Manual del integrador del

Client e

afirma

Índice

1 Introducción

El Cliente de Firma es una herramienta de Firma Electrónica que funciona en forma de Applet de Java integrado en una página Web mediante JavaScript.

El Cliente hace uso de los certificados digitales X.509 y de las claves privadas asociadas a los mismos que estén instalados en el repositorio o almacén de claves y certificados (*KeyStore*) del navegador web (*Internet Explorer, Mozilla, Firefox*) o el sistema operativo así como de los que estén en dispositivos (tarjetas inteligentes, dispositivos *USB*) configurados en el mismo (el caso de los DNI-e).

El Cliente de Firma, como su nombre indica, es una aplicación que se ejecuta en cliente (en el ordenador del usuario, no en el servidor Web). Esto es así para evitar que la clave privada asociada a un certificado tenga que "salir" del contenedor del usuario (tarjeta, dispositivo USB o navegador) ubicado en su PC. De hecho, nunca llega a salir del navegador, el Cliente le envía los datos a firmar y éste los devuelve firmados.

El Cliente de Firma contiene las interfaces y componentes web necesarios para la realización de los siguientes procesos (además de otros auxiliares como cálculos de hash, lectura de ficheros, etc…):

- Firma de formularios Web.
- Firma de datos y ficheros.
- Multifirma masiva de datos y ficheros.
- Co-firma (CoSignature)→ Multifirma al mismo nivel.
- Contrafirma (CounterSignature)→ Multifirma en cascada.

Como complemento al cliente de firma, se encuentra un cliente de cifrado que nos permite realizar las funciones de encriptación y desencriptación de datos atendiendo a diferentes algoritmos y configuraciones. Además permite la generación de sobres digitales.

El Cliente viene acompañadode un BootLoader independiente que, además de cargar el cliente, permite instalaren disco local las librerías que necesita.En cada carga, el instalador comprueba si ya están instaladas y se trata de una versión adecuada, y en caso contrario las instalará automáticamente en el sistema, previo consentimiento y aceptación de las condiciones de uso por parte del usuario.

El Cliente se distribuye en 3 construcciones distintas (LITE, MEDIA y COMPLETA) de tal forma que un usuario no tiene la necesidad de descargar en su sistema una construcción más pesada que incorpore características que no necesite.

Las funcionalidades de las que dispone cada una de estas construcciones son:

- Construcción LITE: Soporta firmas sin formato, CMS/PKCS#7 y CADES, e incorpora todas las capacidades comunes del cliente (firmas, cifrados, acceso a repositorios…).
- Construcción MEDIA: Soporta firmas XMLDSig, XAdES, Facturae, ODF y OOXML, más las funcionalidades de la construcción LITE.

- Construcción COMPLETA: Soporta firmas PDF, además de disponer de las funcionalidades de la construcción MEDIA.

2 Objeto y alcance

El presente documento recoge la descripción del cliente @firma y todas sus funcionalidades, así como la información necesaria para permitir a los integradores del cliente incorporarlo como parte de sus aplicaciones Web para la realización de operaciones criptográficas.

Los aspectos detalladosque se tratan del Cliente de Firmason los siguientes:

- Requisitos del Cliente
  - Sistemas operativos soportados
  - Navegadores soportados
  - Otros requisitos
- Componentes del Cliente
  - Applet *BootLoader*/instalador (cliente ligero)
  - Applet de firma (Componentes de Administración Electrónica)
- Funcionalidad del Instalador
  - Instalar / Actualizar ciertas dependencias del Cliente
- Funcionalidad básica del Cliente:
  - Firma
  - Firma masiva de hashes
  - Multifirma masiva de ficheros
  - Multifirma masiva programática
  - Co-firma
  - Contrafirma
  - Cifrado y descifrado de datos
  - Generación de sobres digitales.
  - Apertura de sobres.
- Configuración del cliente:
  - Algoritmos y formatos
  - Selección de certificados
- Otras funcionalidades
- Ejemplos que abarquen los aspectos anteriormente descritos.

3 Requisitos mínimos

- Sistema Operativo
  - Windows XP SP3 / Vista SP2 / 7 SP1 / Server 2003 SP2 / Server 2008SP2 / 8 y superiores
    - El Applet Cliente @firma no es compatible con Windows 8 RT.
  - Linux 2.6 (Guadalinex y Ubuntu) y superiores.
  - Mac OS X 10.6.8 y superiores (Snow Leopard, Lion y Mountain Lion).
- Navegador web:
  - Firefox 3.0 y superiores.
  - Internet Explorer 7 o superior, en 32 y 64 bits.
  - Google Chrome 4 o superior (no soportado em Mac OS X)
  - Apple Safari 4 o superior (únicamente soportado em Mac OS X)
- JRE:
  - JRE 5 (1.5 update 22) instalado en el navegador.
    - **Advertencias de uso**:
      - El Cliente @firma v3.3.1 será la última versión del Cliente compatible con esta versión de Java.
      - Java 5 no es compatible con Internet Explorer 9, Google Chrome, Apple Safari, ni Firefox 3.6 y superiores.
      - No es posible utilizar Java 5 en sistemas Windows Vista/7 por la restricción de permisos que impide la instalación de las dependencias necesarias.
  - JRE 6 de 32 bits (1.6 update 38 recomendada) instalado en el navegador.
  - JRE 7 de 32 bits o 64 bits (1.7 update 10 recomendada) instalado en el navegador.
    - Se desaconseja el uso de Java 7 update 5 debido a que esta puede causar problemas en la carga del *applet* Cliente.
- Certificado digital de usuario instalado en el navegador / sistema operativo o disponible a través de un módulo PKCS#11 o CSP instalado en el navegador (caso del DNI-e).

El Cliente siempre accederá al almacén de certificados del sistema operativo en el que se ejecute, salvo cuando se ejecute sobre Mozilla Firefox, en cuyo caso accederá al almacén de este navegador.

3.1 ¿Qué versión de mi navegador Web debo usar con mi sistema operativo?

A continuación se muestra la tabla de compatibilidad de versión de navegador Web según producto y sistema operativo.

Es importante recalcar que algunas de las celdas reflejan configuraciones no certificadas por Oracle como compatibles con JSE. Esto quiere decir que, si bien se han hecho las pruebas pertinentes por parte del Cliente @firma para asegurar su correcto funcionamiento, pueden existir problemas no detectados de compatibilidad de JSE con esa versión de navegador en ese sistema operativo, por lo que no se dará soporte a esa combinación mientras Oracle no la certifique.

	Internet Explorer	Google Chrome	Mozilla Firefox	Apple Safari	Opera
Windows XP SP3	7, 8	4, 10 o superior	3, 3.5.x, 3.6.x, 4, 5 o superior	No soportado	No soportado
Windows Vista SP2	8, 9	4, 10 o superior	3, 3.5.x, 3.6.x, 4, 5 o superior	No soportado	No soportado
Windows 7 SP1	8, 9 o superior	4, 10 o superior	3, 3.5.x, 3.6.x, 4, 5 o superior	No soportado	No soportado
Mac OS X Snow Leopard / Lion	N/A	No soportado	3, 3.5.x, 3.6.x, 4, 5 o superior	5 o superior	No soportado
Linux	N/A	4, 10 o superior[1]		N/A	No soportado

3.1.1 Internet Explorer 10 en Windows 8

El Applet Cliente @firma no es compatible con Internet Explorer 10 en su versión Metro, y debe ser ejecutado con la versión de escritorio de internet Explorer 10.

Para automatizar en cierta manera el cambio de Internet Explorer desde Metro hasta el escritorio clásico de Windows 8 se debe incluir la siguiente meta-información en la cabecera de la página HTML:

```
<metahttp-equiv="X-UA-Compatible"content="requiresActiveX=true"/>
```

Puede encontrar más información sobre complementos de navegador (*plugins*) en Internet Explorer 10 sobre Metro en Windows 8 en:

- http://msdn.microsoft.com/en-us/library/ie/hh968248%28v=vs.85%29.aspx

3.2 ¿Qué versión de Java debo usar en Linux?

Existen múltiples versiones de Linux, cada una de las cuales, introduce cambios que pueden afectar al funcionamiento del Cliente @firma.

Según la distribución y versión utilizada de Linux puede funcionar adecuadamente una u otra versión de Java. Se recomienda, por su mayor soporte, que en Linuxse utilice siempre la JRE 6 de Oracle.

3.3 ¿Qué versión de Java debo usar con el navegador Web Mozilla Firefox?

A continuación se nuestra la tabla de compatibilidad de versiones de Java (distinguiendo entre Java 1.6 y 1.7) según versión de Mozilla Firefox (con independencia del sistema operativo y la arquitectura).

En ciertas celdas se indica que la combinación no está certificada por Oracle, lo cual significa que, aunque se han hecho las pertinentes pruebas de correcto funcionamiento con el Cliente @firma, no se da soporte a esa combinación.

	Java 6	Java 7
Firefox 3	6u16 o superior	NO CERTIFICADO
Firefox 3.5.1	6u16 o superior	NO CERTIFICADO
Firefox 3.6	6u18 o superior	7u07 o superior
Firefox 4	6u25 o superior	7u07 o superior
Firefox 5	6u27 o superior	7u07 o superior

Firefox 7 y superiores	NO CERTIFICADO	7u07 o superior

3.4 ¿Qué versión de Java debo usar con el navegador Microsoft Internet Explorer?

A continuación se nuestra la tabla de compatibilidad de versiones de Java (distinguiendo entre Java 1.6 y 1.7) según versión de Internet Explorer (con independencia de la versión y arquitectura de Windows).

En ciertas celdas se indica que la combinación no está certificada por Oracle, lo cual significa que, aunque se han hecho las pertinentes pruebas de correcto funcionamiento con el Cliente @firma, no se da soporte a esa combinación.

	Java 6	Java 7
Internet Explorer 7	6u13 o superior	7u07 o superior
Internet Explorer 8	6u13 o superior	7u07 o superior
Internet Explorer 9	6u25 o superior	7u07 o superior
Internet Explorer 9 (64 bits)	--	7u07(64 bits) o superior

3.5 ¿Qué versión de Java debo usar con el navegador Google Chrome?

A continuación se nuestra la tabla de compatibilidad de versiones de Java (distinguiendo entre Java 1.6 y 1.7) según versión de Google Chrome(con independencia del sistema operativo y la arquitectura).

En ciertas celdas se indica que la combinación no está certificada por Oracle, lo cual significa que, aunque se han hecho las pertinentes pruebas de correcto funcionamiento con el Cliente @firma, no se da soporte a esa combinación.

Recuerde que el Cliente @firma no soporta Google Chrome en Mac OS X.

	Java 6	Java 7
Google Chrome 4	6u21 o superior	7u07 o superior
Google Chrome 10	6u25 o superior	7u07 o superior
Google Chrome 11 y superiores	NO CERTIFICADO	7u07 o superior

3.6 ¿Qué versión de Java debo usar con el navegador Apple Safari?

A continuación se nuestra la tabla de compatibilidad de versiones de Java (distinguiendo entre Java 1.6 y 1.7) y la versión según versión de Google Chrome (con independencia del sistema operativo y la arquitectura).

En ciertas celdas se indica que la combinación no está certificada por Oracle, lo cual significa que, aunque se han hecho las pertinentes pruebas de correcto funcionamiento con el Cliente @firma, no se da soporte a esa combinación.

	Java 6	Java 7
Apple Safari (Mac OS X)	NO CERTIFICADO	7u10 o superior
Apple Safari (Windows XP)	NO COMPATIBLE	NO CERTIFICADO
Apple Safari (Windows 7)	NO CERTIFICADO	NO CERTIFICADO

Para el uso del navegador Apple Safari en cualquier sistema operativo se recomienda tener instalado Java 7u4 o superior. La compatibilidad del Cliente @firma sobre Apple Safari en Windows está limitada por la compatibilidad del navegador con el plugin de Java, por lo que se recomienda el uso de otro navegador en Windows.

3.7 ¿Qué versión de Java debo usar con la variante de 64 bits de mi Navegador Web?

A continuación se nuestra la tabla de compatibilidad de versiones de Java en 64 bits según combinación de versión de 64 bits de sistema operativo y versión de 64 bits de navegador Web. Es importante recalcar que este es un caso excepcional, ya que, incluso si el sistema operativo es de 64 bits, es posible (y de hecho lo normal), usar un navegador Web de 32 bits con java de 32 bits, con lo que no aplicaría esta matriz.

Solo se da soporte a arquitecturas de 64 bits basadas en x64 (también llamada “Intel 64”, “x86-64”, “AMD 64” o “EM64T”).

En ciertas celdas se indica que la combinación no está certificada por Oracle, lo cual significa que, aunque se han hecho las pertinentes pruebas de correcto funcionamiento con el Cliente @firma, no se da soporte a esa combinación.

	Internet Explorer 64	Google Chrome 64	Mozilla Firefox 64	Apple Safari 64	Opera 64
Windows 64-bit (x64)	7u07 y superior	N/A	N/A	N/A	N/A
Mac OS X 64-bit	N/A	N/A	N/A	6u30 y superior	N/A
Linux 64-bit (x64)	N/A	N/A	N/A	N/A	NO CERTIFICADO

Internet Explorer 64 sólo puede utilizarse con seguridad en Java 7 de 64bits ya que las versiones de Java 6 64 bits no incluyen las bibliotecas necesarias para el acceso de los almacenes de Windows y Mozilla y los permisos del sistema del usuario pueden bloquear su instalación.

Las celdas marcadas con “N/A” indican que no está disponible una versión final de navegador Web para arquitecturas x64. No se da soporte a ningún tipo de versión preliminar (“alpha”, “beta”, “reléase candidate”, “nightly build”, etc.).

3.8 ¿Qué versión de Java debo usar con Mac OS X?

	Apple Java 6	Oracle Java 7
Snow Leopard	OK	NO CERTIFICADO
Lion hasta 10.7.3	OK	NO CERTIFICADO
Lion 10.7.4 y superiores	OK[1]	OK
Mountain Lion 10.8.1 y superiores	OK[1]	OK

[1]Como norma general, en Mac OS X debe siempre usarse el entorno de ejecución de Java distribuido desde la “Actualización de Software” de Mac OS X (que debe actualizarse periódicamente).

No obstante, los cambios de seguridad incorporados por Apple a la actualización 10.7.4 de Mac OS X Lion pueden causar problemas aleatorios en la obtención de privilegios de los Applets Java firmados. Si experimenta problemas ejecutando el Applet Cliente @firma en Mac OS X 10.7.4 puede actualizar su entorno de ejecución de Java a la versión 7 usando la versión de Oracle, disponible para libre descarga desde:

- http://www.oracle.com/technetwork/java/javase/downloads/index.html

Adicionalmente, aunque Java esté correctamente instalado, puede ser necesaria la activación del soporte específico de Applets de Java y aplicaciones Java WebStart. Esta activación puede realizarse desde “Preferencias de Java”, en el menú “Utilidades” de Mac OS X:

3.8.1 Applets de Java en versiones posteriores a la actualización 2012-006 de Apple

La actualización 2012-006 de Apple Java para OS X deshabilita por completo la ejecución de Applets de Java y aplicaciones Java WebStart en navegadores Web (con el JRE de Apple), introduciendo una incompatibilidad total con el Cliente @firma.

Puede solventar este inconveniente de dos formas alternativas:

1. Volviendo a habilitar manualmente el soporte de Applets de Java y aplicaciones Java WebStart siguiendo las instrucciones descritas en la siguiente página Web:
   http://support.apple.com/kb/HT5559

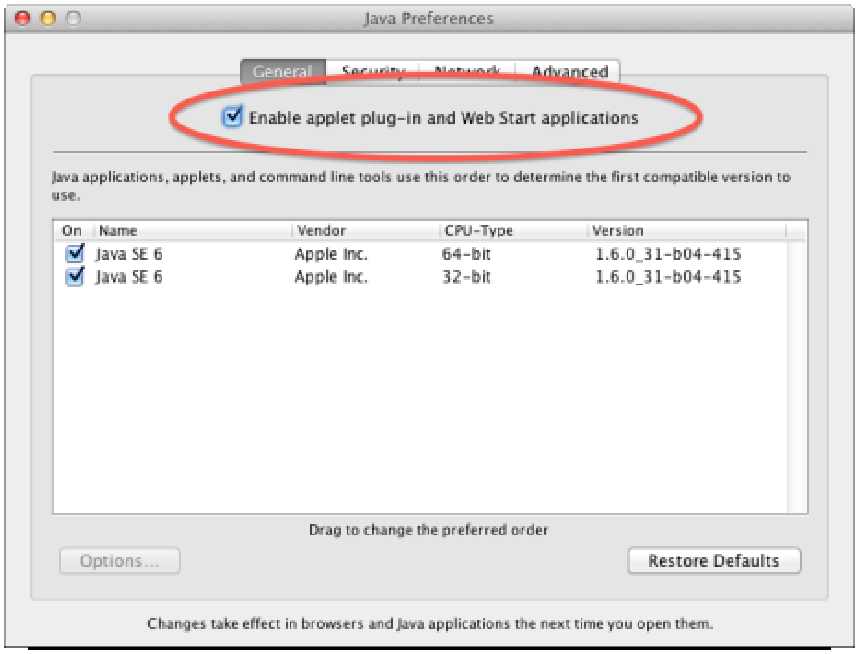
2. Instalando Oracle JRE 7 para Mac OS X

Es importante tener en cuenta que Oracle JRE 7 es incompatible con las versiones de 32 bits del navegador Web Google Chrome (las únicas actualmente disponibles).

3.8.2 Applets de Java en versiones posteriores a la actualización 2012-003 de Apple

Por defecto, tras instalar la actualización de Java 2012-003 de Apple, Mac OS X no permite la ejecución de Applets o aplicaciones Java Web Start, lo cual provoca que el Applet Cliente @firma no funcione.

Para habilitar los Applets de Java y las aplicaciones Web Start en Mac OS X es necesario indicarlo desde el panel de “Preferencias de Java” dentro de las Preferencias generales de Mac OS X y marcar la casilla “Activar módulo de Applet y aplicaciones Web Start”.

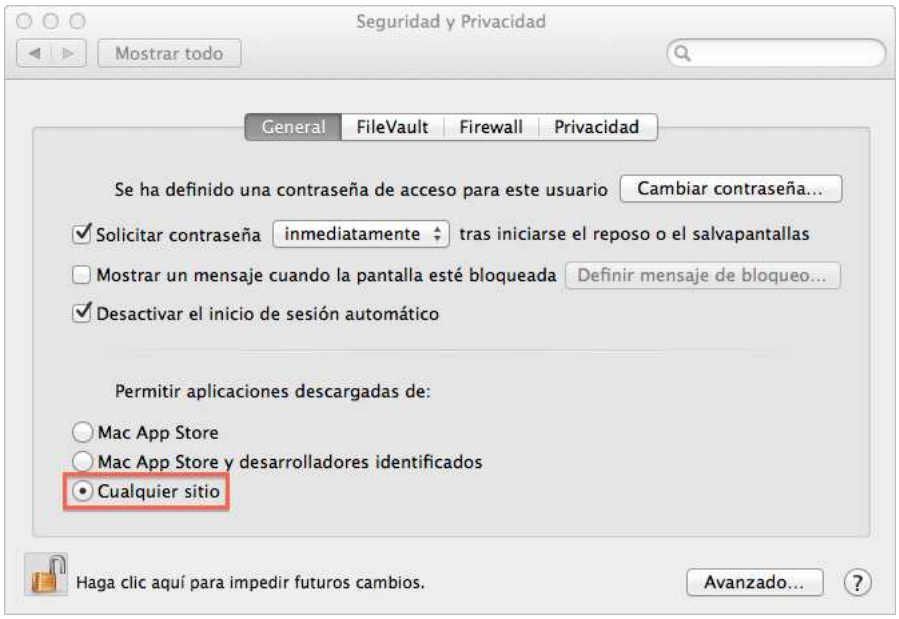


Como medida de seguridad, si el usuario no ejecuta Applets de Java por un periodo de tiempo prolongado, Mac OS X deshabilita automáticamente la ejecución de Applets y aplicaciones Java Web Start, por lo que será necesario comprobar que esta ejecución está permitida antes de iniciar el Applet Cliente @firma, independientemente de si esta ejecución ya fue habilitada anteriormente.

3.8.3 Applets de Java en Mac OS X Mountain Lion (10.8.x)

Mac OS X Mountain Lion introduce, como medida de seguridad una restricción a la ejecución de aplicaciones descargadas a través de Internet, como son los Applets de Java.

Por defecto, Mac OS X no permite esta ejecución a menos las aplicaciones se hayan descargado a través de la Apple Mac App Store (o eventualmente que el desarrollador que firma la aplicación esté autorizado por la propia Apple). Para permitir la ejecución del Applet @firma descargado desde una página Web normal, es necesario indicarlo mediante la opción de Seguridad y Privacidad (dentro de Preferencias) de Mac OS X marcando la opción “Permitir aplicaciones descargadas de: Cualquier sitio”.



3.9 Información adicional

- http://www.oracle.com/technetwork/java/javase/config-417990.html
- http://www.oracle.com/technetwork/java/javase/system-configurations-135212.html
- http://www.oracle.com/technetwork/java/javase/6u10-142936.html
- http://www.oracle.com/technetwork/java/javase/6u12-137788.html
- http://www.oracle.com/technetwork/java/javase/6u18-142093.html
- http://www.oracle.com/technetwork/java/javase/6u21-156341.html
- http://www.oracle.com/technetwork/java/javase/6u25releasenotes-356444.html
- http://www.oracle.com/technetwork/java/javase/6u29-relnotes-507960.html

4 Componentes del Cliente

4.1 Cliente

El cliente se compone de:

- **Clases** de la aplicación, agrupadas en ficheros *.jar*y *.jar.pack.gz*almacenados en un directorio de usuario o en un directorio en el servidor.
- **Librerías** (bibliotecas) adicionales almacenadas durante el proceso de instalación en directorios del sistema. En el servidor y en la distribución para su instalación en local se almacenan en ficheros comprimidos ZIP o JAR.
- **Bibliotecas JavaScript**: Contienen funciones para la automatización de los procesos de firma. Almacenadas en el servidor Web. Son opcionales y se puede operar sin ellas, pero facilitan los procesos más comunes.
  - **El conjunto principal de bibliotecas JavaScript no están diseñadas para ser modificadas directamente por el integrador excepto en caso de necesidades muy específicas.** No obstante, existe una biblioteca JavaScript llamada *constantes.js* que sí contiene parámetros modificables que permiten una mayor personalización del comportamiento del cliente.

4.2 Instalador

El instalador, también llamado *BootLoader* es el encargado de copiar ciertas bibliotecas necesarias para la correcta ejecución del Cliente @firma en cada plataforma específica.

El instalador tiene sus propias clases Java y sus bibliotecas JavaScript.

NOTA IMPORTANTE:

Si el integrador quisiese prescindir de la copia local de ficheros Java al disco del usuario, para así adecuarse mejor a las buenas prácticas de Java, deberá tener en cuenta los siguientes aspectos:

- Sin instalación de bibliotecas locales no es posible realizar firmas XML (XMLDSig, XAdES, Facturae, ODF y OOXML) en versiones de Java 7 anteriores a la 7u1.

5 Instalación del cliente

Aunque el Cliente @firma es una aplicación Applet Java convencional que se descarga desde Internet para ejecutarse (no se instala localmente), en ciertas situaciones necesita la adecuación del entorno operativo para su correcta ejecución.

Para estos casos, el componente instalador (BootLoader), analiza el entorno operativo, determina cuales son las necesidades de este y descarga e instala los componentes necesarios. Estos componentes son:

- El proveedor de seguridad SunPKCS11 para los JRE que no lo integran:
  - JRE 6 y 7 en 64 bits para Windows
- El proveedor de seguridad SunMSCAPI para los JRE que no lo integran:
  - JRE 5 para Windows
  - JRE 6 y 7 anteriores a 7u1 en 64 bits para Windows
- Las bibliotecas Apache Xalan
  - JRE 5, se instalan como ENDORSED
- Las bibliotecas de compatibilidad de @firma
  - JRE 5, son clases de Java 6 instaladas como Endorsed en Java 5 a modo de actualización

Ciertos componentes son específicos de la variante del Cliente @firma que se desea utilizar (LITE, MEDIA o COMPLETA), por lo que el BootLoader distinguirá entre ellas para instalar únicamente los componentes necesarios.

El componente instalador, el BootLoader, actúa de forma completamente transparente para el integrador, que no debe realizar ninguna acción específica para cargarlo o utilizarlo: Cuando se solicita la carga del Cliente @firma automáticamente entra en acción el BootLoader para determinar si es necesario algún componente e instalarlo apropiadamente. Este puede mostrar si es necesario un diálogo con las licencias de los componentes a instalar, que el usuario puede aceptar o rechazar.

5.1 Ficheros para el despliegue del Cliente

El listado completo de archivos que cubren todas las construcciones y configuraciones de entorno soportadas por el cliente y ayudan a su optimización en la carga son:

- `afirmaBootLoader.jar`
  - BootLoader del Cliente @firma. Fichero firmado por el integrador.
- **XXX**`_j`**Y**`_afirma5_core.jar / `**XXX**`_j`**Y**`_afirma5_core.jar.pack.gz`
  - Construcción del núcleo del Cliente @firma, donde **XXX** es la distribución del Cliente (LITE, MEDIA o COMPLETA) e **Y** es la versión de Java compatible (5 ó 6 y superiores). Fichero firmado por el integrador.

  - Por ejemplo: `COMPLETA_j6_afirma5_core__V3.2.jar`

- `XXX_afirma.jnlp`
  - Ficheros de despliegue JNLP del Cliente @firma, donde **XXX** es la distribución del Cliente (LITE, MEDIA o COMPLETA). Debe localizarse en el mismo directorio que los núcleos del Cliente y no debe ser modificado por el integrador.
- `afirma_5_java_5.jar`
  - Componente instalablede Java 5. Fichero firmado por el integrador.
- `sunmscapi.jar`
  - Componente instalablede clases Java del proveedor SunMSCAPI. Fichero firmado por Sun Microsystems).
- `mscapi_`**`XXX`**`_JRE`**`YY`**`.zip`
  - Componente instalablede bibliotecas nativas para el proveedor SunMSCAPI, donde **XXX** es la arquitectura del sistema operativo (x86_64, amd64, x86, etc.) y **YY** es la arquitectura del JRE (32 ó 64). Fichero firmado por el integrador.
    - Por ejemplo: `mscapi_x86_JRE32.zip`
- **`XXX`**`_pkcs11lib_`**`YYY`**`_JRE`**`ZZ`**`.zip`
  - Componente instalablede bibliotecas nativas para el proveedor SunPKCS11, donde **XXX** es el sistema operativo (LINUX, WINDOWS, SOLARIS o MACOSX), **YYY** es la arquitectura del sistema operativo (x86_64, amd64, x86, etc.) y **ZZ** es la arquitectura del JRE (32 ó 64). Fichero firmado por el integrador.
- `sunpkcs11.jar`
  - Componente instalablede clases Java del proveedor SunPKCS#11. Fichero firmado por Sun Microsystems).
- `xalan.zip`
  - Componente instalable de clases Java de Apache Xalan. Fichero firmado por el Integrador.
- `constantes.js`
  - Fichero de variables JavaScript para la instalación y carga del Cliente @firma. Puede ser modificado por el integrador.
- `common-js\*.js`
  - Ficheros JavaScript del Cliente @firma. No deben ser modificador por el integrador.
- `version.properties`
  - Fichero informativo con la versión del Cliente @firma.

No debe eliminarse ninguno de estos ficheros de la carpeta del servidor Web. Únicamente el fichero *constantes.js*puede ser modificado por el integrador para personalizar el despliegue.

5.2 Despliegue del Cliente

Para el despliegue del cliente en un entorno Web, deben situarse todos los ficheros proporcionados, respetando la estructura de directorios, en la misma carpeta que la página Web desde la que se realizará su carga.

