Manual básico de integración de los módulos del Cliente @firma 1.6 en desarrollos Java

Índice de contenidos

[1 Introducción 4](#_Toc499553202)

[2 Núcleo del cliente 5](#_Toc499553203)

[2.1 Interfaces de Utilidad 5](#_Toc499553204)

[es.gob.afirma.core.signers.AOSigner 5](#_Toc499553205)

[es.gob.afirma.core.ciphers.AOCipher 9](#_Toc499553206)

[es.gob.afirma.core.envelopers.AOEnveloper 11](#_Toc499553207)

[2.2 Clases de Utilidad 12](#_Toc499553208)

[es.gob.afirma.core.misc.Base64 12](#_Toc499553209)

[es.gob.afirma.core.misc.MimeHelper 13](#_Toc499553210)

[es.gob.afirma.core.misc.AOUtil 14](#_Toc499553211)

[2.3 Factorías 15](#_Toc499553212)

[es.gob.afirma.core.signers.AOSignerFactory 15](#_Toc499553213)

[es.gob.afirma.core.ui.AOUIFactory 16](#_Toc499553214)

[3 Almacenes de certificados 19](#_Toc499553215)

[3.1 Módulo afirma-core-keystores 19](#_Toc499553216)

[es.gob.afirma.keystores.AOKeyStore 19](#_Toc499553217)

[es.gob.afirma.keystores.AOKeyStoreManagerFactory 20](#_Toc499553218)

[es.gob.afirma.keystores.AOKeyStoreManager 22](#_Toc499553219)

[es.gob.afirma.keystores.AOKeyStoreDialog 23](#_Toc499553220)

[es.gob.afirma.keystores.filters.CertificateFilter 24](#_Toc499553221)

[3.2 Módulo afirma-keystores-mozilla 25](#_Toc499553222)

[3.3 Módulo afirma-keystores-single 25](#_Toc499553223)

[3.4 Módulo afirma-keystores-capiaddressbook 26](#_Toc499553224)

[4 Módulos de firma 27](#_Toc499553225)

[4.1 Módulo afirma-crypto-cades 27](#_Toc499553226)

[4.2 Módulo afirma-crypto-cadestri-client 28](#_Toc499553227)

[4.3 Módulo afirma-crypto-cms 28](#_Toc499553228)

[4.4 Módulo afirma-crypto-pdf 29](#_Toc499553229)

[4.5 Módulo afirma-crypto-padestri-client 29](#_Toc499553230)

[4.6 Módulo afirma-crypto-xades 30](#_Toc499553231)

[4.7 Módulo afirma-crypto-xadestri-client 30](#_Toc499553232)

[4.8 Módulo afirma-crypto-xmlsignature 31](#_Toc499553233)

[4.9 Módulo afirma-crypto-odf 31](#_Toc499553234)

[4.10 Módulo afirma-crypto-ooxml 32](#_Toc499553235)

[5 Firma masiva 33](#_Toc499553236)

[5.1 Módulo afirma-core-massive 33](#_Toc499553237)

[es.gob.afirma.massive.DirectorySignatureHelper 33](#_Toc499553238)

[es.gob.afirma.massive.MassiveSignatureHelper 35](#_Toc499553239)

[6 Módulos de cifrado 38](#_Toc499553240)

[6.1 Módulo afirma-crypto-cipher 38](#_Toc499553241)

[es.gob.afirma.ciphers.AOCipherKeyStoreHelper 39](#_Toc499553242)

[7 Ejemplos generales 41](#_Toc499553243)

[7.1 Selección de almacén y firma 41](#_Toc499553244)

# Introducción

El Cliente @firma se compone de múltiples módulos que le dotan de las distintas funcionalidades de firma, acceso a almacenes y otras características, o que proporcionan funcionalidades de carácter general que se utilizan en otros módulos.

El presente documento explica a rasgos generales los principales módulos y clases Java del Cliente @firma, enfocando la explicación para permitir que un desarrollador los integre en sus aplicaciones Java y pueda utilizar las funcionalidades de alto nivel que ofrece.

La funcionalidad e interfaces de las clases internas del Cliente @firma son susceptibles a cambios sin previo aviso. Se recomienda que los desarrolladores que utilicen los módulos del cliente @firma utilicen versiones estables de los mismos para evitar cambios. Los módulos del Cliente @firma 1.6 y superiores podrán encontrarse en el repositorio central de Maven. Los ficheros fuente del proyecto y todas sus dependencias modificadas están disponibles en GitHub:

<https://github.com/ctt-gob-es/>

No se ofrece soporte sobre las distintas clases del proyecto, si bien sí se corregirá cualquier error que afecte al funcionamiento de los productos con soporte del Cliente @firma.

La descripción completa de las clases y métodos que se mencionan en este documento puede encontrarse en el Javadoc de los distintos módulos del proyecto.

# Núcleo del cliente

El módulo *core* del Cliente @firma (**afirma-core**) es el módulo principal del proyecto. Las principales clases de interés de este módulo son:

* Interfaces de utilidad
  + **AOSigner**: Interfaz de los manejadores de firma.
  + **AOCipher**: Interfaz de los manejadores de cifrado.
  + **AOEnveloper**: Interfaz de los manejadores de envoltorios de datos.
* Clases de utilidad
  + **Base64**: Clase para la codificación y descodificación de cadenas en base 64.
  + **MimeHelper**: Clase para la identificación de tipos de datos.
  + **AOUtil**: Clase con funciones de utilidad general.
* Factorías
  + **AOSignerFactory**: Factoría para la obtención centralizada de manejadores de firma.
  + **AOUIFactory**: Factoría para la obtención centralizada de las clases de gestión de interfaces.

## Interfaces de Utilidad

En este apartado se mostrarán las interfaces que implementan las distintas clases que proporcionan al Cliente @firma las capacidades de gestión de firmas, cifrados y sobres digitales.

*es.gob.afirma.core.signers.AOSigner*

Es la interfaz básica que implementa cada uno de los manejadores de firma del Cliente. Esta clase proporciona varios métodos de utilidad para el tratamiento de las firmas y la generación de firmas, cofirmas y contrafirmas.

Los manejadores de firma que implementan esta interfaz gestionan firmas en un formato concreto. Por ejemplo: CAdES, PAdES, XAdES, Factura Electrónica,... Estas implementaciones se encuentran en los distintos módulos de firma que se describen en el apartado Módulos de firma.

Los métodos de especial interés para el manejo de firma electrónicas son:

* **boolean isValidDataFile(byte[] data) throws IOException**
  + Indica si los datos proporcionados pueden ser firmados por el manejador de firma. Hay formatos de firma que siempre responderán que sí, como el manejador de CAdES que puede firmar cualquier tipo de datos, mientras que otros deberán comprobar que los datos son compatibles, como el manejador de PAdES que sólo puede firmar documentos PDF y tendrá que comprobar que los datos son de este tipo.
* **boolean isSign(byte[] data) throws IOException**
  + Indica si los datos indicados se corresponden con una firma compatible con el formato del manejador. Esto es importante para identificar si esos datos son una firma que podamos cofirmar o contrafirmar. Hay que tener en cuenta que existen formatos compatibles con otros, aunque no en ambos sentidos. Por ejemplo, una firma CAdES puede considerarse una firma CMS, por lo que el manejador de CMS devolvería true al pasarle una firma CAdES en el método isSign(byte[] data). Sin embargo, una firma CMS no tiene por qué ser una firma CAdES, así que el manejador de CAdES podría devolver false si se le pasa una firma CMS.
* **byte[] getData(byte[] signData) throws AOException** **, IOException**
  + Recupera los datos que se firmaron si estos están contenidos en la firma electrónica pasada por parámetro. Si los datos no están contenidos devolverá null, y si los datos no se corresponden con una firma soportada por el manejador o se obtiene un error al recuperarlos, se lanzará una excepción.
* **byte[] sign(byte[] data, String algorithm, PrivateKey key, Certificate[] certChain, Properties extraParams) throws AOException, IOException** 
  + Realiza la firma de unos datos. El manejador de firma debe soportar los datos indicados. Deberemos configurar el algoritmo de firma (consultar la documentación de la clase AOSignConstants para más información), la clave y la cadena de certificación del certificado de firma (consultar el apartado de almacenes de certificados para comprobar cómo se obtienen) y las propiedades de configuración del formato en modo Properties (consultar las propiedades soportadas por cada formato de firma).

