sphereon

eIDAS Advanced Electronic Signature Client

The eIDAS Advanced Electronic Signature (AdES) client, allows to sign documents and digests (hashes), using CAdES (CMS, binary data), JAdES (JSON), PAdES (PDF), XAdES (XML) signatures, as defined by the European Telecommunications Standards Institute (ETSI). These signatures are part of the eIDAS legal framework in the European Union. Next to PAdES it can also create and verify PKCS#7 PDF signatures. These are non-ETSI, but are the more common PDF signatures, provided by companies on Adobe's Approved Trust List.

The purpose of this client is to easily create and verify eIDAS and PKCS#7 compliant signatures for documents and input data, using certificates which are stored in keystore files (PKCS#12) or using hardware (PKCS#11). Documents can be signed by providing the full document or by generating a hash/digest of the document first. Especially with remote signing REST APIs part of the Sphereon VDX platform, we suggest to create the digest first and then use the signature to merge with the original document. This means you are not sending the full document across the wire, which obviously is better from a privacy and security perspective.

Table of Contents

    - Create a hash digest for additional privacy and security
    - Create the signature
    - Signing the original data, merging the signature
    - Check whether a signature is valid
- PDF Signatures
    - Default PKCS#7 PDF signature
        - PKCS#7 configuration options
        - Example PKCS#7 flow
    - ETSI eIDAS PAdES detached PDF signature
        - PAdES configuration options
        - Example PAdES flow
    - Visual PKCS#7 and PAdES signatures
- Verifiable Credentials and SSI
- Building and running the source code
    - Requirements
    - Adding as Maven dependency
    - Gradle build (local maven repo)

Multiplatform library and REST API

This is a multiplatform Kotlin library. Right now it supports Java and Kotlin only. In the future Javascript/Typescript will be added. Please note that a REST API is also available that has integrated this client, allowing to generate and validate signatures using other languages. Next to Java/Kotlin, Javascript/Typescript a .NET SDK is available that integrates with the REST API. SDK code can be generated for other languages based upon the OpenAPI 3 spec provided with the REST API.

Signature flow

Creating a signed AdES document comprises several steps. It starts with the Original Data/Document, for which we first need to determine the Sign Input. The `SignInput` typically either is the full document, or a part of the document (PDF for instance). The `determineSignInput` method which requires the input document together with the signature type and configuration as parameters, automatically determines the Sign Input. The determineSignInput can be run locally without the need to use a REST API for instance.

Next there are two options. Directly signing the `SignInput` object using the `createSignature` method, resulting in a signature, or creating a Digest (Hash) of the `SignInput`. Since the createSignature method could be using a remote REST service or remote Hardware Security Module for instance, it is advisable to use the Digest method in most cases. The Digest method can be run locally, so even if the createSignature method needs to access remote resources, no information from the original data/document would be sent across the wire. The

digest method accepts a `SignInput` object as parameter and results in another `SignInput` object, with its sign method set to `DIGEST` instead of the original method of `DOCUMENT`.

The `createSignature` method accepts the `SignInput` object, which the signMode either being `DOCUMENT` or `DIGEST`, depending on which method was chosen. It is using the supplied 'KeyEntry' or Key alias string to sign the input object. This can either be done locally or remotely depending on the CertProvider implementation. The end result is a `Signature` object.

Lastly the `Signature` object needs to be merged with the original Document. It really depends on the type of signature being used how this is achieved. The document could for instance become part of the signature (ENVELOPING), the signature could become part of the document (ENVELOPED), or the signature could be detached (DETACHED)

The picture below gives a schematic overview of the process

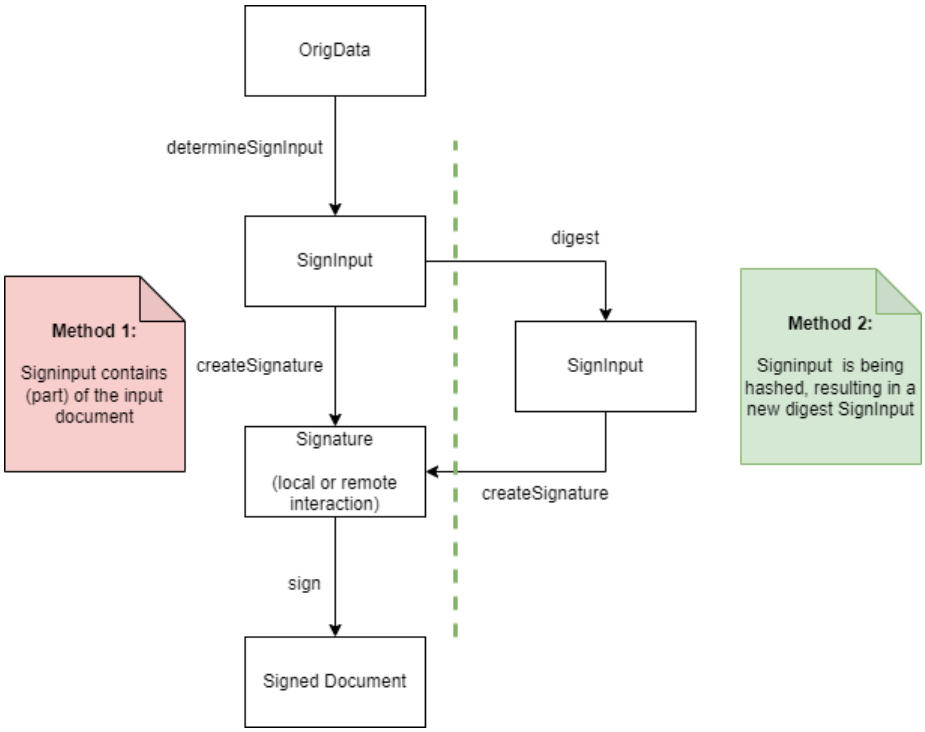

Figure 1: Signature Flow

It is possible to use multiple so called *SignatureServices* with the same *CertificateProvider*. This allows for instance to extract bytes and create a digest/hash from the input file locally, while creating the signature using a REST API or Azure Keyvault for instance. Then the signature is recombined with the

original document locally. The *createSignature* method and its counterpart *verifySignature* methods are typically ran using a REST API, Keyvault or locally with PKCS#11 hardware Certificate Providers. It is up to the caller to determine whether creating the digest/hash, and placing the signature in the input document also should run remotely or not.

