Manual de usuario

DNIeDroid v2.3

13 de julio de 2022

Versión	Fecha	Autores	Descripción
1.0	19.02.2018	CNP-FNMT	Creado
1.1	20.02.2018	CNP-FNMT	Versionado y corrección de elementos gráficos.
1.2	14.12.2018	CNP-FNMT	Actualización de funcionalidad.
1.3	01.02.2019	CNP-FNMT	Actualización de funcionalidad.
1.4	28.02.2019	CNP-FNMT	Migración de cliente HTTP y reorganización de paquetes.
1.5	02.09.2019	CNP-FNMT	Actualización de funcionalidad.
1.6	05.03.2020	CNP-FNMT	Actualización de funcionalidad.
1.7	04.09.2020	CNP-FNMT	Cambios en interfaz relevantes.
1.8	26.04.2021	CNP-FNMT	Actualización de funcionalidad.
1.9	22.10.2021	CNP-FNMT	Actualización de funcionalidad.
1.10	07.07.2022	CNP-FNMT	Actualización de funcionalidad.
1.11	13.07.2022	CNP-FNMT	Actualización de funcionalidad.

Índice

1. Introducción **5**

2. Objetivo **5**

3. Arquitectura **6**

3.1. Lógica 6

3.2. Física 7

4. Integración en aplicaciones Android **7**

4.1. Introducción 7

4.2. Requisitos 8

4.3. Funcionalidad 9

4.4. Paquetes disponibles 9

4.5. Comprobación de edad 18

4.6. Obtener estado del certificado 19

4.7. Información adicional 19

Índice de figuras

1. Reverso DNIe. 5
2. Arquitectura lógica DNIeDroid 7
3. Integración en Android 8
4. Acceso a servicios 9

1. Introducción

En la actualidad se ha generalizado por completo el uso de los dispositivos móviles para el acceso a internet. Esto permite a cualquier usuario estar siempre conectado y tener disponibles multitud de servicios telemáticos. En función del nivel de seguridad que requieran dichos servicios, éstos pueden necesitar de la autenticación del usuario, lo que hace que el Documento Nacional de Identidad (DNIe 3.0/4.0) pueda convertirse en una pieza clave.

La incorporación de la tecnología NFC a los dispositivos móviles de última generación elimina las barreras del lector, drivers, etc. facilitando la conexión online y la autenticación del ciudadano. Por su parte, el DNIe 3.0/4.0 tiene un chip dual-interface que permite su utilización tanto con contactos como en modo contactless.

(a) DNIe 3.0 (b) DNIe 4.0

Figura 1: Reverso DNIe.

2. Objetivo

El objetivo de este documento es detallar la arquitectura y el funcionamiento de una aplicación Android con soporte al DNIe 3.0/4.0 mediante NFC e indicar los requisitos funcionales para el desarrollo de aplicaciones similares.

El documento incluye una descripción de la arquitectura necesaria y el diseño interno básico de la aplicación Android. Estas aplicaciones estarán siempre apoyadas en el conector DNIeDroid, el middleware diseñado para la comunicación con el DNI electrónico.

La autenticación del usuario para el acceso a servicios de la Administración Pública se realiza mediante el uso del DNI electrónico 3.0/4.0. Este DNIe funciona mediante NFC, (Near Field Communication, ver [3]) lo que permite a cualquier dispositivo que disponga de esta tecnología, actuar como lector de tarjetas.

Las comunicaciones entre el dispositivo y el DNIe se realizarán siempre cifradas según la norma [4].

En este documento nos centraremos en la versión inalámbrica del DNIe, en la que la interacción desde el punto de vista físico se realiza por medio de NFC.

Para información relativa a los certificados X509 incluidos en el DNIe 3.0/4.0, véase [1]. En cuanto a la conexión por radiofrecuencia entre dispositivos, ver [2].

3. Arquitectura

3.1. Lógica

El DNIeDroid es un middleware encargado de gestionar la conexión entre dispositivos móviles y el DNIe. Ofrece un API básico de operaciones que permite a los desarrolladores gestionar de manera transparente las conexiones NFC con el DNIe.