**Ejemplo de uso:**

|  |
| --- |
| ...  // Instanciamos el manejador de firmas PAdES incluido en  // el módulo afirma-crypto-pdf  AOSigner signer = **new** AOPDFSigner();  // Comprobamos que el documento que deseamos firmar sea  // compatible con el manejador (que sea un PDF)  **if** (!signer.isValidDataFile(data)) {  **throw** **new** IllegalArgumentException(  "No se ha introducido un documento PDF valido");  }  // Generamos una firma PAdES  **byte**[] signedPDF = signer.sign(  data, AOSignConstants.SIGN\_ALGORITHM\_SHA512WITHRSA, key,  certChain, **null**);  ... |

* **byte[] cosign(byte[] sign, String algorithm, PrivateKey key, Certificate[] certChain, Properties extraParams) throws AOException, IOException** 
  + Realiza la cofirma de unos datos a partir de una firma. Esta firma debe contener los datos originalmente firmados o, si no los tiene, debe haberse realizado con el mismo algoritmo de firma que la cofirma que vamos a realizar (para poder reaprovechar la huella digital de los datos). Es importante saber si nuestro manejador de firma soporta la operación de cofirma.
  + Para realizar la cofirma también deberemos indicar el algoritmo de firma, la clave y la cadena del certificado de firma y la configuración en formato Properties que deseamos aplicar (consultar la documentación del formato de firma para conocer os parámetros soportados).

**Ejemplo de uso:**

|  |
| --- |
| ...  // Instanciamos el manejador de firmas CAdES incluido en los módulos  // afirma-crypto-cades y afirma-crypto-cades-multi  AOSigner signer = **new** AOCAdESSigner();  // Comprobamos que el documento que deseamos firmar sea compatible  // con el manejador  **if** (!signer.isSign(data)) {  **throw** **new** IllegalArgumentException(  "No se ha introducido una firma CAdES valida");  }  // Cofirmamos  **byte**[] cosignature = signer.cosign(  data, AOSignConstants.SIGN\_ALGORITHM\_SHA512WITHRSA,  key, certChain, **null**);  ... |

* **byte[] cosign(byte[] data, byte[] sign, String algorithm, PrivateKey key, Certificate[] certChain, Properties extraParams) throws AOException, IOException** 
  + Este método realiza la cofirma de unos datos tal como el anterior pero, en esta ocasión, se le pueden pasar los datos originalmente firmados por si estos no están incluidos en la firma que proporcionamos.
* **byte[] countersign(byte[] sign, String algorithm, CounterSignTarget targetType, Object[] targets, PrivateKey key, Certificate[] certChain, Properties extraParams) throws AOException, IOException** 
  + Este método genera la contrafirma de una firma. La firma indicada debe estar soportada por el manejador y este debe ser compatible con la operación de contrafirma. Como parámetro se le indica la firma (no tiene porqué incluir los datos originalmente firmados), el algoritmo de firma, el tipo de contrafirma, los nodos del árbol de firmas que se desea contrafirmar, la clave y la cadena del certificado de firma y las propiedades de configuración de la operación en forma de Properties.
  + El tipo de contrafirma indica si se deben firmar todos los nodos del árbol de firmas, sólo lo nodos hoja, una serie de nodos concretos o los nodos correspondientes a unos determinados firmantes. Se determinan los objetivos a contrafirmar mediante los valores del enumerado CounterSignTarget. Sólo en el caso de la contrafirma de nodos (NODES) y firmantes (SIGNERS) será necesario utilizar el parámetro targets en donde se indicarán los índices de los nodos a contrafirmar (en el caso de la contrafirma de nodos) o los nombres de los firmantes (en el caso de la contrafirma de firmantes). La posición de los nodos se determina en preorden a partir del árbol de firmas devuelto con el método getSignersStructure(byte[] signature, boolean asSimpleSignInfo) y el nombre de los firmantes el obtenido como elementos del árbol devuelto por este método cuando asSimpleSignInfo es false.

**Ejemplo de uso:**

|  |
| --- |
| ...  // Instanciamos el manejador de firmas XAdES incluido en el módulo  // afirma-crypto-xades  AOSigner signer = **new** AOXAdESSigner();  // Comprobamos que el documento que deseamos firmar sea compatible  // con el manejador  **if** (!signer.isSign(data)) {  **throw** **new** IllegalArgumentException(  "No se ha introducido una firma XAdES valida");  }  // Contrafirmamos los nodos hoja  **byte**[] countersignature = signer.countersign(  data, AOSignConstants.SIGN\_ALGORITHM\_SHA512WITHRSA,  CounterSignTarget.LEAFS, **null**, key, certChain, **null**);  ... |

Si nuestra aplicación utiliza un único formato de firma podemos instanciar directamente el manejador de firma del formato deseado para realizar las distintas operaciones. Si, por el contrario, nuestra aplicación utiliza más de un formato de firma o puede cambiar de formato en un futuro, se pueden gestionar los manejadores de firma a través de la clase AOSignerFactory descrita en el apartado de Factorías incluidos en el núcleo del Cliente @firma.

*es.gob.afirma.core.ciphers.AOCipher*

Esta interfaz es la que implementan los manejadores de las funciones de cifrado simétrico. Estos manejadores permiten el cifrado y el descifrado de datos para las configuraciones que soporten. También deben permitir obtener claves de cifrado válidas para esas configuraciones, ya sea generándolas u obteniéndolas a partir de una versión codificada de la misma.

Los métodos de principal interés de estos manejadores son:

* **Key generateKey(AOCipherConfig algorithmConfig) throws NoSuchAlgorithmException, AOException** 
  + Genera una clave de cifrado aleatoria compatible con una configuración de cifrado dada. Si la configuración no está soportada o no se puede generar la clave se lanza una excepción.

**Ejemplo de uso**

|  |
| --- |
| ...  AOCipherConfig cipherConfig = **new** AOCipherConfig(  AOCipherAlgorithm.AES, **null**, **null**);  AOCipher cipher = AOSunJCECipher();  Key key = cipher.generateKey(cipherConfig);  ... |

* **Key decodeKey(byte[] keyEncoded, AOCipherConfig algorithmConfig, Object[] params) throws KeyException**
  + Obtiene un objeto clave a partir de su versión codificada. Este método es útil cuando ya existe una clave privada almacenada de forma externa o introducida directamente por el usuario. Recibe la codificación de esta clave, la configuración del algoritmo de cifrado con la que es compatible y una lista de parámetros adicionales por si son necesarios para la descodificación. En caso de error al generar la clave se lanza una excepción.
* **byte[] cipher(byte[] data, AOCipherConfig algorithmConfig, Key cipherKey) throws AOException, KeyException** 
  + Cifra simétricamente unos datos (data) a partir de una configuración de cifrado (algorithmConfig) y una clave de cifrado (cipherKey). La configuración de cifrado debe estar soportada por el manejador y la clave de cifrado ser compatible con este algoritmo. Se lanzará una excepción si la configuración no está soportada, la clave no es válida o no se puede realizar el cifrado.

**Ejemplo de uso**

|  |
| --- |
| ...  // Definimos la configuracion de cifrado  AOCipherConfig cipherConfig = **new** AOCipherConfig(  AOCipherAlgorithm.AES, **null**, **null**);  AOCipher cipher = AOSunJCECipher();  // Generamos la clave  Key key = cipher.generateKey(cipherConfig);  // Ciframos los datos  **byte**[] cipheredData = cipher.cipher(data, cipherConfig, key);  // Obtenemos la codificacion de la clave generada  **byte**[] keyEncoded = key.getEncoded();  ... |

* **byte[] decipher(byte[] data, AOCipherConfig algorithmConfig, Key decipherKey) throws AOException, InvalidKeyException** 
  + Descifra unos datos cifrados (data) a partir de una configuración de cifrado (algorithmConfig) y una clave de cifrado (decipherKey). La configuración de cifrado debe estar soportada por el manejador y la clave de cifrado ser compatible con este algoritmo. Se lanzará una excepción si la configuración no está soportada, la clave no es válida o no se puede realizar el descifrado.

**Ejemplo de uso**

|  |
| --- |
| ...  // Definimos la configuracion de descifrado  AOCipherConfig cipherConfig = **new** AOCipherConfig(  AOCipherAlgorithm.AES, **null**, **null**);  AOCipher cipher = AOSunJCECipher();  // Recogemos la clave codificada utilizada en el proceso de cifrado  Key key = cipher.decodeKey(keyEncoded, cipherConfig, **null**);  // Desciframos los datos  **byte**[] plainData = cipher.decipher(data, cipherConfig, key);  ... |

*es.gob.afirma.core.envelopers.AOEnveloper*

Es la interfaz que implementan los manejadores que gestionan la generación y apertura de sobres digitales y otros envoltorios. La única implementación funcional de este método es AOCMSEnveloper (del módulo **afirma-crypto-cms-enveloper**).