En ciertas ocasiones, puede convenir que los archivos instalables del Cliente residan en una ruta distinta que el núcleo del Cliente y el componente instalador (por ejemplo, para evitar que la descarga de los instalables por parte del BootLoader requiera autenticación SSL con certificado cliente). Para estos casos, deben situarse los archivos instalables en la nueva ruta (del mismo u otro servidor) y configurarlamediante la constante JavaScript "`baseDownloadURL`" del fichero *constantes.js*.

De igual forma, si los núcleos del Cliente, ficheros de despliegue JNLP y el componente instalador no van a residir en la misma carpeta del servidor Web que la página HTML desde la que se va a usar, debe indicarse su situación mediante la constante JavaScript "`base`" de*constantes.js*. En esta ruta se deben localizar el fichero *afirmabootloader.jar* y todos los ficheros del Cliente cuyo nombre comienza por LITE, MEDIA o COMPLETA. Estos son los *core* del Cliente y los ficheros de despliegue JNLP.

Las rutas establecidas mediante las constantes "`base`" y "`baseDownloadURL`" podrán ser absolutas o relativas. Siempre usarán la barra separadora "/" (nunca "\") y no terminar por este carácter.

Rutas de ejemplo:

- **Absolutas:** "file:///C:/ficheros","http://www.minhap.es/ficheros", "https://ficheros"…
- **Relativas:**"instalables", "afirma/ficheros", "/ficheros"…

En caso de que el Cliente se cargue desde una Web creada al vuelo (no existe como un fichero en el servidor) será obligatorio establecer las variables "`base`" y "`baseDownloadURL`" para indicar dónde se encuentran los distintos componentes del Cliente.

5.3 Instalación del Cliente

El instaladordel Cliente, también llamado Bootloader, funciona de forma automática y transparente tanto para el usuario y como para el integrador. El integrador no debe operar directamente con él.

El BootLoader entra en ejecución cuandoel procedimiento de carga del Cliente lo considera necesario (para instalar o actualizar algún componente del entorno operativo), se notifica al usuario mediante un mensaje como el siguiente:

Figura 1: Notificación de instalación

Tras aceptar el mensaje se mostrará el acuerdo de licencia que el usuario deberá aceptar. En caso de no hacerlo, la instalación quedará suspendida y cualquier otra operación sobre él llevará a error.

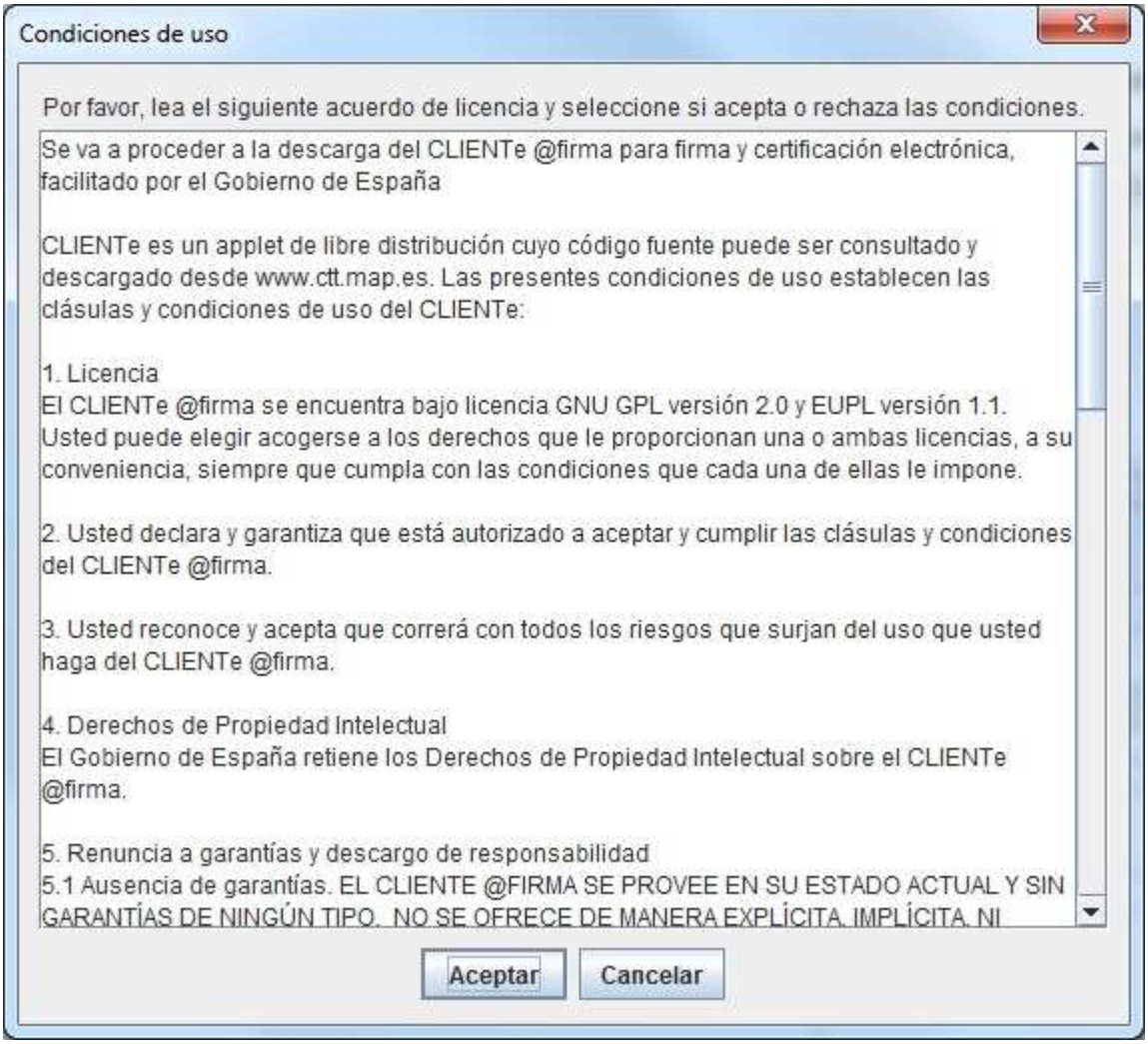


Figura 2: Acuerdo de licencia

Cuando se acepta el acuerdode ususe notifica al usuario si el proceso termino de forma satisfactoria o no.

Figura 3: Confirmación de la instalación

6 Uso del Cliente de Firma como Applet de Java

6.1 Carga del Cliente

Para la carga del Cliente desde una página Web será necesario importar en esta, al menos, las bibliotecas "***constantes.js***", "***instalador.js***" y "***deployJava.js***" que acompañan al Cliente. Para importarlas, se puede utilizar su ruta relativa desde la página Web que las carga o la ruta absoluta de los ficheros.

El proceso de carga ser inicia al invocar la función JavaScript "**`cargarAppletFirma()`**" incluida en el fichero "***instalador.js***".

La función "**`cargarAppletFirma()`**":

1. Invoca al BootLoader, que comprueba si es necesario instalar o actualizar algún componente del entorno operativo y así lo hace si es necesario y el usuario lo autoriza.

2. Carga el Applet de firma (Cliente).

El cliente de firma queda cargado en memoria y puede accederse a las funcionalidades que implementa por medio de la variable JavaScript "`clienteFirma`", localizada en el fichero "***constantes.js***".

Este método admite únicamente un parámetro, que indicaría la construcción mínima que es necesaria para el correcto funcionamiento de la aplicación. Los valores soportados son:

- **`LITE`**. Este es el comportamiento estándar cuando no se indica una construcción por parámetro.
- **`MEDIA`**.
- **`COMPLETA`**.

La carga del cliente en una página puede realizarse con sólo introducir un comando JavaScript en el propio cuerpo de la página que se encargue de invocar al método de carga.

Por ejemplo, si sólo vamos a usar las funcionalidades de firma CAdES usaríamos:

```
<html>
<head>
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
<script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
[…]
</head>
<body>
       <script type="text/javascript">
            cargarAppletFirma();    // Esto carga la construcción por defecto
      </script>
  […]
  </body>
</html>
```

En cambio, si, por ejemplo, quisiésemos realizar firmas PDF (sólo disponibles en la construcción COMPLETA del Cliente) tendríamos que realizar:

```
<html>
<head>
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
<script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
[…]
</head>
<body>
       <script type="text/javascript">
            cargarAppletFirma('COMPLETA');
      </script>
  […]
  </body>
</html>
```

Si a lo largo de la ejecución de nuestra aplicación se tuviese que utilizar en varias ocasiones el cliente y en cada una de ellas se utilizasen distintas funciones, deberemos utilizar siempre como parámetro del método de carga el identificador de la construcción más completa que se requiera. Esto evitará periodos de carga innecesarios.

NOTA IMPORTANTE:

El Cliente @firma utiliza la biblioteca JavaScript "deployJava.js" de Oracle para realizar la carga de los Applets, y esta biblioteca exige que la carga de los Applets (la escritura dinámica de las etiquetas que declaran el Applet) se realice antes de que finalice completamente la carga de la página.

El método onLoad() del cuerpo de las páginas HTML se invoca automáticamente justo después de finalizar completamente la carga de estas, por lo que no es un lugar válido para realizar la llamada. Cualquier otro punto no relacionado con eventos de carga es válido para situar la llamada.

En los ejemplos HTML incluidos con el Cliente puede verse una situación correcta, justo tras la etiqueta HTML de inicio del cuerpo de la página:

```
<html>
        <head>…</head>
        <body>
                <script type="text/javascript">   cargarAppletFirma();    </script>
                …
        </body>
</html>
```

6.2 Tratamiento de errores

Es posible tratar todos los errores que se hayan producido durante la operación del cliente mediante JavaScript. El cliente siempre almacena si la última operación criptográfica que realizó finalizó correctamente o no. Es posible consultar este resultado mediante el método del cliente **`isError()`**. En caso de producirse un error además, se podrá obtener la descripción del mismo mediante el método **`getErrorMessage()`**. De esta forma pueden elaborarse mecanismos JavaScript capaces de detectar y mostrar los errores pertinentes al usuario.

Un ejemplo que ilustra este sistema de tratamiento de errores es:

```
var fichero= document.getElementById("fichero");
      clienteFirma.initialize();
      clienteFirma.setFileuri(fichero.value);
      firmar();

      if(!clienteFirma.isError()){
            var firmaB64 = document.getElementById("firmaB64");
            firmaB64.value = clienteFirma.getSignatureBase64Encoded();
            return true; // Enviar
      }
      else {
            alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
            return false;

      }
}
```

También es posible dejar la tarea de notificación de los errores directamente al cliente. En caso de hacerlo, el cliente mostrará un mensaje de error mediante un dialogo Java por cada error de operación detectado (salvo en multifirmas masivas en donde estas notificaciones harían inviable un uso eficiente del cliente y en donde, por el contrario, se generan trazas de log).

Para activar este mecanismo de notificación de errores es necesario configurar a **`true`** la constante **`showErrors`** del fichero JavaScript "*constantes.js*" y establecerla antes de cada operación mediante la función **`initialize()`** de "*firma.js*" o "*cripto.js*", según se vayan a realizar operaciones de firma o cifrado/ensobrado, respectivamente. Por defecto, esta opción está configurada a **`false`**.

6.3 Firma WEB

6.3.1 ¿Qué es la firma Web?

En el proceso de firma Web una parte de una página Web (como un formulario o la página entera) puede ser firmada digitalmente. Para ello,

1. Se compone un HTML por medio de JavaScript
2. Se muestran al usuario los datos a firmar
3. Se le solicita permiso para firmarlo
4. Se selecciona un certificado con el que firmar
5. Se solicita (si es necesario) la contraseña para acceder tanto al repositorio de certificados como a la clave privada del certificado
6. Se firma el HTML generado

6.3.2 ¿Qué puede firmar el componente firma Web?

Se puede firmar digitalmente cualquier elemento de un documento HTML (o el documento mismo).

En los campos modificables por el usuario, se firman los valores seleccionados por el mismo. Esto incluye también adjuntos, que son firmados por el Cliente. A la hora de firmarse, se muestra al usuariola página Web resultante que le permite verificar lo que realmente va a firmar.

La firma Web del cliente sigue la política WYSIWYS (What You See Is What You Sign), es decir, lo que el usuario ve es lo que firma.

Para llevar a cabo la firma Web es imprescindible que la página generada para la firma esté bien formada.

A continuación se muestra un ejemplo de formulario Web y de su previsualización para la confirmación de firma:

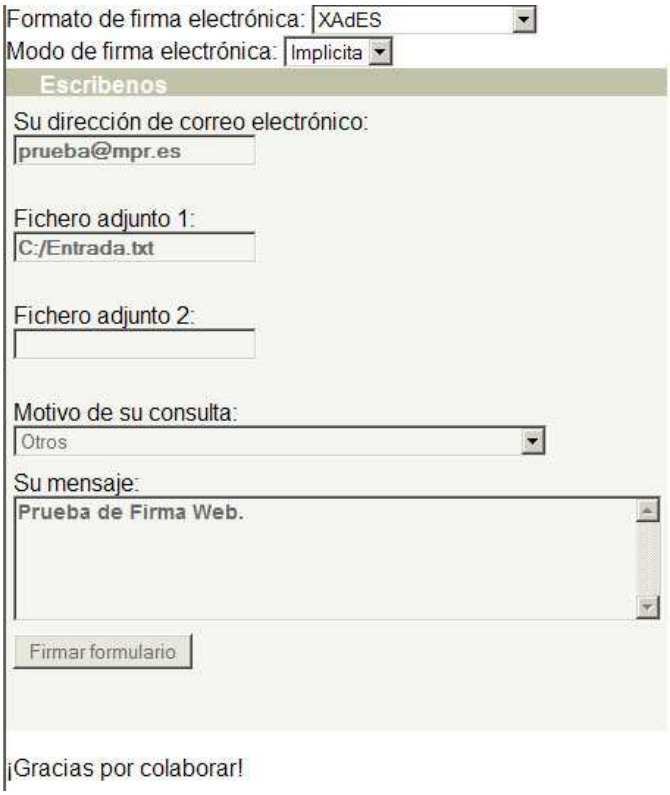


Figura 4: Formulario Web para firmar

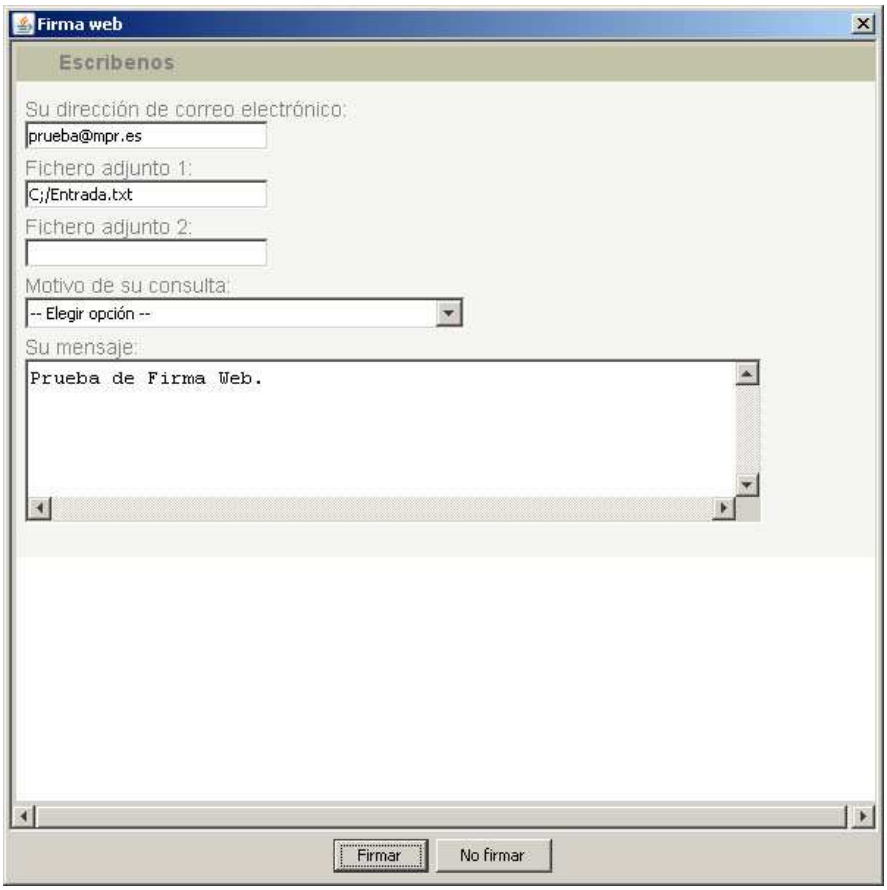


Figura 5: Solicitud de Firma Web

6.3.3 ¿Qué **no** firma el Cliente en la firma Web?

El cliente no firma las imágenes. Esto hay que tenerlo en cuenta a la hora de diseñar la parte del documento que se ha de firmar, pues es la que se mostrará al usuario (sin imágenes).

El cliente recoge todos los estilos *CSS*del documento que se definan mediante `LINK` o `STYLE`. Sin embargo, aquellas hojas de estilo que no se enlacen directamente (sino que se importen mediante la directiva `@import`) no se incluirán en el HTML si el usuario utiliza Mozilla Firefox. Por lo tanto, los estilos necesarios para mostrar correctamente la parte a firmar deben ir incluidos mediante `STYLE` o referenciados directamente mediante `LINK`, nunca mediante `@import`.

6.3.4 ¿Cómo hacer una firma Web?

Para hacer una firma Web, se puede pasar un HTML al Cliente para que muestre al usuario y éste decida si firmarlo mediante el método **`webSign`** del Cliente. Este método recibe una cadena HTML como parámetro.

Este método:

1. Muestra el documento al usuario. Se visualizará la página HTML indicada tal y como se ha especificado. Esto quiere decir que en el visualizador los campos aparecerán habilitados si es así como estaban definidos en el documento original. Así pues, cualquier modificación de los datos contenidos en esta ventana se verá reflejada en la posterior firma.

2. Solicita permiso para firmar el documento mostrado.

3. Solicita (si procede) la contraseña del repositorio de certificados.

4. Si no se ha establecido previamente un certificado para firma, muestra al usuario los certificados disponibles para firmarlo, y le solicita que elija uno.

5. En caso de que esté protegido por contraseña se la solicita al usuario

6. Firma el documento.

Una vez invocado:

1. Si el método **`isError`**del cliente devuelve **`false`**

    a. El valor devuelto (por la función JavaScript **`firmaWeb`** o el método del Cliente **`webSign`**) es la ruta de un fichero local que contiene la firma del HTML en el formato por defecto: CMS (explícito, es decir, que no contiene los datos firmados) y con los algoritmos por defecto: RSA y SHA1 codificada en base 64. En el apartado relativo a la configuración del Cliente se verá cómo cambiar estos parámetros. El contenido del fichero puede ser leído como texto (p. ej. firma XAdES) con el método `getTextFileContent`del cliente (el valor devuelto por este método puede variar dependiendo de la codificación original del texto) o, si es

binario (p. ej. firma CAdES), codificado en base 64 con el método `getFileBase64Encodeddel` cliente. Ambas funciones están descritas en el apartado Otras funcionalidades de este documento. La comunicación con el servidor de firma queda relegada a la aplicación donde se integra el cliente, así pues, el encargado de crear un método para enviar los ficheros devueltos por el cliente de firma al servidor es el propio integrador, ya que el cliente de firma en ningún momento se conecta con el servidor de firma, es un proceso independiente.

2. Si el método **`isError`** devuelve **`true`**el método **`getErrorMessage`**devuelve una cadena con el mensaje de error.

Por ejemplo:

```
clienteFirma.initialize();
var rutaFicheroFirma= clienteFirma.webSign(html);
if(!clienteFirma.isError()) {
      document.body.formulario.inputFicheroFirma.value= rutaFicheroFirma;
} else {
      alert("Se ha producido un error: "+ clienteFirma.getErrorMessage());
}
```

Se provee una función JavaScript llamada **`firmaWeb`** en el fichero "***firmaWeb.js***" que recibe como parámetros un elemento HTML y un documento HTML y compone un HTML y lo firma como se ha descrito (y devuelve la ruta al fichero que contiene la firma). En resumen, este método genera un HTML a partir de un elemento HTML con los valores actuales de los campos, incluyendo el contenido de los ficheros, y lo firma. Esto ahorra al integrador generar un HTML *ad-hoc* con los datos introducidos por el usuario para su firma web. El fichero "***firmaWeb.js***" depende de otros, como vemos a continuación en un ejemplo práctico:

```
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/firma.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/htmlEscape.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/utils.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/styles.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/firmaWeb.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/instalador.js"></script>
[…]

<script type="text/javascript" language="javascript">
       function enviar()
      {
            clienteFirma.initialize();

            var formulario = document.getElementById("formulario");
            var ruta = webSign(formulario, document);
            if(!clienteFirma.isError())
            {
                  var fichero = document.getElementById("fichero");

                  fichero.value = ruta;
                  return true; // Enviar
            }
            else
            {
                  alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
```

```
                return false;
            }
        }
</script>

[…]

<form id="formulario" action="/enviarFirma">
      <input type="text" id="fichero"style="visibility: hidden; display: none;"value="">
      DNI: <input type="text"><br>
      <input type="submit" onclick="return enviar();">
</form>
```

NOTA IMPORTANTE:

La firma Web enformato XMLDSig Enveloped o XAdES Enveloped solo es posible realizarla cuando la página Web a firmar se encuentra en un formato compatible estrictamente con XML, como por ejemplo XHTML. Así mismos, estos formatos exigen que la firma se realice en modo implícito (**`IMPLICIT`**).

6.4 Firma electrónica

El proceso de firma electrónica permite, por defecto, la firma de cualquier tipo de datos, independientemente de su formato. En concreto, los datos de entrada pueden ser.

Se permiten diferentes tipos de datos a firmar (solo se puede firmar un tipo cada vez):

- **un fichero**: se establece qué fichero firmar mediante el método **`setFileuri`**, que recibe como parámetro de entrada una cadena con la ruta al fichero a firmar. Este método no comprueba en ningún momento la existencia de un fichero en la ruta indicada. Si el fichero no existiese se produciría un error durante la operación en cuestión.

- **datos**: se establecen mediante el método **`setData`**, que recibe una cadena con los datos codificados en base 64.

- **un hash**: se establece mediante el método **`setHash`**, que recibe una cadena con el hash codificado en base 64.

- Si no se invoca ninguno de estos métodos, el Cliente solicitará al usuario un fichero para firmar

En las firmas XML (XAdES y XMLsSig), en el caso de que los datos insertados estén en base 64 (ya sea mediante el **`setFileuri`** y un fichero de texto que contenga el base 64 de los datos o a través del **`setData`** y una cadena doblemente codificada en base 64), no se realizará la codificación interna en base 64 que requiere la firma XML para ficheros binarios. Así obtenemos que se firma la codificación base 64 de los datos y no una doble codificación en base 64 de estos. Este mismo comportamiento lo podemos obtener mediante el método **`setFileuriBase64`** que establece como datos de entrada para las firmas electrónicas el contenido descodificado de un fichero en base 64.

Mientras que indicar con **`setFileuri`** un fichero con datos codificados en base 64 sólo aplica a las firmas XAdES y XMLdSig, el método **`setFileuriBase64`** funcionacon todos los formatos de firma. Esto permite indicar los datos a firmar a través de un fichero que los contiene en base 64.

Previamente a la realización de la firma, es aconsejable la inicialización del cliente y su configuración con los parámetros preestablecidos. Esto podemos realizarlo con las funciones JavaScript **`initialize()`** y**`configuraFirmar()`**, que configura los siguientes parámetros según las variables indicadas del fichero *constantes.js"*:

- **Algoritmo de firma**: Determinado por la variable **`signatureAlgorithm`**. Por defecto, SHA1withRSA.

- **Formato de firma**: Determinado por la variable **`signatureFormat`**. Por defecto, CMS.

- **Filtro de certificados**: Determinado por la variable **`certFilter`**. Por defecto, ninguno.

El método del applet que se ha de invocar para firmar es **`sign()`**, aunque también se puede llamar a la función JavaScript **`firmar()`**(en *firma.js*)que, como en los casos anteriores, espera si es necesario a que el cliente esté cargado y actualiza el entorno operativo si es necesario.

Por ejemplo:

```
<script type="text/javascript" language="javascript">
function enviar() {
	var fichero= document.getElementById("fichero");
	initialize();
	configurarFirma();
	clienteFirma.setFileuri(fichero.value);
	firmar();

	if(!clienteFirma.isError()){
		var firmaB64 = document.getElementById("firmaB64");
		firmaB64.value = clienteFirma.getSignatureBase64Encoded();
		return true; // Enviar
	}
	else {
		alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
		return false;
	}
}
</script>

[…]

<form id="formulario" action="/enviarFirma">
	<input type="hidden" id="firmaB64"><br>
	Fichero a firmar:<input type="file" id="fichero">
	<input type="submit" onclick="return enviar();">

</form>
```

Pueden ejecutarse operaciones de firma, así como de cofirma y contrafirma desde el HTML de prueba *demoMultifirma.html*.

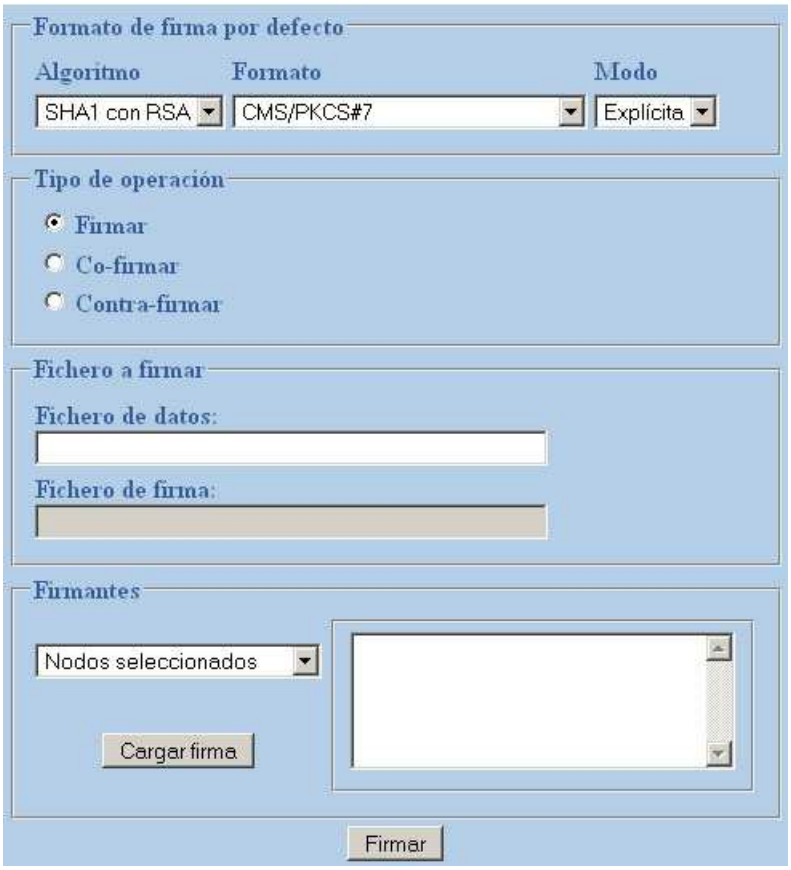


Figura 6: HTML de prueba demoMultifirma.html

6.5 Cofirma (*co-sign*)

La *cofirma* permite a varios usuarios firmar un mismo documento.

Una cofirma siempre firma los datos que se le indican, nunca se aplica ni depende de otra de las firmas del documento.