De esta manera se pretende facilitar la implementación de aplicaciones, ya que no hará falta conocer en detalle el funcionamiento de la tarjeta sino que bastará con incluir el DNIeDroid en la aplicación para poder acceder a las funcionalidades del DNIe.

En el diagrama 2 se describe la arquitectura lógica actual del componente, desarrollado en Java para Android.

DNIeDroid ofrece una capa de alto nivel para la gestión de la conexión con dispositivos lectores de tarjetas. Esta capa, se apoya en una representación de "alto nivel" de los dispositivos, gestionando el envío y recepción de comandos a través del canal de comunicación NFC. Esta gestión se lleva a cabo gracias a la capa de abstracción de bajo nivel proporcionada por Android, y que en su caso se debería proporcionar también en sistemas como iOS.

El objetivo del componente DNIeDroid es interactuar con el DNIe en dispositivos basados en Java (como smartphones y tablets Android), mediante interfaz NFC. De esta manera se facilita el acceso desde esas plataformas a los servicios que hacen uso del DNIe para autenticación y firma electrónica.

En el diagrama 3 de describe el modo de integración del DNIeDroid en DD4J.

Tal como se observa en el diagrama, el componente DNIeDroid se integra como una conexión subyacente más dentro de la arquitectura del Controlador

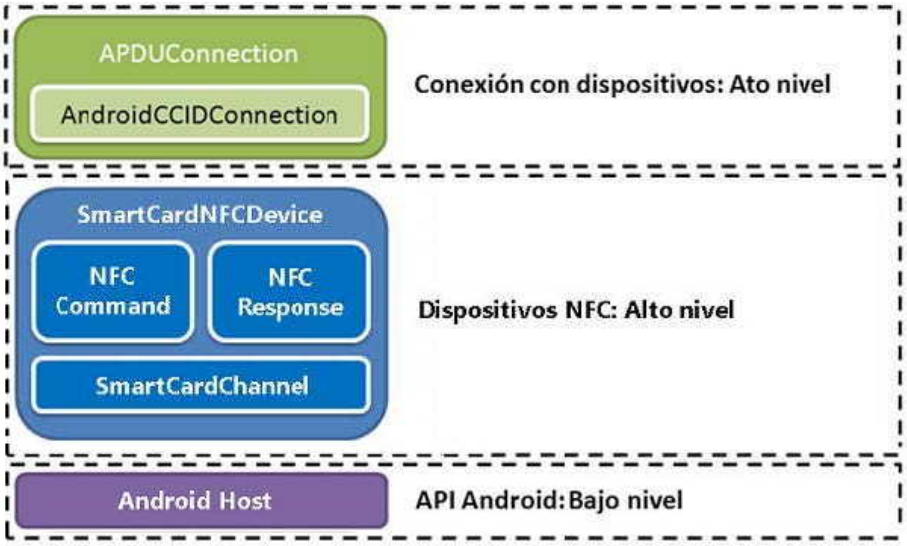

Figura 2: Arquitectura lógica DNIeDroid

Java para el DNIe.

Se abstrae así a las capas superiores de la interacción con las conexiones de tarjetas conectados a los dispositivos Android vía NFC y la gestión de intercambio de comandos de bajo nivel. Es esta última capa la que da acceso al API de NFC en Android y que nos permite el envío de comandos y respuestas entre DNIe y dispositivo.

3.2. Física

A pesar de que el componente es puramente lógico, está diseñado para la comunicación de dispositivos Android mediante la conexión por proximidad con un DNIe v3.0/4.0 a través de la tecnología NFC. Ésta permite la comunicación entre el dispositivo Android y el DNIe sin necesidad de lector de tarjetas.

4. Integración en aplicaciones Android

4.1. Introducción

Tras la integración del controlador DNIeDroid en la aplicación Java, es posible acceder a toda la funcionalidad proporcionada por éste desde aplicaciones desarrolladas para dispositivos Android.