For non Kotlin/Java environments we advise to setup the eIDAS Signature REST Microservice on premise, which connects to PKCS#11 hardware, a QTSP or Azure Keyvault remotely. Then use the REST endpoints, or use an SDK if available for your language. Please note that these SDKs typically have little local processing functionality unlike the Kotlin/Java library. The setup ensures that Personally Identifiable Information (PII) or other sensitive information doesn't leave your premise, and that only the signature is being created remotely from the Digest/Hash value. It also allows you to use authentication and roles/authorization locally on a per certificate and configuration level.

Certificate Provider Service

The Certificate Provider Service allows to manage public/private keys and Certificates using either a PKCS#12 keystore file as byte array or filepath. It also has support for PKCS#11 hardware (HSM and USB cards) as well as support for a Remote REST Certificate Provider and a Azure Keyvault/Managed HSM Certificate Provider. Next to certificate/key management a Certificate Provider also is responsible for creating and verifying signatures themselves. Other operations, like creating a hash, merging a signature into an input document are handled by a Signature Service instead of the Certificate Provider. A Single Certificate Provider can be shared by different Signature Services, as explained above

Given the wide range of supported import/creation methods, this library does not create or import certificates. Please use your method of choice (see below for some pointers).

PKCS#12 Keystore Certificate Provider Service

The below example in Kotlin sets up a certificate service using a PKCS#12 keystore file at a certain path

```
val providerPath = "path/to/pkcs12.p12"
val passwordInputCallback = PasswordInputCallback(password = "password".toCharArray())
val providerConfig = CertificateProviderConfig(
    type = CertificateProviderType.PKCS12,
    pkcs12Parameters = KeystoreParameters(providerPath)
)
val certProvider = CertificateProviderService(
    CertificateProviderSettings(
        id = "my-pkcs12-provider",
```

```
        providerConfig,
        passwordInputCallback
    )
)
```

Use existing tooling to create a certificate and PKCS#12 keystore

How to generate and/or import X.509 certificates and PKCS#12 keystores is out of scope of this project, but we provide some hints below. There are numerous resources on the internet to create X.509 certificates and PKCS#12 keystores.

Creating a PKCS#12 keystore using OpenSSL The private key and certificate must be in Privacy Enhanced Mail (PEM) format (for example, base64-encoded with `----BEGIN CERTIFICATE----` and `----END CERTIFICATE----` headers and footers).

Use the following OpenSSL commands to create a PKCS#12 file from your private key and certificate. If you have one certificate, use the CA root certificate.

```
openssl pkcs12 -export -in <signed_cert_filename> -inkey <private_key_filename> -name ‘tomca
```

If you have a chain of certificates, combine the certificates into a single file and use it for the input file, as shown below. The order of certificates must be from server certificate to the CA root certificate.

See RFC 2246 section 7.4.2 for more information about this order.

```
cat <signed_cert_filename> <intermediate.cert> [<intermediate2.cert>] > cert-chain.txt
openssl pkcs12 -export -in cert-chain.txt -inkey <private_key_filename> -name ‘tomcat’ -out
```

When prompted, provide a password for the new keystore.

PKCS#11 Hardware Security Module and Card based Certificate Provider Service

The PKCS#11 Certificate Provider allows you to use Hardware Security Based solutions that can interact using a PKCS#11 interface. It needs access to the driver library in order to operate.

```
val passwordInputCallback = PasswordInputCallback(password = "password".toCharArray())
val providerConfig = CertificateProviderConfig(
    type = CertificateProviderType.PKCS11,
    pkcs11Parameters = Pkcs11Parameters(
        pkcs11LibraryPath = "/usr/lib/opensc-pkcs11.so", // The PKCS11 driver path
        slotId = 0,
        slotListIndex = 2
    )
)
val certProvider = CertificateProviderService(
```

```
    CertificateProviderSettings(
        id = "my-pkcs11-provider",
        providerConfig,
        passwordInputCallback
    )
)

class Pkcs11Parameters(
    /** The path to the library  */
    val pkcs11LibraryPath: String? = null,

    /** The callback to enter a password/pincode  */
    val callback: PasswordInputCallback? = null,

    /** The slot Id to use  */
    val slotId: Int? = 0,

    /** The slot list index to use  */
    val slotListIndex: Int? = -1,

    /** Additional PKCS11 config  */
    val extraPkcs11Config: String? = null
)
```

Azure Keyvault or Managed HSM Certificate Provider Service

An Azure Keyvault Certificate Provider uses Azure Keyvault and Azure Managed HSM to retrieve certificates, create the Signature and verify a signature. The below example in Kotlin sets up a Certificate Service using Azure Keyvault or Azure Managed HSM. Both Keyvault and Managed HSM support Hardware Security Modules. The Managed HSM service is Microsoft's solution for an HSM not shared with other customers/tenants.

