

Acceso a servicios web

Índice

1 Acceso a la red.....	3
1.1 Conexión a URLs.....	3
1.2 Conexiones asíncronas.....	5
1.3 Estado de la red.....	12
1.4 Carga lazy de imágenes.....	14
2 Ejercicios de acceso a la red.....	23
2.1 Visor de HTML.....	23
2.2 Carga de imágenes.....	24
2.3 Carga lazy de imágenes (*).....	25
3 Acceso a servicios REST.....	27
3.1 Fundamentos de REST.....	27
3.2 Tipos de peticiones HTTP.....	32
3.3 Clientes de servicios REST.....	38
3.4 Parsing de estructuras XML.....	43
3.5 Parsing de estructuras JSON.....	46
4 Servicios de acceso a servicios REST.....	50
4.1 Consulta del libros.....	50
4.2 Timeline público de Twitter.....	51
5 Autenticación en servicios remotos.....	52
5.1 Seguridad HTTP básica.....	52
5.2 Protocolo OAuth.....	54
5.3 Acceso a servicios de terceros.....	59
6 Ejercicios de autenticación en servicios remotos.....	67
6.1 Publicación de libros.....	67
6.2 Acceso a Twitter.....	67
7 Visor web.....	69

7.1 Creación de un visor web.....	69
7.2 Propiedades del visor.....	72
7.3 Comunicación entre web y aplicación.....	75
7.4 PhoneGap.....	78
8 Ejercicios de visor web.....	84
8.1 Integración de contenido web.....	84
8.2 Temporizador web.....	85
8.3 Alarma con PhoneGap.....	86

1. Acceso a la red

En esta sesión vamos a ver cómo acceder a la red desde las aplicaciones Android e iOS. La forma habitual de acceder a servidores en Internet es mediante protocolo HTTP, mediante la URL en la que se localizan los recursos a los que queremos acceder.

Una consideración que debemos tener en cuenta es que las operaciones de red son operaciones lentas, y deberemos llevar cuidado para que no bloqueen la interfaz de nuestra aplicación. En esta sesión veremos cómo establecer este tipo de conexiones de forma correcta desde nuestras aplicaciones para móviles.

1.1. Conexión a URLs

Vamos a comenzar viendo cómo conectar con URLs desde aplicaciones Android e iOS. Lo habitual será realizar una petición GET a una URL y obtener el documento que nos devuelve el servidor, por lo que las APIs de acceso a URLs nos facilitarán fundamentalmente esta operación. Sin embargo, como veremos más adelante también será posible realizar otras operaciones HTTP, como POST, PUT o DELETE, entre otras.

1.1.1. Conexión a URLs en Android

Como paso previo, para todas las conexiones por Internet en Android necesitaremos declarar los permisos en el `AndroidManifest.xml`, fuera del `application` tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Las conexiones por HTTP son las más comunes en las comunicaciones de red. En Android podemos utilizar la clase `HttpURLConnection` en combinación con `Url`. Estas clases son las mismas que están presentes en Java SE, por lo que el acceso a URLs desde Android se puede hacer de la misma forma que en cualquier aplicación Java. Podemos ver información de las cabeceras de HTTP como se muestra a continuación (la información se añade a un `TextView`):

```
TextView textView = (TextView)findViewById(R.id.tvVisor);
textView.setText("Conexión http.\n\n");
try {
    textView.setText("Cabeceras www.ua.es:\n");
    URL url = new URL("http://www.ua.es");
    HttpURLConnection http = (HttpURLConnection)url.openConnection();
    textView.append(" longitud = "+http.getContentLength()+"\n");
    textView.append(" encoding = "+http.getContentEncoding()+"\n");
    textView.append(" tipo = "+http.getContentType()+"\n");
    textView.append(" response code = "+http.getResponseCode()+"\n");
    textView.append(" response message = "+http.getResponseMessage()+"\n");
    textView.append(" content = "+http.getContent()+"\n");
} catch (MalformedURLException e) {
} catch (IOException e) {
}
```

Sin embargo, en Android resulta más común utilizar la librería *Apache Http Client*. La librería de Java SE está diseñada de forma genérica, para soportar cualquier tipo de protocolo, mientras que *HttpClient* se centra en HTTP y con ella resulta más sencillo utilizar este protocolo.

La forma más sencilla de acceso con esta librería es la siguiente:

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet("http://www.ua.es");
try {
    ResponseHandler<String> handler = new BasicResponseHandler();
    String contenido = client.execute(request, handler);
} catch (ClientProtocolException e) {
} catch (IOException e) {
} finally {
    client.getConnectionManager().shutdown();
}
```

Con esta librería tenemos un objeto cliente, y sobre él podemos ejecutar peticiones. Cada petición se representa en un objeto, cuya clase corresponde con el tipo de petición a realizar. Por ejemplo, lo más común será realizar una petición GET, por lo que utilizaremos una instancia de `HttpGet` que construiremos a partir de la URL a la que debe realizar dicha petición.

Para obtener la respuesta, en caso de que queramos obtenerla como una cadena, lo cual también es lo más habitual, podemos utilizar un `BasicResponseHandler`, que se encarga de obtener el contenido en forma de `String`.

Por último, en el `finally` deberemos cerrar todas las conexiones del cliente HTTP si no lo vamos a utilizar más. Hay que destacar que el cliente (`HttpClient`) puede utilizarse para realizar varias peticiones. Lo cerraremos cuando no vayamos a realizar más peticiones con él.

En el ejemplo anterior hemos visto el caso en el que nos interesaba obtener la respuesta como cadena. Sin embargo, puede interesarnos obtener otros formatos, o tener más información sobre la respuesta HTTP (no sólo el contenido):

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet("http://www.ua.es/imagenes/logoua.png");
try {
    HttpResponse response = client.execute(request);
    InputStream in = response.getEntity().getContent();

    Bitmap imagen = BitmapFactory.decodeStream(in);
} catch (ClientProtocolException e) {
} catch (IOException e) {
} finally {
    client.getConnectionManager().shutdown();
}
```

Este es el caso general, en el que obtenemos la respuesta como un objeto `HttpResponse` del que podemos leer todas las propiedades de la respuesta HTTP. La parte más destacada de la respuesta es el bloque de contenido (`HttpEntity`). De la entidad podemos obtener sus propiedades, como su tipo MIME, longitud, y un flujo de entrada para leer el

contenido devuelto. En caso de que dicho contenido sea texto, será más sencillo leerlo como en el ejemplo previo.

1.1.2. Conexión a URLs en iOS

En la API de Cocoa Touch encontramos diferentes objetos para representar las URLs y las conexiones a las mismas. Veremos que se trabaja con estos objetos de forma similar a las librerías para conectar a URLs de Java.

En primer lugar, tenemos la clase `NSURL` que representa una URL. Esta clase se inicializa con la cadena de la URL:

```
NSURL *theUrl = [NSURL URLWithString:@"http://www.ua.es"];
```

Si queremos hacer una petición a dicha URL, deberemos crear un objeto `NSURLRequest` a partir de la URL anterior. De la petición podemos especificar también la política de caché a seguir, o el *timeout* de la conexión. Si queremos modificar más datos de la petición, deberíamos utilizar un objeto `NSMutableURLRequest`.

```
NSURLRequest *theRequest = [NSURLRequest requestWithURL: theUrl];
```

Una vez creada la petición, podemos establecer la conexión con la URL y obtener el contenido que nos devuelva. Para hacer esto tenemos el objeto `NSURLConnection`. La forma más sencilla es utilizar el siguiente método:

```
NSURLResponse *theResponse;  
NSError *theError;  
NSData *datos = [NSURLConnection sendSynchronousRequest:theRequest  
                                returningResponse:&theResponse  
                                error:&theError];
```

Con esto se realiza la conexión y nos devuelve los datos obtenidos encapsulados en un objeto de tipo `NSData`. La clase `NSData` se utiliza para encapsular datos binarios, y podemos leerlos directamente como array de bytes, o bien mediante otros tipos de objetos. Por ejemplo, podemos obtener los datos en forma de cadena mediante el inicializador `initWithData:encoding:` de la clase `NSString`:

```
NSString *contenido = [[NSString alloc] initWithData: datos  
                                encoding: NSASCIIStringEncoding];
```

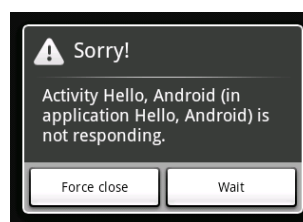
Esta forma síncrona de realizar la conexión es sencilla, pero resulta poco adecuada debido a que dejará bloqueada la aplicación hasta que obtenga la respuesta, y en el caso de las operaciones de acceso a la red esto podría demorarse durante un tiempo considerable. Por lo tanto, lo recomendable será realizar la conexión siempre de forma asíncrona para evitar este problema, como veremos a continuación.