Este apartado pretende servir de guía a la hora de la integración del con-

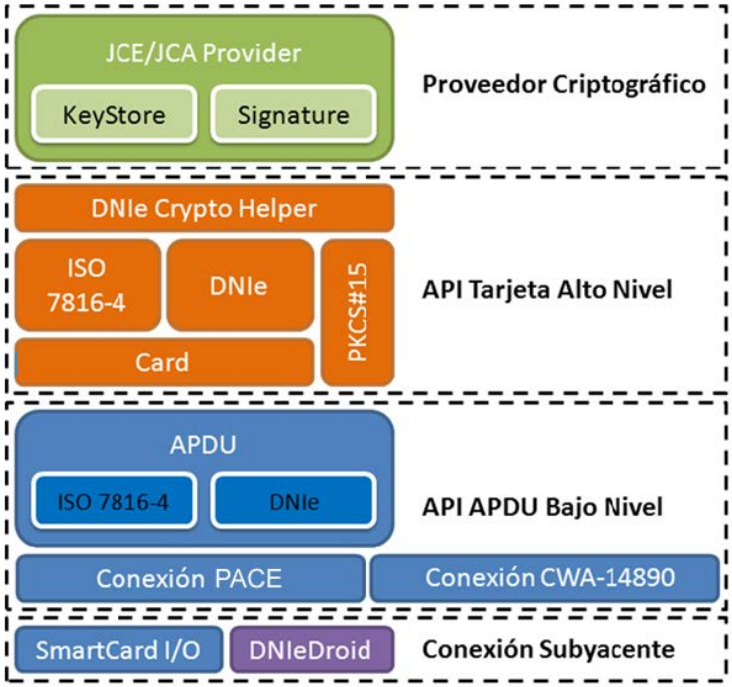

Figura 3: Integración en Android

trolador, adaptado para Android en aplicaciones de terceros.

4.2. Requisitos

Para poder realizar un uso adecuado de la librería es necesario tener experiencia en desarrollo de aplicaciones para dispositivos Android, y se recomienda poseer conocimientos básicos sobre Infraestructura de Clave Pública (PKI) y de Java Cryptography Architecture (JCA), así como de protocolos de comunicación a través de las redes.

Esta librería se apoya en APIs de terceros, por lo que será necesario incluir dependencias en el proyecto. Así, en el archivo *build.gradle* será necesario añadir las siguientes líneas:

```
implementation 'org.bouncycastle:bcprov-jdk15on:1.65'
implementation 'org.jsoup:jsoup:1.13.1

(De forma opcional, cuando se utilice la clase
es.gob.fnmt.dniedroid.net.http.okhttp.OkHTTPClient)
implementation 'com.squareup.okhttp3:okhttp:4.9.1'
implementation 'com.squareup.okhttp3:okhttp-urlconnection:4.9.1'
```

4.3. Funcionalidad

Como funcionalidad básica, la librería dispone de la implementación de la capa criptográfica accesible a través de un Proveedor de Servicios Criptográficos o 'Cryptographic Service Provider', véase [6], proporcionando una interfaz estándar.

También se incluyen clases para las comunicaciones y recuperación de información a través de la red, que abstrae al usuario de la complejidad en la implementación de la autenticación con certificado de cliente a través del protocolo HTTPS.

Figura 4: Acceso a servicios

4.4. Paquetes disponibles

A continuación se indica una breve descripción de los paquetes que engloban toda la funcionalidad necesaria para realizar la integración de la aplicación. La mayor parte del código fuente que se muestra en los ejemplos se ha extraído del proyecto que incluye el SDK.

es.gob.jmulticard.jse.provider contiene la clases que implementan la capa criptográfica, disponible a través de la interfaz JCA.

es.gob.fnmt.dniedroid.help contiene la clase que encapsula operaciones básicas.

es.gob.fnmt.dniedroid.net contiene las clases para la conexión y recuperación de información a través de la red.

es.gob.fnmt.dniedroid.policy contiene las clases para el uso de claves de la tarjeta criptográfica a través de políticas.

es.gob.jmulticard.ui.passwordcallback contiene la interfaz necesaria para la implementación del diálogo de petición de PIN.

de.tsenger.androsmex.mrtd contiene las clases que encapsulan la información de tipo MRTD, véase [5].

de.tsenger.androsmex.data contiene las clases que encapsulan la información del CAN (Card Access Number).

es.gob.fnmt.dniedroid.gui contiene las clases relacionadas con la interfaz de presentación e interacción para el usuario.