Note: *Although the Azure Certificate Provider should work with Azure Managed HSM, the library is not being tested against Azure Managed HSM, as opposed to Azure Keyvault.*

```
val providerConfig = CertificateProviderConfig(
    type = CertificateProviderType.AZURE_KEYVAULT
)
val keyvaultConfig = AzureKeyvaultClientConfig(
    keyvaultUrl = "https://your-keyvault-here.vault.azure.net/",
    tenantId = "<your-directory-id-as-shown-in-keyvault-properties>",
    credentialOpts = CredentialOpts(
        credentialMode = CredentialMode.SERVICE_CLIENT_SECRET, // Use a client id and secret
        secretCredentialOpts = SecretCredentialOpts(
```

```
            clientId = "<client id which has access to keyvault>",
            clientSecret = "<client secret belonging to client id>"
        )
    ),
    hsmType = HSMType.KEYVAULT, // Either KEYVAULT as HSM (FIPS140 Level-2), or MANAGED_HSM
    applicationId = "your-application-id-or-name", // This can be randomly choosen
    exponentialBackoffRetryOpts = ExponentialBackoffRetryOpts(
        maxTries = 10, // let's try max 10 times
        baseDelayInMS = 500, // Wait 0,5 seconds the first time
        maxDelayInMS = 15000 // Wait for max 15 seconds eventually
    )
)

val providerSettings = CertificateProviderSettings(
    id = "my-keyvault-provider",
    providerConfig
)

// From a factory
var certProvider = CertificateProviderServiceFactory.createFromConfig(settings = providerSet


// Or directly:
certProvider = AzureKeyvaultCertificateProviderService(providerSettings, keyvaultConfig)
```

REST Certificate Provider

The REST Certificate Provider uses REST to get Keys/Certificates It also exposes the createSignature and verifySignature methods as REST endpoints. Lastly other methods like creating a digest, determineSignInput and placing the signature in the original document could also be executed remotely if desired. This is however handled by the RESTSignatureService.

Authentication and Authorization support

The REST Certificate Provider has OAuth2, OpenID Connect and bearer token support integrated.

OAuth2 support The REST Certificate Provider client has support for oAuth2 and by extension OpenID Connect. Access to the OAuth2 functionality can be achieved by invoking the `oauth()` method on the REST Certificate Provider after providing the appropriate configuration:

```
val certProviderSettings = CertificateProviderSettings(
    id = "rest-oauth2",
    config = CertificateProviderConfig(
```

```
        cacheEnabled = true,
        type = CertificateProviderType.REST,
    )
)
val restConfig = RestClientConfig(
    baseUrl = "https://example.rest.service/signature/1.0",
    oauth2 = OAuth2Config(
        tokenUrl = "https://example.auth-server.com/auth/realms/sign-test/protocol/openid-co
        flow = OAuthFlow.APPLICATION,       // Use a clientId/clientSecret
        scope = "sign:vdx_sign_cert",       // Request scope, see scope chapter below
        clientId = "<client-id>",           // Provided by the IDP administrator
        clientSecret = "<client-secret>",   // Provided by the IDP administrator
        accessToken = "ey...."              // Typicaly should not be included. Can be used
    )
)

val restCertificateProvider = RestCertificateProviderService(certProviderSettings, restConfi
// Use the respective methods to get certificate or sign at this point. It will use oAuth2 t

// If oauth2 access is needed the oauth() method can be used. For instance to renew the acce
restCertificateProvider.oauth().renewAccessToken()

// From this point on requests will use the new token
```

Bearer Token and JWT support The REST Certificate Provider client has support to include bearer tokens (JWTs) in the authentication header. Access to the JWT functionality can be achieved by invoking the bearerAuth() method on the REST Certificate Provider after providing the appropriate configuration:

```
val certProviderSettings = CertificateProviderSettings(
    id = "rest-bearer",
    config = CertificateProviderConfig(
        cacheEnabled = true,
        type = CertificateProviderType.REST,
    )
)
val restConfig = RestClientConfig(
    baseUrl = "https://example.rest.service/signature/1.0",
    bearerAuth = BearerTokenConfig(
        schema = "bearer", // Can be omitted as it is the default
        bearerToken = "ey.....ffd" // Usefull to set an initial token or when the token is s
    )
)

val restCertificateProvider = RestCertificateProviderService(certProviderSettings, restConfi
// Use the respective methods to get certificate or sign at this point. It will use the bear
```

```
// If the bearer token needs to be updated
restCertificateProvider.bearerAuth().bearerToken = "ey.....<updated.bearer.token>..."

// From this point on requests will use the new token
```

OAuth2, OpenID Scopes The REST Microservice can be configured to support scopes. It is supported for both OAuth2 and OpenID Connect. Scopes are optional and when enabled, protect the API endpoints from users not having access to a certain scope at an endpoint level.

The available scopes are:

- **read:vdx_sign_cert**: Read/List certificates
- **admin:vdx_sign_cert**: Administer certificated
- **sign:vdx_sign_cert**: Sign using certificates
- **read:vdx_sign_config**: Read configurations
- **admin:vdx_sign_config**: Create and update configurations

Please note that this doesn't provide protection at an individual configuration nor certificate level. For these there is role support. The eIDAS REST MS documentation provides more info on this subject.

To enable scopes, you have to use your IAM solution of choice which supports oAuth2 or OpenID Connect and ensure that the appropriate scopes are requested from the IAM solution. How this works is different per IAM and outside the scope of this README.