Los métodos más relevantes para la gestión de los envoltorios son:

* **byte[] envelop(byte[] data, String signAlgorithm, String type, PrivateKeyEntry keyEntry, X509Certificate[] certDest, AOCipherAlgorithm cipherAlgorithm, String dataType, Properties extraParams) throws AOException** 
  + Método para la generación de un envoltorio. Según el tipo (type) este envoltorio puede incluir los datos en plano o estar estos procesados de alguna manera. Los formatos más extendidos cifran estos datos de tal forma que sólo uno de los destinatarios pueda descifrarlo.
  + Una configuración común de este tipo de envoltorios son los sobres electrónicos. En estos sobres, los datos son cifrados mediante un algoritmo simétrico y una clave aleatoria. La clave se incluye a su vez múltiples veces en el sobre, cifrada cada una de ellas con la clave pública del certificado de uno de los destinatarios definidos en el sobre. De esta forma, cada destinatario declarado podrá descifrar con su clave privada la clave con la que se cifraron los datos y con esta ya obtener los datos originales.
  + El método recibe los datos a envolver, el algoritmo de firma que se desee utilizar en los envoltorios que lo requieran y cuyo algoritmo de huella digital se utilizará de ser necesario (por ejemplo, indicando SHA256withRSA se utilizará este algoritmo para la firma de los envoltorios que lo necesiten y el algoritmo SHA156 para los envoltorios que necesiten algoritmo de huella digital), el tipo de envoltorio deseado (cuyos valores se definirán para cada implementación de la interfaz), la clave privada de quien genera el envoltorio para poder identificarlo como remitente (según el tipo de envoltorio), los certificados de los destinatarios (según el tipo de envoltorio), la configuración de cifrado (según el tipo de envoltorio), el tipo de los datos envueltos (opcional) y parámetros adicionales que puedan hacer falta para la configuración de la operación (según las necesidades del envoltorio).
* **byte[] recoverData(byte[] envelop, PrivateKeyEntry addresseePke) throws InvalidKeyException, AOException**
  + Método para recuperar el contenido de un envoltorio de datos. Este método recibe el envoltorio y la clave privada de un destinatario cuando se hayan especificado destinatarios para el envoltorio (según tipo). El método lanza excepciones cuando la clave indicada no se valida o no se pertenezca a un destinario del envoltorio o cuando se produzca otro error durante su apertura.

## Clases de Utilidad

En el núcleo del Cliente @firma se incluyen una serie de clases de utilidad general que se utilizan en casi todos los módulos del Cliente y que un desarrollador/integrador puede utilizar en aquellas aplicaciones que integren el Cliente @firma. A continuación se detalla cada una de estas clases y los métodos que proporcionan para facilitar la labor del integrador/desarrollador.

*es.gob.afirma.core.misc.Base64*

Es una clase para la codificación de datos a Base 64 y viceversa. Esta clase cuenta con varios métodos estáticos que se pueden utilizar para tratar aquellas entradas y salidas del Cliente.

* **static String encode( byte[] source )** 
  + Codifica datos en una cadena en base 64.
* **static String encode( byte[] source, boolean urlSafe )** 
  + Como el anterior, pero, si se indica que la codificación sea *URL Safe*, el base 64 resultante usará el carácter ‘-’ en lugar de ‘+’ y ‘\_’ en lugar de ‘/’. De esta forma, la cadena base 64 podrá enviarse a través de una URL sin provocar errores de codificación.
* **static byte[] decode( String s ) throws IOException** 
  + Decodifica una cadena en base 64 en datos binarios. Se lanza una excepción si la cadena proporcionada no es interpretable como base 64.
* **static byte[] decode( String s, boolean urlSafe ) throws IOException** 
  + Como el anterior, pero, si se indica que la codificación sea *URL Safe*, esperará que la cadena base 64 proporcionada esté codificada de esta manera.

*es.gob.afirma.core.misc.MimeHelper*

Clase para la identificación del tipo de datos. Esta clase proporciona el MimeType de los datos y, si es posible, la extensión asignada a ese tipo de documentos y una descripción textual. Además, incluye métodos estáticos para obtener el OID de un tipo de dato a partir de su MimeType y viceversa.

Para el análisis de unos datos es necesario crear un objeto de esta clase con esos datos y después obtener los valores identificados.

Los métodos de interés de la clase son:

* **MimeHelper(byte[] data)** 
  + Constructor a partir del cual obtenemos un objeto capaz de analizar los datos proporcionados.
* **String getMimeType()throws IOException**
  + Obtiene el MimeType de los datos. Devuelve "application/octet-stream" si no puede identificarlos.
* **String getExtension()** 
  + Obtiene la extensión del tipo de fichero en el que se almacenan estos datos.
* **static String transformMimeTypeToOid(String mimetype)** 
  + Método estático que obtiene el OID identificador del tipo de datos correspondiente a un MimeType. Si no se encuentra, se devuelve el OID genérico "1.2.840.113549.1.7.1".
* **static String transformOidToMimeType(String oid)** 
  + Método estático que obtiene el MimeType asignado a un OID identificador de tipo de datos. Si no se encuentra, se devuelve el MimeType ("application/octet-stream").

*es.gob.afirma.core.misc.AOUtil*

Clase con múltiples métodos estáticos de utilidad general.

* **static boolean copyFile(File source, File dest) throws IOException** 
  + Clase para copiar el fichero indicado por source en la ruta de dest.
* **static URI createURI(String file) throws URISyntaxException** 
  + Método para crear una URI a partir de cualquier cadena de texto con forma de URI o URL (remota o local). Admite los separadores de ruta "/" y "\", y rutas locales que no empiecen por "file://".
* **static InputStream loadFile(URI uri) throws IOException** 
  + Abre un flujo de datos con el contenido de una URI.
* **static byte[] getDataFromInputStream(InputStream input) throws IOException** 
  + Lee todo el contenido de un flujo de datos.
* **static String getCN(X509Certificate c)**
  + Obtiene el nombre común (CN o *Common Name*) del *Subject* de un certificado. Si no tiene definido el nombre común devuelve la unidad organizativa (OU o *Organization Unit*). Si no encuentra tampoco esta se devuelve el valor del RDN más significativo según el orden.
* **static String getCN(String principal)** 
  + Obtiene el nombre común (CN o *Common Name*) a partir del *Principal* de un certificado. Si no tiene definido el nombre común devuelve la unidad organizativa (OU o *Organization Unit*). Si no encuentra tampoco esta se devuelve el valor del RDN más significativo según el orden.

## Factorías

Estas son factorías que permiten gestionar de forma centralizada algunos de los recursos con los que cuenta el Cliente @firma. El uso de estas factorías facilitará cambios posteriores en su aplicación, la ampliación de sus funcionalidades y el soporte de distintos entornos.

*es.gob.afirma.core.signers.AOSignerFactory*

La clase AOSignerFactory es la factoría desde la que se cargan los manejadores de firma para cada formato soportado por el Cliente. Cuando deseemos trabajar con un formato de firma, podemos solicitar a esta clase el manejador adecuado para ese formato. Para que la factoría devuelve el manejador de un formato concreto, será necesario haber importado en el proyecto el módulo de firma correspondiente a ese formato.

Los dos principales métodos de interés son:

* **static AOSigner getSigner(String signFormat)**
  + Este método permite obtener un manejador de firma para el formato de firma indicado. Este método es el que se utiliza comúnmente para seleccionar el manejador de firma a utilizar cuando son varios los formatos soportados por la aplicación o puede cambiarse de formato en un futuro. En caso de soportar un único formato se podría instanciar directamente un objeto del manejador de firma concreto. El manejador de firma instanciado es un objeto que implementa la interfaz AOSigner. Pueden tomarse los nombres de formatos de firma de la clase AOSignConstants.

**Ejemplo de uso:**

|  |
| --- |
| ...  AOSigner signer = AOSignerFactory.getSigner(  AOSignConstants.SIGN\_FORMAT\_CADES);  // Firmamos en formato CAdES  **byte**[] signature = signer.sign(  data, // Datos que se firman  AOSignConstants.SIGN\_ALGORITHM\_SHA1WITHRSA, // Algoritmo de firma  key, // Referencia a la clave privada  certChain, // Cadena de certificación  **null** // Propiedades extra de configuración  ) ;  ... |

* **static AOSigner getSigner(byte[] signData) throws IOException**
  + Este método permite obtener el manejador de firma más adecuado para manipular firmas con el formato de la firma introducida por parámetro. La principal utilidad de este método es comprobar si unos datos son en realidad una firma con un formato soportado. Una vez obtenido el manejador podemos utilizarlo para manipular la firma y, por ejemplo, cofirmarla/contrafirmar, extraer los datos que se firmaron u obtener información de ella. Si los datos introducidos no se corresponde con una firma soportada (podría ser una firma pero en un formato que desconocemos, una para el que ahora no tenemos disponible el manejador o una firma en un formato soportado pero con alguna peculiaridad para la que no está preparada el manejador) se devuelve null.