El caso de la *cofirma* es igual al de la firma simple, pero además de los datos hay que pasar al Cliente la firma electrónica de los demás firmantes. Esto se puede hacer de diferentes maneras:

- Mediante un fichero que contenga la firma electrónica, con el método **`setElectronicSignatureFile()`**, que recibe como parámetro una cadena con la ruta al fichero.
- Introduciendo directamente la firma, con el método **`setElectronicSignature()`**que recibe como parámetro una cadena con la firma en base 64.
- Si no se especifica, se pedirá al usuario que seleccione el fichero de firma.

Una vez especificados los parámetros necesarios, se invoca al método **`coSign()`**. La salida es análoga a la de la operación de firma.

Pueden ejecutarse operaciones de cofirma, así como de firma y contrafirma desde el HTML de prueba "*demoMultifirma.html*".

6.6 Contrafirma (*counter-sign*)

La contrafirma permite a un usuario firmar las firmas de otros usuarios.

El caso de la contrafirma es similar a los anteriores, pero sólo es necesario indicar la firma que deseamos contrafirma (no son necesarios los datos) y, segúnla operación concreta, puede ser necesario conocer la estructura de firmantes que contiene.

Para conocer la estructura de firmantes de una firma el Cliente dispone del método **`getSignersStructure().`** Este método devuelve una cadena que contiene los nombres de los firmantes separados por un retorno de carro ("**`\n`**" en JavaScript). Al comienzo del nombre hay tantos tabulados ("**`\t`**") como nivel ocupe el firmante en el documento. Por ejemplo, si A y B co-firman un documento y C contra-firma la firma de A, entonces la cadena devuelta sería "**`A\n\tC\nB`**".

La firma que deseamos contrafirmar se especifica mediante el método **`setElectronicSignature()`** o **`setElectronicSignatureFile()`**, que reciben la firma en base 64 y la ruta del fichero de firma, respectivamente.

En el fichero ***demoMultifirma.html***se puede ver un ejemplo de cómo tratar esta cadena.

Se puede especificar qué firmas se desean firmar de diferentes maneras:

- Todas las firmas hojas (firmas no contra-firmadas): invocando el método **`counterSignLeafs().`**
- Todas las firmas: invocando el método **`counterSignTree().`**

- Todas las firmas de un firmante: configurando los firmantes con el método **`setSignersToCounterSign()`**que recibe como parámetro una cadena con los nombres de los firmantes separados por "***\n***" e invocando el método **`counterSignSigners().`**

- Firmas concretas: con el método **`setSignersToCounterSign()`**indicamos que firmas deseamos contrafirmara partir de su posición (partiendo de 0) según el orden de aparición en la estructura devuelta por **`getSignersStructure`**(). Las posiciones se indican con números separados por "***\n***". Por ejemplo, "***0\n3\n4***" indica que se contrafirmen las firmas de las posiciones 0, 3 y 4. Se invoca con el método**`counterSignSigners().`**

La salida es análoga a la de la firma digital.

Téngase en cuenta que las contrafirmas siempre aplican a una firma y se colocan bajo esta en el árbol de firmas, al contrario que las cofirmas, que siempre se colocan como un nodo dependiente de los datos.

Pueden ejecutarse operaciones de contrafirma, así como de firma y cofirma desde el HTML de prueba "*demoMultifirma.html*".

NOTA IMPORTANTE: Dado que las contrafirmas se aplican sobre las firmas previas y no sobrelos propios datos,no es posible (no es conceptualmente correcto) realizar contrafirmas multi-fase, es decir, las huellas digitales se calculan al vuelo siempre (no se admiten huellas digitales pre-generadas externamente), ya que estas se generan en base a las firmas, no a los datos.

6.7 Firma y Multifirma Masiva

6.7.1 Consideraciones previas

Un aspecto importante que debe tenerse en cuenta en todas las operaciones de firma y multifirma masiva es que los procesos no son interactivos a nivel de operación individual, es decir, que no se requiere intervención del usuario y este no recibe información ni notificaciones hasta que finaliza el proceso completo, tanto si han ocurrido errores durante su desarrollo como si transcurrió sin incidencias.

Este modo de operar permite que, por ejemplo al iniciar un proceso de 2.000 firmas, el usuario pueda despreocuparse hasta su finalización, y que este no se detendrá en una firma aunque ocurriese un error (sea cual sea este). Siguiendo el ejemplo, si el usuario iniciase la firma de los 2.000 ficheros y desatendiese el proceso pensando que este tardará una o dos horas, el proceso no se habrá detenido porque el fichero número 3 estuviese corrupto, sino que se firmarían los 1.999 restantes y en el informe final de operación se marcarán las incidencias ocurridas.

Una excepción a esta regla es el uso de dispositivos de firma que requieren la introducción de un PIN / contraseña o una confirmación para cada una de las operaciones de firma (como el DNIe). Aunque los mensajes y diálogos de aplicación se pospondrán a la finalización total de las tareas, esta confirmación o introducción de PIN no puede ser omitida, por lo que el usuario debe realizarla por cada operación individual.

Consulte el punto "Tratamiento de errores" para más información sobre cómo se muestran los errores en los procesos de firma y multifirma masiva.

6.7.2 Firma/multifirma de directorios

Este proceso permite establecer un directorio y firmar/multifirmar todos los ficheros que contiene en una única operación, obteniendo la firma individual de cada uno de ellos.

El tipo de operación a realizar se especificará mediante **`setMassiveOperation`**, lo que nos permitirá realizar una firma masiva simple (**`FIRMAR`**), cofirmar (**`COFIRMAR`**) o contrafirmar todas las firmas al completo que encontremos (**`CONTRAFIRMAR_ARBOL`**) o tan sólo las firmas hoja (**`CONTRAFIRMAR_HOJAS`**). La operación se ejecutará mediante el método **`signDirectory`** del cliente y, en caso de no haber especificado ningún directorio, se mostrará la pantalla para su selección.

Los ficheros que se firmaran durante la operación pueden ser filtrados por extensión. Para esto se usará el método **`setInIncludeExtensions`** que recibe las extensiones de los ficheros que se deben procesar separadas por comas (“,”).

Por ejemplo:

```
clienteFirma.setInIncludeExtensions(“txt,xml,p7s”);
```

También es posible indicar que se desean procesar los ficheros de los subdirectorios dela rutaindicada. Esto se configuramediante el método**`setInRecursiveDirectorySign`**.

En el caso de las operaciones de multifirma es muy recomendable utilizar el mismo formato de firma del que ya dispusiese la firma original. Para indicar que se desea respetar este formato debe usarse el método **`setOriginalFormat`**. En caso de tratarse de una operación de firma masiva o no desear respetar el formato original del fichero de firma, se realizará una operación de firma conforme la configuración establecida mediante el mecanismo tradicional.

Según el tipo de operación masiva que se haya solicitado y el tipo de fichero que se encuentre durante la misma se realizará una u otra acción:

- **Firma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se firmará con la configuración de firma establecida.
- **Cofirma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se extraerán, siempre que sea posible, los datos implícitos de la firma y se cofirmará el fichero.
- **Contrafirma:**
  - **Fichero binario:** Se ignorará.
  - **Fichero de firma:** Se contrafirmará completamente o sólo las firmas hoja según tipo de operación (**`CONTRAFIRMAR_ARBOL`** o **`CONTRAFIRMAR_HOJAS`**).

En cada caso, se entenderá como fichero de firma todo aquel que sea una firma en el formato configurado, o en cualquier formato si se ha solicitado mantener el formato original. El resto de ficheros son considerados ficheros binarios.

Las firmas resultado de esta operación se almacenarán en el directorio establecido con el método **`setOutputDirectoryToSign`**. El método creará los ficheros de firma con el mismo nombre que

el fichero original (extensión incluida) y la extensión apropiada según el formato de la firma. En el caso de la cofirma y contrafirma se insertarán las partículas "*.cosign*" y "*.countersign*", respectivamente, antes de la extensión de firma.Si se ejecuta una operación de cofirma, pero el fichero es considerado un binario, se generará una firma, como ya se explicó anteriormente, y se agregará la partícula "*.sign*". En caso de no indicar un directorio de salida se tomará el mismo directorio en donde se encuentren los ficheros de entrada.

En el mismo directorio de salida se creará un fichero de log (***result.log***) en donde se registrará el resultado de cada una de las acciones realizadas durante la operación masiva.

En caso de producirse uno o más errores durante el proceso el método **`signDirectory`** devolverá **`false`**, pero no se detendrá hasta haber finalizado la operación. Para conocer con más detalle la causa de los errores que puedan producirse será necesario consultar el fichero de log.

Un ejemplo del uso de esta funcionalidad es:

```
clienteFirma.initialize();

clienteFirma.setSignatureFormat("CADES");
clienteFirma.setSignatureAlgorithm("SHA1withRSA");
clienteFirma.setInputDirectoryToSign("C:/ficheros");
clienteFirma.setOutputDirectoryToSign("C:/firmas");
clienteFirma.setInIncludeExtensions("csig");
clienteFirma.setInRecursiveDirectorySign(true);
clienteFirma.setMassiveOperation("CONTRAFIRMAR_HOJAS");

clienteFirma.signDirectory();

if(!clienteFirma.isError()){
    alert("La operacion finalizo con exito");
}
else{
    alert("Se detectaron errores durante el proceso de firma consulte el log de error para
más información");
}
```

Puede verse el funcionamiento de la multifirma masiva basada en ficheros en el HTML de prueba*demoFirmaDirectorios.html*.

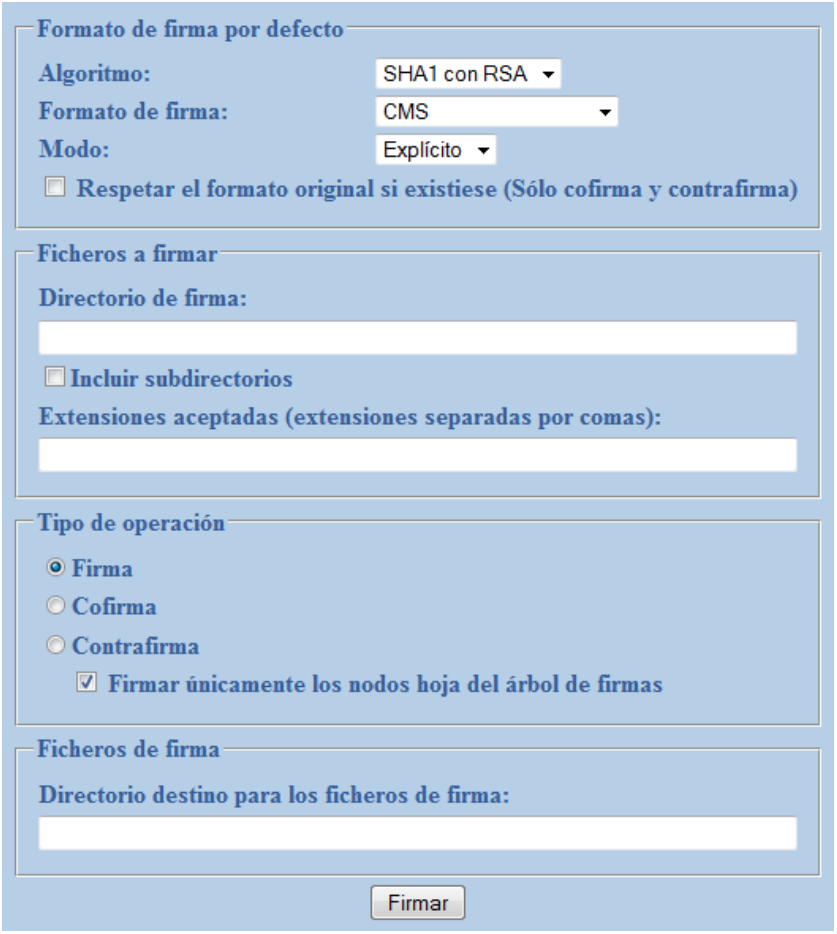


Figura 7: HTML de prueba demoFirmaDirectorios.html

6.7.3 Modo de operación programática

Adicionalmente a la metodología ya comentada se dispone de un procedimiento para la firma independiente de datos, ficheros y hashes en base a una configuración única de firma.

El procedimiento a seguir para realizar esta operación es el siguiente:

1. Configuración del cliente.
2. Inicialización de la operación masiva.
3. Firma masiva de los datos.
4. Finalización de la operación.

Configuración del cliente

Los aspectos configurables del cliente que afectan a la operación masiva son:

- **Operación masiva a realizar** (Firma, cofirma y contrafirma de nodos hoja o del árbol completo de firma).
- **Algoritmo de firma** (`SHA1withRSA`, `MD5withRSA`, `SHA512withRSA`, etc.).
- **Formato**con el que realizar las firmas (CMS, XAdES Detached, PDF, ODF…).
- Si se debe **respetar el formato original** que, en el caso de las operaciones cofirma y contrafirma, significa detectar el formato de las firmas introducidas para multifirmar con el mismo formato.
- **Modo de firma** (Implícita o explícita).
- **Certificado con el que firmar**.

La configuración de estos parámetros se realizarespectivamente mediante los métodos:

- `setMassiveOperation(String)`
- `setSignatureAlgorithm(String)`
- `setSignatureFormat(String)`
- `setOriginalFormat(boolean)`
- `setSignatureMode(String)`
- `setSelectedCertificateAlias(String)`

La mayoría de estos métodos se utilizan en la configuración de la firma simple del applet, pero otros se utilizan únicamente para la firma masiva:

- **`setMassiveOperation(String)`,** que configura el tipo de operación masiva y puede recibir los parámetros:
  - **`FIRMAR:`**Firmar datos.

- **COFIRMAR:** Agrega una nueva firma a los datos indicados. Cofirma si se le indica una firma o firma si se le proporcionan datos. Sólo podrá cofirmar cuando los datos estén contenidos en la firma o exista una referencia a ellos.
- **CONTRAFIRMAR_ARBOL:** Contrafirmar todas las firmas de undocumento de firma.No admite que se le proporcionen datos no reconocibles como una firma.
- **CONTRAFIRMAR_HOJAS:** Contrafirmar todas las firmas hoja de un documento de firma.No admite que se le proporcionen datos no reconocibles como una firma.

- **`setOriginalFormat(boolean)`**, que se configurará a **`true`** o **`false`** según se desee respetar o no el formato original de firma durante las operaciones de cofirma y contrafirma. Durante la operación de firma se ignora este parámetro. El comportamiento de esta opción es el siguiente:
  - Si la opción está activada: Se identificará el formato de la firma original y se multifirmará en este formato.
  - Si no está activada la opción: Se comprobará el formato de la firma y, si es compatible con el formato establecido, se cofirmará/contrafirmará en ese formato. Si no es compatible se actuará según la operación configurada:
    - **`COFIRMAR`**: Firmará el fichero en el formato configurado.
    - **`CONTRAFIRMAR_ARBOL`**: Fallará la operación.
    - **`CONTRAFIRMAR_HOJAS`**: Fallará la operación.

  Es decir, que si está activada esta función y, por ejemplo, indicamos que se cofirme en CAdES una firma CMS, se ignorará el formato indicado y se cofirmará en CMS (el formato original). Si fuese el caso contrario, firma CAdES y se solicita una cofirma CMS, se ignoraría el CMS y se firmaría en CAdES. Si la opción "respetar el formato original" estuviese desactivada (**`setOriginalFormat(false)`**) se multifirmaría siempre en el formato indicado, o se informaría mediante un mensaje de error que el fichero indicado no es un fichero de datos o firma compatible si el formato indicado no lo soportase.

  El mantener activa esta opción es útil cuando no se conozca el formato en el que fuesen originalmente firmados los datos o queramos evitarnos el seleccionarlo para cada elemento de firma, mientras que el desactivarlo evita que se realice el proceso de búsqueda del formato original y, que de seleccionar un formato equivocado, se nos informe.

Inicialización de la operación masiva

El proceso de inicialización configura los parámetros ya comentados en el módulo de firma masiva y reinicia el registro de mensajes (*log*) del módulo. Desde el momento de la inicialización y hasta que se finalice el proceso de firma masiva estos parámetros, a **excepción del tipo de operación** (**`setMassiveOperation(String)`**) y, cuando la operación es FIRMAR, el formato de firma (**`setSignatureFormat(String)`**), permanecen inalterados a lo largo de las operaciones realizadas por el módulo, aunque sí afectarán los cambios de configuración al resto de funcionalidades del cliente.

En caso de que no se hubiesen establecido todas las propiedades necesarias para la configuración de la firma masiva se tomarán los valores por defecto establecidos por el cliente. Estos son:

- **Operación:** Firma.
- **Algoritmo:** SHA1 con RSA.
- **Formato:** CAdES.
- **Respetar formato original:** Activado.
- **Modo:** Explícito.

En el caso del alias, si no se ha establecido ninguno, se mostrará un diálogo para permitir seleccionar el certificado de firma al inicializar el proceso de firma masiva.

Al inicializar el proceso de firma masiva, se solicitará confirmación al usuario ya que durante el proceso es posible que se acceda a ficheros localizados en su sistema.

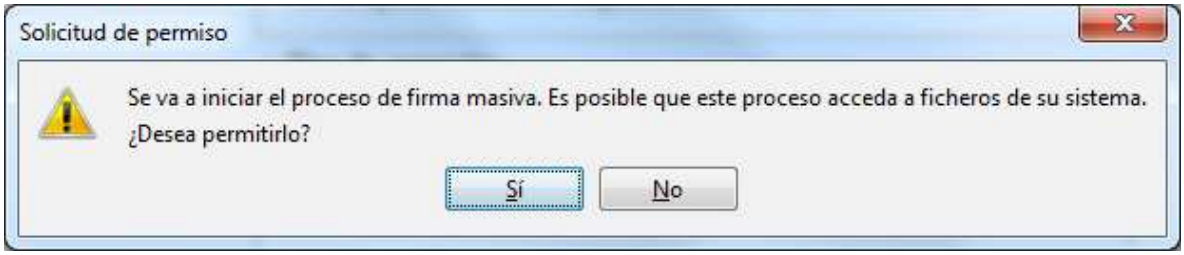


Figura 8: Confirmación del inicio del proceso de firma masiva

La inicialización del proceso de firma masiva se realiza mediante el método **`initMassiveSignature()`**.

Firma masiva de los datos

Existen 3 métodos para firmar, cofirmar o contrafirmar (nodos u hojas), según sea la operación configurada para el proceso:

- `massiveSignatureData(String)`
- `massiveSignatureFile(String)`
- `massiveSignatureHash(String)`

El método **`massiveSignatureData(String)`** realiza la operación configurada sobre los datos que recibe en forma de cadena de texto en base 64; **`massiveSignatureFile(String)`**ejecuta la operación sobre el fichero cuya ruta recibe cómo parámetro y **`massiveSignatureHash(String)`** lo hace sobre un hash en base 64.

A diferencia de cualquier otro método del applet que lea o almacene datos en disco, el método **`massiveSignatureFile(String)`** no pedirá confirmación al usuario para acceder al fichero. El usuario habrá dado su consentimiento para hacer esto al inicio del proceso de firma masiva.

A diferencia de la mayoría de parámetros de la configuración de la firma masiva, es posible modificar el tipo de operación que se desea en cualquier momento durante su desarrollo. Para esto sólo es necesario utilizar el método **`setMassiveSignatureOperation(String)`** en el momento en el que se desee modificar la configuración.

En el caso de realizarse una firma masiva (ni cofirmas, ni contrafirmas) es posible modificara mitad del proceso el formato con el que queremos firmar. Esto se realizará mediante el método **`setSignatureFormat(String)`**, que permitirá generar firmas en el nuevo formato, pero no afectará al formato inicialmente configurado (el establecido antes del **`initMassiveSignature()`**). Si durante la operación de firma masiva establecemos el formato a **`null`**, se establecerá el formato inicialmente configurado. El formato de las cofirmas y contrafirmas masivas no se puede modificar durante la ejecución, pero puede configurarse que se respete el formato original para que se opere siempre en el formato adecuado.

El comportamiento de cada una de las operaciones simples podrá variar según el tipo de fichero que se les proporcione:

- **Firma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se firmará con la configuración de firma establecida.
- **Cofirma:**
  - **Fichero binario:**Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se extraerán, siempre que sea posible, los datos implícitos de la firma y se agregará una nueva firmaal fichero.
- **Contrafirma:**
  - **Fichero binario:** Se ignorará.
  - **Fichero de firma:** Se contrafirmará completamente o sólo las firmas hoja según tipo de operación (**`CONTRAFIRMAR_ARBOL`** o **`CONTRAFIRMAR_HOJAS`**).

IMPORTANTE:Téngase en cuenta las siguientes consideraciones:

- Las operaciones de cofirma y contrafirma no pueden realizarse sobre hashes ya que desde estos no pueden obtenerse los datos originales.
- Determinados formatos de firma pueden exigir que sea necesario firmar sobre los datos o un fichero, no siendo posible firmar hashes. Por ejemplo, los formatos XML enveloped, ODF y PDF.
- La operación de firma recibe los datos (mediante cualquiera de los 3 métodos comentados) mientras que la cofirma y las contrafirmas reciben una firma implícita previamente generada con formato reconocido.
- La contrafirma se aplica sobre firmas y es indiferente que estas almacenen datos implícitos o no, pero la cofirma requiere los datos originales para ser firmados por lo que es obligatorio que se proporcione una firma con los datos implícitos o, al menos, una explícita realizada con el mismo algoritmo de firma con el que se solicita la cofirma, para así poder reutilizar el hash que almacena.
- En el caso de firmas con formato propio del tipo de documento (PDF, ODF y OOXML) la operación de cofirma supondrá agregar una nueva firma al documento.

Las operaciones masivas devuelvensu resultado en forma de cadena en base 64. En caso de producirse algún error se devolverá **`null`** y en ningún caso se lanzará una excepción, permitiendo al integrador obviar la captura de éstas,o se interrumpirá el proceso.

Cada operación individual de la firma masiva realizada generará una entrada en el registro de mensajes (*log*). En el caso de finalizar la operación correctamente esta simplemente lo indicará, mientras que en el caso de error la entrada explicará el error producido.

Finalización de la operación

La finalización de la operación elimina la configuración de operación masiva establecida por lo que ya no es posible continuar operando hasta que se vuelva a inicializar. Tras ser finalizada la operación, la nueva inicialización podría tomar una nueva configuración de firma establecida.

El método para llevar a cabo la finalización de la operación masiva es **`endMassiveSignature()`**.

El finalizar la operación no elimina los mensajes de registro (*log*) generados durante la misma, por lo que es posible seguir accediendo a ellos. Sí, en cambio, los eliminará el iniciar una nueva operación de firma masiva.

Registro de mensajes de la operación masiva

Por cada operación individual de firma/multifirma realizada durante el proceso masivo se genera una entrada en el registro de mensajes. Para obtener, tras una operación individual, el mensaje generado se debe utilizar el método **`getMassiveSignatureCurrentLog()`**. La forma de este registro será:

- Operación sobre TIPO_DATO: MENSAJE.

En donde TIPO_DATO será la palabra "`datos`", "`fichero`" o "`hash`" según el método utilizado para la operación (`massiveSignatureData`, `massiveSignatureFile` o `massiveSignatureHash` respectivamente); y MENSAJE será el mensaje obtenido, "`Correcta`" en el caso de que la operación finalizase correctamente o la explicación del error en caso de que se produjese.

Puede obtenerse todo el log generado hasta el momento para su proceso mediante el método **`getMassiveSignatureLog()`**. El texto que devuelve este método se compone de todas las entradas del mismo con el formato indicado separadas por un retorno de carro ("`\r\n`").

Puede almacenarse este mismo log en disco mediante la función **`saveMassiveSignatureLog()`**, que lo almacenará en la ruta indicada con el método **`setOutFilePath(String)`**. Si no se ha establecido ningún fichero de salida se mostrará un diálogo de guardado para seleccionar en donde se desea almacenar el fichero.

El registro de mensajes permanecerá aun cuando se finalice la operación masiva, pero se reiniciará en cada nueva inicialización del proceso.

Guardado de firmas en disco

Este mecanismo no está optimizado para el guardado de firmas en disco. Si su objetivo es almacenar las firmas resultantes en el sistema del usuario, consulte el apartado "**¡Error!No se encuentra el origen de la referencia.¡Error!No se encuentra el origen de la referencia.**" y evalúe si es preferible su uso.

Si requiere almacenar las firmas en disco y utilizar el mecanismo de firma masiva programática, dese cuenta de que se requerirá al usuario confirmación para el guardado de cada una de las firmas.

Las firmas resultantes de la operación de firma masiva se devuelven en base 64 por cada operación de firma individual (realizadas con `massiveSignatureData(String)`, `massiveSignatureFile(String)` o `massiveSignatureHash(String)`), por lo cual el cliente no las almacena internamente como hace con las operaciones de firma simple. Por este motivo, el simple uso del método de guardado de firma del cliente no aplica a esta situación. Si desea guardar los datos en disco tenga en cuenta que esto requiere confirmación explícita del usuario, por lo que deberá aprobar cada guardado individual de datos.

En su lugar se puede utilizar la siguiente sucesión de llamadas a métodos:

- **`setElectronicSignature(String)`**: Recibe como parámetro la firma en base64 y la guarda internamente.
- **`setOutFilePath(String)`**: Establece el fichero de salida. Para permitir al usuario que seleccione el nombre y directorio de salida para cada fichero firmado, se le pasará el parámetro **`null`**.
- **`saveSignToFile()`**: Pide confirmación al usuario y almacena la firma en el directorio de salida indicado.

Ejemplo Java de operación masiva

```
// Creamos una instancia del applet (innecesario para su uso en Web)
SignApplet clienteFirma = newSignApplet();

// Configuramos la operación que deseamos
clienteFirma.setMassiveOperation("FIRMAR");
clienteFirma.setSignatureFormat("CMS");
clienteFirma.setSignatureMode("IMPLICIT");

// Inicializamos la operación (en este momento se nos pedirá seleccionar un
// certificado de firma)
clienteFirma.initMassiveSignature();

// Una vez inicializada la operación, cualquier cambio en el algoritmo, formato,
// tipo de operación, etc. no será tenido en cuenta para la operación masiva,
// aunque sí para el resto de operaciones del cliente

// Vector en el que almacenar los resultados en base 64
Vector<String>firmasB64 = new Vector<String>();

// Firmamos y almacenamos los datos. Por norma general es recomendable operar
// directamente con las firmas generadas (guardarlas, enviarlas,…) y no
// mantenerlas todas cargadas para evitar problemas de desbordamiento de
// memoria

// Firma de ficheros
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.txt") );
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.xml") );
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.odt") );
```

```
// Firma de datos
firmasB64.add( clienteFirma.massiveSignatureData(
     clienteFirma.getFileBase64Encoded("C:\\Fichero.txt", true)
));
firmasB64.add(clienteFirma.massiveSignatureData(
     clienteFirma.getFileBase64Encoded("C:\\Fichero.xml", true)
));
firmasB64.add( clienteFirma.massiveSignatureData(
     clienteFirma.getFileBase64Encoded("C:\\Fichero.odt", true)
));

// Firma de hashes
clienteFirma.setFileuri("C:\\Fichero.txt");
firmasB64.add(clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));
clienteFirma.setFileuri("C:\\Fichero.xml");
firmasB64.add(clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));
clienteFirma.setFileuri("C:\\Fichero.odt");
firmasB64.add( clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));

// Finalizamos la operación
clienteFirma.endMassiveSignature();

// Almacenamos el log preguntando al usuario donde lo desea almacenar
clienteFirma.saveMassiveSignatureLog();

// Además de almacenarlas en un vector queremos guardarlas en disco (en este
caso no mantenemos referencias a los ficheros originales)
for(int i=0; i<firmasB64.size(); i++) {
     if(firmasB64.get(i) != null) {
     clienteFirma.setElectronicSignature(firmasB64.get(i));
     clienteFirma.setOutFilePath("firma"+i+".csig");
     clienteFirma.saveSignToFile();
     }
}

// Mostramos un mensaje de error al usuario por cada error obtenido
String[] mensajes = clienteFirma.getMassiveSignatureLog().trim().split("\r\n");
for(int i=0; i<firmasB64.size(); i++) {
     if(firmasB64.get(i) == null) {
          JOptionPane.showMessageDialog(
               clienteFirma, mensajes[i], "Error", JOptionPane.ERROR_MESSAGE
          );
     }
}
```

Puede verse el funcionamiento de la multifirma masiva basada en ficheros en el HTML de prueba*demoFirmaDirectorios.html*.