The REST Certificate Provider can set the scopes before requesting the tokens using the configuration as shown above. You can also programaticaly set the scopes:

```
restCertificateProvider.oauth()..setScope("sign:vdx_sign_cert")
```

Roles support The REST Microservice has role based support. It is possible to define roles on individual configurations as well as individual certificates. Meaning you can define which roles and thus users/groups can access and administer certain signing configurations, as well as which users/groups can sign using a particular certificate. This is more finegrained than using the scopes above. Of course both could be used together if desired. The use of roles is explained in the Micro Service documentation. How roles should be made available to the JWT/bearer tokens is outside the scope of this README.

Certificate Provider caching

Especially for remote Certificate Providers like the REST, Azure and PKCS#11 Certificate Providers it might make sense to enable Key/Certificate caching. This means that a key which has been retrieved recently and which has not hit

the configured Time to Live yet, will be returned from the local cache, improving performance. Please be aware that this library is not involved in updating or replacing keys as these are highly implementation specific. As such providing a really high TTL could return a stale Key/Certificate if the Key had been replaced for a certain alias value. Given keys/certificates are not replaced that often in practice this shouldn't be too much of a problem. Please note that the actual signing using a key/certificate is rarely done using a cached key/certificate itself, given the private key is seldomly included. The `createSignature` method typically isn't involved in the caching mechanism.

```
CertProviderConfig(
    cacheEnabled = true, // Enable caching of keys/certificates. Requires a JSR107 Cache imp
    cacheTTLInSeconds = 600, // How long in seconds should certificates be kept in the cache
)
```

List keys, certificates

To list all available certificates of the provider one can use the getKeys() method. A list of IKeyEntry objects is being returned. The interface does not expose private keys, as developers typically should not access the private key directly and not every supported implementation gives access to private keys. If you are sure that key contains private keys, you can cas the result to IPrivateKeyEntry.

```
val keys = certProvider.getKeys()
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(keys))
```


```
[
  {
    "type": "PrivateKeyEntry",
    "alias": "test-key",
    "privateKey": {
      "algorithm": "RSA",
      "value": "MIIJRAIBAD....lpe53o2VXP",
      "format": "PKCS#8"
    },
    "encryptionAlgorithm": "RSA",
    "certificate": {
      "value": "MIIE...ybsgEkgc="
    },
    "certificateChain": [
      {
        "value": "MIIE...ybsgEkgc="
      }
    ]
  }
]
```

Get a key/certificate by alias

Use the geKey(alias: String) method to get a single certificate IKeyEntry object by alias if it exists. If it does not exist null is being returned. The IKeyEntry interface does not expose private keys, as developers typically should not access the private key directly and not every supported implementation gives access to private keys. If you are sure that key contains private keys, you can cast the result to IPrivateKeyEntry. Make sure to never sent private keys accoss unprotected network connections!

```
val key = certProvider.getKey("test-key")
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(key))
```


```
  {
  "type": "PrivateKeyEntry",
  "alias": "test-key",
  "privateKey": {
    "algorithm": "RSA",
    "value": "MIIJRAIBAD....lpe53o2VXP",
    "format": "PKCS#8"
  },
  "encryptionAlgorithm": "RSA",
  "certificate": {
    "value": "MIIE...ybsgEkgc="
  },
  "certificateChain": [
    {
      "value": "MIIE...ybsgEkgc="
    }
  ]
}
```


Create the signature

Depending on the certificate provider this method could be traversing the network as it might call a signature REST API, or use a network/cloud based Hardware Security Module containing the private key to sign. As such we advise to create the digest hash using a SignatureService beforehand so original documents/data is not being sent. Only the hash digest will traverse the network.

```
val signature = certProvider.createSignature(digestInput, keyEntry)
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(signatur
```


```
json lines {   // The actual signature   "value": "SoSsp+Mut3....XEDqEVw==",
"algorithm": "RSA_SHA256",   "signMode": "DIGEST",   // The
certificate used during signing   "certificate": {     "value":
"MIID1D....6Q42vNaS"   },   // The certificate chain including
the Certificate Authority (CA) last   "certificateChain": [
```

```
{       "value": "MIID1D....6Q42vNaS"     },     {       "value":
"MIID6j....GePoU8Ug=="     },     {       "value": "MIIDVzC....PSNfsSBog=="
}   ] }
```

Signature Service

The Signature Service allows you to create and verify signatures, as well as creating a hash/digest of input data

Initialize the Signature Service

The Signature service want to have a certificate provider as single argument. If you want to use multiple certificate providers you will have to instantiate multiple signature services.

```
val signingService = SignatureService(certificateProvider = certProvider)
```

Determine Sign Input

Determines the bytes that will serve as input for the `digest` or `createSignature` methods. Since multiple signature types are supported the configuration and key are required te determine the appropriate mode of extraction. For instance Pades signatures do not need a simple digest of the full file contents, depending on whether the PDF document already contains signatures. This method should be called first when creating a signature.

```
val padesConfig = SignatureConfiguration(
    signatureParameters = SignatureParameters(
        // Make sure the signature becomes part of the file
        signaturePackaging = SignaturePackaging.ENVELOPED,
        // Use RSA and SHA256
        digestAlgorithm = DigestAlg.SHA256,
        encryptionAlgorithm = CryptoAlg.RSA,
        signatureLevelParameters = SignatureLevelParameters(
            // Set the level to PAdES baseline B
            signatureLevel = SignatureLevel.PAdES_BASELINE_B,
        ),
        signatureFormParameters = SignatureFormParameters(
            // PAdES specific parameters
            padesSignatureFormParameters = PadesSignatureFormParameters(
                signerName = "John Doe",
                contactInfo = "support@sphereon.com",
                reason = "Test",
                location = "Online"
            )
        )
```

```
    )
)

val pdfDoc = File("input.pdf")
val origData = OrigData(value = pdfDocInput.readBytes(), name = pdfDoc.name)
val keyEntry = signingService.certificateProvider.getKey("test-key")!!

val signInput = signingService.determineSignInput(
    origData = origData,
    keyEntry = keyEntry,
    signMode = SignMode.DOCUMENT,
    signatureConfiguration = signatureConfiguration
)
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(signInpu
```