**Ejemplo de uso:**

|  |
| --- |
| ...  **byte**[] cosignature = **null**;  // Cofirmamos en el mismo formato en el que esta firmada  AOSigner signer = AOSignerFactory.getSigner(signature);  **if** (signer != **null**) {  cosignature = signer.cosign(  signature, // Firma  AOSignConstants.SIGN\_ALGORITHM\_SHA1WITHRSA, // Algoritmo  key, // Referencia a la clave privada  certChain, // Cadena de certificación  **null** // Propiedades extra de configuracion  );  }  ... |

*es.gob.afirma.core.ui.AOUIFactory*

Esta es una factoría orientada a obtener información del usuario o decisiones por su parte mediante componentes gráficos. AOUIFactory utiliza, según el entorno en el que se ejecute el Cliente, un gestor u otro para generar los diálogos gráficos que se necesiten. Actualmente, esta factoría utiliza por defecto una u otra implementación según el entorno en el que se ejecute que se encuentre. En un futuro, podrían darse de alta nuevos manejadores y gracias a esta factoría no sería necesario hacer cambios en el código, tan sólo establecerla como el manejador activo. Los manejadores implementados se encuentran en el módulo **afirma-ui-core-jse**.

Aunque el diálogo de selección de certificados también se maneja internamente mediante esta factoría, para una integración por separado de este diálogo en otra herramienta, es más sencillo hacerlo a través de la clase AOKeyStoreDialog definida en el módulo **afirma-core-keystores**, debido a la relación intrínseca entre el diálogo de selección de certificados y los distintos almacenes a los que se tiene acceso. Para saber más sobre esta clase, consulte el apartado Módulo afirma-core-keystores para más información.

Los manejadores de interfaces gráficos implementan la interfaz: **es.gob.afirma.core.ui.AOUIManager**.

La selección del gestor de interfaces a utilizar se hará de forma automática pero es posible establecer explícitamente qué gestor utilizar mediante el método:

* **static void setUIManager(AOUIManager manager)**

Los métodos estáticos que proporciona AOUIFactory directamente para su uso son:

* **static File[] getLoadFiles(String dialogTitle, String currentDir, String filename, String[] extensions, String description, boolean selectDirectory, boolean multiSelect, Object icon, Object parentComponent)** 
  + Muestra un diálogo para la selección de ficheros. También se puede mencionar múltiples Se puede configurar mediante parámetros para permitir seleccionar uno o varios ficheros o, en su lugar, permitir seleccionar un directorio. Son configurables también el título del diálogo, el nombre del fichero, las extensiones permitidas, la descripción del tipo de fichero, el icono y el componente padre sobre el que mostrar el diálogo si corresponde.
* **static char[] getPassword(String text, Object parent)** 
  + Muestra un diálogo para la obtención de una contraseña. Permite establecer el texto de solicitud de la contraseña y el componente padre sobre el que mostrar el diálogo, si corresponde.
* **static char[] getPassword(String text, String charset, boolean beep, Object parent)** 
  + Muestra un diálogo para la obtención de una contraseña. Permite establecer el texto de solicitud de la contraseña, el conjunto de caracteres permitidos, indicar si se debe reproducir un *beep* del sistema si se trata de insertar un carácter no válido y configurar el componente padre sobre el que mostrar el diálogo, si corresponde.
* **static File getSaveDataToFile(byte[] data, String dialogTitle, String currentDir, String selectedFile, String[] exts, String description, Object parent)** 
  + Muestra un diálogo para el guardado de datos en disco y la obtención del fichero guardado. Permite establecer los datos a guardar, el título del diálogo, el directorio actual, el fichero seleccionado por defecto, un listado de las extensiones de fichero que se desea que aparezcan en el diálogo, la descripción del tipo de fichero y el componente padre sobre el que mostrar el diálogo, si corresponde.
* **static Object showCertificateSelectionDialog(Object parent, NameCertificateBean[] selectionValues)** 
  + Muestra un diálogo para la obtención de un alias de certificado seleccionado por el usuario. Permite configurar el componente padre sobre el que mostrar el diálogo, si corresponde, y el listado con los certificados (y su información) entre los que puede seleccionarse. Se devuelve el alias del certificado seleccionado.
* **static int showConfirmDialog(Object parent, Object message, String title, int optionType, int messageType)** 
  + Muestra un diálogo para solicitar la confirmación del usuario. Permite configurar el componente padre sobre el que mostrar el diálogo, el mensaje que aparecerá en el mismo, su título, las opciones disponibles y el tipo de diálogo.
* **static Object showInputDialog(Object parent, Object message, String title, int messageType, Object icon, Object[] selectionValues, Object initialSelectionValue)** 
  + Muestra un diálogo para obtener la opción seleccionada por el usuario entre las múltiples que se configuran. Permite configurar el componente padre sobre el que mostrar el diálogo, el mensaje que aparecerá en el mismo, su título, el tipo de diálogo, el icono que deberá mostrarse, las opciones seleccionables y el valor inicialmente seleccionado.

# Almacenes de certificados

El uso de certificados es imprescindible para ejecutar las operaciones de firma y ensobrado de datos. Estos certificados se localizan en almacenes de certificados, desde donde cualquier aplicación compatible puede acceder a ellos.

El Cliente @firma es compatible con múltiples almacenes de certificados (almacenes sistema, almacenes en disco, almacenes con formato estándar...). La gestión de estos almacenes se realiza de forma centralizada, de tal forma que es posible indicar a qué almacén deseamos acceder y extraer los certificados que en ellos se encuentran y las claves necesarias para realizar operaciones criptográficas como la firma de documentos.

La gestión de almacenes centralizada se realiza en el Cliente @firma mediante el módulo **afirma-core-keystores**. Adicionalmente, para el acceso al almacén de certificados de Mozilla Firefox es necesario el módulo **afirma-keystores-mozilla**, y para el uso de certificados simples (que no están en una almacén) se utiliza **afirma-keystores-single**.

## **Módulo afirma-core-keystores**

Es el módulo principal para la gestionar de almacenes de claves y certificados. Este módulo cuenta con una factoría mediante la cual es posible seleccionar el almacén de claves deseado, lo que nos devuelve un manejador para el mismo. Mediante es manejador podremos realizar una serie de operaciones como son, introducir la clave para el acceso al almacén, listar los certificados que contiene y extraer sus claves.

Las clases de especial interés de este módulo son:

*es.gob.afirma.keystores.AOKeyStore*

Enumerado con el listado de almacenes inicialmente soportado por los clientes. Principalmente, estos son:

* **WINDOWS**. Almacén de claves personales de Windows. Sólo disponible en sistemas Windows.
* **APPLE**. Almacén de claves de Mac OS X. Sólo disponible en sistemas Mac OS X. Permite acceder tanto al almacén central del sistema como a almacenes en fichero (si se indica su ruta).
* **MOZ\_UNI**. Almacén de claves de Mozilla Firefox. Sólo disponible en aquellos en los que se encuentre instalado Mozilla Firefox. En caso de encontrarse instalados e insertados dispositivos PKCS#11, los certificados de estos se listarán junto con los del almacén interno de Mozilla. Este almacén sólo puede utilizarse si se encuentra disponible el módulo **afirma-keystores-mozilla**.
* **SHARED\_NSS**. Almacén de claves central de Linux. Funciona igual que el almacén MOZ\_UNI y también depende del módulo **afirma-keystores-mozilla**.
* **PKCS12**. Almacenes en disco acordes con el estándar PKCS#12. Para el uso de este manejador es necesario que se especifique la ruta en donde se encuentra del almacén que se solicita (fichero ".p12" y ".pfx").
* **JAVA**. Almacenes en disco acordes al formato Java de almacén de certificado (JKS). Para el uso de este manejador es necesario que se especifique la ruta en donde se encuentra del almacén que se solicita (fichero ".jks").
* **PKCS11**. Almacenes en dispositivo externo con controlador PKCS#11. Para el acceso a este tipo de dispositivos es necesario que se especifique la ruta de la biblioteca PKCS#11 (".dll", ".so", ".dlib",...) que lo controla.
* **SINGLE**. Certificados "sueltos" en disco. Es necesario que se indique la ruta del certificado concreto que desea cargar (".cer", ".p7b"...). Estos certificados carecen de clave privada, por lo que no pueden utilizarse para la firma de documentos, aunque sí como destinatarios de un sobre electrónico.
* **WINADDRESSBOOK**. Libreta de direcciones de Windows. Sólo disponible en sistemas Windows. Estos certificados carecen de clave privada, por lo que no pueden utilizarse para la firma de documentos, aunque sí como destinatarios de un sobre electrónico.
* **JCEKS**: Almacén de claves según formato Java. Permite el almacén de claves, como las de cifrado de datos. Requiere que se indique el fichero del almacén.
* **DNIEJAVA:** Almacén del DNIe al que se accederá mediante el la biblioteca JMulticard. Es necesario incluir esta biblioteca en el proyecto para poder utilizar este almacén.
* **KNOWN\_SMARTCARDS**: Almacén compuesto por diversas tarjetas inteligentes de las que se buscará el PKCS#11 en el sistema. El Cliente @firma utiliza este almacén cuando se carga desde un perfil temporal de Windows.