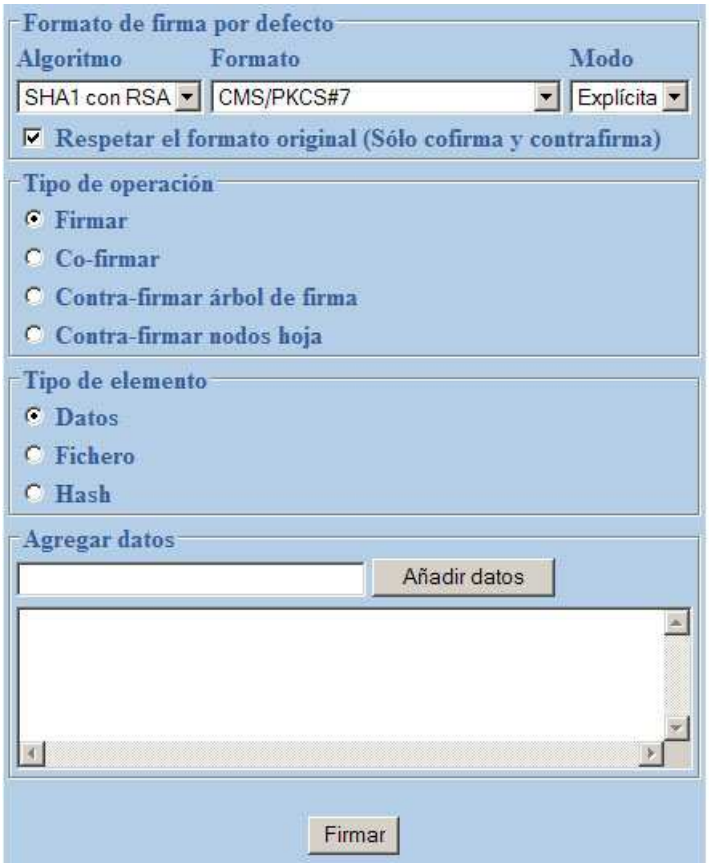


Figura 9: HTML de prueba demoMultifirmaMasiva.html

6.8 Cifrado de datos

El Cliente @firma incorpora funcionalidades de cifrado simétrico de datos que permite encriptar datos o ficheros de tal forma que sólo aquella persona que tenga la clave o contraseña utilizada para el cifrado puede recuperar esos datos.

Antes de proceder al cifrado de datos con el Cliente, conviene reinicializar su configuración debido a que esta funcionalidad comparte recursos con los procesos de firma y podría haber incompatibilidad en la entrada de datos. Para esta tarea puede utilizarse el método "`initialize()`" de la biblioteca JavaScript llamada "***cripto.js***" que reinicia las propiedades del cliente a sus valores por defecto.

Para iniciar el proceso de cifrado habrá que introducir previamente los datos a cifrar. Es posible especificar los datos a cifrar de diferentes formas:

- **datos**: se especifica cuál es la cadena a cifrar mediante el método **`setPlainData`**, que recibe la cadena que se desea cifrar en Base64.

- **fichero**: es posible especificar que se cifre un fichero indicándole la ruta a la llamada del proceso de firma. Para ello, utilizaremos directamente el método de cifrado **`cipherFile`**.

Por defecto el cliente de cifrado define como algoritmo de cifrado AES y generación automática de clave, aunque posteriormente veremos las posibilidades de configuración de estos parámetros. Tras indicar la configuración del cifrador, podemos ejecutar la operación de cifrado. Para cifrar los datos establecidos mediante **`setPlainData`**, utilizaremos el método **`cipherData`**. Para cifrar un fichero, usaremos el método**`cipherFile`**, que recibe la ruta de un fichero en disco. También podemos utilizar las funciones JavaScript (en ***cripto.js***), **`cifrarDatos`** y **`cifrarFichero`**, que reciben los datos en Base64 y la ruta del fichero, respectivamente. El comportamiento de la llamada es análogo al resto de llamadas al Applet, indicando si la ejecución se ha llevado a cabo de forma correcta o los errores en caso negativo.

Los datos cifrados se podrán obtener una vez haya finalizado mediante la llamada al método **`getCipherData`** o la función JavaScript **`obtenerResultadoCifrado`**, que devuelven los datos cifrados codificados en formato Base 64. Es posible almacenar estos datos cifrados en un archivo mediante la función **`saveCipherDataToFile`**, a la cual le pasaremos la ruta absoluta del archivo destino (atención, el archivo destino será sobrescrito para evitar problemas a la hora de descifrar). El contenido del archivo destino son los datos cifrados, por lo que no se recomienda su edición, ya que pudiera alterar gravemente el contenido plano del mensaje cifrado o incluso destruirlo.

Un ejemplo de aplicación de lo anterior para un proceso completo de cifrado sería el siguiente:

```
<html>
<head>
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
<script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
<script type="text/javascript" language="javascript" src="common-js/cripto.js"></script>
<script type="text/javascript" language="javascript">
       function cifrar()
      {
            var texto= document.getElementById("campo1").value;
            clienteFirma.initialize();
            clienteFirma.setKeyMode("GENERATEKEY");
```

```
            clienteFirma.setCipherAlgorithm("AES");
            clienteFirma.setPlainData(clienteFirma.getBase64FromText(texto, null), null);
            clienteFirma.setShowErrors(false);
            cifrarDatos();
            if(!clienteFirma.isError()){
                  var datosCifrados = clienteFirma.getCipherData();
                  var campoCifrado =document.getElementById("campo2");
                  campoCifrado.value = datosCifrados;
                  return true;
            }else{
                  alert("No se ha podido cifrar los datos: "+clienteFirma.getErrorMessage());
                  return false;
            }
      }
</script>


[…]

</head>

<body>
       <script type="text/javascript">
            cargarAppletFirma();
      </script>

  […]
<label>Datos planos</label><br/>
<textarea id="campo1" cols="20" rows="5" nowrap>Introduzca texto plano aquí</textarea>
<br/><br/><input type="button" value="Cifrar" onClick="cifrar();"/><br/><br/>
<label>Datos cifrados</label><br/>
<textarea id="campo2" cols="20" rows="5" nowrap readonly></textarea>
[…]

</body>

</html>
```

Este ejemplo básico captura el texto introducido en un área de texto, la cifra con generación automática de clave y el algoritmo AES y la muestra en un segundo área de texto tras pulsar un botón. Para más información, consultar el ejemplo incluido y el apartado Configuración de cifrado para conocer las opciones de configuración de esta operación.

Puede verse las distintas configuraciones de cifrado y descifrado de datos en el HTML de ejemplo ***demoCifrado.html***.

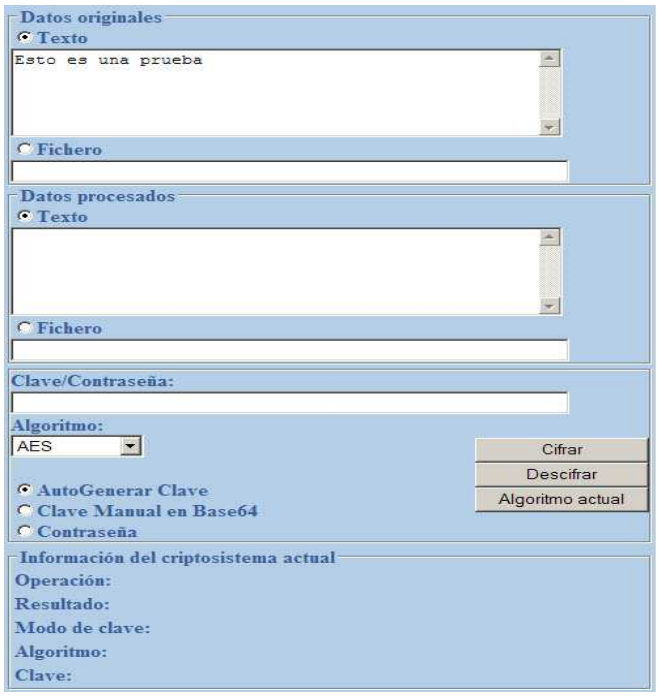


Figura 10: HTML de ejemplo demoCifrado.html

Puede encontrar información adicional sobre el cifrado de datos en el punto Algoritmos de cifrado de este mismo documento.

6.9 Descifrado de datos

De manera similar al cifrado, deberemos especificar cuáles son los datos a descifrar, y al igual que antes podremos especificar los datos cifrados mediante dos métodos distintos:

- **datos o texto cifrado**: se especifica cuál es la cadena a descifrar mediante el método **`setCipherData`**. Los datos de entrada estarán en base 64 (igual que la salida del algoritmo de cifrado) para evitar la aparición de caracteres extraños o no imprimibles. Internamente estos datos se decodificaran a la base apropiada y se descifrarán.

- **fichero**: también es posible especificar que los datos a cifrar provienen de un archivo indicándole la ruta (**`decipherFile`**), o usando la función**`setFileuri`** para especificarla. También aquí se deberá especificar la ruta absoluta del fichero.

Evidentemente para descifrar datos no podremos auto generar una clave, sino que tendremos que especificarle una siempre. En caso que se intente iniciar el método de descifrado sin especificar la clave supondrá un fallo automático. Los datos descifrados se pueden recuperar mediante la llamada a la función **`getPlainData`**. También tenemos un método para escribir estos datos recuperados a un archivo mediante la llamada a **`savePlainDataToFile`** y pasándole la ruta absoluta del archivo destino.

Un ejemplo básico para descifrar sería el siguiente:

```
<html>
<head>
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
<script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
<script type="text/javascript" language="javascript" src="common-js/cripto.js"></script>
<script type="text/javascript" language="javascript">
       function descifrar()
      {
            var textoCifrado= document.getElementById("campo1").value;
            var clave=document.getElementById(“clave”).value;
            var archivoOrigen=document.getElementById(“origen”).value;
            clienteFirma.initialize();
            clienteFirma.setKey(clave);
            clienteFirma.setKeyMode(“USERINPUT”);
            clienteFirma.setCipherAlgorithm(“AES”);
               descifrarArchivo(archivoOrigen);
            if(!clienteFirma.isError()){
                  var datosPlanos = clienteFirma.getTextFromBase64(
clienteFirma.getPlainData(), null);
                  var campoPlano=document.getElementById(“campo2”);
                  campoPlano.value = datosCifrados;
                  var archivoDestino=document.getElementById(“destino”).value;
                  clienteFirma.savePlainDataToFile(archivoDestino);
                      return true;
            }else{
                  alert("No se ha podido descifrar los datos: "+clienteFirma.getErrorMessage());
                  return false;
            }
      }
</script>


[…]

</head>

<body>
       <script type="text/javascript">
            cargarAppletFirma();
      </script>

[…]
<label>Fichero cifrado:</label>
<input type=”file” id=”origen”/>
<label>Fichero plano (introduzca URI):</label>
<input type=”text” id=”destino” value=””/>
<br/><br/><input type=”button” value=”Descifrar” onClick=”descifrar();”/><br/><br/>
<label>Datos descifrados</label><br/>
<textarea id=”campo2” cols=”20” rows=”5” nowrap readonly></textarea>
[…]

</body>

</html>
```

Puede encontrar información adicional sobre el descifrado de datos en el apartado Configuración de cifrado, y ver un ejemplo de uso en el HTML de ejemplo ***demoCifrado.html***.

6.10 Estructuras CMS cifradas/ Sobres Digitales

CMS define una estructura de datos que puede albergar distintos tipos de contenido (datos planos, hashes, datos comprimidos, datos cifrados…). El Cliente @firma permite generar algunas de estas estructuras, entre las que se encuentran los llamados "Sobres Digitales".

Los tipos de contenidos que permite generar el Cliente @firma son:

- CMS encriptado (*EncryptedCMS*).
- CMS envuelto (*EnvelopedCMS*).
- CMS autenticado (*Authenticated&Enveloped*).
- PKCS#7 firmado y envuelto (*Signed&Enveloped*).

Los tipos de contenido considerados sobres digitales son aquellos indicados como "envuelto". Estos son el envuelto, el autenticado y envuelto, y el firmado y envuelto.

6.10.1 Tipo de contenido

6.10.1.1 CMS encriptado

Esta estructura está basada en un mensaje criptográfico que sólo contiene el texto cifrado simétricamente y opcionalmente el algoritmo utilizado para el cifrado. No contiene ninguna información sobre la clave, emisor o receptor, por este motivo no puede considerarse un sobre digital. La metodología para su creación es:

1. Se establece los datos a incluir en el mensaje mediante una llamada a **`setData`**, pasándole en base 64los datos que se desean incluir en el mensaje, o **`setFileuri`**, para incluir un fichero. Opcionalmente se definen el algoritmo de cifrado, la clave y el modo de clave.

2. Se realiza una llamada a **`buildCMSEncrypted`**.

3. El CMS generado se puede recuperar como un String codificado en base 64 mediante el método **`getB64Data`**o guardarla en un archivo con la operación **`saveDataToFile`**.

Consulte el apartado 8.7Configuración de cifrado para conocer las distintas opciones de cifrado que puede utilizar para la configuración del CMS encriptado.

6.10.1.2 CMS envuelto

Mediante la creación de un CMS envuelto obtenemos un sobre digital en el cual podremos incluir contenido cifrado sólo visible por los receptores que le indiquemos. Posteriormente veremos la estructura generada y comentaremos algunos detalles sobre ella.

El procedimiento de creación es el siguiente:

1. Definimos los datos a incluir en el sobre digital de igual manera que en el apartado anterior, indicando los datos en base 64 mediante **`setData`**o un fichero mediante **`setFileuri`**. Definimos también el resto de parámetros opcionales.

2. Opcionalmente, establecemos el alias del certificado que deseamos indicar como remitente del sobre por medio de la función **`setSelectedCertificateAlias`**.

3. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **`setRecipientsToCMS`** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro (“\n”). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden eliminarse los receptores indicados llamando a este método con el parámetro **`null`**. Para la generación del sobre será necesario indicar al menos un receptor válido.

   De forma independiente a los receptores indicados mediante el método **`setRecipientsToCMS`**, es posible configurar receptores adicionales mediante el método **`addRecipientToCMS`** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **`removeRecipientToCMS`**.

4. Hacemos la llamada al método **`buildCMSEnveloped`**. Tras la llamada nos solicitará que indiquemos el emisor del mensaje, mediante la selección de nuestro certificado digital, aunque es opcional indicar el emisor, es recomendable.

5. Una vez concluida la operación, podremos obtener el resultado mediante la llamada a**`getB64Data`** o guardarla en un archivo con **`saveDataToFile`**.

Si se desean agregar más remitentes al sobre, puede realizarse la operación tal como se describe en el apartadoSobres con múltiples remitentes.

6.10.1.3 PKCS#7 firmado y envuelto

Similar al CMS envuelto, pero los datos además de cifrarse son firmados por el emisor. El procedimiento es el siguiente:

1. Definimos los datos a incluir en el sobre digital de igual manera que en el apartado anterior, indicando los datos en base 64 mediante **`setData`**. Para incluir un fichero le indicaremos la dirección absoluta del fichero en la llamada**`setFileuri`**.

2. Opcionalmente, establecemos el alias del certificado que deseamos indicar como remitente del sobre por medio de la función **`setSelectedCertificateAlias`**.

3. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **`setRecipientsToCMS`** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro (“\n”). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden eliminarse los receptores indicados llamando a este método con el parámetro **`null.`**Para la generación del sobre será necesario indicar al menos un receptor válido.

De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales mediante el método **addRecipientToCMS** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

4. Hacemos la llamada al método **signAndPackData** si hemos especificado datos o al método **signAndPackFile** utilizando la ruta del fichero. Tras la llamada, si no indicamos el remitente del sobre en el paso 2, nos solicitará que seleccionemos el emisor del mensaje mediante la selección de nuestro certificado digital. En esta ocasión es obligatorio indicarlo para así para firmar los datos.

5. Una vez concluida la operación, podremos obtener el resultado mediante la llamada a**getB64Data** o guardarla en un archivo con **saveDataToFile**.

Si se desean agregar más remitentes al sobre, puede realizarse la operación tal como se describe en el apartado Sobres con múltiples remitentes.

6.10.1.4 CMS autenticado y envuelto

Similar al PKCS#7 firmado y envuelto. El ensobrado firmado y envuelto contaba con una vulnerabilidad que hacía posible que el sobre fuese modificado sin que se detectase con posterioridad. El ensobrado CMS autenticado corrige este problema generando un código de autenticación para el sobre que no puede ser replicado tras su modificación sin conocer cuál es su contenido. De esta forma, cualquier cambio en el sobre hará fallar el proceso de validación con ese código y este no podría sustituirse por otro sin conocer el contenido del sobre.

El procedimiento para generar un sobre CMS autenticado y envuelto es el siguiente:

1. Definimos los datos a incluir en el sobre digital indicándolos en base 64 mediante **setData**. Para incluir un fichero le indicaríamos la dirección absoluta del fichero en la llamada **setFileuri**.

2. Opcionalmente, establecemos el alias del certificado que deseamos indicar como remitente del sobre por medio de la función **setSelectedCertificateAlias**.

3. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **setRecipientsToCMS** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro (“\n”). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden eliminarse los receptores indicados llamando a este método con el parámetro **null**. Para la generación del sobre será necesario indicar al menos un receptor válido.

   De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales mediante el método **addRecipientToCMS** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

4. Hacemos la llamada al método **`buildCMSAuthenticated`**. Tras la llamada, si no indicamos el remitente del sobre en el paso 2, nos solicitará que indiquemos el emisor del mensaje mediante la selección de nuestro certificado digital, obligatorio para poder autenticar los datos.

5. Una vez concluida la operación, podremos obtener el resultado mediante la llamada a**`getB64Data`** o guardarla en un archivo con **`saveDataToFile`**.

6.10.2 Sobres con múltiples remitentes

El Cliente @firma permite definir múltiplesremitentes para los sobres digitales de tipo “envelopedData” y “signedAndEnvelopedData”.

Para agregar múltiples remitentes a un sobre será necesario generar el sobre normalmente y agregar en una operaciónposterior la información de un nuevo remitente. Si se desean agregar nuevos remitentes al sobre puede repetirse esta operación tantas veces como se desee. El procedimiento a seguir es el siguiente:

Una vez tenemos el sobre generado con el remitente inicial el proceso a seguir es el siguiente:

1. Seleccionamos el sobre digital al que deseemos agregar el nuevo remitente. Podemos hacer esto introduciéndolo en base 64 mediante **`setData`** o indicando la ruta absoluta en la que se encuentra el sobre en disco utilizando**`setFileuri`**.
2. Opcionalmente, seleccionamos el certificado del nuevo remitente configurando el almacén en donde se encuentra mediante **`setKeystore`** y su alias con **`setSelectedCertificateAlias`**. Si no se introducen estos datos, se pedirá el certificado al usuario.
3. Hacemos la llamada al método **`coEnvelop`**. Tras la llamada, si no indicamos el nuevo remitente, se nos solicitará mediante un diálogo modal que lo seleccionemos del almacén configurado y, seguidamente, se agregará la información del nuevo remitente.
4. Una vez concluida la operación, podremos obtener el resultado mediante la llamada a**`getB64Data`** o guardarla en un archivo con la operación **`saveDataToFile`**.

7 Despliegue del Cliente @firma en Servidor

El núcleo del Cliente @firma se distribuye en forma de biblioteca Java, por lo que es posible integrarla en otras aplicaciones Java como una biblioteca independiente. Para esto sólo es necesario agregarla a proyecto Java, ya sea cliente para ejecución en cliente o servidor, en la que se quiera integrar, recordando que el sistema en donde vaya a ejecutarse debe cumplir con los requisitos mínimos del Cliente @firma.

Puede decirse que existen 2 formas de acceder a las funcionalidades de las que dispone el Cliente @firma:

- **Acceso a bajo nivel:** Esto implica hacer uso de toda la API del Cliente @firma para realizar cada operación unitaria necesaria (extracción de certificados de los almacenes, configuración de los módulos de firma,…) para componer la operación que se desea realizar (firma de datos, generación de sobres digitales). El uso de estas funcionalidades requiere un conocimiento más interno del Cliente @firma y el uso intensivo del Javadoc del mismo para identificar los métodos apropiados para componer las operaciones que se desean.
- **Acceso a alto nivel:** Esto implica el uso del Cliente @firma a través de la misma Clase *applet* que se utiliza para la integración en páginas Web. Por medio de esta clase se puede acceder a las funcionalidades ya conocidas del cliente, preparadas para su uso directo por parte del integrador (firma/multifirma de ficheros, datos y hashes; operaciones masivas; sobres digitales; cifrados…).

7.1 Diferencias del despliegue del Cliente en servidor

Existen algunas consideraciones que deben tenerse en cuenta antes de comenzar a integrar el Cliente @firma en servidor, debido a las diferencias al despliegue en la máquina del usuario:

- La integración del Cliente @firma en servidor está orientada a la firma de datos por parte de las entidades que lo integran, no de los usuarios.
- El Cliente no se carga en un navegador Web que tiene un almacén de certificados predefinido, por lo es necesario indicar siempre a qué almacén de certificados se debe acceder.
- Se accede a los almacenes de certificados del servidor, no del usuario.
  - No es común disponer de los certificados instalados en los almacenes de los navegadores o sistemas operativos. Normalmente dispondremos del certificado en un fichero (P12/PFX, JKS…) o dispositivo externo (tarjeta inteligente, HSM…).
  - Es recomendable que se configure el acceso a dispositivos de firma externos (tarjetas inteligentes, tokens USB,…) en alguno de los almacenes locales (CAPI, Mozilla,…) y se utilicen a través de los mismos.
- El uso del Cliente en servidor debe ser 100% programático. No debe requerir la intervención de un usuario.
  - Opciones que antes se delegaban al usuario tienen que realizarse automáticamente. Por ejemplo, la selección del certificado de firma.
  - No pueden utilizarse los métodos de carga y guardado de ficheros en disco, ya que estos, por seguridad, solicitan confirmación al usuario. Los métodos vetados son:

  - getFileBase64Encoded
  - getTextFileContent(String)
  - saveDataToFile(String)
  - savePlainDataToFile(String)
  - saveCipherDataToFile(String)
  - setFileuri(String)
  - setFileuriBase64(String)
  - setElectronicSignatureFile(String)
  - setOutFilePath(String)
  - setKeyStore(String, String, String)
  - setInputDirectoryToSign(String)
  - setOutputDirectoryToSign(String)
  - initMassiveSignature()
  - cipherFile(String)
  - decipherFile(String)
  - signAndPackFile(String)

- No se pueden realizar firmas con el DNIe, ya que este requiere la autorización del usuario para firmar.
- Las operaciones deben terminar siempre, ya sea exitosamente o debido a un error, pero en ningún caso deben bloquearse, por ejemplo, con mensajes modales de aviso.
- **ADVERTENCIA:** Debido a que el *applet* del Cliente no ha sido desarrollado para usarse como interfaz programática para el uso desde servidor, la actual versión del Cliente @firma no permite que todas las operaciones se puedan realizar de forma 100% programática.Esto implica que hay operaciones que no es posible realizar desde servidor.

- Deben evitarse siempre los diálogos gráficos, ya que es posible que el servidor no disponga de un entorno de ventanas. Por ejemplo, deben evitarse las barras de progreso en la carga de ficheros.

7.2 Acceso a las funcionalidades a bajo nivel del Cliente

El uso de las funcionalidades a bajo nivel del Cliente @firma puede ser tedioso debido a que es necesario conocer los distintos elementos que lo componen. En este punto se hace especialmente importante atender a las consideraciones del apartado anterior y gestionar los errores que pudieran surgir resultado de la configuración del Cliente o de la operación.

Para el uso de estas funcionalidades, diríjase al Javadoc del Cliente @firma y a la documentación técnica del mismo.

7.2.1 Ejemplo de integración

A continuación se muestra un ejemplo de cómo se utilizarían las funcionalidades a bajo nivel del cliente para generar un sobre digital autenticado y envuelto tomando el firmante desde un almacén PKCS#12 y el certificado del destinatario desde el almacén de Windows.

ADVERTENCIA: Para simplificar el código se ha obviado en el ejemplo la gestión de errores.

```
publicstaticvoid main(String[] args) throws Throwable {

String dataFilePath     = args[0];   // Ruta del fichero que queremos ensobrar
  String aliasRecipient = args[1];   // Alias destinatario en almacen Windows
  String aliasSender    = args[2];   // Alias remitente del sobre en almacen P12
  String p12Path        = args[3];   // Ruta del almacen P12
String p12Pass     = args[4];   // Password del almacen P12
String envelopPath = args[5];   // Ruta del fichero de salida

// Configuramos el fichero de datos
InputStream dataIs =  new FileInputStream(dataFilePath);

// Configuramos los datos del firmante tomandolos del P12
PasswordCallback psc = newCachePasswordCallback(p12Pass.toCharArray());
AOKeyStoreManager senderKsm =
          AOKeyStoreManagerFactory.getAOKeyStoreManager(
               AOConstants.AOKeyStore.PKCS12, p12Path, null, psc, null);
  PrivateKeyEntry senderKe = senderKsm.getKeyEntry(aliasSender, psc);

// Configuramos los datos del destinatario desde el almacen de Windows
AOKeyStoreManager recipientKsm =
          AOKeyStoreManagerFactory.getAOKeyStoreManager(
               AOConstants.AOKeyStore.WINDOWS, null, null, null, null);
X509Certificate recipientCert = (X509Certificate)recipientKsm.getCertificate(
          aliasRecipient);

// Ejecutamos la operacion
AOCMSSigner enveloper = newAOCMSSigner();
byte[] result = enveloper.envelop(
          dataIs,                                  // Datos
          AOConstants.SIGN_ALGORITHM_SHA1WITHRSA,        // Algoritmo de firma
          AOConstants.CMS_CONTENTTYPE_AUTHENVELOPEDDATA, // Envoltorio
          senderKe,                                // Clave del firmante
          new X509Certificate[] { recipientCert }    // Destinatarios
  );

// Cerramos el fichero de datos
dataIs.close();

// Guardamosel sobre
FileOutputStream fos = newFileOutputStream(envelopPath);
fos.write(result);
fos.flush();
fos.close();

  }
```

7.3 Acceso a las funcionalidades a alto nivel del Cliente

Las funcionalidades de alto nivel del Cliente @firma son las mismas disponibles para los integradores que lo utilizan a modo de *applet* desde HTML. Las clases y métodos públicos del Cliente @firma vienen documentados en la versión del Javadoc orientada a integradores.

Las principales ventajas de utilizar esta clase para hacer uso del Cliente son:

- Nos permiten reutilizar los conocimientos adquiridos en el despliegue del Cliente en el sistema del usuario.
  - Podremos utilizar los mismos métodos que se utilizan en este entorno.
- Nos proporciona el mismo comportamiento del que disponemos en el despliegue común del Cliente.
  - Por ejemplo, para firmar nos bastará con seleccionar la configuración adecuada (formato, modo, almacén, certificado,…) y llamar al método de firma. No tendremos que preocuparnos de cargar el módulo del formato de firma escogido, extraer el certificado del almacén, etc.

Es importante saber que la clase *applet* del Cliente, denominada `SignApplet`, no fue desarrollada con esta finalidad y, en la actual versión del Cliente, no funciona en todos los casos de uso soportados por el Cliente, por existir operaciones que requieren del de interacción con el usuario. Por este motivo, **no se garantiza que se puedan acceder a todas las funcionalidades del Cliente a través de esta interfaz**. Un ejemplo de esto es el acceso a almacenes de certificados en fichero (P12/PFX y JKS), que no están soportados para las operaciones de multifirma y la selección del remitente de los sobres digitales.