Nota: los paquetes y las clases de **es.gob.fnmt.dniedroid.gui.fragment** y **es.gob.fnmt.dniedroid.nfc** se indican como obsoletas, por lo que se desaconseja su uso ya que serán eliminadas en versiones posteriores. Se aconseja migrar el código siguiendo los ejemplos en el proyecto del SDK.

Paquete *es.gob.jmulticard.jse.provider*

DnieProvider: clase que implementa la interfaz *java.security.Provider* como punto de acceso a toda la capa criptográfica.

La implementación de la JCA permite instanciar un objeto de tipo *java.security.KeyStore*, que sirve para acceder a las claves y por tanto a los motores criptográficos. El almacén de claves se identifica con 'DNIeKS', y para realizar la carga es necesario pasar como argumento una instancia de la clase *es.gob.jmulticard.jse.provider.DnieLoadParameter* que implementa la interfaz *java.security.KeyStore.LoadStoreParameter*. La información imprescindible que se indica en este objeto es el tag NFC obtenido en la detección del DNIe a través de la interfaz android.nfc.NfcAdapter.ReaderCallback, y el código de 6 dígitos que corresponde al CAN del DNIe, véase [7]. Esta clase es de gran utilidad ya que permite obtener información relevante, como la de tipo MRTD, así como inyectar una Callback que informará del estado en los procesos de firma (ver JavaDoc)

A continuación se indica un código de ejemplo básico, disponible en *SampleActivity_provider*.

```
1  ...
2  String can;
3  Tag tag;
4  //... (una vez recuperamos Tag y CAN)
5  final DnieProvider p = new DnieProvider();
6  Security.insertProviderAt(p, 1);
```

```
7   KeyStore keyStore = KeyStore.getInstance("DNIeKS");
8   DnieLoadParameter loadParameter = DnieLoadParameter.getBuilder(can,
    ↪  tag).build();
9   keyStore.load(loadParameter);
10  ...
```

Paquete *es.gob.fnmt.dniedroid.help*

Loader: clase que encapsula operaciones básicas de la librería, como la carga del almacén de claves y gestión del CAN. Ejemplo en *SampleActivity_connect_gui_clave*.

```
1   ...
2   //Clase 'Loader' que nos devuelve directamente el keystore
3   Loader.InitInfo initInfo= Loader.init(new String[]{_can}, tag);
4   X509Certificate certificate = (X509Certificate) initInfo.getKeyStore().g
    ↪  etCertificate(initInfo.getKeyStore().aliases().nextElement());
5   ...
6   //Actualizar la BBDD de CAN de la App
7   CANSpecDO canSpecDO = new CANSpecDO(_can, Tool.getCN(certificate),
    ↪  Tool.getNIF(certificate));
8   Loader.saveCan2DB(canSpecDO, this);
9   ...
```

Paquete *es.gob.fnmt.dniedroid.net*

Este paquete a su vez se organiza en otros dependiendo de la funcionalidad que aporta: seguridad (*.ssl*), conectividad (*.http*) o herramientas (*.tool*).

Paquete *es.gob.fnmt.dniedroid.net.ssl*

DNIeSSLSocketFactory: clase que permite instanciar la factoría de sockets *javax.net.ssl.SSLSocketFactory* necesaria para establecer conexiones por protocolo HTTPS, implementando la autenticación de cliente con certificado digital mediante el DNIe. Utilizará el *java.security.KeyStore* del Provider para resolver las firmas digitales necesarias en el protocolo *SSL/TLS handshake*. Ejemplo en *SampleActivity_connect_gui*.

```
...
SSLSocketFactory dnieSSLSocketFactory =
    DNIeSSLSocketFactory.getInstance(NFCCommReaderFragment.getKeyStore(),
    //almacén del DNIe con los certificados de usuario.
    ToolBox.getKeyStoreFromResource(R.raw.truststore,
    getApplicationContext()), //almacén con los certificados de CA
    reconocidos para la conexión con los servidores.
    KeyManagerPolicy.getBuilder().addAlias(DnieProvider.AUTH_CERT_ALIAS).b
    uild(),  //política de selección de certificados de usuario. En este
    caso indicamos el certificado de autenticación.
    this);
...
```

A través de este constructor se pueden indicar, además de los almacenes de certificados de cliente y de certificados de CA de confianza visto en el código fuente anterior, las políticas de selección de certificados y el protocolo seguro de comunicaciones (SSLv3, TLSv1.1, etc.).