1.2. Conexiones asíncronas

En Internet no se puede asumir que ninguna operación de red vaya a ser rápida o vaya a durar un tiempo limitado (el límite lo establece, en todo caso, el *timeout* de la conexión).

En los dispositivos móviles, todavía menos, ya que continuamente pierden calidad de la señal o pueden cambiar de Wifi a 3G sin preguntarnos, y perder conexiones o demorarlas durante el proceso.

Si una aplicación realiza una operación de red en el mismo hilo de la interfaz gráfica, el lapso de tiempo que dure la conexión, la interfaz gráfica dejará de responder. Este efecto es indeseable ya que el usuario no lo va a comprender, ni aunque la operación dure sólo un segundo. Es más, si la congelación dura más de dos segundos, es muy probable que el sistema operativo muestre el diálogo ANR, "Application not responding", invitando al usuario a matar la aplicación:



Mensaje ANR

Para evitar esto hay que realizar las conexiones de forma asíncrona, fuera del hilo de eventos de nuestra aplicación. En iOS podemos realizar la conexión de forma asíncrona de forma sencilla mediante la misma clase `NSURLConnection` vista anteriormente y objetos delegados. En Android deberemos ser nosotros lo que creamos otro hilo (`Thread`) de ejecución en el que se realice la conexión.

Durante el tiempo que dure la conexión la aplicación podrá seguir funcionando de forma normal, será decisión nuestra cómo interactuar con el usuario durante este tiempo. En algunos casos nos puede interesar mostrar una diálogo de progreso que evite que se pueda realizar ninguna otra acción durante el acceso. Sin embargo, esto es algo que debemos evitar siempre que sea posible, ya que el abuso de estos diálogos entorpecen el uso de la aplicación. Resulta más apropiado que la aplicación siga pudiendo ser utilizada por el usuario durante este tiempo, aunque siempre deberemos indicar de alguna forma que se está accediendo a la red. Para ello normalmente contaremos en la barra de estado con un indicador de actividad de red, que deberemos activar al comenzar la conexión, y desactivar cuando se cierre. A continuación veremos cómo utilizar estos elementos en las diferentes plataformas móviles.



Indicador de actividad

1.2.1. Conexiones asíncronas en iOS

La forma habitual de realizar una conexión en iOS es de forma asíncrona. Para ello deberemos utilizar el siguiente constructor de la clase `NSURLConnection`:

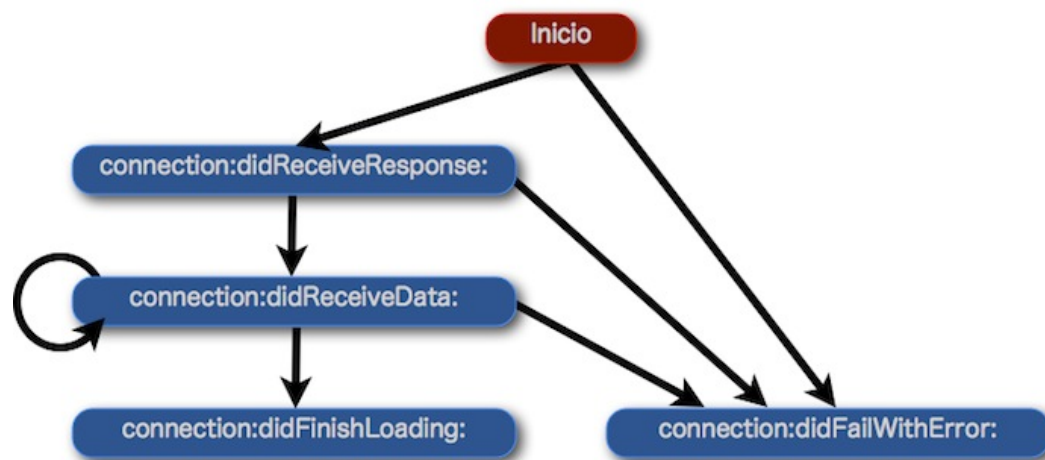
```
NSURLConnection *theConnection = [NSURLConnection  
    connectionWithRequest: theRequest delegate: self];
```

Como podemos ver, en este caso a la conexión sólo le pasamos la petición y un objeto delegado. No hace falta pasarle ningún parámetro de salida para la respuesta HTTP ni para el error, ni esperamos que nos devuelva ningún contenido. Conforme vayan llegando datos desde el servidor se irá llamando a los métodos del delegado para proporcionarle la información recibida, pero nuestra aplicación seguirá funcionando durante ese tiempo. Es recomendable que al poner en marcha la conexión asíncrona, también activemos el indicador de actividad de red que aparece en la barra de estado, para que así el usuario sepa que se está realizando una conexión:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

El objeto delegado que especificamos al crear la conexión deberá implementar el protocolo `NSURLConnectionDataDelegate`. Algunos métodos destacados de este protocolo son:

- `connection:didReceiveResponse:` Se ejecuta una vez recibimos la respuesta del servidor. En este momento ya tenemos las cabeceras de la respuesta, pero aun no su contenido.
- `connection:didReceiveData:` Se ejecuta cada vez que llega un nuevo bloque de datos del servidor. La respuesta puede llegar troceada, por lo que cada vez que nos llegue un nuevo bloque deberemos añadirlo a la respuesta que tenemos hasta el momento.
- `connection:didFailWithError:` Se ejecuta en caso de que la conexión falle. Recibiremos el objeto de error correspondiente.
- `connectionDidFinishLoading:` Se ejecuta cuando la conexión ha terminado de recibir la respuesta. En este momento sabremos que ya tenemos la respuesta completa, y podremos procesarla.



Secuencia de llamadas en el delegado de la conexión

Dado que, como hemos visto, al delegado le va llegando la respuesta por trozos, deberemos tener algún objeto al que añadirla. La respuesta se obtiene mediante un objeto de tipo `NSData`, por lo que lo más sencillo será crear en nuestro delegado una propiedad de tipo `NSMutableData` a la que podamos ir añadiendo cada fragmento recibido:

```
// En la declaración de propiedades
@property (nonatomic, strong) NSMutableData *downloadedData;
```

El mejor momento para inicializar dicho objeto será cuando recibamos el mensaje de respuesta en `connection:didReceiveResponse:`, ya que aquí contaremos con cabeceras que nos dirán el tipo y la longitud del contenido de la respuesta. Conociendo la longitud, podemos inicializar el objeto `NSMutableData` con la capacidad adecuada:

```
- (void)connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response
{
    if(response.expectedContentLength != NSURLResponseUnknownLength) {
        self.downloadedData = [NSMutableData dataWithCapacity:
                               response.expectedContentLength];
    } else {
        self.downloadedData = [NSMutableData data];
    }
}
```

Tras esto, cada vez que nos llegue un nuevo fragmento de datos en `connection:didReceiveData:` lo añadiremos al objeto anterior:

```
- (void) connection:(NSURLConnection *)connection
didReceiveData:(NSData *)data
{
    [self.downloadedData appendData: data];
}
```