*es.gob.afirma.keystores.AOKeyStoreManagerFactory*

Es la factoría a partir de la cual se obtienen los distintos manejadores de almacén de certificados. El método para obtener estos manejadores es:

* **static AggregatedKeyStoreManager getAOKeyStoreManager(AOKeyStore store, String lib, String description, PasswordCallback pssCallback, Object parentComponent) throws AOKeystoreAlternativeException, IOException** 
  + Recibe el tipo de almacén deseado, la ruta de la biblioteca o almacén y su descripción si no se trata de un almacén de sistema, un *callback* a través del cual se pueda insertar la clave del almacén y un componente visual para la visualización de los posibles diálogos modales sobre el mismo. Puede lanzar una IOException si no puede acceder al almacén, o una AOKeyStoreAlternativeException si el almacén no está disponible y se propone una alternativa (declarada en la propia excepción).

**Ejemplo de uso**

|  |
| --- |
| ...  // Obtenemos el gestor del almacen de certificados de Mozilla Firefox  AOKeyStoreManager ksm = AOKeyStoreManagerFactory.getAOKeyStoreManager(  AOKeyStore.MOZ\_UNI,  **null**,  **null**,  AOKeyStore.MOZ\_UNI.getStorePasswordCallback(null),  **null**);  ... |

**Ejemplo de uso**

|  |
| --- |
| ...  // Creamos un PasswordCallback que ya incluye la contraseña su uso  // directo  PasswordCallback psc = **new** CachePasswordCallback("123456".getChars());  // Obtenemos el gestor para un almacén en fichero local de tipo PKCS12  // sin interacción del usuario (valido para su uso en servidor)  AOKeyStoreManager ksm = AOKeyStoreManagerFactory.getAOKeyStoreManager(  AOKeyStore.PKCS12,  "/ruta/almacen.pfx",  **null**,  psc,  **this**.parent);  ... |

*es.gob.afirma.keystores.AOKeyStoreManager*

Manejador para la gestión de un almacén de certificados. Un objeto de este tipo permite obtener los alias de los certificados del almacén, los propios certificados y sus claves.

Los métodos de principal interés de esta clase son:

* **String[] getAliases()**
  + Recupera un *array* con los alias de todos los certificados del almacén. Estos alias serán los que utilizaremos para obtener los certificados y las claves del almacén.
* **X509Certificate getCertificate(String alias)**
  + Recupera del almacén el certificado con el alias indicado.
* **X509Certificate[] getCertificateChain(String alias)**
  + Recupera del almacén todo lo posibles de la cadena de certificación del certificado con el alias indicado.
* **void setEntryPasswordCallBack(PasswordCallback pwc)**
  + Establece el PasswordCallback para la autorizar la obtención de las PrivateKeyEntry del almacén.
* **KeyStore.PrivateKeyEntry getKeyEntry(String alias) throws KeyStoreException, NoSuchAlgorithmException, UnrecoverableEntryException** 
  + Recupera del almacén la referencia a la clave privada del certificado cuyo alias se indica. Utiliza el PassworCallback indica mediante setEntryPasswordCallBack o, si no se indicó, el por defecto para el almacén de claves configurado. Se lanzará una excepción en caso de ocurrir un error en el tratamiento del almacén de claves (KeyStoreException), la clave que se intenta recuperar no sea válida (NoSuchAlgorithmException), no se consiga recuperar la clave (UnrecoverableEntryException) o el usuario cancele la operación (AOCancelledOperationException).

**Ejemplo de uso**

|  |
| --- |
| ...  // Recuperamos los alias del almacen y la referencia a clave privada  // del primero de ellos  PrivateKeyEntry scKeyEntry = **null**;  String[] aliases = ksm.getAliases();  **if** (aliases.length > 0) {  scKeyEntry = ksm.getKeyEntry(aliases[0]);  }  ... |

*es.gob.afirma.keystores.AOKeyStoreDialog*

Clase que controla el diálogo de selección de certificados. Su implementación depende de los módulos gráficos del proyecto: **afirma-ui-core-jse** y **afirma-ui-core-jse-keystores**.

La configuración acerca del almacén del que se deben mostrar los certificados y cuales de ellos se deben visualizar se establece por medio de sus constructores:

* **AOKeyStoreDialog(AOKeyStoreManager ksm, Object parentComponent, boolean checkPrivateKeys, boolean showExpiredCertificates, boolean checkValidity)**
  + Crea el diálogo para la selección de un certificado permitiendo configurar:
    - **ksm**: Manejador del almacén de claves al que acceder.
    - **parentComponent**: Componente gráfico sobre el que se muestra el diálogo.
    - **checkPrivateKeys**: Establece si deben mostrarse únicamente los certificados que posean clave privada con la cual firmar.
    - **showExpiredCertificates**: Establece si deben mostrarse también los certificados que se encuentren fuera de su periodo de validez.
    - **checkValidity**: Habilita que se muestre un diálogo de advertencia si se selecciona un certificado que pueda ser considerado no válido (fuera del periodo de validez,...).
* **AOKeyStoreDialog(AOKeyStoreManager ksm, Object parentComponent, boolean checkPrivateKeys, boolean showExpiredCertificates, boolean checkValidity, List<? extends CertificateFilter> certFilters, boolean mandatoryCertificate)**
  + Crea el diálogo para la selección de un certificado permitiendo configurar:
    - **ksm**: Manejador del almacén de claves al que acceder.
    - **parentComponent**: Componente gráfico sobre el que se muestra el diálogo.
    - **checkPrivateKeys**: Establece si deben mostrarse únicamente los certificados que posean clave privada con la cual firmar.
    - **showExpiredCertificates**: Establece si deben mostrarse también los certificados que se encuentren fuera de su periodo de validez.
    - **checkValidity**: Habilita que se muestre un diálogo de advertencia si se selecciona un certificado que pueda ser considerado no válido (fuera del periodo de validez,...).
    - **certFilters**: Listado de filtros de certificados. Sólo se mostrarán los certificados que pasen, al menos, por uno de los filtros indicados.
    - **mandatoryCertificate**: Establece que si sólo hay un certificado pasa el proceso de filtrado, o si no hay filtros y sólo hay un certificado en el almacén, se devuelva el alias de este certificado sin ni siquiera mostrara el diálogo de selección.

Para la visualización y uso del diálogo contamos principalmente con el método:

* **String show() throws AOCertificatesNotFoundException;**
  + Muestra el diálogo de selección de certificados según la configuración establecida en el constructor. Devuelve el alias del certificado seleccionado por el usuario, o directamente el único que hubiese que mostrar si se configuró el parámetro mandatory a true.
  + Si no se encuentran certificado se lanza la excepción AOCertificatesNotFoundException, mientras que si el usuario cierra el diálogo sin seleccionar un certificado se lanza una AOCancelledOperationException.

*es.gob.afirma.keystores.filters.CertificateFilter*

Clase abstracta de la que debe heredar un filtro de certificados. Existen varios filtros de certificados implementados en el módulo **afirma-keystores-filters**.

Los métodos que se deben implementar son:

* **boolean matches(X509Certificate cert)**
  + Debe devolver true si el certificado indicado cumple con los criterios establecidos en el filtro.
* **String[] matches(String[] aliases, AOKeyStoreManager ksm)** 
  + Debe devolver el listado con los alias de los certificados que cumplen con los criterios establecidos en el filtro.
  + Si no se implementa este método, se utilizará el método anterior por cada certificado individual si supera el filtro establecido, en cuyo caso pasa a la lista de certificados válidos.
  + Implementar este método es útil cuando se desean establecer condiciones especiales que afectan a diversos certificados. Por ejemplo, un certificado sólo se mostrará si está declarado como un certificado de firma o, si no está declarado como tal pero no hay ningún otro certificado con ese número de serie del *Subject*.

## **Módulo afirma-keystores-mozilla**

Módulo particular para el acceso al almacén de Java. Salvo que se desee obtener exclusivamente acceso al almacén de Mozilla Firefox, no es recomendable manipular directamente las clases de este módulo. Obtenga este manejador mediante la factoría de **afirma-core-keystores** indicando el tipo de almacén AOKeyStore.MOZ\_UNI.

Este almacén gestiona de forma unificada los certificados del almacén interno de Mozilla y los certificados de los dispositivos externos instalados en el almacén mediante su PKCS#11.

## **Módulo afirma-keystores-single**

Módulo particular para el manejo de certificados en disco no contenidos en ningún almacén de certificados. Salvo que se desee únicamente utilizar estos certificados, no es necesario manipular directamente las clases de este módulo. Obtenga este manejador mediante la factoría de **afirma-core-keystores** indicando el tipo de almacén AOKeyStore.SINGLE.

Los certificados gestionados mediante este manejador carecen de clave privada, por lo que no se pueden utilizar para realizar firmas electrónicas ni otras operaciones criptográficas. En cambio, pueden utilizarse para designar a los destinatarios de sobres electrónicos.

Para utilizar el diálogo gráfico de selección con los certificados de este almacén será necesario establecer correctamente el parámetro que determina si se deben mostrar los certificados que carecen de clave privada.