Paquete *es.gob.fnmt.dniedroid.net.http*

En este paquete se pueden utilizar dos clases que encapsulan la funcionalidad básica que nos permiten gestionar la comunicaciones a través de la red: HTTPClient y OkHTTPClient. Una tercera clase encapsula la complejidad de las conexiones necesarias para establecer una autenticación a través de la plataforma Cl@ve.

HTTPClient: clase que encapsula la funcionalidad de un cliente sencillo para peticiones HTTP de tipo 'GET' y 'POST', a través de la API *javax.net.ssl.HttpsURLConnection*. Permite opcionalmente el paso de parámetros, establecer la factoría de sockets SSL/TLS para el establecimiento del canal seguro, así como indicar propiedades específicas requeridas para poder realizar la conexión, como son las cabeceras HTTP. Ejemplo en *SampleActivity_connect_gui*.

```
HTMLParser parser = new HTMLParser(this);  //clase de ayuda para obtener
    información de las páginas HTML.
Map<String,String> postParams = new HashMap<>(); //Lista de parámetros a
    pasar a la petición 'POST'.
...
updateProgressDlg("Descargando datos...","Llamada "+ ++llamada);
```

```
5   String _htmlBody = new HTTPClient(URL_SS_LOGIN).getStringResponse();
6   parser.setContent(_htmlBody);
7
8   Element form = parser.getDocument().getElementById("P017_login2");
9   String action = URL_SEDE_TUSS_DOMAIN;
10
11  postParams.put("loginFormSubmit",
    ↪  form.getElementById("loginFormSubmit").val());
12  action += form.attr("action");
13  updateProgressDlg(null,"Llamada "+ ++llamada);
14
15  _htmlBody = new HTTPClient(action, postParams).getStringResponse();
16  parser.setContent(_htmlBody);
17  updateProgressDlg(null,"Llamada "+ ++llamada);
18  form = parser.getDocument().getElementById("P017_login");
19  action = form.attr("action");
20  Elements elements = form.getElementsByTag("input");
21  postParams.clear();
22  for( Element element : elements){
23    postParams.put(element.attr("name"), element.attr("value"));
24  }
25
26  _htmlBody = new HTTPClient(action, postParams).setSSLSocketFactory(dnieS
    ↪  SLSocketFactory).getStringResponse(); //Se establece la factoría de
    ↪  sockets SSL para la autenticación.
27  parser.setContent(_htmlBody);
28  ...
```

Para establecer la conexión y recuperar los datos, se llamará a uno de los métodos disponibles, dependiendo del formato requerido.

```
1   int httpCode = new HTTPClient(URL_SS_LOGIN).getResponse(); //Simplemente
    ↪  obtenemos el código HTTP de respuesta (p.ej. 200); obviando el
    ↪  contenido de la respuesta.
2   InputStream is = new HTTPClient(URL_SS_LOGIN).getStreamResponse();
    ↪  //Devuelve el flujo de datos.
3   String html = new HTTPClient(URL_SS_LOGIN).getStringResponse();
    ↪  //Devuelve una cadena.
4   byte[] data = new HTTPClient(URL_SS_LOGIN).getByteArrayResponse();
    ↪  //Devuelve una array de bytes.
```

Por defecto las peticiones se realizan a través del método 'GET' y, en el caso de recuperar caracteres, se resuelve la codificación a través de la información de las cabeceras en la respuesta. Si no se incluye esta información, la codificación por defecto es *UTF-8*. Sin embargo, las funciones sobrecargadas permiten establecer estos parámetros según sea necesario.

