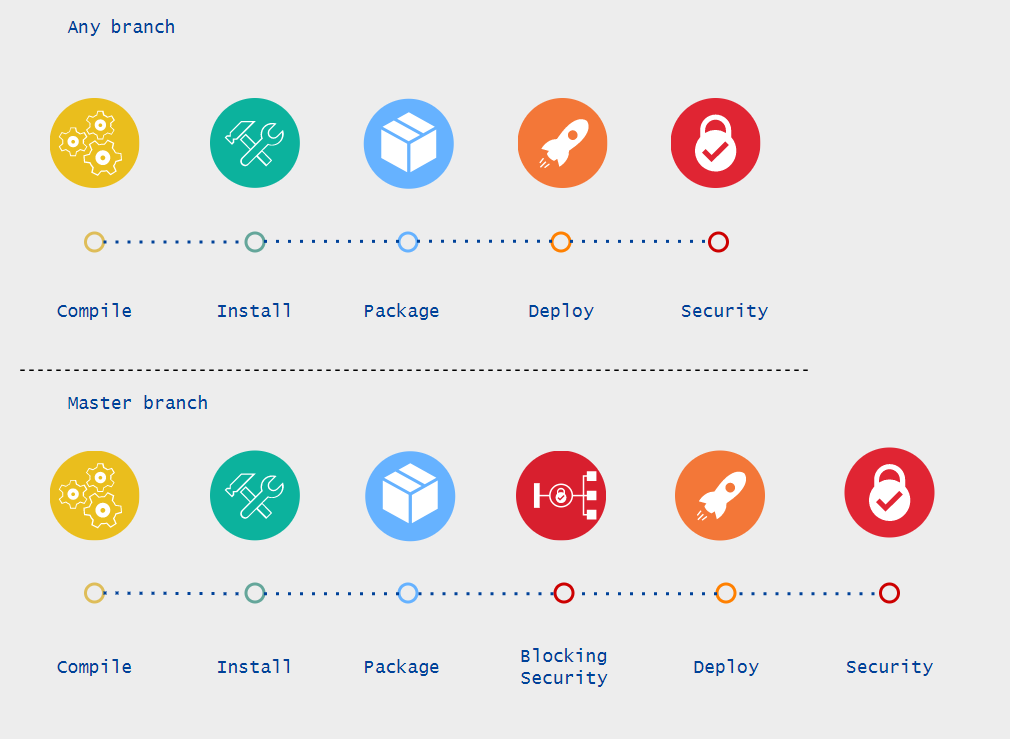
From the above components, the pipelines are distinguished for the back-end and the front-end:

* Pipelines for the front-end: Gui-sign and Gui-idp (written in Javascript)
* Pipelines for the back-end: Sign-validation, IDP and eSealing (written mainly in JAVA)

***Deployment stages***

For the above components, namely Gui-sign, Sign-validation, Gui-idp, IDP, eSealing, deployment takes place in stages. The preparation, package, and deployment stages constitute the “functional” part of the pipeline: compile, install, package the application in a docker container, and deploy the docker container. Security is applied after deployment in all branches. The deployment is rolled out as an Openshift update in the corresponding environment. This order is followed for functionality reasons, so that the security stage does not slow down or interfere with development.

When the code is aimed for deployment in the master branch, it goes through the INT and then to the PRD environment. In this case, the order of deployment stages changes and blocking security (unit tests and coverage) is added between packaging the code and deployment. This is to ensure that all the necessary checks are performed before the code is deployed into production.

For a visual representation of the stages in the pipeline, see the image below:

***Hidden jobs***

As explained in previous sections, GitLab is used for the building of the application, including the pipelines. The inclusion of hidden jobs relates to the volume of jobs in the configuration file and the specific needs that different branches/release targets have. Hidden jobs are essentially templates of jobs for reusable configuration, that allows for adding parameters for different instances. Each hidden job is further specified by including rules on what job needs to be implemented and how. In this way, if a modification is necessary, the hidden job is modified, and the last modification will be implemented to all instances of a job without having to apply it manually to every job.

Examples of hidden jobs can currently be spotted in the “Hidden jobs to extend” part of every pipeline.