## **Módulo afirma-keystores-capiaddressbook**

Módulo particular para el acceso a la libreta de direcciones de sistemas Windows. Salvo que se desee únicamente utilizar los certificados de es almacén, no es necesario manipular directamente la clase de este módulo. Obtenga este manejador mediante la factoría de **afirma-core-keystores** indicando el tipo de almacén AOKeyStore.WINADDRESSBOOK.

Los certificados gestionados mediante este manejador carecen de clave privada, por lo que no se pueden utilizar para realizar firmas electrónicas ni otras operaciones criptográficas. En cambio, pueden utilizarse para designar a los destinatarios de sobres electrónicos.

Para utilizar el diálogo gráfico de selección con los certificados de este almacén será necesario establecer correctamente el parámetro que determina si se deben mostrar los certificados que carecen de clave privada.

# Módulos de firma

Un módulo de firma se compone del conjunto de clases que permiten la gestión de un formato de firma (generación, identificación y extracción de datos de firma) donde, al menos una de las cuales, debe implementar la interfaz **es.gob.afirma.core.signers.AOSigner** para permitir su uso de forma homogénea al resto de módulos.

El que el Cliente @firma soporte o no un formato de firma depende de si integra o no el módulo de firma que se encarga de la gestión de ese formato. Cada uno de los módulos tiene sus propias dependencias, de tal forma que una construcción del Cliente puede disponer de distintos módulos, no necesariamente todos ellos, mientras se incluyan las dependencias de cada uno.

Para gestionar firmas en un formato concreto deberemos utilizar las clases del módulo que implementa AOSigner y se encarga de la gestión de este formato. Para consultar cómo utilizar la clase AOSigner, revisa el apartado Interfaces de Utilidad de este documento.

Para gestionar de forma centralizada los formatos de firma disponibles utilice la clase **es.gob.afirma.core.signers.AOSignerFactory**. Puede consultar más información sobre esta clase en el apartado Factorías de este módulo.

A continuación se listan los distintos módulos de firma de los que dispone el Cliente @firma y los formatos que soporta:

## **Módulo afirma-crypto-cades**

Este módulo permite la generación y gestión de firmas en formato CAdES y firmas empaquetadas CAdES-ASiC-S. Todas las firmas CAdES generadas son acordes al formato B-Level.

Este módulo depende del *core* del proyecto (**afirma-core**) y del módulo de funciones de gestión de firmas binarias (**afirma-crypto-core-pkcs7**). El módulo **afirma-crypto-cades** no soporta las operaciones de cofirma y contrafirma para lo que requiere que se incluya el módulo **afirma-crypto-cades-multi** para estas funciones. Utilizaremos el módulo **afirma-crypto-cades** sin incluir el módulo **afirma-crypto-cades-multi** sólo cuando nuestra aplicación no realice nunca multifirmas CAdES. Como dependencias externas utiliza las bibliotecas de SpongyCastle.

El manejador de firma que implementa la interfaz AOSigner para el manejo de firmas CAdES es **es.gob.afirma.signers.cades.AOCAdESSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_CADES.

El manejador de firma que implementa la interfaz AOSigner para el manejo de firmas CAdES-ASiC-S es **es.gob.afirma.signers.cades.asic.AOCAdESASiCSSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_CADES\_ASIC\_S.

Las opciones de configuración específicas para la generación de firmas, cofirmas y contrafirmas CAdES se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.cades.AOCAdESExtraParams**.

## **Módulo afirma-crypto-cadestri-client**

Este módulo permite la generación de firmas en formato CAdES y empaquetados CAdES-ASiC-S de forma trifásica. Las firmas soportadas por este módulo son exactamente iguales a las del módulo **afirma-crypto-cades**. Este módulo sólo permite generar firmas de los formatos soportados y no implementa el resto de métodos de la interfaz AOSigner (detección de datos válidos, identificación de formatos de firma, extracción de los datos…)

Este módulo sólo tiene dependencia con el módulo *core* (**afirma-core**) y, para funcionar, requiere conectar con el servicio de firmas trifásicas del Cliente @firma.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas CAdES es **es.gob.afirma.signers.cadestri.client.AOCAdESTriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_CADES\_TRI.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas empaquetadas CAdES-ASiC-S es **es.gob.afirma.signers.cadestri.client.asic.AOCAdESASiCSTriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_CADES\_ASIC\_S\_TRI.

## **Módulo afirma-crypto-cms**

Este módulo permite la generación y gestión de firmas en formato CMS/PKCS#7.

Este módulo depende del *core* del proyecto (**afirma-core**) y de **afirma-crypto-core-pkcs7**, el módulo de funciones de gestión de firmas binarias. Como dependencias externas utiliza las bibliotecas de SpongyCastle.

El manejador de firma que implementa la interfaz AOSigner es **es.gob.afirma.signers.cms.AOCMSSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_CMS.

Las opciones de configuración específicas para la generación de firmas, cofirmas y contrafirmas CMS se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.cms.AOCMSExtraParams**.

## **Módulo afirma-crypto-pdf**

Este módulo permite la generación y gestión de firmas en formato PAdES/PDF. Las firmas PAdES generadas serán acordes al formato B-Level cuando se especifique el subfiltro ETSI.CAdES.detached a través de las propiedades extraParams.

Este módulo depende del *core* del proyecto (**afirma-core**), el módulo de funciones de gestión de firmas binarias (**afirma-crypto-core-pkcs7**), el módulo para la generación de sellos de tiempo en firmas binarias (**afirma-crypto-core-pkcs7-tsp**) y el módulo de generación de firmas CAdES (**afirma-crypto-cades**). Como dependencias externas utiliza las bibliotecas de SpongyCastle e iText (versión propia generada sobre la versión 2.1.7 de iText).

El manejador de firma que implementa la interfaz AOSigner es **es.gob.afirma.signers.pades.AOPDFSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_PADES.

Las opciones de configuración específicas para la generación de firmas y multifirmas PAdES se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.pades.PDFExtraParams**.

## **Módulo afirma-crypto-padestri-client**

Este módulo permite la generación de firmas en formato PAdES de forma trifásica. Las firmas soportadas por este módulo son exactamente iguales a las del módulo **afirma-crypto-pdf**. Este módulo sólo permite generar firmas de los formatos soportados y no implementa el resto de métodos de la interfaz AOSigner (detección de datos válidos, identificación de formatos de firma, extracción de los datos…)

Este módulo sólo tiene dependencia con el módulo *core* (**afirma-core**) y, para funcionar, requiere conectar con el servicio de firmas trifásicas del Cliente @firma.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas PAdES es **es.gob.afirma.signers.cadestri.client.AOPAdESTriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_PADES\_TRI.

## **Módulo afirma-crypto-xades**

Este módulo permite la generación y gestión de firmas en formato XAdES, firmas de Factura Electrónica (FacturaE) y firmas empaquetadas XAdES-ASiC-S. Todas las firmas XAdES generadas son acordes al formato B-Level.

Este módulo depende del *core* del proyecto (**afirma-core**) y de el módulo de funciones de gestión de firmas XML (**afirma-crypto-core-xml**). Como dependencia externa utiliza la biblioteca JXAdES.

Los manejadores de firma que implementa la interfaz AOSigner son:

* **es.gob.afirma.signers.xades.AOXAdESSigner**: Para las firmas XAdES.
* **es.gob.afirma.signers.xades.AOFacturaESigner**: Para la firma de facturas electrónicas.
* **es.gob.afirma.signers.xades.asic.AOXAdESASiCSSigner**: Para las firmas empaquetadas XAdES-ASiC-S.

Para obtener estos manejadores a partir de la factoría AOSignerFactory, deben utilizarse las constantes es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_XADES (para el formato XAdES) es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_FACTURAE (para el formato FacturaE) y es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_XADES-ASIC-S (para el formato XAdES-ASiC-S)

Las opciones de configuración específicas para la generación de firmas, cofirmas y contrafirmas XAdES se detallan en el manual del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.xades.AOXAdESExtraParams**. Salvo que se configure explícitamente el tipo de firma XAdES, se generarán firmas XAdES Enveloping.

Las opciones de configuración específicas para la generación de firmas de facturas electrónicas son un subconjunto de las permitidas en las firmas XAdES y se detallan en el manual de integración del Cliente @firma y el Javadoc del método sign() de la clase **es.gob.afirma.signers.xades.AOFacturaESigner**.

Las opciones de configuración específicas para la generación de firmas, cofirmas y contrafirmas XAdES-ASiC-S se detallan en el Javadoc de la clase **es.gob.afirma.signers.xades.AOXAdESASiCSExtraParams**.