Por último, una vez realizada la conexión y obtenidos los datos, se puede recuperar el código HTTP devuelto para verificar que la conexión ha sido exitosa, o realizar alguna acción en función de su valor (4XX, 5XX, etc).

```
1  HTTPClient httpClient = new HTTPClient(URL_SS_LOGIN);
2  InputStream is = httpClient.getStreamResponse();
3  int httpCode = httpClient.getLastResponse(); //Recuperamos el código
   ↪ devuelto por la conexión.
```

CookiePolicyHandler: clase que permite gestionar la cookies de los diferentes sitios que se van visitando. Para almacenar las cookies y así dar persistencia a la autenticación realizada en los diferentes dominios, es necesario registrar un gestor de cookies *java.net.CookieManager* a través de *java.net.CookieHandler*. Con un objeto CookiePolicyHandler asociado al CookieManager, tendremos acceso para poder eliminar las cookies de un sitio en concreto cuando sea necesario. Ejemplo en *SampleActivity_main* y *SampleActivity_connect_gui*.

```
1  ...
2  public static CookieManager _cookieManager = new CookieManager();
3  public static CookiePolicyHandler _cookiePolicy = new
   ↪ CookiePolicyHandler();
4
5  static{
6    _cookieManager.setCookiePolicy(_cookiePolicy);
7    CookieHandler.setDefault(_cookieManager);
8  }
9  ...
10 //Establecemos el dominio para tener un control de las cookies que se
   ↪ van guardando.
11 SampleActivity_main._cookiePolicy.setSiteHandling("TUSS");
12 ...
13 //Eliminamos las cookies que pertenecen a este dominio.
14 SampleActivity_main._cookiePolicy.deleteCookies(SampleActivity_main._coo
   ↪ kieManager.getCookieStore());
```

```
15    ...
```

MultipartData: clase que facilita el envío de archivos en las peticiones HTTP, a través de una interfaz sencilla para la construcción del contenido. Se muestra un ejemplo en el siguiente código:

```
1     ...
2     //Request
3     MultipartData multipartData = new MultipartData();
4     multipartData.addPart("fileData", file, context);  //Archivo a enviar.
5     multipartData.addPart("codigoSeguridad", code);   //Parámetro requerido.
6     HTTPClient httpClient = new HTTPClient(VALIDE_FIRMA, multipartData);
      ↪  //Cliente HTTP al que le pasamos el objeto.
7     String response = httpClient.getStringResponse();
8     if (httpClient.getLastResponse() != HttpURLConnection.HTTP_OK)
9       throw new ConnectException("Cannot send the file to Valide");
10     ...
```

OkHTTPClient: clase que encapsula la funcionalidad de un cliente sencillo para peticiones HTTP, a través de la API *okhttp3.OkHttpClient*. Al contrario que la clase *HTTPClient*, se deberá utilizar la misma instancia *OkHTTPClient* para las comunicaciones con un sitio web, una vez se haya establecido la autenticación (utilizando la factoría de sockets del DNIe). El pool de conexiones se gestiona internamente y se reutiliza para mantener una sesión autenticada. Si se quiere realizar otra autenticación en otro sitio web, será necesario instanciar un nuevo objeto *OkHTTPClient*.

```
1     ...
2     OkHTTPClient okHttpClient = OkHTTPClient.getBuilder(_sslSocketFactory)
3     .setTrustCertStore(ToolBox.getKeyStoreFromResource(R.raw.geotrustsslcag
      ↪  3,
      ↪  getApplicationContext()))
4     .build();
5
6     InputStream stream = okHttpClient.setRequest(new
      ↪  OkHTTPClient.RequestBuilder(_url)).getStreamResponse(); //Con una
      ↪  petición 'GET'.
7
8     String html = okHttpClient.setRequest(new
      ↪  OkHTTPClient.RequestBuilder(_url)
```

```
9     .addParam("param1", "John")
10    .addParam("param2","Doe"))
11    .getStringResponse(); //Con una petición 'POST' con parámetros.
12    ...
```

ClaveAuthentication: clase que facilita la autenticación con sitios web integrados con la plataforma Cl@ve. Para utilizarla correctamente es necesario indicarle el código HTML de la página de inicio de la autenticación del sitio web y el atributo de la etiqueta *form* y su valor en donde reside la petición a Cl@ve y, por tanto, se encuentra la petición SAML. Se puede ver un ejemplo de uso en *SampleActivity_connect_gui_clave*

Paquete *es.gob.fnmt.dniedroid.net.tool*

HTMLParser: clase que encapsula la funcionalidad de un simple 'parser' de documentos con sintaxis HTML. Internamente hace uso del paquete *org.jsoup*. Se puede ver su uso en los varios ejemplos de código que se muestran en el documento.