***Flow control of the pipelines***

Text

Description automatically generated with medium confidenceIn the description of the jobs, the parameters “interruptible” and “allow failure” are set as true or false. See the following example:

* If “interruptible” is set to true, then when a newer pipeline starts before the job completes, the job can be interrupted/cancelled.
* If “allow failure” is true, it means that when running the pipeline and an issue is detected, this cannot cause the job to stop. Conversely, if it is set to false, it causes the job to stop at that stage and not go into deployment. This parameter is important, for instance, in the case of a major release or when the code goes into production, because if in the security testing stage, the unit tests fail, then the parameter “allow failure/false” ensures that the next stage is blocked, and the code does not go into production.

The compile stage is differentiated for the front-end or back-end pipelines, because they are written in different programming languages.

In the deployment stage, the docker container/image and the project/pipeline need to be specified. The deployment process is different based on the branch:

* deployment for the master branch is done automatically to QA;
* deployment for the develop branch is done automatically to TA;
* deployment for any other branch is done manually to TA.

#### Deployment principle

As the development branch is automatically synchronised with the GitLab project, it is important for sensitive information (i.e. proxy address) to remain safe. This is achieved by including in the GitLab configuration file of all branches, with the exception of the “cicd” branch, a reference to another file from the “eidas/fts-config-files” project. The file is located in the master branch.

To update the file on GitLab, copy the file from the “cicd” branch of any project and paste it into the particular “gitlab-ci.yaml” file (i.e. gitlab-ci/sign-validation.yaml) on the master branch.

Attention: The “cicd” branch should never be merged with another branch!

#### Security testing

Security testing for the Gui-sign, Sign-validation, Gui-idp, IDP and eSealing components is done using various tools and it involves the following:

* Unit testing, written by the developer and aimed at covering as much as possible of the solution .
* Coverage

Coverage calculates the percentage of code covered by the executed unit tests. This allows monitoring of the actual coverage of the code as well as of the coverage history over a period of few months. For the back-end pipelines (coded in JAVA), Cobertura is the tool used for coverage reporting, while for the front-end pipelines (coded in javascript), the coverage is specified as a command in the “NPM” integration tool for unit tests.

* Dependency checking

Dependency checking is done using OWASP Dependency Check for back-end and NPM Audit for front-end components. It looks into every dependency included in the code and checks for publicly disclosed vulnerabilities. For dependency checks, a particular threshold[[6]](#footnote-7) is set to determine if a given vulnerability is critical. If the criticality of a vulnerability is above the threshold, then the test fails, and it generates a report linking the vulnerability to the corresponding CVE entry.

* Docker container scanning

The container scanning step is performed by using the “Snyk” tool, integrated in the Docker tool, to scan the containers created by the Docker image and identify any known vulnerabilities in the container. If such vulnerability is detected, it is reported and it is mapped to the docker file, so as to ensure that any vulnerability in the code is flagged before the latter is deployed in production. The docker container scans are as many as the docker images created for each branch.

* SAST[[7]](#footnote-8)

SAST is static code testing and here, it is done using the SonarCloud tool, which goes through the written code and detects any vulnerabilities, bugs or errors in the code.

#### Jobs

* To update the pipelines
* Create a new branch in the project and copy the contents of the complete “gitlab-ci.yaml” file from the “fts-config-file”;
* Replace the “include” command inside the gitlab-ci file in your new branch by the actual content of the file.;
* Edit the pipeline to add your new changes
* Test the changes thoroughly to validate the new pipeline
* Copy the new file content and push it in “fts-config-file”
* Delete the working branch in the initial project.

The pipelines for Gui-sign and Sign-validation are running in Gitlab’s docker executor to allow each job to run in a different docker container.

* To run the pipelines:

The pipeline for these projects run with a similar flow:

* Preparation of the application
* Blocking security tests
* Package deployment and finally
* Non-blocking security tests. In both projects, the blocking security tests consist of the unit test battery and those are only blocking tests in case of a QA release. In any TA release, security tests are non-blocking in order to not slow down the development.

#### The BeID Connect pipeline

* To update the BeID Connect pipeline:
* Create another branch;
* Edit the “gitlab-ci.yaml” file, and
* Test any changes before merging them into the ‘master’ branch.
* To run the BeID Connect pipeline

The BeID Connect pipeline builds the source code using the rpmbuild tool before packaging it in a binary format for every supported architecture and Linux distribution. For Windows and Mac release of the application, all steps are manual, however for a full Linux release of the application, the first steps of the release are automated but subsequent steps are manual.