## **Módulo afirma-crypto-xadestri-client**

Este módulo permite la generación de firmas en formato XAdES, FacturaE y empaquetados XAdES-ASiC-S de forma trifásica. Las firmas soportadas por este módulo son exactamente iguales a las del módulo **afirma-crypto-xades**. Este módulo sólo permite generar firmas de los formatos soportados y no implementa el resto de métodos de la interfaz AOSigner (detección de datos válidos, identificación de formatos de firma, extracción de los datos…)

Este módulo sólo tiene dependencia con el módulo *core* (**afirma-core**) y, para funcionar, requiere conectar con el servicio de firmas trifásicas del Cliente @firma.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas XAdES es **es.gob.afirma.signers.cadestri.client.AOXAdESTriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_XADES\_TRI.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas de factura electrónica es **es.gob.afirma.signers.cadestri.client.AOFacturaETriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_FACTURAE\_TRI.

El manejador de firma que implementa la interfaz AOSigner para la generación de firmas trifásicas empaquetadas XAdES-ASiC-S es **es.gob.afirma.signers.cadestri.client.asic.AOXAdESASiCSTriPhaseSigner** y la constante para obtener el manejador a partir de la factoría AOSignerFactory es es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_XADES\_ASIC\_S\_TRI.

## **Módulo afirma-crypto-xmlsignature**

Este módulo permite la generación y gestión de firmas en formato XMLdSign.

Este módulo depende del *core* del proyecto (**afirma-core**) y del módulo de funciones de gestión de firmas XML (**afirma-crypto-core-xml**).

El manejador de firma que implementa la interfaz AOSigner es **es.gob.afirma.signers.xmldsig.AOXMLDSigSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_XMLDSIG.

Las opciones de configuración específicas para la generación de firmas, cofirmas y contrafirmas XMLdSig se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.xmldsig.AOXMLDSigExtraParams**. Salvo que se configure explícitamente el tipo de firma XMLdSig, se generarán firmas XMLdSig Enveloping.

## **Módulo afirma-crypto-odf**

Este módulo permite la generación y gestión de firmas en formato ODF (Open Document Format).

Este módulo depende del *core* del proyecto (**afirma-core**) y del módulo de funciones de gestión de firmas XML (**afirma-crypto-core-xml**).

El manejador de firma que implementa la interfaz AOSigner es **es.gob.afirma.signers.odf.AOODFSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_ODF.

Las opciones de configuración específicas para la generación de firmas y multifirmas ODF se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.odf.AOODFExtraParams**.

## **Módulo afirma-crypto-ooxml**

Este módulo permite la generación y gestión de firmas en formato OOXML (Office Open XML).

Este módulo depende del *core* del proyecto (**afirma-core**) y del módulo de funciones de gestión de firmas XML (**afirma-crypto-core-xml**) y los módulos de firmas XAdES (**afirma-crypto-xades**) y XMLdSig (**afirma-crypto-xmlsignature**), de los que se vale para generar las firmas internas de OOXML.

El manejador de firma que implementa la interfaz AOSigner es **es.gob.afirma.signers.ooxml.AOOOXMLSigner**.

Para obtener este manejador a partir de la factoría AOSignerFactory, debe utilizarse la constante es.gob.afirma.core.signers.AOSignConstants.SIGN\_FORMAT\_OOXML.

Las opciones de configuración específicas para la generación de firmas y multifirmas OOXML se detallan en el manual de integración del Cliente @firma y el Javadoc de la clase **es.gob.afirma.signers.ooxml.AOOOXMLExtraParams**.

# Firma masiva

Los procesos de firma masiva consisten en la realización de varias firmas/multifirmas electrónicas por medio de una única operación de cara al usuario. Aunque este es un procedimiento que se puede programar directamente a partir de la repetición en bucle de las operaciones de firma, el Cliente @firma cuenta con 2 clases que gestionan buena parte de la lógica deseable de este tipo de operaciones.

Estas 2 clases se encuentran en el módulo **afirma-core-massive** y son:

* **es.gob.afirma.massive.DirectorySignatureHelper**: Para la gestión de firmas/multifirmas masivas de directorios.
* **es.gob.afirma.massive.MassiveSignatureHelper**: Para la gestión de firmas/multifirmas masivas de datos, fichero o hashes con una configuración prefijada.

## **Módulo afirma-core-massive**

*es.gob.afirma.massive.DirectorySignatureHelper*

Es la clase que permite la firma masiva de ficheros en disco. Esta clase cuenta con un constructor con el que se puede establecer el algoritmo, el formato por defecto y el modo de firma que se debe utilizar una vez creado el objeto pueden utilizarse las distintas funciones con las que cuenta para establecer la configuración de operación y ejecutarla. Los métodos más relevantes son:

* **void setFileFilter(java.io.FileFilter fileFilter)** 
  + Establece los filtros de fichero que deben pasar los ficheros de los directorios sobre los que se operan.
* **void setOverwritePreviuosFileSigns(boolean overwrite)** 
  + Establece si debe sobre escribirse cualquier fichero con el mismo nombre que una de las firmas generadas.
* **void setActiveLog(boolean activeLog)** 
  + Establece si debe generarse un fichero de log con el resultado. Por defecto, se genera.
* **void setLogPath(String path)** 
  + Establece la ruta del fichero de log. Por defecto, el fichero de salida de firmas.
* **boolean massiveSign(MassiveType type, String startDir, boolean recurse, String outDir, boolean createOutDir, boolean originalFormat, PrivateKeyEntry keyEntry, Properties config) throws AOException** 
  + Ejecuta la operación determinada por type sobre el directorio startDir y sus subdirectorios (en caso de que recurse sea true) y genera las firmas resultantes en outDir (creándolo si así se indica en createOutDir). Si se especifica originalFormat las multifirmas se realizarán con el formato de la firma original en lugar del establecido en el constructor. La clave de firma es keyEntry y config determina la configuración de firma.

**Ejemplo de uso:**

|  |
| --- |
| ...  // Creamos el gestor de firmas de directorios  DirectorySignatureHelper massiveHelper =  **new** DirectorySignatureHelper(  AOSignConstants.SIGN\_ALGORITHM\_SHA1WITHRSA,  AOSignConstants.SIGN\_ALGORITHM\_PADES,  AOSignConstants.SIGN\_MODE\_IMPLICIT);  // Restringimos la firma solo a ficheros PDF  massiveHelper.setFileFilter(**new** FileFilter {  **public** **boolean** accept(File file) {  **return** file.getName().toLowerCase().endsWith(".pdf");  }  });  // Desactivamos la generación del fichero de log  massiveHelper.setActiveLog(**false**);  // Establecemos una configuracion de firma  Properties config = **new** Properties();  config.setProperty("signReason", "Prueba");  config.setProperty("signatureProductionCity", "Madrid");  config.setProperty("signerContact", "fulanito@mail.es");  // Ejecutamos una operación masiva de firma sobre todos  // los ficheros del directorio indicado  massiveHelper.massiveSign(  MassiveType.SIGN, "C:/pruebas", **true**, "C:/salida", **true**,  **true**, pke, config);  ... |

* **boolean massiveSign(MassiveType type, String[] filenames, String outDir, boolean createOutDir, boolean originalFormat, PrivateKeyEntry keyEntry, Properties config) throws AOException** 
  + Ejecuta la operación determinada por type sobre los ficheros de los que se proporciona la ruta mediante filenames. Genera las firmas resultantes en outDir (creándolo si así se indica en createOutDir). Si se especifica originalFormat las multifirmas se realizarán con el formato de la firma original en lugar del establecido en el constructor. La clave de firma es keyEntry y config determina la configuración de firma.

*es.gob.afirma.massive.MassiveSignatureHelper*

Esta es la clase que contiene la lógica para la realización de firmas masivas de modo programático. La clase se construye con una configuración completa con todos los parámetros necesarios para la ejecución de la operación masiva. Para aquellos parámetros que no estén establecidos se utilizará la configuración por defecto. En ningún caso debe bloquearse la ejecución mostrando un diálogo visual al usuario o solicitándole información.

El constructor de la clase y los métodos de mayor interés son:

* **MassiveSignatureHelper(MassiveSignConfiguration configuration) throws AOException** 
  + Construye el objeto para la ejecución de la operación masiva proporcionándole la configuración completa de firma. Durante la construcción del objeto se establecen los valores para la operación y se selecciona el manejador de firma a partir del formato configurado. Si no existe ningún manejador disponible para el formato indicado, se lanzará una excepción.
* **void setMassiveOperation(MassiveType massiveOperation)** 
  + Permite modificar, después de la construcción del objeto e incluso iniciado el proceso masivo, la operación que se debe ejecutar (firma, cofirma, contrafirma de nodos hoja o contrafirma completa).
* **void setSignatureFormat(String signatureFormat)** 
  + Permite modificar, después de la construcción del objeto e incluso iniciado el proceso masivo, el formato de firma que ese debe utilizar. Este método sólo afecta a la operación de firma masiva. A ninguno de los procesos de multifirma.
* **byte[] signData(byte[] data)** 
  + Firma/multifirma los datos indicados con la configuración establecida por defecto. Si es una firma masiva y se configuró un nuevo formato con setSignatureFormat(String) después de la construcción del objeto, se utilizará este último.
* **byte[] signFile(String fileUri)** 
  + Firma/multifirma el contenido del fichero indicado con la configuración establecida por defecto.
* **byte[] signHash(byte[] hash)** 
  + Firma el hash indicado con la configuración establecida por defecto. Este método sólo puede utilizarse cuando la operación masiva configurada sea la firma, ni cofirma ni contrafirma.
* **String getCurrentLogEntry()** 
  + Recupera un texto descriptivo con el resultado de la última operación ejecutado mediante los métodos signData(byte[]), signFile(String) o signHash(byte[]).