The below SignInput object could be used directly for the createSignature method or a digest can be created first, so that the input data will never traverse a network if a remote Sign REST API or remote Hardware Security Module is being used.

```
json lines {   "input": "MYHeMBgGCSqG....TAkxVAgEK",   "signMode":
"DOCUMENT",   "digestAlgorithm": "SHA256",   "name": "input.pdf"
}
```

Create a hash digest for additional privacy and security

The `digest` method creates a hash digest out of the SignInput. The hash digest is a one way function that creates the fingerprint of the file. From the digest you cannot get back to the original input data/document. This means any Personally Identifiable Data or Data which needs to stay private will not be available to methods which need access to a remote REST API or remote Hardware Security Module. This obviously is preferable from a privacy and security perspective. It allows users of the library to execute all methods on premise and then depending on the chosen Certificate Provider sign the data either on premise or remotely. In no circumstance will the input data leave the premise.

```
val digestInput = signingService.digest(signInput)
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(digestIn
```

Notice that the below SignInput object is different from the passed in SignInput. The input value is shorter as it now is a hash digest. The signMode moved from `DOCUMENT` to `DIGEST` so that the `createSignature` method knows not to create a hash digest out of the input anymore.

```
json lines {   "input": "fSx6BzHxJ8p3Mn9E52DJ1eNrchfcMa1ZHaSjAi9D5z8=",
"signMode": "DIGEST",   "digestAlgorithm": "SHA256",   "name":
"input.pdf" }
```

Create the signature

The default Signature Service implementations delegate this method to the corresponding method of the CertificateProvider, given most CertificateProviders to not expose private keys for security reasons.

Depending on the certificate provider settings this method could be traversing the network as it might call a signature REST API, or use a network/cloud based Hardware Security Module containing the private key to sign. As such we advise to create the digest beforehand so original documents/data is not being sent. Only the hash digest will traverse the network.

```
val signature = signingService.createSignature(digestInput, keyEntry)
println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(signatur
```

```
json lines {   // The actual signature   "value": "SoSsp+Mut3....XEDqEVw==",
"algorithm": "RSA_SHA256",   "signMode": "DIGEST",   // The
certificate used during signing   "certificate": {     "value":
"MIID1D....6Q42vNaS"   },   // The certificate chain including
the Certificate Authority (CA) last   "certificateChain": [
{       "value": "MIID1D....6Q42vNaS"     },     {       "value":
"MIID6j....GePoU8Ug=="     },     {       "value": "MIIDVzC....PSNfsSBog=="
}   ] }
```

Signing the original data, merging the signature

This method takes the original input document, the created signature and merges them together to provide a signed output document. It needs access to the configuration to know how and where the signature should be merged with the document.

```
val signOutput = signingService.sign(origData, signature, signatureConfiguration)

// Write the signed bytes to a file
File("signed-output.pdf").writeBytes(signOutput.value)

println(Json { prettyPrint = true; serializersModule = serializers }.encodeToString(signOutp
```

The result of the above is a new file which is the input PDF, but now signed.

```
json lines {   // The signed data/document   "value": "JVBERi0xLjYNJ....4cmVmCjc2NzUwCiUlRU9
"signMode": "DIGEST",   "digestAlgorithm": "SHA256",   "name":
"input-pades-baseline-b.pdf",   "mimeType": "application/pdf",
"signature": {     "value": "SoSsp+Mut3....XEDqEVw==",     "algorithm":
"RSA_SHA256",     "signMode": "DIGEST",     "certificate": {
"value": "MIID1DCC....XxY1e6Q42vNaS"     },     "certificateChain":
[       {         "value": "MIID1DCC....XxY1e6Q42vNaS"       },
{         "value": "MIID6jCC....Y+TpJGePoU8Ug=="       },       {
"value": "MIIDVzCCAj....PSNfsSBog=="       }     ]   } }
```

Check whether a signature is valid

The default Signature Service implementations delegate this method to the corresponding method of the CertificateProvider.

In order to check whether a signature is valid the SignInput is needed. If a reference to that data is not available anymore the `determineSignInput` and depending on whether a hash digest was create the `digest` method are needed to get back the SignInput object.

```
val valid = signingService.isValidSignature(digestInput, signature, signature.certificate!!)
// Returns a boolean
```

PDF Signatures

This library supports electronically signing PDF documents, including approval and certify signatures as well as visual signatures. Supported PDF signature types are:

- **adbe.pkcs7.detached**, which is the default PDF document signature as used by Adobe and the most common type of signature
- **ETSI.PAdES/ETSI.CAdES.detached**, which is ETSI/eIDAS compliant (needs special Certificates provided by eIDAS Trust Service Providers!)

Default PKCS#7 PDF signature

This is the default PDF Signature type, typically used with Certificates provided by an organization on the Adobe Approved Trusted List (AATL).