## eSealing

Diagram

Description automatically generated

Explanation of the steps:

1. The requester (i.e. a Belgian Ministry) sends a request to the eSealing service controller to return the list with all the sealing credentials. To certify the identity of the requestor, a username and password (slot label) must be provided to the service
2. If the identity is certified, the service returns the full list with the credentials (key store)
3. The requester selects the applicable certificate to be used for eSealing and launches the “/info” call to get its certificate chain and other attributes
4. The eSealing service returns its certificate chain and other attributes through the “/info” call
5. The requester obtains the Hash of the data to be signed (i.e. by calling the FTS through the “getDataToSign” operation as in the diagram)
6. The requester signs the Hash with a pre shared certificate called the “SAD certificate” and calls the “/signHash” eSealing operation
7. The eSealing service verifies the signature of the Hash and returns the signed Hash value (Digest)
8. If the Hash was obtained through the “getDataToSign” call, the requester calls “signDocument” to the server (i.e. FTS Sign-validation server)
9. The server returns the signed document with eSealing

## Future developments

#### MS Sharepoint integration

***Step-by-step explanation***

MS Sharepoint, a document library for secure storage of files, is planned to be used as part of the signing solution. The signing extension of MS Sharepoint can be accessed through the MS Sharepoint website.

* To sign using the MS Sharepoint library, there is an external menu outside the context menu or a menu appearing inside the taskbar of MS Sharepoint. The document to be signed should be a PDF file and in the initial version of MS Sharepoint integration, the selection of file will only allow one file to be selected for signing.
* Graphical user interface

  Description automatically generatedThe user selects ‘I want to sign’;
* A new box appears asking for confirmation ‘Do you want to sign the file?’;

Graphical user interface, text, application, email

Description automatically generated

* If the answer is ‘Yes’,
  + Graphical user interface, application

    Description automatically generatedThe MS Sharepoint extension needs to detect if there is a smartcard connected to the device to start the signing process. The MS Sharepoint extension connects with the browser extension, which calls the BeID Connect.
  + This step also checks if inside the selected document, there are Acroform fields (i.e. allowing for two different signatures, one for each party of a contract.).
    - If there are none, it goes directly to signing the document;
    - If there is one or two Acroforms, the user can select which Acroform field the signature should be inserted in and the signature is placed in the indicated Acroform. It also allows for insertion of the eID photo next to the signature;

Graphical user interface, application

Description automatically generated

* The MS Sharepoint extension communicates with the Sign-validation server;
* The next step is the validation of the signature by inserting the PIN code for the eID. If a PIN pad reader is connected to the device, the PIN code can be typed on the PIN pad. If there is no PIN pad reader, a dialogue box will appear and ask for the insertion of the PIN code;

Graphical user interface, application

Description automatically generated

* Once inserted, the signing of the document is performed and the signed file is inserted to the MS Sharepoint library as a new file, that has as a title the same name as the original file and the name of the person who signed it.

Graphical user interface, text, application

Description automatically generated

***Diagram

Description automatically generatedBusiness overview of MS Sharepoint integration***

Explanation

1. The user browser connects to the Sharepoint extension to upload the document to be signed or for signature validation.
2. If the browser extension is not already installed, the browser is redirected to download the browser extension for the eID software from the BOSA download server.
3. Two flows are possible:

Red flow:

* + The document is uploaded to the FTS;
  + The user will be asked to insert the PIN code to create the hash to sign;
  + If the PIN code is correct, the FTS returns the signed document.

Green flow:

* The document is uploaded to the FTS;
* The FTS checks the signature;
* The FTS sends back to the user browser the result of the check.

1. It means it is running inside the operating system [↑](#footnote-ref-2)
2. The Demo webapp shipped with the DSS Demo WebApp, can be found here: <https://ec.europa.eu/cefdigital/DSS/webapp-demo/sign-a-document>. [↑](#footnote-ref-3)
3. The freshness of the revocation status information is the maximum accepted difference between the issuance date of the revocation status information and the current time. [↑](#footnote-ref-4)
4. ETSI TS 119 172-1 V1.1.1 (2015-07), clause A.4.2.1, table A.2 row (m)2.2 RevocationFreshnessConstraints. [↑](#footnote-ref-5)
5. The signature is considered qualified when the Trust provider is a Qualified Trust Provider and for XAdES format, the qualification level must be able to be validated in the long term (LTA). [↑](#footnote-ref-6)
6. This threshold can be subject to change by the developers, and it can easily be found in the code of the pipelines. [↑](#footnote-ref-7)
7. This part of the security testing is currently under development. [↑](#footnote-ref-8)