ToolBox: clase con varios métodos que pueden ser de ayuda (tratamiento de flujo de datos, almacén de claves, etc.).

Paquete *es.gob.fnmt.dniedroid.gui*

PasswordUI: clase que establece la apariencia de los elementos visuales del cuadro de diálogo de petición de PIN. Se puede incorporar un cuadro de diálogo de petición de PIN propio que sustituya al de por defecto (ver más adelante en *es.gob.jmulticard.ui.passwordcallback*).

```
1   PasswordUI.setTextColor(Color.YELLOW);
2   PasswordUI.setBackgroundColor(Color.BLUE);
3   PasswordUI.setTextTypeFace(Typeface.createFromAsset(...));
```

CertificateUI: clase que establece la apariencia de los elementos visuales de la ventana de diálogo de selección de certificado. Por defecto, esta ventana no aparece cuando las políticas de selección de certificados establecidas, es decir *KeyManagerPolicy*, sólo filtran un certificado.

Paquete *es.gob.fnmt.dniedroid.policy*

KeyManagerPolicy: clase que permite indicar las políticas de selección de certificado cuando se va a realizar una operación de firma. Se puede añadir políticas por alias conocido, uso de clave o uso extendido de clave (definidas en *KeyManagerPolicy.ExtendedKeyUsage* y *KeyManagerPolicy.KeyUsage*). Para su construcción la propia clase provee un *Builder*.

```
1  KeyManagerPolicy keyManagerPolicy = new
   ↪ KeyManagerPolicy.getBuilder().addAlias(DnieProvider.AUTH_CERT_ALIAS)
   ↪ .addKeyUsage(KeyManagerPolicy.KeyUsage.digitalSignature).build();
```

Paquete *es.gob.jmulticard.ui.passwordcallback*

DialogUIHandler: interfaz a implementar para proporcionar un cuadro de diálogo de petición de PIN a la clase *PasswordUI* en el método *setPasswordDialog()*, si se quiere sustituir el de por defecto. En el código siguiente se muestra la línea de ejemplo.

```
1  PasswordUI.setPasswordDialog(new MyPasswordDialog());
   ↪ //'MyPasswordDialog' implementa la interfaz 'DialogUIHandler'
```

El ejemplo completo se encuentra en *SampleActivity_connect_gui*.

Paquete *de.tsenger.androsmex.mrtd*

DG1_Dnie, DG2, DG7, DG11, DG13, EF_COM: clases que encapsulan la información devuelta por la clase *es.gob.jmulticard.card.baseCard.mrtd.MrtdCard*, instancia que se recupera a través del objecto DnieLoadParameter en la carga del almacén de claves. Se muestra un ejemplo de acceso a los datos en el código siguiente, disponible en *SampleActivity_read_data*.

```
1  ...
2  MrtdCard mrtdCardInfo = Loader.init(new String[]{_can},
   ↪ tag).getMrtdCardInfo();
3  updateInfo("Leyendo datos...", "Obteniendo fecha de nacimiento...");
```

```
4   DG1_Dnie data = mrtdCardInfo.getDataGroup1();
5   Date date = new SimpleDateFormat("yyMMdd").parse(data.getDateOfBirth());
6
7   updateInfo("Leyendo datos...", "Obteniendo foto...");
8   DG2 data2 = mrtdCardInfo.getDataGroup2();
9
10  updateInfo("Fecha de nacimiento",new
    ↪  SimpleDateFormat("dd-MMMM-yyyy").format(date), new
    ↪  com.gemalto.jp2.JP2Decoder(data2.getImageBytes()).decode());
11  ...
```