There are 2 types of signatures possible:

- CERTIFICATION
  - Can only be applied once to a PDF document!
  - It acts like a seal, which typically is organization or department wide.
  - A blue bar will appear with name of the signer, the company and the CA that issued the Certificate
  - Allows to protect the document for further modifications at several levels
  - Optionally showing an image of the signature. Clickable to show more information
- APPROVAL
  - Can be applied multiple times.
  - This is what typically is being used for people signing the document.
  - It is comparable to a user signing a paper based document.
  - The signature shows the name and additional information.
  - Optionally showing an image of the signature. Clickable to show more information

PKCS#7 configuration options

The below options are part of a configuration, but can typically also be provided on every invocation. This allows to use the same certificate for instance for signing by multiple people by changing the signerName and related properties.

```
class Pkcs7SignatureFormParameters(
    /**
     * The signature mode, according to the PDF spec. Either needs to be APPROVAL or CERTIFI
     *
     * - CERTIFICATION can only be applied once to a PDF document. It acts like a seal, whic
     * A blue bar will appear with name of the signer, the company and the CA that issued th
     * - APPROVAL can be applied multiple times. This is what typically is being used for pe
     * The signature shows the name and additional information. Optionally showing an image
     */
    val mode: PdfSignatureMode? = PdfSignatureMode.APPROVAL,

    /**
     * This attribute allows to explicitly specify the SignerName (name for the entity signi
     * The person or authority signing the document.
     */
    val signerName: String,


    /** The signature creation reason  */
    val reason: String? = null,

    /** The contact info  */
    val contactInfo: String? = null,

    /** The signer's location  */
    val location: String? = null,

    /**
     * Defines the preserved space for a signature context. Only change if you know what yo
     *
     * Default : 9472 (default value in pdfbox)
     */
    val signatureSize: Int? = 9472,

    /**
     * This attribute allows to override the used Filter for a Signature.
     *
     * Default value is Adobe.PPKLite
     */
    val signatureFilter: String? = PdfSignatureFilter.ADOBE_PPKLITE.specName,
```

```
    /**
     * This attribute allows to override the used subFilter for a Signature.
     *
     * Default value is adbe.pkcs7.detached
     */
    val signatureSubFilter: String? = PdfSignatureSubFilter.ADBE_PKCS7_DETACHED.specName,

    /**
     * This attribute is used to create visible signature
     */
    val signatureImageParameters: SignatureImageParameters? = null,

    /**
     * This attribute allows to set permissions in case of a "certification signature". Tha
     * future change(s).
     */
    val permission: CertificationPermission? = null,

    /**
     * Password used to encrypt a PDF
     */
    val passwordProtection: String? = null
)
```

Example PKCS#7 flow

Below an example is provided where a local Signing Service and a Local Azure Keyvault Certificate Provider is being used to sign with a certificate on the AATL list, resulting in "blue-bar" signatures. The example key vault settings can be found above. The createSignature/verifySignature/getCert(s) methods would use the Azure Keyvault REST API, so we will be creating a digest first to ensure we are not sending the document to Azure Keyvault.

```
// Gets the file and set the orig data object
val pdfDocInput = this::class.java.classLoader.getResource("example-unsigned.pdf")
val origData = OrigData(value = pdfDocInput.readBytes(), name = "example-unsigned.pdf")


val certProvider = CertificateProviderServiceFactory.createFromConfig(
    settings = providerSettings, // See above for examples
    azureKeyvaultClientConfig = keyvaultConfig // See example above
)
// The factory has returned an Azure Keyvault Certificate Provider at this point

// Create a local signature service, which uses alias/strings to denote the certificates to
```

```
val signingService = AliasSignatureService(certProvider)

val alias = "example:3f98a9a740fb41b79e3679cce7a34ba6" // The alias is Azure Keyvault certif

val signatureConfiguration = SignatureConfiguration(
    signatureParameters = SignatureParameters(
        signaturePackaging = SignaturePackaging.ENVELOPED,
        digestAlgorithm = DigestAlg.SHA256,
        encryptionAlgorithm = CryptoAlg.RSA,
        signatureAlgorithm = SignatureAlg.RSA_SHA256,
        signatureLevelParameters = SignatureLevelParameters(
            signatureLevel = SignatureLevel.PKCS7_B, // This sets the mode to PDF PKCS#7 (Bo
        ),
        signatureFormParameters = SignatureFormParameters(
            // PKCS#7 specific parameters
            pkcs7SignatureFormParameters = Pkcs7SignatureFormParameters(
                mode = PdfSignatureMode.APPROVAL,   // Use an approval signature
                signerName = "Example User",
                contactInfo = "example@sphereon.com",
                reason = "Example",
                location = "Amsterdam",
            )
        )
    )
)

// Locally extract the bytes to be signed from the PDF document
val signInput = signingService.determineSignInput(
    origData = origData,
    alias = alias,
    signMode = SignMode.DOCUMENT,
    signatureConfiguration = signatureConfiguration
)

// Locally create a hash/digest of the extracted bytes
val digestInput = signingService.digest(signInput)

// Calls Azure Keyvault using the hash/digest and the certificate associated with the alias
val signature = signingService.createSignature(digestInput, alias)

// Locally combine the original document with the created signature
val signOutput = signingService.sign(origData, signature, signatureConfiguration)
```

ETSI eIDAS PAdES detached PDF signature

This is the eIDAS/ETSI compliant PDF Signature type, used with Certificates provided by eIDAS Trust service providers.