Deberemos implementar también métodos que nos notifiquen cuando la respuesta se ha completado o cuando ha fallado la conexión. En cualquiera de los dos deberemos desactivar el indicador de actividad de red:


```
- (void) connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error
{
    [UIApplication
        sharedApplication].networkActivityIndicatorVisible = NO;
}

- (void) connectionDidFinishLoading:(NSURLConnection *)connection
{
    [UIApplication
        sharedApplication].networkActivityIndicatorVisible = NO;
    [self actualizarDatos];
}
```

1.2.2. Conexiones asíncronas en Android

Una forma sencilla de realizar una conexión de forma asíncrona es utilizar hilos, de la misma forma que en Java SE:

```
ImageView imageView = (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
    public void run() {
        Drawable imagen = cargarLaImagen("http://...");
        //Desde aquí NO debo acceder a imageView
    }
}).start();
```

Pero hay un problema: tras cargar la imagen no puedo acceder a la interfaz gráfica porque la GUI de Android sigue un modelo de hilo único: sólo un hilo puede acceder a ella. Se puede solventar de varias maneras. Una es utilizar el método `View.post(Runnable)`.

```
ImageView imageView = (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
    public void run() {
        Drawable imagen = cargarLaImagen("http://...");
        imageView.post(new Runnable() {
            public void run() {
                imageView.setDrawable(imagen);
            }
        });
    }
}).start();
```

Con esto lo que se hace es indicar un fragmento de código que debe ejecutarse en el hilo principal de eventos. En dicho fragmento de código se realizan los cambios necesarios en la interfaz. De esta forma, una vez la conexión ha terminado de cargar de forma asíncrona, desde el hilo de la conexión se introduce en el hilo principal de la UI el código que realice los cambios necesarios para mostrar el contenido obtenido.

Como alternativa, contamos también con el método `Activity.runOnUiThread(Runnable)` para ejecutar un bloque de código en el hilo de la UI:

```
ImageView imageView = (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
    public void run() {
```

```

        Drawable imagen = cargarLaImagen("http://...");
        runOnUiThread(new Runnable() {
            public void run() {
                imageView.setDrawable(imagen);
            }
        });
    }
}).start();

```

Con esto podemos crear conexiones asíncronas cuyo resultado se muestre en la UI. Sin embargo, podemos observar que generan un código bastante complejo. Para solucionar este problema a partir de Android 1.5 se introduce la clase `AsyncTask` que nos permite implementar tareas asíncronas de forma más elegante. Se trata de una clase creada para facilitar el trabajo con hilos y con interfaz gráfica, y es muy útil para ir mostrando el progreso de una tarea larga, durante el desarrollo de ésta. Nos facilita la separación entre tarea secundaria e interfaz gráfica permitiéndonos solicitar un refresco del progreso desde la tarea secundaria, pero realizarlo en el hilo principal.

```

TextView textView;
ImageView[] imageView;

public void bajarImágenes(){
    textView = (TextView)findViewById(R.id.TextView01);
    imageView[0] = (ImageView)findViewById(R.id.ImageView01);
    imageView[1] = (ImageView)findViewById(R.id.ImageView02);
    imageView[2] = (ImageView)findViewById(R.id.ImageView03);
    imageView[3] = (ImageView)findViewById(R.id.ImageView04);

    new BajarImágenesTask().execute(
        "http://a.com/1.png",
        "http://a.com/2.png",
        "http://a.com/3.png",
        "http://a.com/4.png");
}

private class BajarImágenesTask extends
    AsyncTask<String, Integer, List<Drawable>> {
    @Override
    protected List<Drawable> doInBackground(String... urls) {
        ArrayList<Drawable> imagenes = new ArrayList<Drawable>();
        for(int i=1;i<urls.length; i++) {
            cargarLaImagen(urls[i]);
            publishProgress(i);
        }
        return imagenes;
    }

    @Override
    protected void onPreExecute() {
        setProgressBarIndeterminateVisibility(true);
        textView.setText("Comenzando la descarga ...");
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        textView.setText(values[0] + " imágenes cargadas...");
    }

    @Override
    protected void onPostExecute(List<Drawable> result) {
        setProgressBarIndeterminateVisibility(false);
    }
}

```

```

        textView.setText("Descarga finalizada");

        for(int i=0; i<result.length; i++){
            imageView[i].setDrawable(result.getItemAt(i));
        }

        @Override
        protected void onCancelled() {
            setProgressBarIndeterminateVisibility(false);
        }
    }

```

Nota:

La notación (String ... values) indica que hay un número indeterminado de parámetros, y se accede a ellos con values[0], values[1], ..., etcétera. Forma parte de la sintaxis estándar de Java.

Lo único que se ejecuta en el segundo hilo de ejecución es el bucle del método `doInBackground(String...)`. El resto de métodos se ejecutan en el mismo hilo que la interfaz gráfica.

Podemos observar que en la `AsyncTask` se especifican tres tipos utilizando genéricos:

```
class Tarea extends AsyncTask<Entrada, Progreso, Resultado>
```

El primer tipo es el que se recibe como datos de entrada. Realmente se recibe un número variable de objetos del tipo indicado. Cuando ejecutamos la tarea con `execute` deberemos especificar como parámetros de la llamada dicha lista de objetos, que serán recibidos por el método `doInBackground`. Este método es el que implementará la tarea a realizar de forma asíncrona, y al ejecutarse en segundo plano deberemos tener en cuenta que **nunca** deberemos realizar cambios en la interfaz desde él. Cualquier cambio en la interfaz deberemos realizarlo en alguno de los demás métodos.

El segundo tipo de datos que se especifica es el tipo del progreso. Conforme avanza la tarea en segundo plano podemos publicar actualizaciones del progreso realizado. Hemos dicho que desde el método `doInBackground` no podemos modificar la interfaz, pero si que podemos llamar a `publishProgress` para solicitar que se actualice la información de progreso de la tarea, indicando como información de progreso una lista de elementos del tipo indicado como tipo de progreso, que habitualmente son de tipo entero (`Integer`). Tras hacer esto se ejecutará el método `onProgressUpdate` de la tarea, que recibirá la información que pasamos como parámetro. Este método si que se ejecuta dentro del hilo de la interfaz, por lo que podremos actualizar la visualización del progreso dentro de él, en función de la información recibida. Es importante entender que la ejecución de `onProgressUpdate(...)` no tiene por qué ocurrir inmediatamente después de la petición `publishProgress(...)`, o puede incluso no llegar a ocurrir.

Por último, el tercer tipo corresponde al resultado de la operación. Es el tipo que devolverá `doInBackground` tras ejecutarse, y lo recibirá `onPostExecute` como parámetro. Este último método podrá actualizar la interfaz con la información resultante

de la ejecución en segundo plano.

También contamos con el método `onPreExecute`, que se ejecutará justo antes de comenzar la tarea en segundo plano, y `onCancelled`, que se ejecutará si la tarea es cancelada (una tarea se puede cancelar llamando a su método `cancel`, y en tal caso no llegará a ejecutarse `onPostExecute`). Estos métodos nos van a resultar de gran utilidad para gestionar el indicador de actividad de la barra de estado. Este indicador se activa y desactiva mediante el siguiente método de la actividad:

```
// Mostrar indicador de actividad en la barra de estado
setProgressBarIndeterminateVisibility(true);

// Ocultar indicador de actividad en la barra de estado
setProgressBarIndeterminateVisibility(false);
```

Lo habitual será activarlo en `onPreExecute`, y desactivarlo tanto en `onPostExecute` como en `onCancelled`. Este indicador informa al usuario de que la aplicación está trabajando, pero no da información concreta sobre el progreso (por eso se llama *Indeterminate*). Para poder utilizar este tipo de progreso, al crear la actividad deberemos haberlo solicitado:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
    ...
}
```

También contamos con una barra de progreso en la barra de estado, que podemos activar con `setProgressBarVisibility`, habiéndola solicitado previamente con:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getWindow().requestFeature(Window.FEATURE_PROGRESS);
    ...
}
```

En este caso, deberemos modificar el progreso mientras se realiza la carga. El progreso se indicará mediante un valor de 0 a 10000, y se actualizará mediante el método `setProgress(int)` de la actividad. Podemos llamar a este método desde `onProgressUpdate`, tal como hemos visto anteriormente:

```
@Override
protected void onProgressUpdate(Integer... values) {
    setProgress(values[0] * 10000 / numero_imagenes);
}
```

1.3. Estado de la red

En algunas aplicaciones puede convenir comprobar el estado de red. El estado de red no es garantía de que la conexión vaya a funcionar, pero sí que puede prevenirnos de intentar establecer una conexión que no vaya a funcionar. Por ejemplo, hay aplicaciones que

requieren el uso de la WIFI para garantizar mayor velocidad.