Para cargar el Cliente en una aplicación Java y utilizar el Applet como interfaz de acceso a sus funcionalidades, haremos:

```
…
// Cargamos el cliente
SignApplet afirma = newSignApplet();
…
```

Una vez cargado el Cliente haremos uso de las funciones descritas en este manual y el JavaDoc para integradores del Cliente para realizar las distintas operaciones soportadas.

Siempre que se desee realizar una nueva operación debe invocarse al método `initialize()` del Cliente @firma para restaurar los valores por defecto y eliminar los resultados de anteriores operaciones.

7.3.1 Ejemplo de integración

A continuación se muestra un ejemplo de uso del cliente en donde se realiza una firma electrónica usando un certificado de un almacén PKCS#12:

```
…
// Cargamos el cliente
SignApplet afirma = newSignApplet();
```

```
// Configuramos la operacion de firma
afirma.setKeyStore("C:/almacen.p12", "1111", "P12");           // Almacen PKCS12
// System.out.println(afirma.getCertificatesAlias());          // Imprime los alias
afirma.setSelectedCertificateAlias("aliasCertificado");        // Certificado
afirma.setFileuri("C:/entrada.txt");                           // Fichero de datos
afirma.setSignatureFormat("CAdES");                            // Formato
afirma.setSignatureMode("Implicit");                           // Modo

// Ejecutamos la operacion
afirma.sign();

// Comprobamos si ocurrio un error durante la firma
if (afirma.isError()) {
      System.err.println("Error en la firma: " + afirma.getErrorMessage());
      return;
}

// Almacenamos la firma
afirma.setOutFilePath("C:/salida.csig");
afirma.saveSignToFile();

// Mostramos el resultado
if (afirma.isError()) {
      System.err.println("Error al almacenar la firma: " + afirma.getErrorMessage());
} else {
      System.err.println("La operacion finalizo correctamente");

}

…
```

8 Configuración del Cliente

8.1 Configuración de idioma

El Cliente @firma tiene configurado por defecto el idioma español para los textos. Sin embargo, al iniciarse el applet se toma la configuración de idioma del sistema del usuarioy se configura este idioma para la aplicación.

También es posible forzar el cambio de idioma para asegurar que los mensajes del Cliente se mostrarán en un idioma concreto. Es requisito indispensable que el idioma esté soportado por el Cliente. Si se indica un idioma no soportado o no válido, se configurará el idioma del sistema, o el idioma por defecto si este tampoco estuviese. Igualmente, si alguno de los textos necesarios no estuviese disponible en el idioma solicitado, se tomará del idioma por defecto.

Para forzar un idioma será necesario indicarlo en los parámetros de inicialización del applet, tanto a través del *constantes.js*, como en los ficheros JNLP.

En el fichero *constantes.js* estableceremos la variable **`locale`** al valor, conforme las ISO 639 y 3166. Por ejemplo:

```
var locale = “en_UK”;
```

En los ficheros JNLP será necesario agregar un nodo **`<param>`**dependiente del nodo **`<jnlp>`**, en donde se defina el atributo **`name=”locale”`**y el atributo **`value`** con el valor conforme a las ISO 639 y 3166 que deseemos utilizar. Por ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" href="COMPLETA_afirma.jnlp">
<information>
<title>Cliente @firma</title>
<vendor>Ministerio de Hacienda y Administraciones Publicas</vendor>
<homepage href="http://www.minhap.es" />
<description>Cliente de firma @firma</description>
</information>
<offline-allowed/>
<security>
<all-permissions/>
</security>
<resources>
<property name="jnlp.packEnabled" value="true"/>
<j2se  version="1.6+"  href="http://java.sun.com/products/autodl/j2se"  initial-heap-
size="512m" max-heap-size="512m" />
<jar href="COMPLETA_j6_afirma5_core.jar" main="true" />
</resources>
<applet-desc
name="Cliente @firma"
main-class="es.gob.afirma.applet.SignApplet"
width="1"
height="1" />
<param name="locale" value="en"/>
<update check="background"/>
</jnlp>
```

8.2 Inicialización de las operaciones

Antes de iniciar una operación criptográfica se debe invocar el método **`initialize()`**del Cliente, que borra las entradas y salidas de operaciones anteriores.

En las bibliotecas JavaScript "*firma.js*" y "*constantes.js*" se incluye un método **`initialize()`** que lo invoca al **`initialize()`** del cliente y configura diversos parámetros, como el formato de firma por defecto o el filtro de certificados.

8.3 Cambio de almacén de certificados

Al ejecutar el cliente @firma como Applet se configura por defecto el almacén de certificados del navegador o sistema operativo sobre el que se ejecuta. Según la configuración navegador/sistema operativo el almacén de certificados por defecto será:

	Internet Explorer	**Mozilla Firefox 32 Bits**	**Mozilla Firefox 64 Bits**	**Google Chrome / Opera**	**Apple Safari**
Windows	Almacén Windows	Almacén Mozilla	Almacén Windows	Almacén Windows	Almacén Windows
Linux / Solaris		Almacén Mozilla	Almacén Mozilla	Almacén Mozilla	
Mac OS X		Almacén Mozilla	Almacén Mozilla	Llavero Mac OS X	Llavero Mac OS X

Leyenda: Gris = no aplica. Para detalles sobre compatibilidad consulte con la sección Requisitos mínimos de este mismo documento.

El cliente @firma, sin embargo, permite la configuración de este almacén de certificados de tal forma que es posible indicar de qué almacén deben extraerse los certificados. Esta configuración se establece mediante el método **`setKeystore(String path, String pass, String type).`**

Este método recibe, por orden:

- **`path`**: La ruta al almacén de certificados que se desea utilizar (sólo para almacenes en disco). Si es necesaria para el tipo de almacén seleccionado y no se indica, se le mostrará un diálogo al usuario para que lo seleccione.
- **`pass`**: La contraseña para abrir el almacén. Aplica a cualquier almacén que pueda estar protegido por contraseña (PKCS#12/PFX, Mozilla Firefox configurado con clave maestra,…). Si no se indica y es necesaria se le mostrará un diálogo al usuario para que la inserte.
- **`type`**: Tipo de almacén de certificados. Los distintos parámetros admitidos son:
  - **`WINDOWS`**: Repositorio de Microsoft Windows (MSCAPI).
  - **`APPLE`**: Repositorio de Apple Macintosh (Llavero o KeyChain).
  - **`MOZILLA`**: Repositorio Mozilla. Para su uso en Windows es obligatorio tener instalado Mozilla Firefox.
  - **`P11`**: Repositorio de tipo PKCS#11 accesible desde una biblioteca nativa del sistema. No es recomendable el uso directo de este tipo de almacén, en su lugar debería aconsejarse al usuario que instale el dispositivo y acceda a él a través del almacén de certificados de su navegador. Si no se indica, se le solicitará al usuario la ruta y contraseña de la biblioteca.
  - **`P12`**: Repositorios en disco en formato PKCS#12 o PFX. Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén. Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.
  - **`JKS`**: Repositorios en disco en formato JKS.Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén.Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.
  - **`SINGLE`**: Certificado suelto en disco.Estos certificados sólo disponen de clave pública, por lo que no son aptos para firmar. Si no se indica, se le solicitará al usuario la ruta del certificado.
  - **`JAVACE`**: Repositorios en disco en formato Java Case Exact.Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén.Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.
  - **`WINADDRESSBOOK`**: Repositorio de Certificados de Otras Personas de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.
  - **`WINDOWS-CA`**: Repositorio de Certificados de Autoridades de Certificación de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.

- **WINDOWS-ROOT**: Repositorio de Certificados Raíz de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.

En caso de seleccionar un almacén no válido (el almacén de Apple en Windows, por ejemplo) u ocurrir un error durante su inicialización, el cliente se reconfigurará al almacén que se tuviese configurado en ese momento.

8.4 Selección y filtrado de certificados

8.4.1 Selección de los certificados para operaciones criptográficas

Muchas operaciones criptográficas de las soportadas por el cliente @firma requieren que se seleccione un certificado de usuario como, por ejemplo, la firma. Los certificados accesibles por el applet de firma son aquellos disponibles desde el repositorio de certificados del sistema o navegador y es posible seleccionar uno de ellos mediante el método del cliente **`setSelectedCertificateAlias(String)`** al que debe pasarse uno de los certificados recogidos mediante el método **`getCertificatesAlias()`**.

Es posible permitir al usuario seleccionar un certificado directamente a través de un diálogo de selección de certificados. Podemos mostrar este diálogo a través del método **`showCertSelectionDialog()`**, que devuelve el alias del certificado. Cuando el usuario selecciona un certificado a través de este método, este queda automáticamente seleccionado, de modo que es posible recuperarlo mediante los métodos **`getSignCertificate()`** y **`getSignCertificateBase64Encoded()`**, detallados en el apartado "Obtener el certificado usado para firmar".

En caso no seleccionarse un certificado, al realizar una operación criptográfica que lo requiera, se solicitará éste automáticamente al usuario mediante el diálogo de selección.

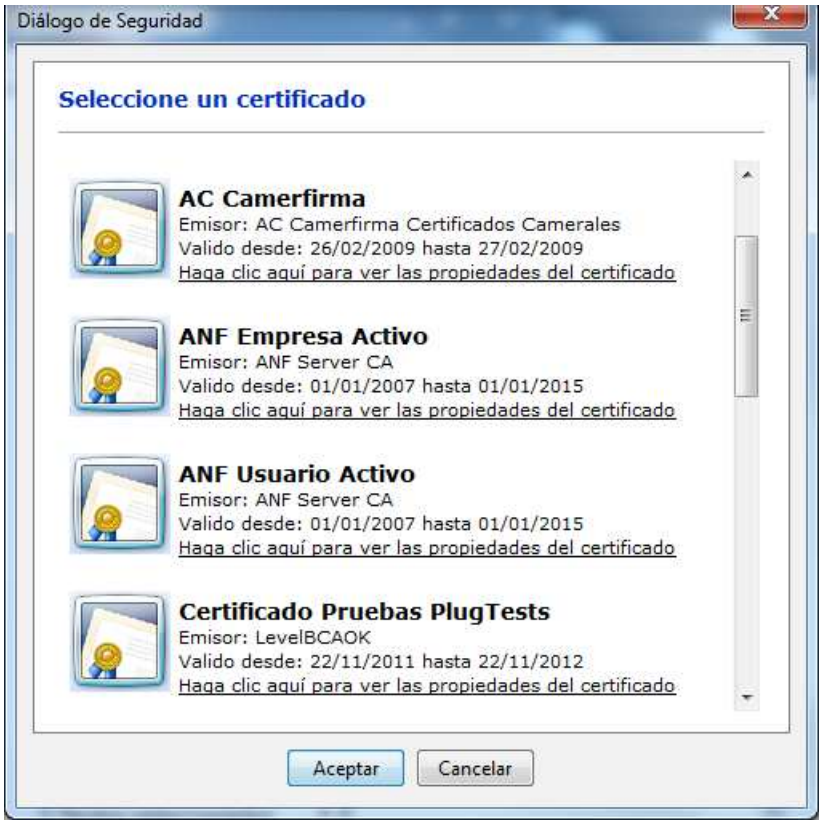


Figura 11: Selección de certificado

Por defecto, sólo se mostrarán aquellos certificados aptos para realizar una firma electrónica (independientemente de como se declare en campo KeyUsage explicado en siguiente apartado). Si desea que se muestren todos los certificados, a fin de seleccionar uno con un fin distinto al de firma, utilice el método **`setShowOnlySignatureCertificates(boolean)`**. Al pasar el valor, false a este método se mostrarán todos los certificados del almacén seleccionado, independientemente de si son válidos para firma o no.

Para indicar los receptores de los sobre digitales se deberán introducir las direcciones de sus certificados exportados (ficheros CER o DER). El método utilizado es **`setRecipientsToCMS(String)`** y recibe una cadena con las rutas de los certificados separadas por el carácter '\n'. Por ejemplo:

```
clienteFirma.setRecipientsToCMS("C:/detinatario1.cer\nC:/destinatario2.cer");
```

De forma independiente a los receptores indicados mediante el método **`setRecipientsToCMS`**, es posible configurar receptores adicionales de sobre digital mediante el método **`addRecipientToCMS`** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **`removeRecipientToCMS`**.

Un punto importante a destacar es que el método **`getCertificateAlias()`** proporciona los **alias reales** con los que los certificados han sido dados de alta en los almacenes (que son siempre los que deben usarse con **`setSelectedCertificateAlias(String)`**), pero que el diálogo de selección de certificado solicita la selección en base a un "nombre descriptivo", que se compone a partir del Nombre Común (CN) del titular, el alias real si procede y el nombre de la entidad emisora. Este "nombre descriptivo" se usa porque en muchas ocasiones los alias reales no son realmente descriptivos o están en formatos poco prácticos (como X.500).

8.4.2 Filtros de certificados

El Cliente de firma incorpora una funcionalidad que permite hacer una preselección de los certificados que se muestran para selección al usuario, de forma que se puedan descartar a priori los no aceptados o no apropiados y así disminuir la probabilidad de que el usuario erre en la elección del certificado adecuado.

El establecimiento de los filtros se realiza mediante el método **`addRFC2254CertificateFilter(String, String, boolean)`**, que admite tres parámetros:

1. Filtro a aplicar en el campo *Principal* del titular del certificado X.509.
   - Debe proporcionarse una cadena de texto con una condición de filtro según la normativa RFC 2254.
2. Filtro a aplicar en el campo *Principal* del emisor del certificado X.509
   - Debe proporcionarse una cadena de texto con una condición de filtro según la normativa RFC 2254.
3. Se indicará **`true`** si se desea que solo se muestren los certificados aptos para firma electrónica según el campo *KeyUsage* del certificado X.509, **`false`** si no se desea hacer distinción por el valor de este campo.

El paso de *null* en cualquiera de los parámetros indica que, por el criterio correspondiente, no se aplicará ningún filtro. Para más información, consulte la documentación JavaDoc.

Ejemplos de uso:

- Selección entre certificados de firma de DNIe:
  - `clienteFirma.addRFC2254CertificateFilter(null, "cn=AC DNIE*", true);`
- Selección entre cualquier certificado marcado como apto para firma electrónica:
  - `clienteFirma.addRFC2254CertificateFilter(null, null, true);`
- Selección únicamente entre certificados emitidos por Camerfirma:
  - `clienteFirma.addRFC2254CertificateFilter(null, "o=Camerfirma", false);`
- Selección con diversos criterios en un mismo Principal:
  - `clienteFirma.addRFC2254CertificateFilter("(&(OU=Clase 2 persona fisica)(C=ES))", null, false);`
- Selección entre certificados de un titular cuyo número de DNI sea “123456789Z” (funciona con la mayoría de los emisores de certificados, como DNIe, FNMT, etc.):
  - `clienteFirma.addRFC2254CertificateFilter("SERIALNUMBER=123456789Z", null, false);`
- Selección del certificado de firma del DNIe de un titular con número de DNI “123456789Z”:
  - `clienteFirma.addRFC2254CertificateFilter("SERIALNUMBER=123456789Z", "cn=AC DNIE*", true);`

Un filtro establecido mediante este método bloquearía todos aquellos certificados que no cumpliesen las 3 condiciones indicadas en el filtro: *subject*, *issuer* y *keyusage*.

Utilizando varias veces este método podemos agregar nuevos filtros, de tal forma que un certificado será válido cuando cumpla al menos uno de ellos.

Ejemplo de uso:

- Selección del certificado de firma del DNIe y los certificados expedidos por Camerfirma:
  - `clienteFirma.addRFC2254CertificateFilter(null, "cn=AC DNIE*", true);`
  - `clienteFirma.addRFC2254CertificateFilter(null, "o=Camerfirma", false);`

Para borrar los filtros establecidos, ya sea para obtener todos los certificados del almacén o introducir nuevos filtros, usaremos el método **`resetFilters()`**.

Es posible solicitar al cliente que, en caso de que sólo exista un certificado en el almacén o sólo un certificado pase los filtros indicados, se seleccione automáticamente, sin dar al usuario la posibilidad de elegir. Esto se puede hacer mediante el método**`setMandatoryCertificate(boolean)`**. Por defecto, se mostrará el diálogo de selección aunque sólo haya un certificado disponible para seleccionar.

Si el filtro devolviese más de un certificado, se ignoraría el valor establecido mediante **`setMandatoryCertificate(boolean)`**.

NOTA IMPORTANTE SOBRE FILTROS RFC2254: Los nombres de los campos X.500/LDAP que encontramos en la identificación del titular y el emisor de un certificado, se codifican internamente mediante un OID ASN.1 consistente en una sucesión de números separados por puntos.

Estos OID se traducen para una mayor facilidad de lectura en palabras asociadas. Por ejemplo, el OID *2.5.4.4* identifica al campo apellido, y se traduce por la palabra *SURNAME*.

No obstante, el diccionario de traducción de OID a palabras legibles no es unívoco, y existe la posibilidad de que a un mismo OID le correspondan dos palabras. Siguiendo con el ejemplo anterior, aunque la mayoría de los diccionarios asocian la palabra *SURNAME* al OID *2.5.4.4*, algunos le asocian la palabra *SN*.

Adicionalmente, otro inconveniente que podemos encontrar es que un determinado diccionario de OID no contenga ninguna palabra asociada a un OID particular. Un ejemplo de este caso podríamos encontrarlo en el OID *0.2.262.1.10.12.0*, que aunque corresponde a la palabra *liabilityLimitationFlag* no figura en los diccionarios comunes. Cuando un OID no figura en un diccionario se usa directamente el OID.

El Cliente @firma utiliza el diccionario de OID de Java, que puede diferir del usado por el sistema operativo y de los usados por otras aplicaciones.

Para evitar problemas de filtrado debidos a estas circunstancias, **debe construir siempre los filtros previendo que un campo puede aparecer referenciado por cualquiera de sus nombres o por su OID**.

Por ejemplo, si desea filtrar por apellido, la expresión de filtrado debe ser construida de forma que funcione correctamente tanto si este aparece referenciado por su OID (*2.5.4.4*) como por cualquiera de sus palabras descriptivas (*SURNAME*, *SN*, etc.).

Independientemente de que se utilicen en el filtro todas las palabras posibles con las que se identifique un OID, listamos a continuación las opciones preferentes para los OID más comunes para los que se conocen varias de estas palabras:

- **`2.5.4.4: SURNAME`**
- **`2.5.4.4: GIVENNAME`**

- **1.2.840.113549.1.9.1: EMAILADDRESS**
- **2.5.4.12 = T**
- **2.5.4.46 = DNQ**
- **2.5.4.43 = INITIALS**
- **2.5.4.44 = GENERATION**

Consulte la documentación sobre la normativa RFC 2254 para obtener más información de cómo construir adecuadamente sus filtros.

Información adicional:

- Copia de la normativa RFC 2254:
  - http://www.faqs.org/rfcs/rfc2254.html
- Ejemplo de diccionario de OID:
  - http://www.cs.auckland.ac.nz/~pgut001/dumpasn1.cfg

La marca de certificado apto para firma electrónica en el atributo KeyUsage de un certificado X.509

La inmensa mayoría de los certificados digitales usan el atributo X.509 *KeyUsage* para determinar el uso de un certificado (autenticación, firma electrónica, SSL servidor, etc.), por lo que distinguir por este para la selección del certificado apropiado para las operaciones de firma es en general una buena opción.

No obstante, la mayoría de los certificados emitidos por la FNMT-RCM (CERES, APE, etc.) no siguen las normativas internacionales en este sentido y en el atributo *KeyUsage* no marcan que son adecuados para firma electrónica pese a que se publicitan como aptos para dicho uso. Debido a esta falta de adecuación, si se marca mediante el último parámetro del método anteriormente comentado que solo deben mostrarse certificados aptos para firma, no se mostrará ningún certificado emitido por la FNMT-RCM.

Los certificados del DNIe sin embargo si siguen las normativas internacionales y marcan con los atributos correspondientes el uso, encontrándonos en cada DNIe un certificado apto para firma y otro que no lo es (el de autenticación).

Para consultar el significado preciso de cada uno de los valores del campo *KeyUsage* consulte con el emisor de sus certificados.

8.5 Configuración de firma

8.5.1 Algoritmos de firma digital

El cliente permite usar distintos algoritmos de firma digital, siempre especificados con el formato A*with*B, donde A es el algoritmo de huella digital y B el de cifrado. Entre los algoritmos soportados encontramos:

- **`MD2withRSA`**
- **`MD5withRSA`**
- **`SHA1withRSA`** (por defecto)
- **`SHA256withRSA`**
- **`SHA384withRSA`**
- **`SHA512withRSA`**(es el más seguro)

El algoritmo a utilizar se puede cambiar con el método **`setSignatureAlgorithm`,** que recibe como parámetro una de las cadenas citadas.

NOTAS IMPORTANTES:

- No todas las operaciones soportan todos los algoritmos:
  - Los formatos de firma XAdES y XMLDSig solo soportan `SHA1withRSA` en versiones anteriores a Java 6 update 18.
- No todos los algoritmos pueden considerarse seguros:
  - `MD2withRSA`y `MD5withRSA`son algoritmos ofrecen un nivel de seguridad inferior al mínimo recomendado y algunos formatos de firma no los admiten.
- No todos los repositorios soportan todos los algoritmos:
  - `SHA256withRSA`, `SHA384withRSA` y `SHA512withRSA` no están soportados en la configuración tradicional del almacén de Windows.
- Los formatos de firma no genéricos (ODF, OOXML, PDF) ignorarán cualquier configuración especificada por el integrador que no esté soportada por sucorrespondiente normativa. Por ejemplo, modo de firma explícito, algoritmos de firma no soportados…

8.5.2 Formato de firma electrónica

El cliente permite crear firmas digitales en distintos formatos (por defecto CAdES).

Globalmente se soportan los siguientes formatos y normativas de firma electrónica:

- CMS: Representado por la cadena "**CMS/PKCS#7**".
- CAdES: Representado por la cadena "**CAdES**".
- XMLDSig Internally Detached: Representado por la cadena "**XMLDSig Detached**".
- XMLDSig Enveloping: Representado por la cadena "**XMLDSig Enveloping**".
- XMLDSig Enveloped: Representado por la cadena "**XMLDSig Enveloped**".
- XAdES Internally Detached: Representado por la cadena "**XAdES Detached**".
- XAdES Enveloping: Representado por la cadena "**XAdES Enveloping**".
- XAdES Enveloped: Representado por la cadena "**XAdES Enveloped**".
- Factura Electrónica: Representado por la cadena "**FacturaE**".
- PAdES: Representado por la cadena "**Adobe PDF**".
- ODF (Open Document Format): Representado por la cadena "**ODF**".
- OOXML (Office Open XML): Representado por la cadena "**OOXML**".

El formato se puede cambiar con el método **setSignatureFormat**que recibe como parámetro la cadena que representa al formato en cuestión.

Las variantes EPES de los formatos de firma que las soportan se generarán automáticamente al configurar el formato de firma correspondiente y una política de firma (consulte el apartado Política de Firma).

8.5.3 Modos de firma electrónica

Determinados formatos de firma electrónica soportan los llamados, modos de firma. El modo de firma determina si los datos firmados se incorporarán o no junto con la firma electrónica generada. Los modos de firma existentes son:

- Implícito: Representado por la cadena "**implicit**".
- Explícito: Representado por la cadena "**explicit**".

El modo de firma se puede cambiar con el método **setSignatureMode**que recibe como parámetro la cadena que representa al modo en cuestión. Los parámetros son insensibles a mayúsculas y minúsculas.

Los formatos soportados por el cliente @firma que admiten configuración de modo son:

- CMS/PKCS#7
- CAdES
- XMLdSig Detached
- XMLdSig Enveloping

- XAdES Detached
- XAdES Enveloping

Un formato de firma puede definir modos propios válidos para su configuración.

8.5.4 Política de Firma

El cliente permite especificar, para cada firma electrónica, la política a la que esta se restringe. Los formatos de firma, soportados por el cliente, que admiten políticas de firma son CAdES, PDF/PAdES yXAdES(en sus variantes Detached, Enveloping y Enveloped). En el momento de establecer en el cliente la política de firma para una firma CAdES, se generará una firma CAdES-EPES en lugar de la firma CAdES-BES tradicional. De igual manera, al establecer la política de firma para una firma XAdES o PAdES se generará una firma XAdES-EPES o PAdES-EPES, respectivamente.

Las firmas con política generadas por el cliente @firma son de referencia externa. Es decir, la política no se incluye en la propia firma, tan sólo una referencia a la misma.

Es posible establecer una política de firma en el cliente @firma mediante el método **`setPolicy`** que recibe como parámetros 4 cadenas:

- Identificador: URL identificadora de la política de firma (normalmente una URL hacia el XML o el ASN.1 que formaliza la política) u OID (que puede estar en forma de URN) identificador de la política.
  - Las firmas XAdES deben usar URL.
  - Las firmas CAdES y PAdES deben usar OID
- Descripción: Descripción breve de la política.
- Calificador: URL calificadora de la política de firma (normalmente la URL apunta a un documento en formato PDF que describe la política).
- Hash: Huella digital SHA-1 en base64 de la política de firma.

En cualquier caso, para el establecimiento de estos parámetros, consulte con detenimiento los documentos de su política de firma y establezca los valores que allí se indiquen.

8.5.4.1 Restricción de formatos, algoritmos y modos de firma en las políticas de firma

La mayoría de las políticas de firma restringen la forma en la que se puede realizar una firma electrónica acorde a la política concreta. Esta restricción puede afectar a casi cualquier aspecto de la firma, y entre estos aspectos encontramos:

- Los algoritmos de firma (SHA1withRSA, SHA512withRSA, etc.) que pueden utilizarse.
- El tipo de firma admitido (CAdES, PAdES, XAdES, etc.).
- Las variantes de firma admitidas (explícitas, implícitas, *enveloping*, *enveloped*, *externally detached*, *internally detached*, etc.).
  - Por ejemplo, las firmas XAdES-EPES conformes a la política de firma de la AGE en su versión 1.8 deben ser siempre de tipo "*Internally Detached*".
- Etc.

Si especifica una política de firma en sus firmas electrónicas, debe asegurarse de que la totalidad de los parámetros de esta sean compatibles con dicha política, leyendo con detenimiento tanto la documentación del Cliente @firma como la versión PDF de descripción de la política de firma.

8.6 Configuración de sobres digitales

8.6.1 Selección de destinatarios desde LDAP

Además de la posibilidad de seleccionar los destinatarios de un sobre digital a partir de sus certificados de clave pública almacenados en disco, el cliente @firma permite la configuración de un LDAP para seleccionar los certificados que este tenga publicados.

El procedimiento para la selección de estos certificados es la siguiente:

1. Configuración del servidor LDAP al que se desea acceder. Esto lo conseguimos mediante el método **`setLdapConfiguration(String address, String port, String root)`**. Este método recibe:
   - **`address`**: Dirección URL del LDAP.
   - **`port`**: Puerto a través del que se realiza la conexión. Si no se indica se usará el puerto 389, el por defecto para LDAP.
   - **`root`**: Dirección raíz del LDAP (actualmente sin uso).
2. Selección del certificado que se desea recuperar del LDAP. Para ello se utilizará el método **`setLdapCertificatePrincipal`**, que recibe como parámetro el *principal* del certificado que deseamos.
3. Recuperación del certificado en base 64 mediante el método **`getLdapCertificate`**.
4. Configuración del destinatario del sobre indicándolo mediante el método **`addRecipientToCMS`**, que recibe como parámetro el certificado en base 64 recuperado del LDAP. Pueden agregase más de un destinatario de esta manera. Una vez establecido un destinatario, puede eliminarse mediante el método **`removeRecipientToCMS`** al que se le pasa como parámetro el mismo certificado en base 64 con el que se estableció.

8.7 Configuración de cifrado

8.7.1 Algoritmos de cifrado

Los algoritmos de cifrado permitidos son los siguientes:

- Cifrado con clave
  - **AES** (por defecto)
  - **ARCFOUR**
  - **Blowfish**
  - **DES**
  - **DESede**(triple DES o 3DES)
  - **RC2**
- Cifrado con contraseña
  - **PBEWithSHA1AndDESede**(basado en DESede/3DES)
  - **PBEWithSHA1AndRC2_40**(basado en RC2)
  - **PBEWithMD5AndDES** (basado en DES)

Para establecer el algoritmo deberemos invocar la función **setCipherAlgorithm** y podemos recuperar el algoritmo actual con el método **getCipherAlgorithm**.

8.7.2 Modo de clave

Definen de qué manera se trata la clave de cifrado. Existen tres posibilidades **GENERATEKEY, USERINPUT** y **PASSWORD.**

- GENERATEKEY: La clave se generará automáticamente.
- USERINPUT: El usuario deberá establecer la clave en base 64.
- PASSWORD: La clave de usuario se generará a partir de una contraseña. Esto requiere el uso de algoritmos de cifrado diseñados con este objetivo (algoritmos PBE).

El modo de clave se establece mediante **setKeyMode** y se recupera con **getKeyMode**.

8.7.3 Clave y contraseña de cifrado

Para obtener la clave que se ha utilizado para el cifrado/descifrado deberemos ejecutar el método **getKey**, el cual nos devolverá la clave codificada en base 64. Para fijar una clave para el cifrado o descifrado de datos usaremos **setKey**, adjuntando como parámetro la clave deseada en base 64.

En el caso de haber especificado el modo de clave **PASSWORD** (consultar apartadoModo de clave), en lugar de una clave de cifrado será necesario especificar una contraseña de cifrado. Para establecer la contraseña de cifrado/descifrado se utilizará el método **setPassword**. Para recuperar la contraseña establecida se utilizará el método **getPassword**.

ADVERTENCIA: Las contraseñas de cifrado/descifrado no podrán contener caracteres no ASCII.

8.7.4 Almacén de claves de cifrado

El cliente @firma v3.2 y superiores permiten a los usuarios almacenar sus claves de cifrado en un almacén de claves protegido por contraseña.

Es posible configurar el cliente @firma para que, en el momento de autogenerar una clave de cifrado se ofrezca al usuario la posibilidad de almacenarla en su almacén personal de claves. Para esto será necesario configurar el Cliente en modo "GENERATEKEY"tal como se indica en el apartadoModo de clave.

En caso de que el usuario acepte almacenar la clave en su almacén, se comprobará que este ya exista. Si existía, se le solicitará al usuario la contraseña para abrirlo y el alias con el que desea almacenar la clave. Si no existía, se le indicará al usuario y se le dará la posibilidad de crearlo para lo que se le solicitará la contraseña con la que desea protegerlo. Tras crear el almacén se procederá a almacenar la clave tal como ya se indicó.

El almacén de claves se guardacon el nombre ***ciphkeys.jceks*** en el directorio raíz del usuario activo. Al encontrarse en este directorio, el almacén será distinto para cada usuario del sistema que utilice el Cliente y un usuario no podrá acceder al almacén del resto de usuarios. Tenga en cuenta que si este fichero es eliminado no se podrán recuperar las claves almacenadas en él, por lo que es posible que no pueda recuperar los datos cifrados con el Cliente.

El integrador puede permitir al usuario utilizar sus claves ya almacenadas en el almacén para cifrar nuevos datos. Para esto sólo sería necesario configurar el modo de clave del Cliente al valor "USERINPUT" (consultar apartadoModo de clave) y ejecutar la operación de cifrado.

Cuando se desee descifrar un contenido y no se haya indicado directamente la clave para el descifrado, se le preguntará al usuario si desea tomar la clave de su almacén personal. En caso de aceptar, se le pedirá la contraseña del almacén y se le dará a elegir mediante un diálogo modal entre las claves almacenadas (de las que se mostrará el alias asignado y el algoritmo de cifrado para el que fueron generadas). Si no existiese el almacén de claves o el usuario no quisiera utilizarlo, se le preguntaría directamente por la clave de cifrado.

Cuando se activa el modo de clave para el cifrado/descifrado con contraseñas (modo "PASSWORD" establecido según el apartadoModo de clave) el almacén de claves queda inhabilitado.

Queda a elección del integrador la posibilidad de permitir que el usuario pueda o no almacenar la clave de cifrado en su almacén personal de claves o utilizar las almacenadas para cifrar. Esto puede hacerlo mediante el método **`setUseCipherKeystore`**al que se le puede pasar un `true` o un `false` para permitir o no su uso (por defecto se permitirá almacenarlas). Estemétodo no afecta al descifrado de datos. Si no se indicase la clave para el descifrado y el usuario dispusiese de un almacén de claves, siempre se le dará la posibilidad de descifrar mediante una de las claves almacenadas.

A continuación se muestran algunos ejemplos para el uso del almacén de claves de cifrado:

- Cifrado con las opciones por defecto (algoritmo de cifrado AES con una clave autogenerada)permitiendo que el usuario almacene la clave en su almacén:

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.cipherData();
```

```
var cipheredData = clienteFirma.getCipherData();
…
```