PAdES configuration options

The below options are part of a configuration, but can typically also be provided on every invocation. This allows to use the same certificate for instance for signing by multiple people by changing the signerName and related properties.

```
class PadesSignatureFormParameters(
    /**
     * This attribute allows to explicitly specify the SignerName (name for the Signature).
     * The person or authority signing the document.
     */
    val signerName: String? = null,

    /** The signature creation reason  */
    val reason: String? = null,

    /** The contact info  */
    val contactInfo: String? = null,

    /** The signer's location  */
    val location: String? = null,

    /**
     * Defines the preserved space for a signature context
     *
     * Default : 9472 (default value in pdfbox)
     */
    val signatureSize: Int? = 9472,

    /**
     * This attribute allows to override the used Filter for a Signature.
     *
     * Default value is Adobe.PPKLite
     */
    val signatureFilter: String? = PdfSignatureFilter.ADOBE_PPKLITE.specName,

    /**
     * This attribute allows to override the used subFilter for a Signature.
     *
     * Default value is ETSI.CAdES.detached
     */
    val signatureSubFilter: String? = PdfSignatureSubFilter.ETSI_CADES_DETACHED.specName,
```

```
    /**
     * This attribute is used to create visible signature in PAdES form
     */
    val signatureImageParameters: SignatureImageParameters? = null,

    /**
     * This attribute allows to create a "certification signature". That allows to remove pe
     * future change(s).
     */
    val permission: CertificationPermission? = null,

    /**
     * Password used to encrypt a PDF
     */
    val passwordProtection: String? = null,

    /** Defines if the signature shall be created according to ETSI EN 319 122  */
    val en319122: Boolean? = true,

    /** Content Hints type  */
    val contentHintsType: String? = null,

    /** Content Hints description  */
    val contentHintsDescription: String? = null,

    /** Content identifier prefix  */
    val contentIdentifierPrefix: String? = null,

    /** Content identifier suffix  */
    val contentIdentifierSuffix: String? = null

)
```

Example PAdES flow

Below an example is provided where a local Signing Service and a Certificate Provider using a Hardware Security Module accessed using the PKCS#12 interface with a certificate provided by a Qualified Trust Service Provider.

```
// Gets the file and set the orig data object
val pdfDocInput = this::class.java.classLoader.getResource("example-unsigned.pdf")
val origData = OrigData(value = pdfDocInput.readBytes(), name = "example-unsigned.pdf")
```

```
val certProvider = CertificateProviderServiceFactory.createFromConfig(
    settings = providerSettings, // See above for examples
)
// The factory has returned a Local Certificate Provider with PKCS#12 HSM support at this po

// Create a local signature service, which uses alias/strings to denote the certificates to
val signingService = AliasSignatureService(certProvider)

val alias = "example" // The alias is HSM specific and is <certificate Id>

val signatureConfiguration = SignatureConfiguration(
    signatureParameters = SignatureParameters(
        signaturePackaging = SignaturePackaging.ENVELOPED,
        digestAlgorithm = DigestAlg.SHA256,
        encryptionAlgorithm = CryptoAlg.RSA,
        signatureAlgorithm = SignatureAlg.RSA_SHA256,
        signatureLevelParameters = SignatureLevelParameters(
            signatureLevel = SignatureLevel.PAdES_BASELINE_B, // This sets the mode to PDF
        ),
        signatureFormParameters = SignatureFormParameters(
            // PAdES specific parameters
            padesSignatureFormParameters = PadesSignatureFormParameters(
                signerName = "Example User",
                contactInfo = "example@sphereon.com",
                location = "Amsterdam",
            )
        )
    )
)

// Locally extract the bytes to be signed from the PDF document
val signInput = signingService.determineSignInput(
    origData = origData,
    alias = alias,
    signMode = SignMode.DOCUMENT,
    signatureConfiguration = signatureConfiguration
)

// Locally create a hash/digest of the extracted bytes
val digestInput = signingService.digest(signInput)

// Calls the HSM using the hash/digest and the certificate associated with the alias value
val signature = signingService.createSignature(digestInput, alias)

// Locally combine the original document with the created signature
val signOutput = signingService.sign(origData, signature, signatureConfiguration)
```

Visual PKCS#7 and PAdES signatures

It is possible to create visual signatures. These signatures show an image of a 'wet signature' by providing an image file, or alternatively they are created from provided text. These visual signatures will show up in the document, and can be clicked upon to show more information.

Both PAdES and PKCS#7 type of PDF signatures have option to add visual signature options in their respective SignatureFormParameters

PKCS#7 Signature Form Parameters using an image

```
/**
 * This attribute is used to create visible signature inside the Pkcs7FormParameters
 */
signatureImageParameters = SignatureImageParameters(
    image = ByteArray(),                // The company logo or "wet signature" image as byte
    zoom = 150,                         // Scale the image to 150%, defaults to 100%
    imageScaling = ImageScaling.STRETCH,// Stretches the image in both directions

    signatureFieldParameters = SignatureFieldParameters(
        fieldId = "Signature 1",    // Signature field id/name
        page = 2,                   // Page number where the signature field is added (defa
        originX = 10f,              // Coordinate X where to add the signature field (origi
        originY = 50f,              // Coordinate Y where to add the signature field (origi
        width = 100f,               // Signature field width
        height = 30f                // Signature field height
    )

)
```

PKCS#7 Signature Form Parameters using text

```
/**
 * This attribute is used to create visible signature inside the Pkcs7FormParameters
 */
signatureImageParameters = SignatureImageParameters(
    signatureImageTextParameters = SignatureImageTextParameters(
        text = "John Doe, CEO",                 // This variable defines the text
        textWrapping = TextWrapping.FILL_BOX,   // Fills the box adjusting the font-size to
        padding = 10,                           // padding in pixels, defaults to 5
        textColor = "RED",
        backgroundColor = "WHITE"
    ),
    signatureFieldParameters = SignatureFieldParameters(
        fieldId = "Signature 1",    // Signature field id/name
        page = 2,                   // Page number where the signature field is added (defa
        originX = 10f,              // Coordinate X where to add the signature field (origi
```

```
        originY = 50f,              // Coordinate Y where to add the signature field (origi
        width = 100f,               // Signature field width
        height = 30f                // Signature field height
    )
)
```