Cuando desarrollemos una aplicación que acceda a la red deberemos tener en cuenta que el usuario normalmente contará con una tarifa de datos limitada, en la que una vez superado el límite o bien se le tarificará por consumo, o bien se le reducirá la velocidad de conexión. Por este motivo, deberemos llevar especial cuidado con las operaciones de red, y velar al máximo por reducir el consumo de datos del usuario.

En ciertas ocasiones esto puede implicar limitar ciertas funcionalidades de la aplicación a las zonas en las que contemos con conexión Wi-Fi, o por lo menos avisar al usuario en caso de que solicite una de estas operaciones mediante 3G, y darle la oportunidad de cancelarla.

1.3.1. Comprobación de la conectividad en Android

A continuación se muestra cómo usar el `ConnectivityManager` para comprobar el estado de red en dispositivos Android.

```
ConnectivityManager cm = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo wifi = cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
NetworkInfo mobile = cm.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean hayWifi = wifi.isAvailable();
boolean hayMobile = mobile.isAvailable();
boolean noHay = (!hayWifi && !hayMobile); //¡¡¡iinteerneer!!
```

El `ConnectivityManager` también puede utilizarse para controlar el estado de red, o bien estableciendo una preferencia pero permitiéndole usar el tipo de conectividad que realmente esté disponible,

```
cm.setNetworkPreference(NetworkPreference.PREFER_WIFI);
```

o bien pidiéndole explícitamente que se desconecte de la red móvil y se conecte a la red WiFi:

```
cm.setRadio(NetworkType.MOBILE,false);
cm.setRadio(NetworkType.WIFI,true);
```

1.3.2. Conectividad en iOS

En iOS también podemos comprobar el tipo de conexión con el que contamos. De hecho, uno de los ejemplos publicados en la documentación de Apple consiste precisamente en esto. El ejemplo se llama *Reachability* (se puede encontrar en la sección *Sample Code* de la ayuda), y su código se puede utilizar directamente en nuestras aplicaciones. En el caso de iOS la comprobación no es tan sencilla como en Android, y se requiere el acceso a la conexión a bajo nivel. Por este motivo, lo más habitual entre los desarrolladores es incorporar el código del ejemplo *Reachability*.

Lo que necesitaremos incorporar al proyecto son los ficheros `Reachability.m` y `Reachability.h`. Además, también necesitaremos el *framework* `SystemConfiguration`

(deberemos añadirlo en la pantalla *Build Phases* del *Target*).

```
internetReach = [[Reachability reachabilityForInternetConnection] retain];
[internetReach startNotifier];
if([internetReach currentReachabilityStatus] == ReachableViaWWAN) {
    // Tenemos conexión a Internet
}

wifiReach = [[Reachability reachabilityForLocalWiFi] retain];
[wifiReach startNotifier];
if([wifiReach currentReachabilityStatus] == ReachableViaWiFi) {
    // Estamos conectados mediante Wi-Fi
}
```

1.4. Carga lazy de imágenes

Otro caso típico en el trabajo con HTTP es el de cargar una lista de imágenes para almacenarlas o bien mostrarlas. Lo más habitual es tener un componente de tipo lista o tabla, en el que para cada elemento se muestra una imagen como icono. En una primera aproximación, tal como hemos visto en alguno de los ejemplos anteriores, podríamos cargar todas las imágenes al cargar los datos de la lista, y tras ello actualizar la interfaz. Sin embargo, esto tiene serios problemas. El primero de ellos es el tiempo que pueden tardar en cargarse todas las imágenes de una lista. Podría dejar al usuario en espera durante demasiado tiempo. Por otro lado, estaríamos cargando todas las imágenes, cuando es posible que el usuario no esté interesado en recorrer toda la lista, sino sólo sus primeros elementos. En este caso estaríamos malgastando la tarifa de datos del usuario de forma innecesaria.

Un mejor enfoque para la carga de imágenes de listas es hacerlo de forma *lazy*, es decir, cargar la imagen de una fila sólo cuando dicha fila se muestre en pantalla. Además, cada imagen se cargará de forma asíncrona, mediante su propio hilo en segundo plano, y cuando la carga se haya completado se actualizará la interfaz. El efecto que esto producirá será que veremos como van apareciendo las imágenes una a una, conforme se completa su carga. Vamos a ver ahora algunas recetas para implementar este comportamiento en Android e iOS.

Como mejora, también se suele hacer que la carga *lazy* sólo se produzca en el caso en el que no estemos haciendo *scroll* en la lista. Es posible que el usuario esté buscando un determinado elemento en una larga lista, o que esté interesado en los últimos elementos. En tal caso, mientras hace *scroll* rápidamente para llegar al elemento buscado será recomendable evitar que las imágenes por las que pasemos se pongan en la lista de carga, ya que en principio el usuario no parece interesado en ellas. Esto se puede implementar de forma sencilla atendiendo a los eventos del *scroll*, y añadiendo las imágenes a la cola de descargas sólo cuando se encuentre detenido.

Según la aplicación, también podemos guardar las imágenes de forma persistente, de forma que en próximas visitas no sea necesario volver a descargarlas. En caso de tener un conjunto acotado de elementos a los que accedamos frecuentemente, puede ser recomendable almacenarlos en una base de datos propia, junto con su imagen. De no ser

así, podemos almacenar las imágenes en una caché temporal (`Context.getCacheDir()` en Android, `NSCachesDirectory` en iOS).

1.4.1. Carga lazy en Android

En Android podemos implementar la carga *lazy* de imágenes en el mismo adaptador que se encargue de poblar la lista de datos. Por ejemplo, imaginemos el siguiente adaptador que obtiene los datos a partir de un *array* de elementos de tipo `Elemento` (con los campos `texto`, `imagen`, y `urlImagen`):

```
public class ImagenAdapter extends ArrayAdapter<Elemento> {

    public ImagenAdapter(Context context, List<Elemento> objects) {
        super(context, R.id.tvTitulo, objects);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        if(convertView == null) {
            LayoutInflater li = (LayoutInflater) this.getContext()
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            convertView = li.inflate(R.layout.item, null);
        }

        TextView tvTexto = (TextView) convertView.findViewById(R.id.tvTitulo);
        ImageView ivIcono = (ImageView) convertView
            .findViewById(R.id.ivIcono);

        Elemento elemento = this.getItem(position);
        tvTexto.setText(elemento.getTexto());

        if(elemento.getImagen()!=null) {
            ivIcono.setImageBitmap(elemento.getImagen());
        }

        return convertView;
    }
}
```

El *layout* de cada fila se puede definir de la siguiente forma:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" android:orientation="horizontal">
    <ImageView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/icon"
        android:id="@+id/ivIcono"></ImageView>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:id="@+id/tvTitulo"></TextView>
</LinearLayout>
```

El adaptador anterior funcionará siempre que las imágenes se encuentren ya cargadas en memoria (dentro del campo `imagen` del objeto `Elemento`). Sin embargo, si queremos implementar carga *lazy*, deberemos hacer que al rellenar cada fila, en caso de no estar

todavía cargada la imagen, ponga en marcha una `AsyncTask` que se encargue de ello. Para evitar que se pueda crear más de una tarea de descarga para un mismo elemento, crearemos un mapa en memoria con todas las imágenes que se están cargando actualmente, y sólo comenzaremos una carga si no hay ninguna en marcha para el elemento indicado:

```
public class ImagenAdapter extends ArrayAdapter<Elemento> {

    // Mapa de tareas de carga en proceso
    Map<Elemento, CargarImagenTask> imagenesCargando;

    public ImagenAdapter(Context context, List<Elemento> objects) {
        super(context, R.id.tvTitulo, objects);
        imagenesCargando = new HashMap<Elemento, CargarImagenTask>();
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ...

        if(elemento.getImagen()!=null) {
            ivIcono.setImageBitmap(elemento.getImagen());
        } else {
            // Si la imagen no está cargada, comienza una tarea de carga
            ivIcono.setImageResource(R.drawable.icon);
            this.cargarImagen(elemento, ivIcono);
        }

        return convertView;
    }

    private void cargarImagen(Elemento elemento, ImageView view) {
        // Solo carga la imagen si no esta siendo cargada ya
        if(imagenesCargando.get(elemento)==null) {
            CargarImagenTask task = new CargarImagenTask();
            imagenesCargando.put(elemento, task);
            task.execute(elemento, view);
        }
    }

    class CargarImagenTask extends AsyncTask<Object, Integer, Bitmap> {

        Elemento elemento;
        ImageView view;

        @Override
        protected Bitmap doInBackground(Object... params) {
            this.elemento = (Elemento)params[0];
            this.view = (ImageView)params[1];

            HttpClient client = new DefaultHttpClient();
            HttpGet request = new HttpGet(this.elemento.getUrlImagen());
            try {
                HttpResponse response = client.execute(request);
                Bitmap imagen = BitmapFactory
                    .decodeStream(response.getEntity().getContent());

                return imagen;
            } catch(IOException e) {
            }

            return null;
        }
    }
}
```



```
@Override
protected void onPostExecute(Bitmap result) {
    if(result!=null) {
        this.elemento.setImagen(result);
        this.view.setImageBitmap(result);
    }
}
}
```

Como mejora, se puede hacer que las imágenes sólo carguen si no se está haciendo *scroll* en la lista. Para ello podemos hacer que el adaptador implemente `OnScrollListener`, y registrarlo como oyente de la lista:

```
this.getListView().setOnScrollListener(adaptador)
```

En el adaptador podemos crear una variable que indique si está ocupado o no haciendo *scroll*, y que sólo descargue imágenes cuando no esté ocupado. Cuando pare el *scroll*, recargaremos los datos de la lista (`notifyDataSetChanged()`) para que carguen todas las imágenes que haya actualmente en pantalla:

```
public class ImagenAdapter extends ArrayAdapter<Elemento>
    implements OnScrollListener {
    boolean busy = false;

    ...

    private void cargarImagen(Elemento elemento, ImageView view) {
        if(imagenesCargando.get(elemento)!=null && !busy) {
            new CargarImagenTask().execute(elemento, view);
        }
    }

    public void onScroll(AbsListView view, int firstVisibleItem,
        int visibleItemCount, int totalItemCount) {
    }

    public void onScrollStateChanged(AbsListView view, int scrollState) {
        switch(scrollState) {
            case OnScrollListener.SCROLL_STATE_IDLE:
                busy = false;
                notifyDataSetChanged();
                break;
            case OnScrollListener.SCROLL_STATE_TOUCH_SCROLL:
                busy = true;
                break;
            case OnScrollListener.SCROLL_STATE_FLING:
                busy = true;
                break;
        }
    }
}
```

Si la tabla tuviese una gran cantidad de elementos, y cargásemos las imágenes de todos ellos, podríamos correr el riesgo de quedarnos sin memoria. Una posible forma de evitar este problema es utilizar la clase `SoftReference`. Con ella podemos crear una referencia débil a datos, de forma que si Java se queda sin memoria será eliminada automáticamente

de ella. Esto es bastante adecuado para las imágenes de una lista, ya que si nos quedamos sin memoria será conveniente que se liberen y se vuelvan a cargar cuando sea necesario. Podemos crear una referencia débil de la siguiente forma:

```
public class Elemento {
    ...

    SoftReference<Bitmap> imagen;

    ...
}
```

Para obtener la imagen referenciada débilmente deberemos llamar al método `get()` del objeto `SoftReference`:

```
if(elemento.getImagen().get()!=null) {
    ivIcono.setImageBitmap(elemento.getImagen().get());
} else {
    ...
}
```

Para crear una nueva referencia débil a una imagen deberemos utilizar el constructor de `SoftReference` a partir de la imagen que vamos a referenciar:

```
protected void onPostExecute(Bitmap result) {
    if(result!=null) {
        this.elemento.setImagen(new SoftReference<ImagenCache>(result));
        this.view.setImageBitmap(result);
    }
}
```

Cuando el dispositivo se esté quedando sin memoria, podrá liberar automáticamente el contenido de todos los objetos `SoftReference`, y sus referencias se pondrán a `null`.

1.4.2. Carga lazy en iOS

En iOS contamos en la documentación con un ejemplo proporcionado por Apple que realiza esta función. Vamos a ver aquí como ejemplo una solución simplificada. La forma de trabajar será similar a la vista en el caso de Android. En el controlador de nuestra tabla podemos incluir un diccionario que contenga las imágenes que se están descargando actualmente (equivale al mapa de tareas de descarga que teníamos en Android):

```
@property(nonatomic, strong) NSMutableDictionary *downloadingImages;
```

Conforme se vayan rellenando las celdas, se solicita la carga de las imágenes que no hayan sido cargadas todavía:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    ...
    // Configure the cell.
    UElemento *theItem = [self.listadoElementos
                        objectAtIndex: indexPath.row];
    cell.textLabel.text = theItem.texto;
}
```

```

if(theItem.imagen!=nil) {
    cell.imageView.image = theItem.imagen;
} else if(theItem.urlImagen!=nil) {
    [self cargarImagen: theItem paraIndexPath: indexPath];
    cell.imageView.image = [UIImage imageNamed: @"Placeholder.png"];
}

return cell;
}

```

En caso de no contar todavía con la imagen, podemos poner una imagen temporal en la celda. La carga de imágenes se puede hacer de la siguiente forma:

```

- (void) cargarImagen: (UAElemento *) item
    paraIndexPath: (NSIndexPath *) indexPath
{
    if([self.downloadingImages objectForKey: indexPath] == nil) {

        UIImageDownloader *theDownloader = [[UIImageDownloader alloc]
            initWithUrl:item.urlImagen indexPath:indexPath delegate:self];
        [self.downloadingImages setObject: theDownloader
            forKey: indexPath];
        [theDownloader release];
    }
}

```

Cada descarga es gestionada por un objeto de la clase `UIImageDownloader`, que se encargará de gestionar la descarga asíncrona de la imagen cuya URL se le pasa en el inicializador, y estos objetos se introducen en el diccionario de descargas en progreso. Utilizamos el patrón delegado para que, una vez se haya finalizado la descarga, se nos proporcione la imagen obtenida. Definimos el protocolo para el delegado junto a la clase `UIImageDownloader`:

```

@class UIImageDownloader;

@protocol UIImageDownloaderDelegate

- (void) imageDownloader: (UIImageDownloader *) downloader
    didFinishDownloadingImage: (UIImage *) image
    forIndexPath: (NSIndexPath *) indexPath;

@end

@interface UIImageDownloader : NSObject

- (id)initWithUrl: (NSString *) url
    indexPath: (NSIndexPath *)indexPath
    delegate: (id<UIImageDownloaderDelegate>) delegate;

@property(nonatomic,strong) NSString *url;
@property(nonatomic,strong) NSURLConnection *connection;
@property(nonatomic,strong) NSMutableData *data;
@property(nonatomic,strong) UIImage *image;

@property(nonatomic,unsafe_unretained)
    id<UIImageDownloaderDelegate> delegate;
@property(nonatomic,unsafe_unretained) NSIndexPath *indexPath;

@end