- Cifrado con las opciones por defecto (algoritmo de cifrado AES con una clave autogenerada) NO permitiendo que el usuario almacene la clave en su almacén:

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.setUseCipherKeyStore(false);
clienteFirma.cipherData();
var cipheredData = clienteFirma.getCipherData();
…
```

- Cifrado con una clave tomada del almacén del usuario (si no existiese se solicitaría directamente al usuario):

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.setKeyMode("USERINPUT");
clienteFirma.cipherData();
var cipheredData = clienteFirma.getCipherData();
…
```

- Descifrado con una clave tomada del almacén del usuario:

```
…
clienteFirma.setFileuri("fichero_cifrado");
clienteFirma.decipherData();
var plainData = clienteFirma.getPlainData();
…
```

9 Otras funcionalidades

9.1 Guardar la firma en un fichero

El método **saveSignToFile** permite guardar la última firma generada en un fichero. Se puede especificar la ruta al fichero con **setOutFilePath**, que recibe una cadena con la ruta al fichero de salida. Si no se especifica, se permitirá elegir al usuario.

Si el integrador es quien ha decidido la ruta de guardado mediante el método **setOutFilePath**, se pedirá confirmación al usuario para el guardado del fichero.

Si el fichero ya existe, se pide confirmación:

Figura 12: Diálogo para confirmar la sobreescritura de ficheros

9.2 Obtener el certificado usado para firmar

Es posible recuperar el certificado utilizado en la última operación de firma mediante el método **getSignCertificate**. Este método obtiene una instancia de la clase X509Certificate de Java.

El método **getSignCertificateBase64Encoded**devuelve una cadena de texto con el certificado, codificado en base 64,utilizado en para la última firma realizada. El certificado no estará delimitado por ninguna cadena ASCII ni carácter extra.

9.3 Leer el contenido de un fichero de texto

El método **getTextFileContent**que recibe como parámetro una URI a un fichero devuelve el contenido del mismo como una cadena. Si el fichero está almacenado en local, la URI comenzará por “file:///”.

9.4 Leer el contenido de un fichero en Base64

El método **getFileBase64Encoded**que recibe dos parámetros (ruta al fichero y un booleano que indica si mostrar o no gráficamente al usuario el progreso en la lectura del fichero). En caso de producirse un error se devolverá **null**.

9.5 Convertir un texto plano a Base64

El método **`getBase64FromText`**recibe como parámetros un texto plano y el nombre de un juego de caracteres y codifica este primero a base 64 en base al juego de caracteres indicado. En caso de producirse un error se devolverá **`null`**.

Es importante tener en cuenta que el indicar un juego de caracteres u otro en este método no implica que el texto se recodifique antes de convertirse en Base64, sino que la secuencia interna de caracteres se interprete de una u otra manera. En caso de dudas sobre el juego de caracteres más apropiado a usar, se debe especificar **`null`**, y el sistema usará el por defecto o el más adecuado. Este aspecto es de especial relevancia en el caso de textos que representen XML bien formados, ya que la especificación de un juego de caracteres inadecuado provocará la introducción de caracteres extraños que invalidarán el XML.

9.6 Obtener el hash de un fichero

El método **`getFileHashBase64Encoded`** devuelve una cadena con el hash de un fichero codificado en base 64. En caso de producirse un error se devolverá **`null`**.

9.7 Obtener la estructura de un envoltorio CMS

Como método de diagnóstico podemos obtener la estructura de un CMS encriptado, envuelto o autenticado mediante la llamada a los métodos **`formatEncryptedCMS`**o**`formatEnvelopedCMS`** (ambos son compatibles con los distintos tipos de sobres), pasándole como parámetros la cadena en base 64 correspondiente al CMS del que queramos obtener su estructura.

9.8 Obtener la ruta de un fichero

Para permitir a un usuario obtener la ruta completa de un fichero el cliente dispone del método **`loadFilePath(String, String, String).`** Este método abre una ventana modal para la selección de un fichero. Es posible configurar el diálogo de selección a través de los parámetros que recibe la función y que son respectivamente:

- El título de la ventana de selección.
- El listado de extensiones permitidas separadas por “$%$”.
- La descripción del fichero que se busca.

Todos los parámetros pueden ser nulos.

La salida de este método puede utilizarse para configurar la entrada del cliente mediante el método **`setFileuri`**.

ADVERTENCIA: Este método bloquea el script desde el que se ejecuta a la espera de que el usuario seleccione un fichero mediante el diálogo mostrado. Este comportamiento puede hacer que el navegador Mozilla Firefox 3.6 muestre al usuario una advertencia informando que el script está ocupado y puede ser dañino, dándole la posibilidad de bloquearlo. En caso de que se desee evitar esta interferencia, es responsabilidad del integrador ejecutar este método de forma asíncrona al resto del script (por ejemplo, mediante AJAX).

9.9 Obtener la ruta de un directorio

Para permitir a un usuario obtener la ruta completa de un directorio puede hacerse uso el método **`selectDirectory`**. Este método devuelve la ruta absoluta al directorio.

ADVERTENCIA: Este método bloquea el script desde el que se ejecuta a la espera de que el usuario seleccione un directorio mediante el diálogo mostrado. Este comportamiento puede hacer que el navegador Mozilla Firefox 3.6 muestre al usuario una advertencia informando que el script está ocupado y puede ser dañino, dándole la posibilidad de bloquearlo. En caso de que se desee evitar esta interferencia, **es responsabilidad del integrador ejecutar este método de forma asíncrona al resto del script**.

10 Ejemplos de uso

Junto al cliente se distribuyen los siguientes ficheros HTML de ejemplo de uso del cliente:

- ***demoMultifirma.html*:** Ejemplos de firma, co-firma y contra-firma
- ***demoMultifirmaMasiva.html***: Ejemplo de multifirma masivaprogramática.
- ***demoFirmaDirectorios.html***: Ejemplo de multifirma masiva sobre directorios.
- ***demoCifrado.html:*** Ejemplo de cifrado.
- ***demoSobreDigital.html***: Ejemplo de CMS encriptado, CMS envuelto y CMS firmado y envuelto.
- ***demoKeyStores.html***: Ejemplo de la funcionalidad de cambio de almacén de certificados.
- ***demoLdap.html***: Ejemplo de la carga de certificados desde LDAP.

11 Buenas prácticas enla integración del cliente

11.1 Localizar el *Bootloader* y el directorio de instalables

Aunque la librería del cliente que facilita el uso del cliente (*instalador.js*) toma como dirección por defecto de los recursos del cliente la del HTML que lo carga, es muy recomendable el establecer estas direcciones explícitamente. En concreto, los parámetros a establecer se encuentran en el fichero *constantes.js* y son:

- **base**: Ruta del directorio en el que se encuentra el Applet de instalación y carga del cliente (*bootloader.jar*). Esta ruta debe apuntar al directorio en donde se encuentra este Applet, no al mismo. Por ejemplo, si la localización del Applet fuese "`http://www.minhap.es/clienteAfirma/afirmaBootLoader.jar`" la dirección que se debería establecer sería "`http://www.minhap.es/clienteAfirma`".
- **baseDownloadUrl**: Ruta del directorio con los ficheros instalables del cliente (librerías nativas, fichero *version.properties* etc.). Esta ruta debe apuntar al directorio en donde se encuentran los instalables, no a un fichero concreto.

Las rutas indicadas pueden ser absolutas o relativas. Las rutas absolutas deben comenzar por "`file:///`" (nótese la triple barra), "`http://`" o "`https://`" (por ejemplo, "`file:///C:/ficheros`", "`http://www.minhap.es/ficheros`",...) y las rutas relativas no pueden empezar por "/" (por ejemplo, "`afirma/ficheros`"). Se debe usar siempre el separador "/", nunca "\".

La configuración de estas rutas, asegura la completa localización del cliente independientemente de la distribución de los HTML de la aplicación Web o de si estos se generan automáticamente. En este último caso sería necesario establecer la ruta absoluta de los directorios.

11.2 Indicar siempre la construcción mínima requerida del cliente

Aunque puede utilizarse el método JavaScript "`cargarAppletFirma()`" para asegurar que se carga el cliente @firma es aconsejable el pasarle siempre como parámetro a este método la construcción mínima que requiera nuestra aplicación. Esto es utilizar los parámetros 'LITE' (por defecto), 'MEDIA' o 'COMPLETA'. Esto asegurará que la construcción instalada dispone de todas las capacidades requeridas por nuestra aplicación.

No es recomendable cargar siempre el cliente con el parámetro 'COMPLETA' salvo que se requiera de alguna de las capacidades exclusivas de esta construcción, ya que si no se necesitan éstas podemos estar obligando a los usuarios a descargarse una mayor cantidad de datos que no le reportarán beneficio alguno. Por este motivo se debe utilizar siempre como parámetro el identificador de la construcción mínima exigida.

11.3 Reducir las opciones de configuración

Siempre debe ofrecerse al usuario el menor número de opciones de configuración posibles sobre el proceso de firma o cualquier otra operación criptográfica. Son dos los aspectos que llevan a esta decisión:

- El cliente de firma comúnmente se integra en las aplicaciones Web para un fin determinado como puede ser el envío de un formulario Web firmado, por ejemplo, por lo que es el sistema de *backend* el receptor de los datos generados y el que finalmente debe almacenarlos y gestionarlos. En este caso, es lógico que sea el integrador el que decida la configuración y condiciones de la operación.
- La finalidad del usuario, no suele ser el propio uso del cliente, sino el acceso al servicio dado por la aplicación que lo integra. De esta forma, los usuarios no tienen porqué conocer detalles de las operaciones criptográficas que se realizan y ni siquiera conocimientos de los conceptos relacionados con la firma electrónica. En estos casos conviene simplificarle la tarea y no llevarle a dudar acerca de la opción más acertada para su fin concreto.

11.4 Configuración y uso del cliente en operaciones únicas

En el caso de que la ejecución de las operaciones del cliente dependan de una configuración introducida por el usuario o generada en tiempo de ejecución, es recomendable el realizar la configuración y ejecución de la operación criptográfica sin dar posibilidad de alterar el proceso. Por ejemplo, una forma de proceder sería el inicializar y configurar el cliente nada más cargarlo (filtro de certificados, datos obtenidos de una ventana anterior…) y establecer el resto de la configuración a medida que el usuario inserta los datos (formato de firma, datos a firmar, certificado de usuario,…) para, finalmente, sólo ejecutar la operación de firma. Este mecanismo tiene el inconveniente que cualquier interrupción en el cliente puede desechar toda esa información y terminar operando con una configuración por defecto en lugar de la indicada por el usuario.

En su lugar, es recomendable que, una vez se vaya a realizar la operación criptográfica, sea cuando se configure el cliente. Como ejemplo, en una implementación genérica JavaScript de invocación al cliente esto sería:

```
// Inicializamos la configuración para asegurar que no hay preestablecido
// ningún valor de operaciones anteriores
clienteFirma.initialize();

// Configuramos todos los parámetros del cliente, ya sea con datos directorios o
// extraídos de la página (formularios, contexto de la aplicación,…)
clienteFirma.setSignatureFormat("CADES");
clienteFirma.setSignatureAlgorithm("SHA1withRSA");
clienteFirma.setFileuri(document.getElementById("fichero").value);

// Ejecutamos la operación que corresponda
clienteFirma.sign();
```

Este modo de ejecución ayudará a evitar que, por ejemplo, el refrescar la página Web con F5 se pierda la sincronización con la configuración real del cliente con la que pueda verse en un momento determinado en la página Web. El uso de la tecla F5 o el botón “Refrescar Pantalla” debe evitarse siempre cuando nos encontremos a medias de un procedimiento online. En el caso de que el entorno de despliegue pueda detectarlo, incluso es recomendable que se obligue al usuario a reiniciar el procedimiento completo.

12 Funciones y métodos en la interfaz Applet del cliente @firma v3.x añadidos respecto a versiones anteriores

public String getCertificate(final String alias)

Obtiene el certificado X.509 correspondiente al alias proporcionado. El resultado es el certificado en Base64 delimitado por las cadenas ASCII `-----BEGIN CERTIFICATE-----` y `-----END CERTIFICATE-----`.

public String getCertificatePublicKey(final String alias)

Obtiene la clave pública del certificado X.509 correspondiente al alias proporcionado. El resultado es una clave RSA en Base64 delimitado por las cadenas ASCII `-----BEGIN RSA PUBLIC KEY-----` y `-----END RSA PUBLIC KEY-----`.

public String getCertificates()

Obtiene todos los certificados del almacén actual en una única cadena en donde los elementos se dividen mediante el separador STRING_SEPARATOR definido como constante en el cliente.El formato individual de los certificados es el mismo que el devuelto por el método `public String getCertificate(final String alias)`. También es posible obtener de forma segura un array con los certificados mediante el método JavaScript getCertificates() definido en “firma.js”.

public String[] getArrayCertificates()

Obtiene todos los certificados del almacén actual en un array unidimensional, con el mismo formato individual que el devuelto por el método `public String getCertificate(final String alias)`. **El uso de este método no está recomendado** debido a la incompatibilidad existente entre el formato de `array` de Java 5 y el motor JavaScript de Microsoft Internet Explorer.

public String getCertificatesAlias()

Se ha considerado útil que el integrador, vía JavaScript, pueda obtener los alias del almacén de certificados utilizado por el navegador Web activo. Este método obtiene los alias de los certificados en una única cadena separándolos mediante la constante STRING_SEPARATOR definida en el cliente. También es posible obtener de forma segura un array con los alias de los certificados mediante el método JavaScript getCertificatesAlias() definido en “firma.js”. Para más información, consulte la documentación en formato JavaDoc.

public String[] getArrayCertificatesAlias()

Se ha considerado útil que el integrador, vía JavaScript, pueda obtener los alias del almacén de certificados utilizado por el navegador Web activo. Para más información, consulte la información en formato JavaDoc.**El uso de este método no está recomendado** debido a la incompatibilidad existente entre el formato de `array` de Java 5 y el motor JavaScript de Microsoft Internet Explorer.

public void setSelectedCertificateAlias(String certAlias)

Como complemento al método anterior, se ha considerado útil que el integrador, vía JavaScript, pueda establecer el alias del certificado a utilizar por el Applet en el navegador Web activo. Para más información, consulte la información en formato JavaDoc.

public boolean signDirectory()

Para las funciones de firma masiva, firma todos los archivos de un directorio según la configuración establecida. Para más información, consulte la información en formato JavaDoc.

publicbooleaninitMassiveSignature()

Inicializa una operación de firma masiva programática. Esto toma la configuración actual de certificado, formato de firma, algoritmo, modo, etc y la aplica a cada firma individual generara mediante los métodos `massiveSignatureData()`, `massiveSignatureFile()` y `massiveSignatureHash()`. La operación de firma masiva programática finaliza al invocarse al método `endMassiveSignature()`. Para más información, consulte la información en formato JavaDoc.

publicvoidendMassiveSignature()

Finaliza un proceso de firma masiva. Hasta que no se inicie un nuevo proceso mediante el método `initMassiveSignature()` no será posible realizar firmas/multifirmas mediante los métodos `massiveSignatureData()`, `massiveSignatureFile()` y `massiveSignatureHash()`. Para más información, consulte la información en formato JavaDoc.

publicStringmassiveSignatureData(String b64Data)

Genera una firma/multifirma, dentro de un proceso de firma, a partir de los datos indicados en base 64. La operación concreta realizada se debe indicar con `setMassiveOperation(String)`. El método devuelve el resultado de la operación criptográfica. Para más información, consulte la información en formato JavaDoc.

publicStringmassiveSignatureFile(String path)

Genera una firma/multifirma, dentro de un proceso de firma, a partir del fichero cuya ruta se ha indicado. La operación concreta realizada se debe indicar con `setMassiveOperation(String)`. El método devuelve el resultado de la operación criptográfica. Para más información, consulte la información en formato JavaDoc.

publicStringmassiveSignatureHash(String b64Hash)

Genera una firma/multifirma, dentro de un proceso de firma, a partir del hash indicado en base 64. Este método requiere que se haya establecido la operación de firma mediante el método `setMassiveOperation(String)`. El método devuelve el resultado de la operación criptográfica. Para más información, consulte la información en formato JavaDoc.

public void setMassiveOperation(String massiveOperation)

Para las funciones de firma masiva, establece la operación masiva a realizar en el proceso generado por el método`signDirectory()` o los métodos de firma masiva programática (`massiveSignatureData()`, `massiveSignatureFile()` y `massiveSignatureHash()`). Las operaciones masivas aceptadas son “FIRMAR”, “COFIRMAR”, “CONTRAFIRMAR_ARBOL” y “CONTRAFIRMAR_HOJAS”. Para más información, consulte la información en formato JavaDoc.

public void setOriginalFormat(boolean originalFormat)

Para las funciones de firma masiva, indica si se debe respetar el formato de firma original para las operaciones de multifirmamasiva. Para más información, consulte la información en formato JavaDoc.

public void setOriginalFormat(boolean originalFormat)

Para las funciones de firma masiva, indica si se debe respetar el formato de firma original para las operaciones de multifirmamasiva o, si en cambio, se usará la configuración de firma establecida paratodas las firmas. Para más información, consulte la información en formato JavaDoc.

public String getInputDirectoryToSign()

Para las funciones de firma masiva, devuelve la ruta absoluta del directorio donde se ubican los ficherosa ser firmados de forma masiva. Para más información, consulte la información en formato JavaDoc.

public void setInputDirectoryToSign(String directory)

Para las funciones de firma masiva, establece el directorio de donde se tomarán los ficheros de firma y datos para laoperación de firma masiva. Para más información, consulte la información en formato JavaDoc.

public String getOutputDirectoryToSign()

Para las funciones de firma masiva, devuelve la ruta absoluta del directorio donde se almacenarán las firmas resultado dela operación de firma masiva. Para más información, consulte la información en formato JavaDoc.

public void setOutputDirectoryToSign(String directory)

Para las funciones de firma masiva, establece el directorio donde se depositarán las firmas masivas de los archivossituados en `InputDirectoryToSign`. Para más información, consulte la información en formato JavaDoc.

public void setInIncludeExtensions(String extensions)

Para las funciones de firma masiva, define las extensiones que se incluirán en la firma de directorios. Para más información, consulte la información en formato JavaDoc.

public void setInRecursiveDirectorySign(boolean recursiveSignDir)

Para las funciones de firma masiva, establece si la firma de directorios se efectuará de forma recursiva o no. Para más información, consulte la información en formato JavaDoc.

public void setFileuriBase64(Stringuri)

Establece los datos contenidos en el fichero indicado (en donde se encontrarán codificados en base 64), como los datos de entrada para las operaciones criptográficas y establece la ruta introducida como ruta de entrada.

El contenido del fichero se interpretará siempre como datos en base 64 no realizándose la comprobación previa de los mismos.

publicStringloadFilePath(Stringtitle, Stringexts, Stringdescription)

Muestra un diálogo modal para la selección de un fichero del que se recuperará su ruta completa. Para más información, consulte la información en formato JavaDoc.

addRFC2254CertificateFilter(String subjectFilter, String issuerFilter, Boolean signatureKeyUsage)

Agrega un filtro al listado de filtros de certificados deusuario. Los filtros limitarán los certificados que se muestran al usuario para su selección a sólo aquellos que cumplan, al menos, uno de los filtros definidos.

Para más información consulte el apartado Filtros de certificados.

resetFilters()

Elimina todos los filtros de certificado definidos hasta el momento.

Para más información consulte el apartado Filtros de certificados.

setMandatoryCertificate(boolean mandatory)

Establece que debe seleccionarse automáticamente un certificado de firma si este es el único del almacén de certificados o, en caso de establecer filtros, si es el único que los cumple.

Para más información consulte el apartado Filtros de certificados.

13 Casos problemáticos de despliegue e integración del cliente

13.1 Despliegue del cliente en servidores Web que requieren identificación de los usuarios mediante certificado cliente

13.1.1 Applets de Java y Autenticación con Certificado Cliente

Durante la instalación delas bibliotecas del entorno operativo requerido por el Applet en el sistema local del usuario, el Applet BootLoader establece varias conexiones con el servidor Web para descargarse los ficheros necesarios.

En los servidores en los que se requiere al cliente que se identifique mediante un certificado (autenticación por certificado cliente) cuando solicita datos o una conexión, los Applets Java no son una excepción, por lo que el BootLoader debe identificase de forma independiente del navegador Web, ya que las conexiones que establece son independientes de este.

El Plugin de Java contempla esta posibilidad, y gestiona la autenticación por certificado cliente de forma independiente de los Applets que ejecute, por lo que estos no deben implementar ningún cambio para adaptarse a estos entornos, siendo todo el proceso completamente transparente para ellos.

El almacén que usa el Plugin de Java para seleccionar el certificado en muestra al servidor Web varía según la configuración cliente, pero sigue en todos los casos el mismo proceso:

1. Intenta acceder primero al almacén nativo del navegador Web (MS-CAPI, Apple KeyRing o Mozilla/Firefox NSS).
2. Si por problemas de configuración el almacén nativo no pudiese ser accedido por el Plugin de Java, se selecciona el almacén propio del JRE.
3. Se pide al usuario que seleccione uno de los certificados del almacén finalmente seleccionado, que será el que se envíe al servidor Web.

La clasificación de los navegadores Web por almacén utilizado para los certificados y por sistema operativo es la siguiente:

- Almacén de certificados MS-CAPI
  - Windows
    - Internet Explorer
    - Google Chrome
    - Apple Safari
    - Opera
- Almacén de certificados Apple KeyRing
  - Mac OS X

    - Apple Safari
    - Google Chrome
    - Opera
    - Internet Explorer
- Almacén de certificados Mozilla / Firefox (NSS)
  - Windows
    - Mozilla / Firefox
  - Linux / Solaris
    - Mozilla / Firefox
    - Google Chrome
    - Opera

A continuación, detallamos la configuración adicional necesaria en cada uno de los casos para el correcto funcionamiento en servidores Web que soliciten certificado cliente. Esta configuración no es específica para el Cliente @firma, sino que será necesaria para cualquier otro Applet de Java que establezca conexiones independientes del navegador con el servidor Web:

MS-CAPI

No es necesario ningún proceso adicional de configuración. El entorno de ejecución de Java, desde su versión 1.5 accede directamente al almacén CAPI.