Signature Image Parameters is the main class to configure Visual Signatures.

```
class SignatureImageParameters(

    /**
     * This variable contains the image to use (company logo,...)
     */
    val image: ByteArray? = null,

    /**
     * This variable defines a `SignatureFieldParameters` like field positions and dimensio
     */
    val fieldParameters: SignatureFieldParameters? = null,

    /**
     * This variable defines a percent to zoom the image (100% means no scaling).
     * Note: This does not touch zooming of the text representation.
     */
    val zoom: Int = NO_SCALING,

    /**
     * This variable defines the color of the image
     */
    val backgroundColor: String? = null,

    /**
     * This variable defines the DPI of the image
     */
    val dpi: Int? = null,

    /**
     * Use rotation on the PDF page, where the visual signature will be
     */
    val rotation: VisualSignatureRotation? = null,

    /**
     * Horizontal alignment of the visual signature on the pdf page
     */

    val alignmentHorizontal: VisualSignatureAlignmentHorizontal? = VisualSignatureAlignmentH
```

```
    /**
     * Vertical alignment of the visual signature on the pdf page
     */
    val alignmentVertical: VisualSignatureAlignmentVertical? = VisualSignatureAlignmentVerti

    /**
     * Defines the image scaling behavior within a signature field with a fixed size
     *
     * DEFAULT : ImageScaling.STRETCH (stretches the image in both directions to fill the s
     */
    val imageScaling: ImageScaling? = ImageScaling.STRETCH,

    /**
     * This variable is use to defines the text to generate on the image
     */
    val textParameters: SignatureImageTextParameters? = null
)
```

The Signature Field Parameters define the location and dimensions of the signature

```
class SignatureFieldParameters(
    /** Signature field id/name (optional)  */
    val fieldId: String? = null,

    /** Page number where the signature field is added  */
    val page: Int = DEFAULT_FIRST_PAGE,

    /** Coordinate X where to add the signature field (origin is top/left corner)  */
    val originX: Float = 0f,

    /** Coordinate Y where to add the signature field (origin is top/left corner)  */
    val originY: Float = 0f,

    /** Signature field width  */
    val width: Float = 0f,

    /** Signature field height  */
    val height: Float = 0f
)
```

Signature Image Text parameters define text and the appearance to place in the visual signature

```
class SignatureImageTextParameters(
    /**
     * This variable allows to add signer name on the image (by default, LEFT)
     */
```

```
val signerTextPosition: SignerTextPosition = SignerTextPosition.LEFT,

/**
 * This variable defines the image from text vertical alignment in connection
 * with the image<br></br>
 * <br></br>
 * It has effect when the [SignerPosition][SignerTextPosition] is
 * [LEFT][SignerTextPosition.LEFT] or [ RIGHT][SignerTextPosition.RIGHT]
 */
val signerTextVerticalAlignment: SignerTextVerticalAlignment = SignerTextVerticalAlignme

/**
 * This variable set the more line text horizontal alignment
 */
val signerTextHorizontalAlignment: SignerTextHorizontalAlignment = SignerTextHorizontalA

/**
 * This variable defines the text to sign
 */
val text: String? = null,

/**
 * This variable defines the font to use
 * (default is PTSerifRegular)
 */
val font: String? = null,

/**
 * This variable defines how the given text should be wrapped within the signature fiel
 *
 * Default : TextWrapping.FONT_BASED - the text is computed based on the `Font` configu
 */
val textWrapping: TextWrapping = TextWrapping.FONT_BASED,

/**
 * This variable defines a padding in pixels to bound text around
 * (default is 5)
 */
val padding: Float = DEFAULT_PADDING,

/**
 * This variable defines the text color to use
 * (default is BLACK)
 * (PAdES visual appearance: allow null as text color, preventing graphic operators)
 */
val textColor: String? = DEFAULT_TEXT_COLOR,
```

```
    /**
     * This variable defines the background of a text bounding box
     */
    val backgroundColor: String? = DEFAULT_BACKGROUND_COLOR
)
```

Verifiable Credentials and SSI

This library can be used as part of a Self Sovereign Identity solution to sign Verifiable Credentials. For more information we refer to the accompanying SSI Proof Client

Building and running the source code

Requirements

This library has the following minimal requirements:

Java 11 and higher (tested up to Java 17) for the build is required. At runtime Kotlin and Java can be used currently

Gradle 7.X and higher;

We strongly recommend using the latest available version of JDK, in order to have the most recent security fixes and cryptographical algorithm updates. Before processing the integration steps, please ensure you have successfully installed Maven and JVM with a required version.

Adding as Maven dependency

The simplest way to include DSS to your Maven project is to add a repository into the pom.xml file in the root directory of your project as following:

```
<project>
  <repositories>

    .....

    <repository>
      <id>sphereon-public</id>
      <name>Sphereon Public</name>
      <url>https://nexus.qa.sphereon.com/repository/sphereon-public/</url>
    </repository>
  </repositories>
```

```
  <dependencies>

    ....

    <dependency>
      <groupId>com.sphereon.vdx</groupId>
      <artifactId>eidas-signature-client-jvm</artifactId> <!-- The Java implementation if th
      <version>0.9.1</version>
    </dependency>
  </dependencies>
</project>
```

Gradle build (local maven repo)

```
gradlew clean deployToMavenLocal
```