4.5. Comprobación de edad

La librería permite realizar una consulta directa para verificar la edad del propietario del DNIe. Para ello se envía una fecha de interés, para comprobar si la fecha de nacimiento del sujeto es posterior a ésta. Esto permite verificar diferentes edades según sea la necesidad. Así, la función devolverá un valor booleano 'verdadero' si el nacimiento del sujeto es anterior o igual a la fecha indicada, o 'falso' si es posterior. Por ejemplo, si se quiere comprobar si el sujeto es mayor (o igual) de 18 años a día 12 de mayo del 2018 (12/05/2018), se enviará al DNIe la fecha 12 de mayo del 2000 (12/05/2000). Si queremos saber si es mayor de 16, se enviará la fecha 12 de mayo del 2002.

Para su invocación es necesario haber cargado el almacén de claves del Provider (KeyStore) como se ha visto en ejemplos anteriores. En el Activity *SampleActivity_age_verification* de la aplicación de ejemplo se puede observar el uso de esta función.

```
1   ...
2     KeyStore keyStore =
    ↪  KeyStore.getInstance(DnieProvider.KEYSTORE_PROVIDER_NAME);
3   keyStore.load(DnieLoadParameter.getBuilder(_can, tag).build());
4
5   updateInfo("Leyendo datos...", "Verificando edad...");
6   final Calendar calendar = Calendar.getInstance();
7   calendar.set(_date.getYear(), _date.getMonth(), _date.getDayOfMonth());
8   _image.setCensored(DnieProvider.verifyAge(calendar.getTime())?
    ↪  CanvasView.CENSORED.FALSE: CanvasView.CENSORED.TRUE);
9   // Actualizamos la imagen a mostrar
10  updateInfo("Comprobar edad", null);
11  ...
```

4.6. Obtener estado del certificado

Para obtener el estado de revocación de los certificados digitales del DNIe es necesario utilizar el servicio OCSP [8] indicado en la extensión *Acceso a información de autoridad* o **AIA** (**A**uthority **I**nformation **A**ccess), dentro del propio certificado. En el proyecto con el código fuente de ejemplos se implementa la clase *com.fnmt.sample_dnie_app.utils.pki.OSCP* que encapsula la funcionalidad de obtención de estado de revocación del certificado. Un ejemplo de uso se puede encontrar en *SampleActivity_provider*.

```
1  ...
2  CertificateStatus status =
   ↪ OSCP.checkCertificateStatus(_certificateChain[0],
   ↪ _certificateChain[1], "http://ocsp.dnie.es");
3  if(status == CertificateStatus.GOOD || status instanceof UnknownStatus){
4    updateInfo("Realizando firma...", null);
5    _signature = getSignature(_privateKey);
6  }
7  ...
```

Este es sólo un código de ejemplo. Las políticas de actuación según la información recuperada del servicio OCSP corresponden al desarrollador (en este caso aunque se desconozca el estado del certificado, se realiza la firma).

4.7. Información adicional

Para más información acerca de la funcionalidad disponible, las clases y sus usos, se dispone de documentación en formato *JavaDoc*, así como de un proyecto para *Android Studio* donde se puede encontrar el código utilizado para los ejemplos del documento.

Referencias

[1] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley y W. Polk, *RFC 5280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, 2008.

[2] ISO/IEC JTC 1/SC 17, *ISO/IEC 14443. Identification cards – Contactless integrated circuit cards – Proximity cards.*, 3ª edición, 2016.

[3] NFC Forum, NXP Semiconductors, *NFC Forum Type Tags White Paper V1.0*, 2009.

[4] *CWA 14890-1: Application Interface for smart cards used as Secure Signature Creation Device*, 2009.

[5] ICAO, *DOC 9303-11, Machine Readable Travel Documents Part 11-Security Mechanisms for MRTDs*, 2015.

[6] ORACLE, *Java Cryptography Architecture (JCA) Reference Guide.*

[7] Dpto. Documentos de identificación y tarjetas, FNMT, *Guía de referencia del DNIe con NFC*, 2017.

[8] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin y C. Adams, *RFC 6960, X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*, 2013.