Mozilla / Firefox (NSS)

Es necesario instalar previamente en el entorno de ejecución de Java las bibliotecas JSS (Netscape Java Security Services), atendiendo a las siguientes precauciones:

- Ha de seguirse el proceso de instalación exactamente como se describe en la documentación de Java: http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/keystores.html.
  - Las instrucciones publicadas por Sun Microsystems están desactualizadas y no aplican para las últimas versiones de Mozilla Firefox, para las cuales deben seguirse los siguientes pasos:
    1. Copiar el fichero ***jss4.jar*** al directorio de extensiones del entorno de ejecución de Java (JRE) en uso:
       - `%JAVA_HOME%\lib\ext` en sistemas Windows
       - `$JAVA_HOME/lib/ext` en sistemas basados en UNIX (Linux, Solaris, Mac OS X)

2. Copiar el fichero la biblioteca nativa de JSS en el directorio principal de bibliotecas del sistema operativo. Es necesario cerciorarse de que la biblioteca que copiemos corresponda con la versión y arquitectura de nuestro sistema operativo:
    - En sistemas Windows, el fichero ***jss4.dll*** debe copiarse al directorio `%SystemRoot%\system32`
    - En sistemas Linux y Solaris, el fichero ***jss4.so*** debe copiarse al directorio `/lib` o al directorio `/usr/lib`
    - En sistemas Mac OS X, el fichero ***jss4.dylib*** o el fichero ***jss4.so*** (dependiendo de la compilación utilizada) debe copiarse al directorio `/lib` o al directorio `/usr/lib`

- Hemos también de asegurarnos de que la versión instalada de JSS sea compatible con nuestra versión de Mozilla / Firefox. Consulte atentamente la documentación de los productos antes de proceder a instalarlos.
    - Firefox 3 solo es completamente compatible con JSS 4.3.2 y superiores:https://developer.mozilla.org/En/JSS/4_3_ReleaseNotes, pero aun presenta ciertos problemas de compatibilidad en sistemas Windows. En cualquier caso, si encuentra dificultades, pruebe a instalar siempre la versión más actualizada.
- JSS puede descargarse de forma libre desde: ftp://ftp.mozilla.org/pub/mozilla.org/security/jss/releases/
    - No obstante, para ciertas versiones de JSS, la Comunidad Mozilla no distribuye binarios, sino únicamente el código fuente.

Almacén propio de Java

Como se ha comentado anteriormente, cuando el *Plugin* del entorno de ejecución de Java (JRE) no puede acceder al almacén del navegador Web, solicita al usuario la selección de un certificado de su propio almacén.

El principal problema de esta opción es que JRE no accede a los módulos PKCS#11 o CSP del sistema, por lo que los certificados residentes en tarjetas inteligentes (incluido DNIe) o dispositivos externos (USB, etc.) no son accesibles.

Para comprobar los certificados existentes en el almacén de Java e importar certificados en él, se pueden seguir los siguientes pasos (sistemas operativos Windows):

1. Abrir la opción de “Java” en el Panel de Control y seleccionar la pestaña “Seguridad”:

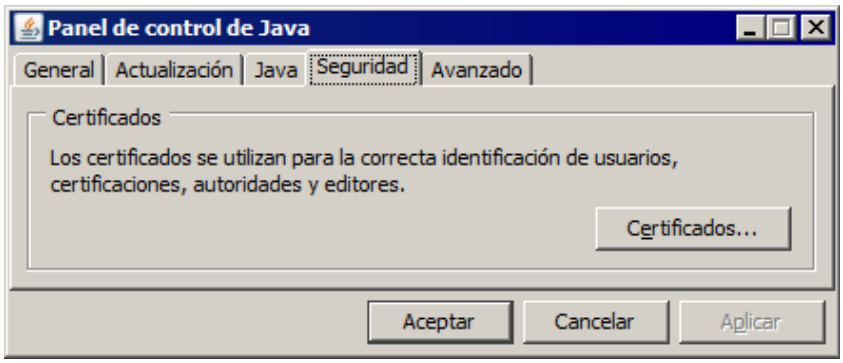


Figura 13: Panel de control de Java

2. Pulsar el botón “Certificados” y seleccionar el tipo de certificado como “Autenticación de cliente”, dentro de la pestaña “Usuario”.

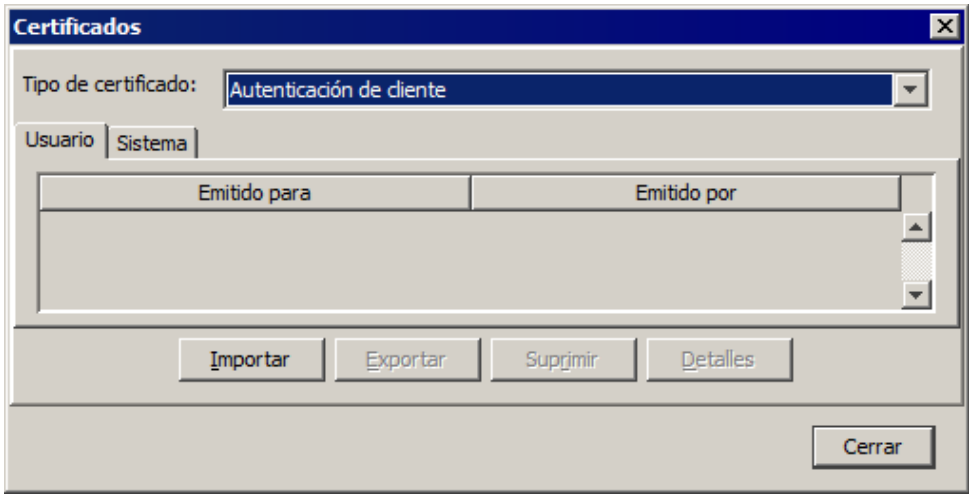


Figura 14: Certificados del almacén de Java

3. Los certificados mostrados en la lista son los disponibles por el Plugin de Java para autenticar a un Applet ante un servidor Web que requiere certificado cliente. Podemos importar nuevos certificados usando el botón “Importar”.

Para realizar las mismas comprobaciones en otros sistemas operativos, consulte la documentación del entorno de Java instalado (JRE) y del Java Plugin.

13.1.2 Alternativa de despliegue

Una variante sobre lo arriba expuesto es el no publicar el cliente @firma al completo en un servidor con autenticación con certificado cliente, tan sólo las páginas Web que dan acceso al mismo. Es posible situar, tanto el *Applet*de carga como los ficheros instalables en un segundo servidor (o una ruta del mismo configurada para no pedir certificados). Con esto obtenemos que:

- Páginas Web: En servidor con conexión SSL y autenticación con certificado cliente.
- Binarios Java: En servidor con conexión SSL o sin ella (según se desee) y sin autenticación.

El principal beneficio de esta alternativa está claro, no es necesario que los binarios java se autentiquen contra el servidor, por lo que no es necesario que el usuario configure ningún certificado en el repositorio de Java.

Para configurar este entorno basta con configurar las siguientes variables del fichero de despliegue "*constantes.js*":

- **base:** Esta variable apunta al directorio en el que se sitúa el applet de carga (bootloader.jar). Deberá establecerse para que apunte a la ruta en la que se sitúa este fichero (en un servidor que no requiere autenticación cliente). Por ejemplo, si la ruta del fichero es "`https://www.minhap.es/afirma/bootloader.jar`" el valor de la variable deberá ser "`https://www.minhap.es/afirma`". También es posible establecer una ruta relativa hasta el directorio del aplicativo.
- **baseDownloadURL:** Esta variable apunta al directorio en el que se encuentran los ficheros necesarios para la instalación delas necesidades en cuanto a entorno operativo del cliente. Deberá establecerse para que apunte a la ruta en la que se sitúan estos (en un servidor o un directorio en donde no se requiera autenticación cliente). Por ejemplo, "`https://www.minhap.es/afirma/instalables`" o "`../instalables`".

Los ficheros que deberán situarse en el directorio de instalables son la totalidad de los indicados para despliegue.

13.2 Problema con el objeto HTML File en los nuevos navegadores

La nueva generación de navegadores Web (Internet Explorer 8, Firefox 3,…) ha restringido el comportamiento del objeto *File* de HTML por motivos de seguridad.

La finalidad de este componente es únicamente el permitir la carga de ficheros a un servidor. Sin embargo, antes se le permitía obtener a este servidor excesiva información sobre el sistema del usuario ya que, determinados navegadores, mediante JavaScript, proporcionaban la ruta completa en la que estaba almacenado el fichero.

Los nuevos navegadores no permiten obtener más que el nombre del fichero o, a lo sumo, este y una ruta genérica y no descriptiva.

Esto podría inhabilitar la práctica que han seguido muchos integradores del cliente @firma, que utilizaban este componente para, además de la carga del fichero, obtener la ruta del mismo para así utilizar el método `setFileuri()` y operar con el fichero.

Con objeto de solventar en parte este problema, se ha incluido en el Applet cliente el método `loadFilePath()` que muestra al usuario un diálogo para la selección de un fichero de datos y devuelve la ruta completa de ese fichero. Con esto el usuario es libre de usar la ruta con el método `setFileuri()` para realizar la firma del fichero y/o mostrársela al usuario en un cuadro de texto, por ejemplo. Puede encontrarse la descripción completa del método en el JavaDoc del cliente @firma.

Sin embargo, al igual que no permite obtener la ruta del fichero seleccionado, el objeto *File* no permite establecer de forma externa una ruta de fichero para su posterior subida al servidor, ya que esto posibilitaría a cualquier página Web maliciosa a obtener ficheros del disco duro del usuario sin su consentimiento. Esto imposibilita el tomar la ruta del fichero mediante el método `loadFilePath()` para luego cargarlo con un objeto *File*.

La solución a este problema puede llevarse a cabomediante el uso de diferentes mecanismos, cada cual ajustado al entorno, el fin y las tecnologías con las que cuente el integrador del sistema.

Una solución simple sería, por ejemplo, el leer el fichero mediante el método `getFileBase64Encoded()` y anexar la cadena en base 64 resultante como campo oculto al formulario del usuario (configurar con el método POST). La ruta del fichero se habrá obtenido previamente con la ayuda del método `loadFilePath()` y especificado al cliente mediante `setFileuri()`.

ADVERTENCIA: Si su sistema requiere o permite que el usuario envíe ficheros mayores de 4 megas de tamaño, consulte el apartadoProcedimiento de carga para ficheros mayores de 4MB.

13.3 Procedimiento de carga para ficheros mayores de 4MB

Al ejecutar el Cliente @firma en un entorno con Java 6u10 o superior y el plugin de nueva generación activado (configuración por defecto), nos encontramos con que no es posible convertir ficheros de datos mayores de 4MB a cadenas Base64. Esta operación es necesaria para posteriormente adjuntar los datos firmados (o la firma implícita generada) al formulario Web a través del cual se enviará la información al servidor. Esta limitación también puede afectar a la generación de firmas XML implícitas de ficheros mayores de 4MB.

Este problema no tiene solución actualmente pero es posible realizar algunas prácticas con las que es posible evitarlo en caso de que el propio fichero de datos no sea mayor de este tamaño.

1. Evalúe si es necesario que su sistema firme los ficheros adjuntos a una transacción o si basta con firmar la propia transacción. Esto podría hacerse mediante un XML en el que se almacenen los datos de la transacción (identificador, los datos proporcionados por el usuario, nombre de los ficheros adjuntos y su hash,…).
2. Si su sistema realiza firmas de ficheros seleccionados por el usuario y se van a admitir ficheros mayores de 4MB, evalúe el uso de firmas binarias (CAdES), que son de menor tamaño, en lugar de firmas XML (XAdES).El problema comentado puede afectar a la generación de firmas XML (XMLdSig / XAdES) de ficheros binarios mayores de 4MB.
3. Si es necesario el envío de ficheros mayores de 4MB al servidor,deberán enviarse mediante el componente File de los formularios HTML. Para esto,tendremos que firmar

previamente los datos y obligar a que sea el propio usuario quien seleccione los ficheros de firma generados.Se propone el siguiente modelo de aplicación Web:

a. Mostrar al usuario el formulario Web con la información que debe rellenar. Esto puede hacerse en una única página Web o en varias si la cantidad de datos lo requiere.

b. En el punto que corresponda del formulario, se dará la opción al usuario de seleccionar los ficheros que desea adjuntar al mismo. Esto abrirá una nueva ventana en donde se cargará el Cliente @firma y, mediante el método descrito en el apartado13.2Problema con el objeto HTML File en los nuevos navegadores, se dará al usuario la posibilidad de firmar los ficheros. En este caso, en lugar de adjuntar el resultado de la firma al formulario Web, se le permitirá almacenarla en disco,notificándole que esta es la firma electrónica generada que posteriormente se deberá adjuntar al formulario y que, si lo desea, puede conservar como parte del resguardo de la transacción. En este paso se pueden firmar tantos ficheros como se deseen. Consulte el apartado 9.1Guardar la firma en un fichero para conocer como almacenar las firmas en el sistema del usuario.

c. De vuelta al formulario principal y al final del mismo se mostrará un botón Aceptar que redirigirá al usuario a una nueva página en la que se cargará el Cliente @firma y se mostrará el resumen de los datos del formulario para que confirme que son válidos. También en esta página se mostrarán los componentes necesarios de tipo File de HTML para que el usuario cargue los ficheros de firma generados en el paso anterior (y los documentos firmados en caso de firmas explícitas). En esta ocasión no se utilizará el Cliente para cargar los ficheros de firma, únicamente el componente File

d. Tras revisar los datos y seleccionar los ficheros necesarios, el usuario podrá enviar el formulario para finalizar el trámite. Al pulsar el botón Enviar, se firmará la transacción con el Cliente @firma y seguidamente se enviará el formulario con esta firma.

    - El concepto de transacción deberá definirse para cada sistema. Puede ser, por ejemplo, un XML que contenga todos los datos del formulario y la relación de ficheros adjuntos (nombres y hashes).

NOTA: En sistemas con el Plugin de próxima generación desactivadoel límite se encuentra en torno a los 50MB. Sin embargo, no debe presuponerse que el usuario operará desde este entorno.

13.4 Mensajes de confirmación durante el proceso de firma masiva

A partir del Cliente @firma v3.3, cualquier acceso a disco como leer y guardar datos requiere del consentimiento expreso del usuario. Este procedimiento no afecta a la mecánica de las aplicaciones que integran el Cliente, por lo que no requerirán ningún tipo de modificación, salvo en casos concretos de firma masiva de datos.

El proceso de firma masiva programática dispone de un método para la firma de ficheros, asegúrese de utilizar este método (**`massiveSignatureFile()`**) y no el de firma de datos (**`massiveSignatureData()`**) si va a firmar ficheros en disco.

Por otra parte, el proceso de firma masiva programática no dispone de un método propio para el guardado de las firmas en disco. En anteriores versiones del Cliente era posible utilizar los métodos comunes de guardado para almacenar las firmas pero a partir de la versión 3.3 esto supone que se pida confirmación para el guardado de cada firma individual. Por motivos de seguridad, este comportamiento no puede evitarse.

Si se desea firmar ficheros y almacenar el resultado en disco, consulte el apartado "**¡Error!No se encuentra el origen de la referencia.¡Error!No se encuentra el origen de la referencia.**" en donde se detalla el mecanismo de firma de directorios. Este mecanismo permitiría firmar y almacenar todos los ficheros sin necesidad de que el usuario lo apruebe individualmente.

14 Refirmado de los componentes del Cliente

Si el integrador decide refirmar los componentes de firma con su propio certificado de firma de código debe tener los siguientes factores en cuenta:

- Antes de firmar el fichero bootloader.jar debe introducirse el certificado raíz del nuevo firmante en la ruta `/resources/integrator_Code_Signing_CA.cer`. El nombre del certificado debe ser siempre "`integrator_Code_Signing_CA.cer`" y estar codificado directamente en ASN.1 DER.
  - El certificado debe corresponder a la autoridad de certificación que firmó directamente el certificado con el que se desea firmar el código (el inmediatamente siguiente en la cadena de certificación).
  - Para introducir un fichero dentro de un JAR puede utilizar cualquier herramienta común de manejo de ficheros ZIP (WinZip, WinRAR, 7Zip, etc.). Recuerde que cualquier operación debe hacerse siempre antes de firmar el fichero.
- Los ficheros `sunmscapi.jar` y `sunpkcs11.jar` jamás deben ser refirmados, puesto que se deben distribuir con la firma de Sun Microsystems / Oracle específica para extensiones criptográficas (JCE, Java Cryptography Extension).
- Se deben firmar el resto de los ficheros JAR.
- Se deben firmar la totalidad de los ficheros ZIP, utilizando firmas JAR y la misma herramienta de firmado JAR (habitualmente *JarSigner* de Su Microsystems / Oracle).

15 Siglas

CAdES	*CMS Advanced Electronic Signature*
CMS	*Cryptographic Message Standard*
CSP	*Cryptographic Service Provider* (proveedor de servicios criptográficos)
DNIe	DNI electrónico
JAR	*Java Archive*
JCE	*Java Cryptography Extension*
JNLP	*Java Networking Launching Protocol*
JRE	*Java Runtime Environment*
PDF	*Portable Document Format*
PKCS#1	*Public Key Cryptography Standard number 1* (estándar de criptografía de clave pública nº 1)
PKCS#7	*Public Key Cryptography Standard number 7* (estándar de criptografía de clave pública nº 7)
PKCS#11	*Public Key Cryptography Standard number 11* (estándar de criptografía de clave pública nº 11)
PKCS#12	*Public Key Cryptography Standard number 12* (estándar de criptografía de clave pública nº 12)
PKI	*Public Key Infrastructure*
SHA	*Secure Hash Algorithm*
URI	*Uniform Resource Identifier*(Identificador Uniforme de Recursos)
URL	*Uniform Resource Locator*(Localizador Uniforme de Recursos)
WYSIWYS	*What You See Is What You Sign*(lo que ves es lo que firmas)
XAdES	*XML Advanced Electronic Signature*(firma electrónica avanzada XML)
XML	*eXtensible Markup Language*(Lenguaje de marcas extensible)
XMLDSig	*XML Digital Signature* (firma digital XML)

16 Documentos de Referencia

[JAVADOC] Documentación de los métodos públicos de los Applets Bootloader y de Firma en la carpeta `javadoc`.

Anexo A. Formatos de firma binaria genérica soportados por el cliente

Matriz de formatos soportados en formatos binarios (CMS y CAdES)

		Firma Digital		Sobre Digital				
		Implícito	Explícito	Cifrado	Envuelto	Envuelto y Firmado	Autenticado	Autenticado y Envuelto
CMS	Firma							
	Cofirma							
	Contrafirma							
CAdES	Firma							
	Cofirma							
	Contrafirma							

Leyenda:	
	Soportado y acorde a estándar
	Soportado pero con problemas de adecuación a estándar
	No soportado
	No contemplado en el estándar

Adicionalmente, deben observarse las siguientes aclaraciones sobre los formatos:

- Las firmas CMS generadas son compatibles PKCS#7
- Las firmas CAdES generadas son compatibles con la especificación CAdES-BES o CAdES-EPES.

Algoritmos de huella digital

El cliente de firma soporta (con las salvedades indicadas en las notas posteriores) la aplicación de los siguientes algoritmos de huella digital para las firmas binarias:

- MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512.

NOTAS IMPORTANTES SOBRE LOS SOBRES DIGITALES:

Ciertos dispositivos de seguridad, incluido el DNIe y las tarjetas CERES de la FNMT-RCM no soportan operaciones de descifrado RSA usando clave privada, por lo que son incapaces de abrir sobres digitales. Debe informarse convenientemente a los usuarios para que no traten de enviar sobres digitales usando claves públicas de DNIe o de certificados CERES cuya clave privada resida en una tarjeta FNMT-RCM CERES. Esta limitación afecta a todas las plataformas (implementada tanto en el controlador MS-CAPI/CSP como en el PKCS#11).

Uso de los parámetros de funcionamiento

El modo de uso del cliente para establecer los parámetros de funcionamiento del Cliente consiste en realizar llamadas a ciertos métodos del Applet indicando cadenas de texto que identifican los valores que queremos establecer.

En paticular, se indican los formatos y sub-formatos (modos) de firma mediante unas cadenas de texto específicas. Además, cada formato o sub-formato introducido va asociado a una extensión de fichero (que define su tipo), extensiones que se usan como filtro de entrada o de salida a la hora de abrir o salvar archivos.

Las cadenas de identificación de formato deben usarse con la llamada JavaScript:

```
clienteFirma.setSignatureFormat(String format)
```

Y las cadenas de sub-formato con la llamada JavaScript:

```
clienteFirma.setSignatureMode(String mode)
```

Ambas funciones están documentadas en el JavaDoc del Applet, Remítase a estos documentos para más información. El orden de llamada de ambos métodos no es significativo.

Parámetros de funcionamiento

- Cadenas (se ignoran las diferencias entre mayúsculas y minúsculas) de identificación de formato (varias alternativas por cada uno de ellos, por flexibilidad de uso, se muestran separadas por "/"). **Aplica únicamente a firma digital, y no a sobres digitales**:
  - CMS
    - `"CMS" / "PKCS7" / "PKCS#7"`
  - CAdES
    - `"CAdES" / "CAdES-BES"`
- Cadenas de identificación del modo de firma (insensibles a mayúsculas/minúsculas). **Aplica únicamente a firma digital, y no a sobres digitales**:
  - Firma Explícita
    - `"Explicit"`
  - Firma Implícita
    - `"Implicit"`
- Ficheros de entrada (todos: CMS y CAdES, Implícitas y Explícitas)
  - Binarios (*.*)
- Ficheros de salida
  - CMS y CAdES
    - Ficheros de firma ASN.1 (*.csig)

Cofirmas cruzadas entre CMS y CAdES

Las cofirmas de un documento dan como resultado dos firmas sobre este mismo documento que se encuentran a un mismo nivel, es decir, que ninguna envuelve a la otra ni una prevalece sobre la otra.

A nivel de formato interno, esto quiere decir que cuando cofirmamos un documento ya firmado previamente, esta firma previa no se modifica. Si tenemos en cuenta que CAdES es en realidad un subconjunto de CMS, el resultado de una cofirma CAdES sobre un documento firmado previamente con CMS (o viceversa), son dos firmas independientes, una en CAdES y otra en CMS. Dado que todas las firmas CAdES son CMS pero no todas las firmas CMS son CAdES, el resultado global de la firma se adecúa al estándar más amplio, CMS en este caso.

Otro efecto de compatibilidad de formatos de las cofirmas con varios formatos de un único documento es la ruptura de la compatibilidad con PKCS#7, ya que, aunque las firmas generadas por el cliente mediante CMS son compatibles con PKCS#7, las generadas con CAdES no lo son, por lo que, en el momento que se introduzca una estructura CAdES, se perderá la compatibilidad PKCS#7 en el global de la firma.

Formato CMS de Firma Digital

Al igual que otros elementos CMS que describiremos posteriormente, la estructura de una firma CMS viene definida en la RFC 3852, aunque en este caso en especial, y para mantener la compatibilidad con PKCS#7, el elemento *signedData* especificado en la RFC indicada se encuentra limitado. La estructura empleada por el cliente de firma es la siguiente:

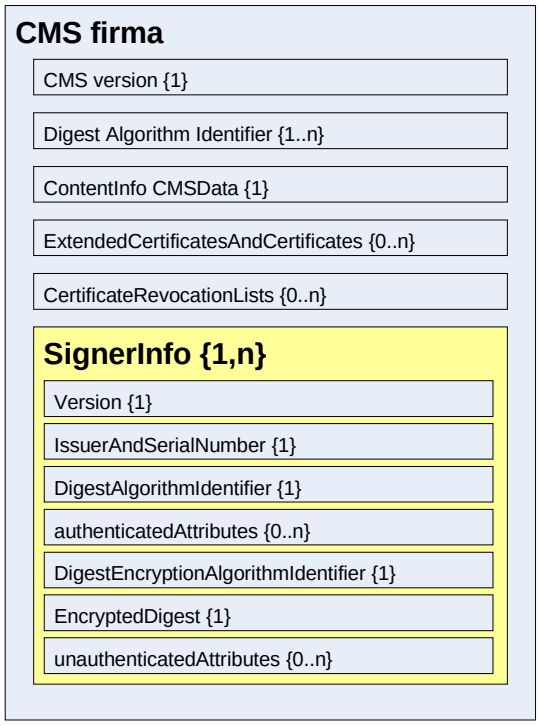


Figura 15: Estructura PKCS#7 SignedData

- **CMS Version**. Indica las diferentes versiones del mensaje. Para que la compatibilidad con PKCS#7 se mantenga debe ser 0.
- **Digest Algorithm Identifier**. Identifica el algoritmo utilizado.
- **ContentInfo**. Secuencia de parámetros que identifican el contenido del mensaje. Comprende el tipo de contenido (*contentType*) que en nuestro caso será *Data* y *content* que se refiere a la secuencia de bytes correspondiente a los datos mismos.
- **Extended Certificates And Certificates**. Opcional. Permite especificar una cadena de certificados para la validación de los distintos certificados firmantes.
- **Certificate revocation Lists**. Opcional. Permite especificar las CRL para los certificados utilizados.
- **Signer Info**. Estructura que especifica la información de los diferentes firmantes del contenido del mensaje. Se subdividen en los siguientes campos:
  - **Versión**. Especifica la versión de esta estructura y será siempre 1 para PKCS#7.

- **IssuerAndSerialNumber**. Especifica el certificado usado mediante el emisor y número de serie de éste.
- **DigestAlgorithm Identifier**. Identifica el algoritmo utilizado.
- **AuthenticatedAttributes**. Opcional. Secuencia de atributos firmados que especifican ciertos parámetros importantes para la interpretación del contenido. Si el tipo del contenido fuese distinto de *Data*, sería obligatorio incorporar como atributos el tipo empleado y el hash del contenido, pero en nuestro caso esto no es posible ya que siempre tendremos el *ContentType Data*.
- **DigestEncryptionAlgorithm**. Describe que algoritmo se ha usado en la encriptación de la firma y resumen del documento.
- **Encrypted Digest**. Hash del mensaje encriptado empleando la clave privada del certificado y el algoritmo especificado antes
- **UnauthenticatedAttributes**. Opcional. Atributos no firmados definidos en PKCS#9, como por ejemplo las contrafirmas.

Formato de sobre digital CMS encriptado

La estructura vendría definida como sigue:

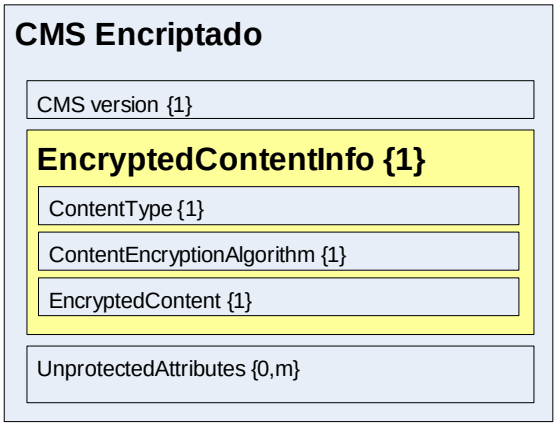


Figura 16: Estructura PKCS#7 EncryptedData

- **CMS Version**. Será 0 si no existen ***UnprotectedAtributes*** o 2 en caso contrario.
- **Encrypted Content Info**. Subestructura que se define mediante los siguientes campos:
  - **Content Type**. Define el tipo de contenido.
  - **Content Encryption Algorithm**. Define el algoritmo utilizado para encriptar el contenido.
  - **Encrypted Content**. Contenido encriptado usando el algoritmo especificado anteriormente.
- **Unprotected Attributes**. Opcional. Secuencia de parámetros auxiliares definidos por otros estándares.

Como se puede apreciar, esta estructura no contiene la clave de cifrado ni ningún método de transmisión de esta, por lo que si se usa como mensaje se debe buscar un método para compartir una clave privada.