**Ejemplo de uso**

|  |
| --- |
| ...  // Establecemos la configuracion propia del formato  Properties extraParams = **new** Properties();  extraParams.setProperty("includeOnlySignningCertificate", "true");  // Establecemos la configuracion de la operación  MassiveSignConfig config = **new** MassiveSignConfig(pke);  config.setMassiveOperation(MassiveType.COUNTERSIGN\_LEAFS);  config.setSignatureFormat(AOSignConstants.SIGN\_FORMAT\_CADES);  config.setOriginalFormat(**true**);  config.setExtraParams(extraParams);  MassiveSignatureHelper massiveHelper =  **new** MassiveSignatureHelper(config);  // Contrafirmamos los ficheros cuyas rutas se encuentran en  // el array FILES.  // Si es un PDF, lo cofirmamos en su lugar  **byte**[] result = **null**;  **for** (String path : FILES) {  **if** (path.toLowerCase().endsWith(".pdf")) {  massiveHelper.setSignatureFormat(  AOSignConstants.SIGN\_FORMAT\_PADES);  result = massiveHelper.signFile(path);  massiveHelper.setSignatureFormat(  AOSignConstants.SIGN\_FORMAT\_CADES);  }  **else** {  result = massiveHelper.signFile(path);  }  // Operamos como corresponda con el resultado  procesar(result);  }  ... |

# Módulos de cifrado

A diferencia de los módulos de firma, el Cliente @firma sólo cuenta actualmente con un módulo de cifrado simétrico de datos. Este módulo es el que se utiliza en los procesos de cifrado del *applet* Cliente.

Un módulo de cifrado debe poseer al menos una clase que implemente la interfaz **es.gob.afirma.core.ciphers.AOCipher**, así es posible utilizar de forma homogénea todos los manejadores con las funciones de cifrado que se puedan crear para uso del Cliente.

Actualmente no existe ninguna factoría que permita recuperar los distintos manejadores de funciones de cifrado por lo que se deberán utilizar estos directamente a través de su clase AOCipher.

## **Módulo afirma-crypto-cipher**

Este módulo de cifrado está basado en el proveedor de seguridad SunJCE de Java. El módulo mantiene internamente la lista de configuraciones soportadas, tomadas estas de las disponibles en Java 6, por lo que no será posible utilizar otro tipo de configuraciones de cifrado. Los algoritmos soportados son:

* AES
* ARCFOUR
* Blowfish
* DES
* DESede
* RC2
* PBEWithMD5AndDES
* PBEWithSHA1AndDESede
* PBEWithSHA1AndRC2\_40

Cada uno de estos algoritmos soporta varias configuraciones de modo y *padding*, pero de no indicarse se utilizará la que por defecto define el proveedor.

Este módulo sólo tiene como dependencia al módulo con el núcleo del proyecto (**afirma-core**).

La clase del módulo que implementa la interfaz AOCipher es **es.gob.afirma.ciphers.jce.AOSunJCECipher**.

Además del propio manejador de funciones de cifrado, este módulo incorpora una clase para la creación y gestión de una almacén de claves de cifrado, AOCipherKeyStoreHelper.

*es.gob.afirma.ciphers.AOCipherKeyStoreHelper*

Esta clase permite la gestión de un único almacén de claves por usuario del sistema. Este almacén se sitúa inevitablemente en el fichero *ciphkeys.jceks*, situado en el directorio del usuario. Esta clase no utiliza ningún tipo de interfaz gráfico, por lo que advertencias y cualquier otro tipo de acceso deberán realizarse de forma externa. Los métodos principales para la gestión de este almacén son:

* **static boolean storeExists()** 
  + Método estático para comprobar la existencia del almacén de claves del usuario actual.
* **AOCipherKeyStoreHelper(char[] p) throws AOException, IOException, GeneralSecurityException** 
  + Constructor para obtener acceso al almacén de claves del usuario o para crearlo en caso de que no exista.
* **String[] getAliases()** 
  + Lista los alias de las claves del almacén.
* **Key getKey(String alias) throws AOException** 
  + Recupera una clave de cifrado del almacén, a partir de su alias.
* **void storeKey(String alias, Key key) throws AOException** 
  + Almacena una clave en el almacén identificándola con el alias indicado.

**Ejemplo de uso**

|  |
| --- |
| ...  **char**[] password = **null**;  // Si el almacen existe solicitamos la contrasena establecida, si no  // existe solicitamos la contrasena para su creacion  **if** (AOCipherKeyStoreHelper.storeExists()) {  password = mostrarDialogoSolicitarContrasenaParaApertura();  } **else** {  password = mostrarDialogoSolicitarContrasenaParaCreacion();  }  AOCipherKeyStoreHelper cipherKs = AOCipherKeyStoreHelper(password);  // Listamos los alias del almacen y recuperamos la clave del primero  // de ellos  Key key = **null**;  String[] aliases = cipherKs.getAliases();  **if** (aliases.length > 0) {  key = cipherKs.getKey(aliases[0]);  }  **return** key;  ... |

# Ejemplos generales

## **Selección de almacén y firma**

**Secuencia:**

1. Selecciona un almacén de certificados en base al sistema operativo en el que se ejecute, car
2. Carga el almacén de claves.
3. Muestra un diálogo de selección de certificados.
4. Obtiene el alias del certificado seleccionado.
5. Selecciona el manejador de firma del formato CAdES.
6. Genera una firma de implícita de los datos proporcionado.
7. Devuelve el resultado.

**Código:**

|  |
| --- |
| **public** **byte**[] firma(**byte**[] datos, Component componentePadre)  **throws** KeyStoreException, NoSuchAlgorithmException,  UnrecoverableEntryException, AOKeystoreAlternativeException,  IOException, AOCertificatesNotFoundException, AOException {  // Selección del almacén de claves por defecto en base al  // sistema operativo  AOKeyStore ks;  **switch** (Platform.*getOS*()) {  **case** ***WINDOWS***:  ks = AOKeyStore.***WINDOWS***; // Almacén de Windows  **break**;  **case** ***MACOSX***:  ks = AOKeyStore.***APPLE***; // Llavero de macOS  **break**;  **case** ***LINUX***:  ks = AOKeyStore.***MOZ\_UNI***; // Almacén de Mozilla  **break**;  **default**:  ks = AOKeyStore.***PKCS12***; // Almacén en fichero P12/PFX  }  // Carga del almacén de claves  **final** AOKeyStoreManager ksm = AOKeyStoreManagerFactory.*getAOKeyStoreManager*(  ks,  **null**,  "Almacén de claves PKCS#12", // Este texto sólo se mostrará para PKCS#12  ks.getStorePasswordCallback(componentePadre),  componentePadre);  // Creamos el diálogo de selección de certificados  **final** AOKeyStoreDialog dialog = **new** AOKeyStoreDialog(  ksm,  componentePadre,  **true**, // Sólo certificados con claves privadas  **true**, // No mostrar los caducados  **false**); // Mostrar advertencia cuando se seleccionase alguno caducado  // Mostramos el dialogo de selección  **final** String selectedAlias = dialog.show();  // Si no se selecciono un certificado, se aborta la operacion  **if** (selectedAlias == **null**) {  **throw** **new** AOCancelledOperationException(  "Operacion cancelada por el usuario");  }  // Obtención de la clave de firma  **final** PrivateKeyEntry pke = ksm.getKeyEntry(selectedAlias);  // Seleccion del manejador de firma  **final** AOSigner signer =  AOSignerFactory.*getSigner*(AOSignConstants.***SIGN\_FORMAT\_CADES***);  // Configuramos las propiedades que deseemos del formato  **final** Properties extraParams = **new** Properties();  // Firma implicita (Attached)  extraParams.setProperty(CAdESExtraParams.***MODE***, "implicit"); //$NON-NLS-1$  **final** **byte**[] firma = signer.sign(  datos, // Datos que deseamos firmar  AOSignConstants.***SIGN\_ALGORITHM\_SHA256WITHRSA***, // Algoritmo de firma  pke.getPrivateKey(), // Referencia a la clave privada  pke.getCertificateChain(), // Cadena de certificación  extraParams); // Parámetros extra de configuración  **return** firma;  } |

**Módulos requeridos (además de sus dependencias):**

* afirma-core
* afirma-core-keystore
* afirma-keystores-mozilla
* afirma-crypto-cades
* afirma-ui-core-jse-keystores

[Creative Commons License](#Licencia_Creative_Commons)

Esta obra está bajo una licencia [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported](#Licencia_Creative_Commons).