Formato de Sobre Digital CMS Envuelto

Esta estructura se identifica con el sobre digital identificado en la RFC 3852 como **Enveloped CMS** y sigue la siguiente estructura:

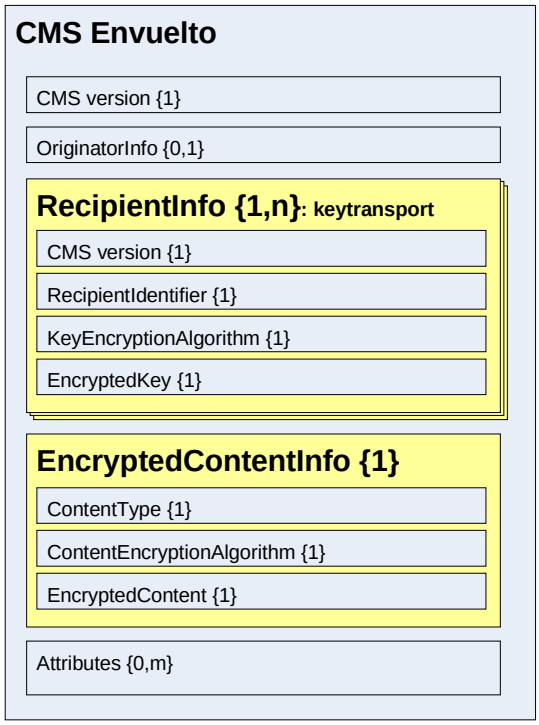


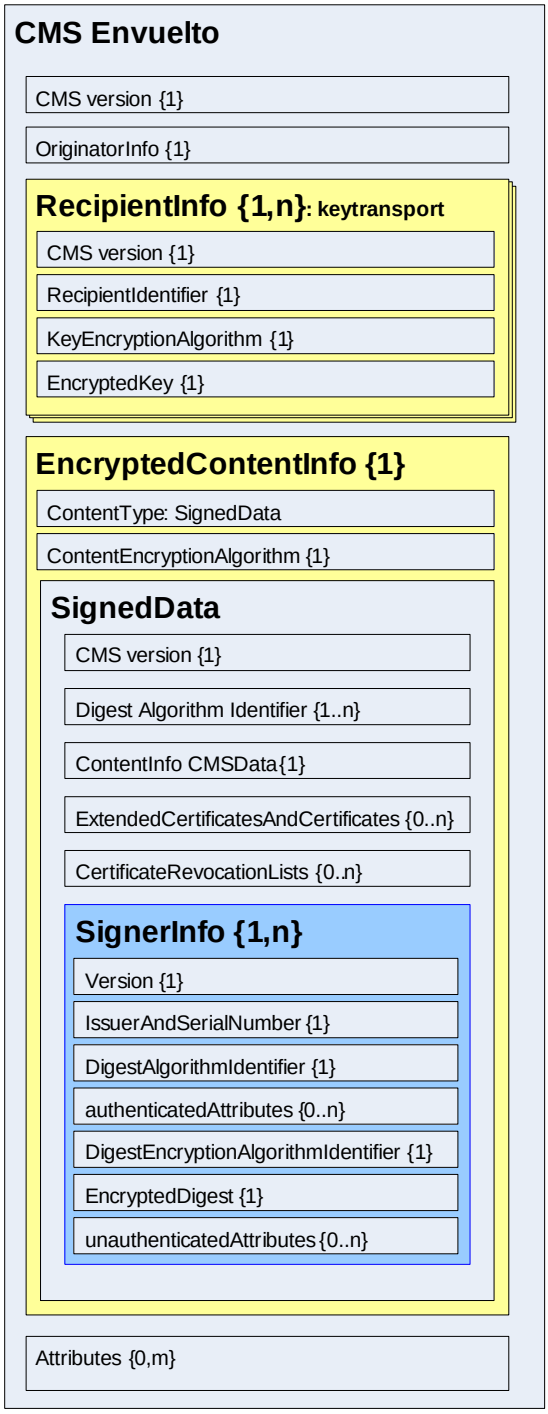
Figura 17: Estructura PKCS#7 EnvelopedData

- **CMS Version**. Viene determinada en función de los parámetros presentes en la estructura generada. Para que fuese compatible con la estructura especificada en PKCS#7 debería ser 0, pero quitaría mucha de las opciones más importantes que incorpora esta solución.
- **Originator Info**. Define el emisor del mensaje. Aunque es opcional, su presencia viene determinada por los algoritmos utilizados internamente.
- **Recipient Info**. Define los receptores válidos para el mensaje actual. Por requerimientos de la tecnología utilizada serán de tipo **keytransport**, ya que necesitamos incorporar la clave simétrica utilizada en el cifrado. Se distinguen los siguientes campos:
  - **Version**. Puede ser 0 o 2 en función de los datos incluidos en **Recipient Identifier**. En nuestro caso será 2.
  - **Key Encryption Algorithm**. Define el algoritmo por el cual se ha encriptado la clave simétrica adjunta. Se utilizan algoritmos asimétricos y en nuestro caso RSA.
  - **Encrypted Key**. Clave utilizada para encriptar el contenido del mensaje cifrada utilizando el algoritmo anteriormente definido.
- **Encrypted Content Info**. Estructura igual que la contenida en CMS encriptado:
  - **Content Type**. Define el tipo de contenido.

- **Content Encryption Algorithm**. Define el algoritmo utilizado para encriptar el contenido.
- **Encrypted Content**. Contenido encriptado usando el algoritmo especificado especificado.

- **Unprotected Attributes**. Conjunto de atributos no cifrados definidos o necesarios por otros estándares.

Formato de sobre digital CMS Firmado y envuelto



Esta estructura es un atajo para crear un CMS envuelto en cuyo interior se encuentra un mensaje firmado. Esto significa que la única diferencia en cuanto a la estructura es que el *conten type* de la subestructura *Encrypted Content Info* sería un *Signed Data* como el definido en el CMS Firmado.

La diferencia fundamental es que los parámetros a especificar no son tan libres, ya que por ejemplo es obligatorio especificar el emisor, ya que tenemos que firmar el mensaje con su certificado.

Este es un ejemplo de cómo se pueden anidar estructuras CMS. Por ejemplo, podríamos insertar un CMS envuelto en un CMS firmado (obviando la utilidad que pudiese tener o no) simplemente generando el CMS envuelto y especificando el resultado de la salida como datos de entrada para la creación del CMS firmado, y así sucesivamente.

Figura 18: Estructura PKCS#7 SignedAndEnvelopedData

Formato de sobre digital CMS Autenticado

Esta estructura se identifica con el sobre digital identificado en la RFC 3852 como Authenticated CMS y sigue la siguiente estructura:

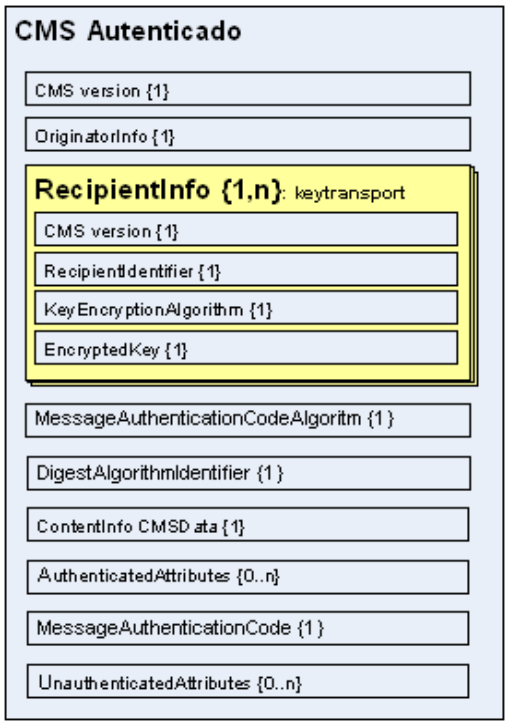


Figura 19: Estructura CMS AuthenticatedData

- **CMS Version.** Viene determinada en función de los parámetros presentes en la estructura generada. Para que fuese compatible con la estructura especificada en PKCS#7 debería ser 0, pero quitaría mucha de las opciones más importantes que incorpora esta solución.
- **Originator Info**. Define el emisor del mensaje. Aunque es opcional, su presencia viene determinada por los algoritmos utilizados internamente.
- **Recipient Info**. Define los receptores válidos para el mensaje actual. Por requerimientos de la tecnología utilizada serán de tipo **keytransport**, ya que necesitamos incorporar la clave simétrica utilizada en el cifrado. Se distinguen los siguientes campos:
  - **Version**. Puede ser 0 o 2 en función de los datos incluidos en **Recipient Identifier**. En nuestro caso será 2.
  - **RecipientIdentifier:** Identifica al usuario al que va dirigido el sobre.
  - **Key Encryption Algorithm**. Define el algoritmo por el cual se ha encriptado la clave simétrica adjunta. Se utilizan algoritmos asimétricos y en nuestro caso RSA.

- **Encrypted Key**. Clave utilizada para encriptar el contenido del mensaje cifrada utilizando el algoritmo anteriormente definido.

- **MessageAuthenticationCodeAlgorithm**: Define el algoritmo con el que se creará la MAC.
- **DigestAlgorithmIdentifier**:Identifica el algoritmo utilizado.
- **ContentInfo**: Secuencia de parámetros que identifican el contenido del mensaje. Comprende el tipo de contenido (*contentType*) que en nuestro caso será *Data* y *content* que se refiere a la secuencia de bytes correspondiente a los datos mismos.
- **AuthenticatedAtributes:** Opcional. Secuencia de atributos firmados que especifican ciertos parámetros importantes para la interpretación del contenido. Si el tipo del contenido fuese distinto de *Data*, sería obligatorio incorporar como atributos el tipo empleado y el hash del contenido, pero en nuestro caso esto no es posible ya que siempre tendremos el *ContentType Data*.
- **MessageAuthenticationCode:** Código que autentificael mensaje.
- **UnauthenticatedAttributes**. Opcional. Atributos no firmados definidos en PKCS#9, como por ejemplo las contrafirmas.

AnexoB. Configuración específica para el formato CAdES

El Cliente de firma genera firmas CAdES compatibles por estructura y atributos tanto con la versión 1.7.3 como con la 1.8.1, pero en ambas versiones, el atributo **Signing Certificate** se puede generar de dos formas distintas, la V1 y la V2.

Por defecto, y para una mayor compatibilidad, este atributo se genera de la forma V1 cuando la firma se genera con un algoritmo de firma cuya operación de huella digital es SHA1 y con la forma V2 cuando se usa cualquier otro algoritmo.

Este comportamiento se puede variar indicando explícitamente si deseamos usar o no la versión 2 del atributo. Esto se hará mediante el método del Applet (que es posible invocar vía JavaScript): `clienteFirma.addExtraParam(String paramName, String paramValue)`, y el siguiente uso:

```
clienteFirma.addExtraParam("signingCertificateV2", "true");
```

Desde la invocación de este método todas las firmas CAdES que se realicen hasta el reinicio del Applet tendrán la forma V2 del atributo Signing Certificate. Si queremos restablecer el comportamiento normal de generación en la forma V1 debemos invocar el paso de parámetro adicional de este otra forma:

```
clienteFirma.addExtraParam("signingCertificateV2", "false");
```

Para las firmas en CAdES que van a sufrir un tratamiento posterior acorde a la versión CAdES 1.8.1 (como por ejemplo sellos de tiempo complejos), se recomienda usar siempre la forma V2 del atributo Signing Certificate.

Creative Commons

Reconocimiento-NoComercial-CompartirIgual 3.0 Unported

Usted es libre de:

Compartir - copiar, distribuir, ejecutar y comunicar públicamente la obra

hacer obras derivadas

Bajo las condiciones siguientes:

Atribución — Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciante (pero no de una manera que sugiera que tiene su apoyo o que apoyan el uso que hace de su obra).

NoComercial — No puede utilizar esta obra para fines comerciales.

Compartir bajo la Misma Licencia — Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Entendiendo que:

Renuncia — Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor

Dominio Público — Cuando la obra o alguno de sus elementos se hallen en el dominio público según la ley vigente aplicable, esta situación no quedará afectada por la licencia.

Otros derechos — Los derechos siguientes no quedan afectados por la licencia de ninguna manera:

- Los derechos derivados de usos legítimosu otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
- Los derechos morales del auto;

- Derechos que pueden ostentar otras personas sobre la propia obra o su uso, como por ejemplo derechos de imagen o de privacidad.

Aviso — Al reutilizar o distribuir la obra, tiene que dejar muy en claro los términos de la licencia de esta obra. La mejor forma de hacerlo es enlazar a esta página.

Licencia

LA OBRA O LA PRESTACIÓN (SEGÚN SE DEFINEN MÁS ADELANTE) SE PROPORCIONA BAJO LOS TÉRMINOS DE ESTA LICENCIA PÚBLICA DE CREATIVE COMMONS (CCPL O LICENCIA). LA OBRA O LA PRESTACIÓN SE ENCUENTRA PROTEGIDA POR LA LEY ESPAÑOLA DE PROPIEDAD INTELECTUAL Y/O CUALESQUIERA OTRAS NORMAS QUE RESULTEN DE APLICACIÓN. QUEDA PROHIBIDO CUALQUIER USO DE LA OBRA O PRESTACIÓN DIFERENTE A LO AUTORIZADO BAJO ESTA LICENCIA O LO DISPUESTO EN LA LEY DE PROPIEDAD INTELECTUAL.

MEDIANTE EL EJERCICIO DE CUALQUIER DERECHO SOBRE LA OBRA O LA PRESTACIÓN, USTED ACEPTA Y CONSIENTE LAS LIMITACIONES Y OBLIGACIONES DE ESTA LICENCIA, SIN PERJUICIO DE LA NECESIDAD DE CONSENTIMIENTO EXPRESO EN CASO DE VIOLACIÓN PREVIA DE LOS TÉRMINOS DE LA MISMA. EL LICENCIADOR LE CONCEDE LOS DERECHOS CONTENIDOS EN ESTA LICENCIA, SIEMPRE QUE USTED ACEPTE LOS PRESENTES TÉRMINOS Y CONDICIONES.

1. Definiciones

a. La **obra** es la creación literaria, artística o científica ofrecida bajo los términos de esta licencia.

b. En esta licencia se considera una **prestación** cualquier interpretación, ejecución, fonograma, grabación audiovisual, emisión o transmisión, mera fotografía u otros objetos protegidos por la legislación de propiedad intelectual vigente aplicable.

c. La aplicación de esta licencia a una **colección** (definida más adelante) afectará únicamente a su estructura en cuanto forma de expresión de la selección o disposición de sus contenidos, no siendo extensiva a éstos. En este caso la colección tendrá la consideración de obra a efectos de esta licencia.

d. El **titular originario** es:

   a. En el caso de una obra literaria, artística o científica, la persona natural o grupo de personas que creó la obra.

   b. En el caso de una obra colectiva, la persona que la edite y divulgue bajo su nombre, salvo pacto contrario.

   c. En el caso de una interpretación o ejecución, el actor, cantante, músico, o cualquier otra persona que represente, cante, lea, recite, interprete o ejecute en cualquier forma una obra.

   d. En el caso de un fonograma, el productor fonográfico, es decir, la persona natural o jurídica bajo cuya iniciativa y responsabilidad se realiza por primera vez una fijación exclusivamente sonora de la ejecución de una obra o de otros sonidos.

e. En el caso de una grabación audiovisual, el productor de la grabación, es decir, la persona natural o jurídica que tenga la iniciativa y asuma la responsabilidad de las fijaciones de un plano o secuencia de imágenes, con o sin sonido.

f. En el caso de una emisión o una transmisión, la entidad de radiodifusión.

g. En el caso de una mera fotografía, aquella persona que la haya realizado.

h. En el caso de otros objetos protegidos por la legislación de propiedad intelectual vigente, la persona que ésta señale.

e. Se considerarán **obras derivadas** aquellas obras creadas a partir de la licenciada, como por ejemplo: las traducciones y adaptaciones; las revisiones, actualizaciones y anotaciones; los compendios, resúmenes y extractos; los arreglos musicales y, en general, cualesquiera transformaciones de una obra literaria, artística o científica. Para evitar la duda, si la obra consiste en una composición musical o grabación de sonidos, la sincronización temporal de la obra con una imagen en movimiento (synching) será considerada como una obra derivada a efectos de esta licencia.

f. Tendrán la consideración de **colecciones** la recopilación de obras ajenas, de datos o de otros elementos independientes como las antologías y las bases de datos que por la selección o disposición de sus contenidos constituyan creaciones intelectuales. La mera incorporación de una obra en una colección no dará lugar a una derivada a efectos de esta licencia.

g. El **licenciador** es la persona o la entidad que ofrece la obra o prestación bajo los términos de esta licencia y le concede los derechos de explotación de la misma conforme a lo dispuesto en ella.

h. **Usted** es la persona o la entidad que ejercita los derechos concedidos mediante esta licencia y que no ha violado previamente los términos de la misma con respecto a la obra o la prestación, o que ha recibido el permiso expreso del licenciador de ejercitar los derechos concedidos mediante esta licencia a pesar de una violación anterior.

i. La **transformación** de una obra comprende su traducción, adaptación y cualquier otra modificación en su forma de la que se derive una obra diferente. La creación resultante de la transformación de una obra tendrá la consideración de obra derivada.

j. Se entiende por **reproducción** la fijación directa o indirecta, provisional o permanente, por cualquier medio y en cualquier forma, de toda la obra o la prestación o de parte de ella, que permita su comunicación o la obtención de copias.

k. Se entiende por **distribución** la puesta a disposición del público del original o de las copias de la obra o la prestación, en un soporte tangible, mediante su venta, alquiler, préstamo o de cualquier otra forma.

l. Se entiende por **comunicación pública** todo acto por el cual una pluralidad de personas, que no pertenezcan al ámbito doméstico de quien la lleva a cabo, pueda tener acceso a la obra o la prestación sin previa distribución de ejemplares a cada una de ellas. Se considera comunicación pública la puesta a disposición del público de obras o prestaciones por procedimientos alámbricos o inalámbricos, de tal forma que cualquier persona pueda acceder a ellas desde el lugar y en el momento que elija.

m. La **explotación** de la obra o la prestación comprende la reproducción, la distribución, la comunicación pública y, en su caso, la transformación.

2. Límites de los derechos. Nada en esta licencia pretende reducir o restringir cualesquiera límites legales de los derechos exclusivos del titular de los derechos de propiedad intelectual de acuerdo con la Ley de propiedad intelectual o cualesquiera otras leyes aplicables, ya sean derivados de usos legítimos, tales como la copia privada o la cita, u otras limitaciones como la resultante de la primera venta de ejemplares (agotamiento).

3. Concesión de licencia. Conforme a los términos y a las condiciones de esta licencia, el licenciador concede, por el plazo de protección de los derechos de propiedad intelectual y a título gratuito, una licencia de ámbito mundial no exclusiva que incluye los derechos siguientes:

a. Derecho de reproducción, distribución y comunicación pública de la obra o la prestación.
b. Derecho a incorporar la obra o la prestación en una o más colecciones.
c. Derecho de reproducción, distribución y comunicación pública de la obra o la prestación lícitamente incorporada en una colección.
d. Derecho de transformación de la obra para crear una obra derivada siempre y cuando se incluya en ésta una indicación de la transformación o modificación efectuada.
e. Derecho de reproducción, distribución y comunicación pública de obras derivadas creadas a partir de la obra licenciada.
f. Derecho a extraer y reutilizar la obra o la prestación de una base de datos.
g. Para evitar cualquier duda, el titular originario:
    i. Conserva el derecho a percibir las remuneraciones o compensaciones previstas por actos de explotación de la obra o prestación, calificadas por la ley como irrenunciables e inalienables y sujetas a gestión colectiva obligatoria.
    ii. Renuncia al derecho exclusivo a percibir, tanto individualmente como mediante una entidad de gestión colectiva de derechos, cualquier remuneración derivada de actos de explotación de la obra o prestación que usted realice.

Estos derechos se pueden ejercitar en todos los medios y formatos, tangibles o intangibles, conocidos en el momento de la concesión de esta licencia. Los derechos mencionados incluyen el derecho a efectuar las modificaciones que sean precisas técnicamente para el ejercicio de los derechos en otros medios y formatos. Todos los derechos no concedidos expresamente por el licenciador quedan reservados, incluyendo, a título enunciativo pero no limitativo, los derechos morales irrenunciables reconocidos por la ley aplicable. En la medida en que el licenciador ostente derechos exclusivos previstos por la ley nacional vigente que implementa la directiva europea en materia de derecho sui generis sobre bases de datos, renuncia expresamente a dichos derechos exclusivos.

4. Restricciones. La concesión de derechos que supone esta licencia se encuentra sujeta y limitada a las restricciones siguientes:

a. Usted puede reproducir, distribuir o comunicar públicamente la obra o prestación solamente bajo los términos de esta licencia y debe incluir una copia de la misma, o su Identificador Uniforme de Recurso (URI). Usted no puede ofrecer o imponer ninguna condición sobre la obra o prestación que altere o restrinja los términos de esta licencia o el ejercicio de sus derechos por parte de los concesionarios de la misma. Usted no puede sublicenciar la obra o prestación. Usted debe mantener intactos todos los avisos que se refieran a esta licencia y a la ausencia de garantías. Usted no puede reproducir, distribuir o comunicar públicamente la obra o prestación con medidas tecnológicas que controlen el acceso o el uso de una

manera contraria a los términos de esta licencia. Esta sección 4.a también afecta a la obra o prestación incorporada en una colección, pero ello no implica que ésta en su conjunto quede automáticamente o deba quedar sujeta a los términos de la misma. En el caso que le sea requerido, previa comunicación del licenciador, si usted incorpora la obra en una colección y/o crea una obra derivada, deberá quitar cualquier crédito requerido en el apartado 4.b, en la medida de lo posible.

b. Si usted reproduce, distribuye o comunica públicamente la obra o la prestación, una colección que la incorpore o cualquier obra derivada, debe mantener intactos todos los avisos sobre la propiedad intelectual e indicar, de manera razonable conforme al medio o a los medios que usted esté utilizando:

   i. El nombre del autor original, o el seudónimo si es el caso, así como el del titular originario, si le es facilitado.

   ii. El nombre de aquellas partes (por ejemplo: institución, publicación, revista) que el titular originario y/o el licenciador designen para ser reconocidos en el aviso legal, las condiciones de uso, o de cualquier otra manera razonable.

   iii. El título de la obra o la prestación si le es facilitado.

   iv. El URI, si existe, que el licenciador especifique para ser vinculado a la obra o la prestación, a menos que tal URI no se refiera al aviso legal o a la información sobre la licencia de la obra o la prestación.

   v. En el caso de una obra derivada, un aviso que identifique la transformación de la obra en la obra derivada (p. ej., "traducción castellana de la obra de Autor Original," o "guión basado en obra original de Autor Original").

   Este reconocimiento debe hacerse de manera razonable. En el caso de una obra derivada o incorporación en una colección estos créditos deberán aparecer como mínimo en el mismo lugar donde se hallen los correspondientes a otros autores o titulares y de forma comparable a los mismos. Para evitar la duda, los créditos requeridos en esta sección sólo serán utilizados a efectos de atribución de la obra o la prestación en la manera especificada anteriormente. Sin un permiso previo por escrito, usted no puede afirmar ni dar a entender implícitamente ni explícitamente ninguna conexión, patrocinio o aprobación por parte del titular originario, el licenciador y/o las partes reconocidas hacia usted o hacia el uso que hace de la obra o la prestación.

c. Para evitar cualquier duda, debe hacerse notar que las restricciones anteriores (párrafos 4.a y 4.b) no son de aplicación a aquellas partes de la obra o la prestación objeto de esta licencia que únicamente puedan ser protegidas mediante el derecho sui generis sobre bases de datos recogido por la ley nacional vigente implementando la directiva europea de bases de datos

5. Exoneración de responsabilidad

A MENOS QUE SE ACUERDE MUTUAMENTE ENTRE LAS PARTES, EL LICENCIADOR OFRECE LA OBRA O LA PRESTACIÓN TAL CUAL (ON AN "AS-IS" BASIS) Y NO CONFIERE NINGUNA GARANTÍA DE CUALQUIER TIPO RESPECTO DE LA OBRA O LA PRESTACIÓN O DE LA PRESENCIA O AUSENCIA DE ERRORES QUE PUEDAN O NO SER DESCUBIERTOS. ALGUNAS JURISDICCIONES NO PERMITEN LA EXCLUSIÓN DE TALES GARANTÍAS, POR LO QUE TAL EXCLUSIÓN PUEDE NO SER DE APLICACIÓN A USTED.

6. Limitación de responsabilidad. SALVO QUE LO DISPONGA EXPRESA E IMPERATIVAMENTE LA LEY APLICABLE, EN NINGÚN CASO EL LICENCIADOR SERÁ RESPONSABLE ANTE USTED POR CUALESQUIERA DAÑOS RESULTANTES, GENERALES O ESPECIALES (INCLUIDO EL DAÑO EMERGENTE Y EL LUCRO CESANTE), FORTUITOS O CAUSALES, DIRECTOS O INDIRECTOS, PRODUCIDOS EN CONEXIÓN CON ESTA LICENCIA O EL USO DE LA OBRA O LA PRESTACIÓN, INCLUSO SI EL LICENCIADOR HUBIERA SIDO INFORMADO DE LA POSIBILIDAD DE TALES DAÑOS.

7. Finalización de la licencia

a. Esta licencia y la concesión de los derechos que contiene terminarán automáticamente en caso de cualquier incumplimiento de los términos de la misma. Las personas o entidades que hayan recibido de usted obras derivadas o colecciones bajo esta licencia, sin embargo, no verán sus licencias finalizadas, siempre que tales personas o entidades se mantengan en el cumplimiento íntegro de esta licencia. Las secciones 1, 2, 5, 6, 7 y 8 permanecerán vigentes pese a cualquier finalización de esta licencia.

b. Conforme a las condiciones y términos anteriores, la concesión de derechos de esta licencia es vigente por todo el plazo de protección de los derechos de propiedad intelectual según la ley aplicable. A pesar de lo anterior, el licenciador se reserva el derecho a divulgar o publicar la obra o la prestación en condiciones distintas a las presentes, o de retirar la obra o la prestación en cualquier momento. No obstante, ello no supondrá dar por concluida esta licencia (o cualquier otra licencia que haya sido concedida, o sea necesario ser concedida, bajo los términos de esta licencia), que continuará vigente y con efectos completos a no ser que haya finalizado conforme a lo establecido anteriormente, sin perjuicio del derecho moral de arrepentimiento en los términos reconocidos por la ley de propiedad intelectual aplicable.

8. Miscelánea

a. Cada vez que usted realice cualquier tipo de explotación de la obra o la prestación, o de una colección que la incorpore, el licenciador ofrece a los terceros y sucesivos licenciatarios la concesión de derechos sobre la obra o la prestación en las mismas condiciones y términos que la licencia concedida a usted.

b. Cada vez que usted realice cualquier tipo de explotación de una obra derivada, el licenciador ofrece a los terceros y sucesivos licenciatarios la concesión de derechos sobre la obra objeto de esta licencia en las mismas condiciones y términos que la licencia concedida a usted.

c. Si alguna disposición de esta licencia resulta inválida o inaplicable según la Ley vigente, ello no afectará la validez o aplicabilidad del resto de los términos de esta licencia y, sin ninguna acción adicional por cualquiera las partes de este acuerdo, tal disposición se entenderá reformada en lo estrictamente necesario para hacer que tal disposición sea válida y ejecutiva.

d. No se entenderá que existe renuncia respecto de algún término o disposición de esta licencia, ni que se consiente violación alguna de la misma, a menos que tal renuncia o consentimiento figure por escrito y lleve la firma de la parte que renuncie o consienta.

e. Esta licencia constituye el acuerdo pleno entre las partes con respecto a la obra o la prestación objeto de la licencia. No caben interpretaciones, acuerdos o condiciones con respecto a la obra o la prestación que no se encuentren expresamente especificados en la presente licencia. El licenciador no estará obligado por ninguna disposición complementaria

que pueda aparecer en cualquier comunicación que le haga llegar usted. Esta licencia no se puede modificar sin el mutuo acuerdo por escrito entre el licenciador y usted.