```

Podemos implementar esta clase de la siguiente forma:

```
@implementation UIImageDownloader

// Sintetización de propiedades
...

- (id)initWithUrl: (NSString *) url
    indexPath: (NSIndexPath *)indexPath
    delegate: (id<UIImageDownloaderDelegate>) delegate;
{
    self = [super init];
    if (self) {
        NSURL *urlImagen = [NSURL URLWithString: url];
        NSURLRequest *theRequest =
            [NSURLRequest requestWithURL: urlImagen];
        NSURLConnection *theConnection =
            [NSURLConnection connectionWithRequest: theRequest
                                             delegate: self];

        self.url = url;
        self.indexPath = indexPath;
        self.delegate = delegate;

        self.connection = theConnection;
        self.data = [NSMutableData data];
    }
    return self;
}

- (void) connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data
{
    [self.data appendData: data];
}

- (void) connectionDidFinishLoading:(NSURLConnection *)connection
{
    UIImage *theImage = [UIImage imageWithData: self.data];
    self.image = theImage;

    [self.delegate imageDownloader: self
                  didFinishDownloadingImage:self.image
                  forIndexPath:self.indexPath];
}

@end
```

La carga de la imagen se realiza de la misma forma que hemos visto anteriormente para descargar contenido de una URL. Una vez finalizada la descarga se notifica al delegado. Utilizaremos el controlador de la tabla como delegado, que responderá de la siguiente forma cuando una imagen haya sido descargada:

```
- (void) imageDownloader:(UIImageDownloader *)downloader
    didFinishDownloadingImage:(UIImage *)image
    forIndexPath:(NSIndexPath *)indexPath {

    UAElemento *theItem =
        [self.listadoElementos objectAtIndex: indexPath.row];
    theItem.imagen = image;
}
```

```
UITableViewCell *cell =
    [self.tableView cellForRowAtIndexPath: indexPath];
cell.imageView.image = image;
}
```

Al recibir la imagen descargada la almacena en el ítem correspondiente, y la muestra en la tabla.

Tal como hemos visto en el caso de Android, podemos evitar que se descarguen las imágenes mientras se está haciendo *scroll* con un código como el siguiente:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    ...
    // Configure the cell.
    UAElemento *theItem =
        [self.listadoElementos objectAtIndex: indexPath.row];
    cell.textLabel.text = theItem.texto;

    if(theItem.imagen!=nil) {
        cell.imageView.image = theItem.imagen;
    } else if(theItem.urlImagen!=nil) {
        if (self.tableView.dragging == NO &&
            self.tableView.decelerating == NO) {
            [self cargarImagen: theItem paraIndexPath: indexPath];
        }
        cell.imageView.image = [UIImage imageNamed: @"Placeholder.png"];
    }

    return cell;
}
```

Como podemos observar, impedimos la carga tanto si el usuario está arrastrando la lista, como si el *scroll* se está realizando por la inercia de un movimiento anterior. Cuando el scroll pare deberemos hacer que carguen las imágenes de todas las celdas visibles en pantalla. Para ello deberemos implementar los siguientes métodos del protocolo `UIScrollViewDelegate`:

```
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
    willDecelerate:(BOOL)decelerate {
    if (!decelerate) {
        [self cargarImagenesEnPantalla];
    }
}

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView {
    [self cargarImagenesEnPantalla];
}

- (void)cargarImagenesEnPantalla {
    if ([self.listadoElementos count] > 0) {
        NSArray *visiblePaths =
            [self.tableView indexPathsForVisibleRows];
        for (NSIndexPath *indexPath in visiblePaths) {
            UAElemento *item =
                [self.listadoElementos objectAtIndex:indexPath.row];

            if (!item.imagen) {
```

```
        [self cargarImagen: theItem paraIndexPath: indexPath];  
    }  
}
```

Respecto a la memoria, si queremos evitar que las imágenes de una tabla puedan llenar la memoria, deberemos implementar el método `didReceiveMemoryWarning`. En él deberíamos detener las descargas pendientes, y liberar de memoria las imágenes que no sean necesarias. En este caso la liberación de los objetos la deberemos implementar manualmente nosotros.

Hemos visto que en ambos casos utilizamos un mapa o diccionario para almacenar todas las descargas que ya se han iniciado, y así evitar que se repita una descarga para la misma entrada de la tabla. Hemos utilizado la entrada de la tabla para indexar este mapa o diccionario. Sin embargo, en algunos casos podría ocurrir que múltiples entradas con la misma imagen (por ejemplo varios *tweets* de una misma persona). Si es probable que distintas entradas tengan la misma imagen, sería recomendable utilizar la propia URL de la imagen para indexar el mapa o diccionario de descargas.

2. Ejercicios de acceso a la red

Nota

Los ejercicios de esta sesión se pueden realizar tanto en Android como en iOS. Debes realizar al menos uno de ellos para cada plataforma. Implementar un ejercicio para las dos plataformas se considera parte optativa.

2.1. Visor de HTML

Vamos a hacer una aplicación que nos permita visualizar el código HTML de la URL que indiquemos. En las plantillas de la sesión encontramos el proyecto `LectorHtml` que podemos utilizar como base, tanto en versión iOS como Android. Este proyecto contiene una pantalla con un cuadro de edición texto para introducir la URL, un botón para cargar la URL, y un visor de texto donde deberemos mostrar los resultados obtenidos cuando se pulse el botón. Se pide:

a) Implementar el código necesario para que cuando se pulse el botón se realice una conexión a la URL indicada, se obtenga el resultado como texto, y se muestre en el visor de texto (por el momento podemos realizar la conexión de forma síncrona para simplificar).

Ayuda iOS

Trabajaremos con la clase `UAViewController`. Concretamente la conexión deberá ser iniciada en el método `conectarUrl`.

Ayuda Android

¡IMPORTANTE!, en primer lugar debemos añadir los permisos necesarios a `AndroidManifest.xml` (de no hacer esto, nos parecerá que el emulador no tiene acceso a la red). Trabajaremos con la clase `LectorHtmlActivity`. Implementaremos la conexión en el método `getContent`, que deberá devolver el resultado como una cadena de texto. En el evento del botón deberemos realizar la conexión llamando al método anterior.

b) Modifica el código anterior para que la conexión se realice de forma asíncrona.

Ayuda iOS

La conexión se deberá iniciar en el mismo método `conectarUrl`, pero ahora se lanzará de forma asíncrona utilizando `self` como delegado.

Ayuda Android

Seguiremos trabajando en la misma clase. Deberemos utilizar la tarea asíncrona

`TareaConexionUrl`, iniciándola desde el evento del botón, en lugar de llamar directamente al método que realiza la conexión.

c) Muestra en la barra de estado un indicador de actividad de red mientras se descarga el HTML. Haz también que el botón se deshabilite mientras dura la descarga. Recuerda que la tarea de descarga podría terminar tanto de forma normal, como por ser cancelada.

Ayuda iOS

Activa o desactiva el indicador de actividad desde los métodos adecuados del delegado.

Ayuda Android

En Android será en la tarea asíncrona donde deberemos mostrar u ocultar el indicador de progreso indeterminado cuando corresponda.

Para probar el proceso de carga en Android podemos simular conexiones lentas, yendo en Eclipse a la perspectiva DDMS, y en la vista *Emulator Control* buscar la sección *Telephony Status* y dentro de ella el campo *Speed*. Aquí podremos seleccionar el tipo de red que queremos simular (para conexiones muy lentas podemos coger GSM). También se puede cambiar conectando vía telnet al emulador:

```
telnet localhost 5554
network delay gprs
OK
network speed gsm
OK
```

2.2. Carga de imágenes

Vamos a implementar una aplicación que nos permita visualizar una lista de *tweets*, y que al pulsar sobre cada uno de ellos nos aparezca la imagen de perfil del usuario que lo envió, junto al mensaje completo. Se proporciona en las plantillas una aplicación `ClientTwitter` que utilizaremos como base. En la pantalla principal veremos un listado de *tweets*, pero nos centraremos en la pantalla de detalles del *tweet*. En esta pantalla recibiremos la URL de la imagen de perfil, pero no la imagen. Tendremos que descargarla de la red de forma asíncrona y mostrarla en pantalla en el `ImageView` destinado a tal fin.

Ayuda iOS

Trabajaremos con la clase `UADetallesTweetViewController`, en su método `configureView` deberemos inicializar la conexión para obtener la imagen del tweet de forma asíncrona. Una vez obtenida la respuesta, en el método `connectionDidFinishLoading`: creamos la imagen a partir de los datos descargados y la mostramos en pantalla.

Ayuda Android

Trabajaremos con la clase `DatosTweetActivity`. Implementa en ella una `AsyncTask` para

cargar las imágenes. En el método `getImagen` construye un `Bitmap` a partir del contenido devuelto. En la `AsyncTask`, una vez obtenida la imagen muéstrala en el `ImageView` disponible. En `onCreate` lanza la tarea asíncrona para conectar a la URL de la imagen.

2.3. Carga lazy de imágenes (*)

Sobre el proyecto del ejercicio anterior, vamos ahora a cargar las imágenes conforme visualizamos el listado de *tweets*, y no sólo al entrar en los detalles de cada uno de ellos. Esta carga de imágenes la vamos a implementar de forma *lazy*, es decir, deberemos cargarlas conforme se solicita que se muestren las celdas en pantalla, no antes. La carga deberá realizarse en segundo plano.

Nota

Una vez se ha descargado una imagen, podemos guardarla en el campo `image` del objeto `Tweet` correspondiente. De esta forma la próxima vez que mostremos el mismo *tweet* no será necesario volverla a descargar.

Tenemos una implementación parcial de la descarga *lazy* de imágenes, sólo tendremos que introducir el código para lanzar la descarga de cada imagen. Ya damos implementados los componentes para descargar las imágenes en segundo plano, sólo tendremos que inicializarlos cada vez que se visualiza una imagen.

Ayuda iOS

La descarga *lazy* en iOS deberá inicializarse en la clase `UAListadoTweetsViewController`. En `cargarImagen:paraIndexPath:` debemos inicializar la descarga de la imagen de una fila utilizando la clase `UIImageDownloader` que ya se proporciona. Actuaremos como delegados de dicha clase, de forma que al finalizar la descarga llamará a nuestro método `imageDownloader:didFinishDownloadingImage:forIndexPath:`, en el que recibiremos la imagen descargada. Deberemos introducir el código necesario en dicho método para mostrar la imagen en la tabla.

Ayuda Android

La descarga *lazy* en Android se implementa en el adaptador `TweetAdapter`. En su método `cargarImagen` deberemos lanzar la tarea para cargar las imágenes de forma asíncrona, que ya se proporciona implementada. Así se iniciará la descarga conforme se las imágenes se muestran en pantalla. Añade en ese método las imágenes que se están descargando al mapa `imagenesCargando`, para no descargar dos veces el mismo *tweet*.

De forma opcional, como optimización puedes implementar también las siguientes funcionalidades:

- No descargar imágenes mientras se hace *scroll*.
- Permitir que las imágenes se eliminen en condiciones de memoria baja.

- Guardar las imágenes de forma persistente en el directorio de caché (se puede utilizar la URL como nombre de los ficheros).

3. Acceso a servicios REST

Cuando las aplicaciones móviles se conectan a un servidor web, normalmente no buscan obtener un documento web (como en el caso de los navegadores, que normalmente solicitarán documentos HTML), sino que lo que hacen es acceder a servicios. Estos servicios nos pueden servir para realizar alguna operación en el servidor, o para obtener información en un formato que nuestra aplicación sea capaz de entender y de procesar. Un servicio consiste en una interfaz que nos da acceso a ciertas funcionalidades. Al realizarse el acceso a estos servicios mediante protocolos web, como HTTP, hablamos de servicios web.

Existen diferentes tipos de servicios web. Uno de los principales tipos de servicios web son los servicios SOAP. Se trata de un estándar XML que nos permite crear servicios con un alto grado de operabilidad, dado que la forma de consumirlos será idéntica, independientemente de la plataforma en la que estén implementados. Además, al definirse de forma estándar existen herramientas que nos permiten integrarlos casi de forma automática en cualquier lenguaje y plataforma. Sin embargo, tienen el inconveniente de que para conseguir dicha interoperabilidad y facilidad de integración necesitan definir un XML demasiado pesado y rígido, lo cual resulta poco adecuado para dispositivos móviles y para la web en general. Estos servicios se utilizan comúnmente para integración de aplicaciones, y en el ámbito de grandes aplicaciones transaccionales.

Para la web y dispositivos móviles se ha impuesto otro estilo para crear servicios web. Se trata del estilo REST, que acerca los servicios web a la web. Se trata de servicios fuertemente vinculados a los protocolos web (HTTP) sobre los que se invocan, en los que tendremos una gran flexibilidad a la hora de elegir el formato con el que queremos intercambiar la información. Vamos a centrarnos en estudiar este estilo REST.

3.1. Fundamentos de REST

El estilo REST (*Representational State Transfer*) es una forma ligera de crear Servicios Web. El elemento principal en el que se basan estos servicios son las URLs. En líneas generales podemos decir que estos servicios consisten en URLs a las que podemos acceder, por ejemplo mediante protocolo HTTP, para obtener información o realizar alguna operación. El formato de la información que se intercambie con estas URLs lo decidirá el desarrollador del servicio. Este tipo de servicios acercan los Servicios Web al tipo de arquitectura de la *web*, siendo especialmente interesantes para su utilización en AJAX.

El término REST proviene de la tesis doctoral de Roy Fielding, publicada en el año 2000, y significa ***REpresentational State Transfer***. REST es un conjunto de restricciones que, cuando son aplicadas al diseño de un sistema, crean un estilo arquitectónico de software. Dicho estilo arquitectónico se caracteriza por:

- Debe ser un sistema cliente-servidor
- Tiene que ser sin estado, es decir, no hay necesidad de que los servicios guarden las sesiones de los usuarios (cada petición al servicio tiene que ser independiente de las demás)
- Debe soportar un sistema de *cachés*: la infraestructura de la red debería soportar *caché* en diferentes niveles
- Debe ser un sistema uniformemente accesible (con una interfaz uniforme): cada recurso debe tener una única dirección y un punto válido de acceso. Los recursos se identifican con URIs, lo cual proporciona un espacio de direccionamiento global para el descubrimiento del servicio y de los recursos.
- Tiene que ser un sistema por capas: por lo tanto debe soportar escalabilidad
- Debe utilizar mensajes auto-descriptivos: los recursos se desacoplan de su representación de forma que puedan ser accedidos en una variedad de formatos, como por ejemplo XML, HTML, texto plano, PDF, JPEG, JSON, etc.

Estas restricciones no dictan qué tipo de tecnología utilizar; solamente definen cómo se transfieren los datos entre componentes y qué beneficios se obtienen siguiendo estas restricciones. Por lo tanto, un sistema RESTful puede implementarse en cualquier arquitectura de la red disponible. Y lo que es más importante, no es necesario "inventar" nuevas tecnologías o protocolos de red: podemos utilizar las infraestructuras de red existentes, tales como la Web, para crear arquitecturas RESTful.

Antes de que las restricciones REST fuesen formalizadas, ya disponíamos de un ejemplo de un sistema RESTful: la Web (estática). Por ejemplo, la infraestructura de red existente proporciona sistemas de *caché*, conexión sin estado, y enlaces únicos a los recursos, en donde los recursos son todos los documentos disponibles en cada sitio web y las representaciones de dichos recursos son conjuntos de ficheros "legibles" por navegadores web (por ejemplo, ficheros HTML). Por lo tanto, la web estática es un sistema construido sobre un estilo arquitectónico REST.

A continuación analizaremos las abstracciones que constituyen un sistema RESTful: recursos, representaciones, URIs, y los tipos de peticiones HTTP que constituyen la interfaz uniforme utilizada en las transferencias cliente/servidor

3.1.1. Recursos

Un recurso REST es cualquier cosa que sea direccionable a través de la Web. Por direccionable nos referiremos a recursos que puedan ser accedidos y transferidos entre clientes y servidores. Por lo tanto, un recurso es una correspondencia lógica y temporal con un concepto en el dominio del problema para el cual estamos implementando una solución.

Algunos ejemplos de recursos REST son:

- Una noticia de un periódico
- La temperatura de Alicante a las 4:00pm

- Un valor de IVA almacenado en una base de datos
- Una lista con el historial de las revisiones de código en un sistema CVS
- Un estudiante en alguna aula de alguna universidad
- El resultado de una búsqueda de un ítem particular en Google

Aun cuando el mapeado de un recurso es único, diferentes peticiones a un recurso pueden devolver la misma representación binaria almacenada en el servidor. Por ejemplo, consideremos un recurso en el contexto de un sistema de publicaciones. En este caso, una petición de la "última revisión publicada" y la petición de "la revisión número 12" en algún momento de tiempo pueden devolver la misma representación del recurso: cuando la última revisión sea efectivamente la 12. Por lo tanto, cuando la última revisión publicada se incremente a la versión 13, una petición a la última revisión devolverá la versión 13, y una petición de la revisión 12, continuará devolviendo la versión 12. En definitiva: cada uno de los recursos puede ser accedido directamente y de forma independiente, pero diferentes peticiones podrían "apuntar" al mismo dato.

Debido a que estamos utilizando HTTP para comunicarnos, podemos transferir cualquier tipo de información que pueda transportarse entre clientes y servidores. Por ejemplo, si realizamos una petición de un fichero de texto de la CNN, nuestro navegador mostrará un fichero de texto. Si solicitamos una película flash a YouTube, nuestro navegador recibirá una película flash. En ambos casos, los datos son transferidos sobre TCP/IP y el navegador conoce cómo interpretar los *streams* binarios debido a la cabecera de respuesta del protocolo HTTP *Content-Type*. Por lo tanto, en un sistema RESTful, la representación de un recurso depende del tipo deseado por el cliente (tipo MIME), el cual está especificado en la petición del protocolo de comunicaciones.

3.1.2. Representación

La representación de los recursos es lo que se envía entre los servidores y clientes. Una representación muestra el estado del dato real almacenado en algún dispositivo de almacenamiento en el momento de la petición. En términos generales, es un *stream* binario, juntamente con los metadatos que describen cómo dicho *stream* debe ser consumido por el cliente y/o servidor (los metadatos también pueden contener información extra sobre el recurso, como por ejemplo información de validación y encriptación, o código extra para ser ejecutado dinámicamente).

A través del ciclo de vida de un servicio web, pueden haber varios clientes solicitando recursos. Clientes diferentes son capaces de consumir diferentes representaciones del mismo recurso. Por lo tanto, una representación puede tener varias formas, como por ejemplo, una imagen, un texto, un fichero XML, o un fichero JSON, pero tienen que estar disponibles en la misma URL.

Para respuestas generadas para humanos a través de un navegador, una representación típica tiene la forma de página HTML. Para respuestas automáticas de otros servicios web, la legibilidad no es importante y puede utilizarse una representación mucho más

eficiente como por ejemplo XML.

El lenguaje para el intercambio de información con el servicio queda a elección del desarrollador. A continuación mostramos algunos formatos comunes que podemos utilizar para intercambiar esta información:

Formato	Tipo MIME
Texto plano	text/plain
HTML	text/html
XML	application/xml
JSON	application/json

De especial interés es el formato JSON. Se trata de un lenguaje ligero de intercambio de información, que puede utilizarse en lugar de XML (que resulta considerablemente más pesado) para aplicaciones AJAX. De hecho, en Javascript puede leerse este tipo de formato simplemente utilizando el método `eval()`.

3.1.3. URI

Una URI, o **Uniform Resource Identifier**, en un servicio web RESTful es un hiper-enlace a un recurso, y es la única forma de intercambiar representaciones entre clientes y servidores. Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes.

El conjunto de restricciones REST no impone que las URIs deban ser hiper-enlaces. Simplemente hablamos de hiper-enlaces porque estamos utilizando la Web para crear servicios web. Si estuviésemos utilizando un conjunto diferente de tecnologías soportadas, una URI RESTful podría ser algo completamente diferente. Sin embargo, la idea de direccionabilidad debe permanecer.

En un sistema REST, la URI no cambia a lo largo del tiempo, ya que la implementación de la arquitectura es la que gestiona los servicios, localiza los recursos, negocia las representaciones, y envía respuestas con los recursos solicitados. Y lo que es más importante, si hubiese un cambio en la estructura del dispositivo de almacenamiento en el lado del servidor (por ejemplo, un cambio de servidores de bases de datos), nuestras URIs seguirán siendo las mismas y serán válidas mientras el servicio web siga estando "en marcha" o el contexto del recurso no cambie.

Sin las restricciones REST, los recursos se acceden por su localización: las direcciones web típicas son URIs fijas. Si por ejemplo renombramos un fichero en el servidor, la URI será diferente; si movemos el fichero a un directorio diferente, la URI también será diferente.

Por ejemplo, si en nuestra aplicación tenemos información de cursos, podríamos acceder

a la lista de cursos disponibles mediante una URL como la siguiente:

```
http://jtech.ua.es/resources/cursos
```

Esto nos devolverá la lista de cursos en el formato que el desarrollador del servicio haya decidido. Hay que destacar por lo tanto que en este caso debe haber un entendimiento entre el consumidor y el productor del servicio, de forma que el primero comprenda el lenguaje utilizado por el segundo.

Esta URL nos podría devolver un documento como el siguiente:

```
<?xml version="1.0"?>
<j:Cursos xmlns:j="http://www.jtech.ua.es"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <Curso id="1"
    xlink:href="http://jtech.ua.es/resources/cursos/1"/>
  <Curso id="2"
    xlink:href="http://jtech.ua.es/resources/cursos/2"/>
  <Curso id="4"
    xlink:href="http://jtech.ua.es/resources/cursos/4"/>
  <Curso id="6"
    xlink:href="http://jtech.ua.es/resources/cursos/6"/>
</j:Cursos>
```

En este documento se muestra la lista de cursos registrados en la aplicación, cada uno de ellos representado también por una URL. Accediendo a estas URLs podremos obtener información sobre cada curso concreto o bien modificarlo.

3.1.4. Uniformidad de las interfaces a través de peticiones HTTP

Ya hemos introducido los conceptos de recursos y sus representaciones. Hemos dicho que los recursos son *mappings* de los estados reales de las entidades que son intercambiados entre los clientes y servidores. También hemos dicho que las representaciones son negociadas entre los clientes y servidores a través del protocolo de comunicación en tiempo de ejecución (a través de HTTP). A continuación veremos con detalle lo que significa el intercambio de estas representaciones, y lo que implica para los clientes y servidores el realizar acciones sobre dichos recursos.

El desarrollo de servicios web REST es similar al desarrollo de aplicaciones web. Sin embargo, la diferencia fundamental entre el desarrollo de aplicaciones web tradicionales y las más modernas es cómo pensamos sobre las acciones a realizar sobre nuestras abstracciones de datos. De forma más concreta, el desarrollo moderno está centrado en el concepto de **nombres** (intercambio de recursos); el desarrollo tradicional está centrado en el concepto de verbos (acciones remotas a realizar sobre los datos). Con la primera forma, estamos implementando un servicio web RESTful; con la segunda un servicio similar a una llamada a procedimiento remoto- RPC). Y lo que es más, un servicio RESTful modifica el estado de los datos a través de la representación de los recursos (por el contrario, una llamada a un servicio RPC, oculta la representación de los datos y en su lugar envía comandos para modificar el estado de los datos en el lado del servidor). Finalmente, en el desarrollo moderno de aplicaciones web limitamos la ambigüedad en el

diseño y la implementación debido a que tenemos cuatro acciones específicas que podemos realizar sobre los recursos: *Create, Retrieve, Update, Delete (CRUD)*. Por otro lado, en el desarrollo tradicional de aplicaciones web, podemos tener otras acciones con nombres o implementaciones no estándar.

A continuación mostramos la correspondencia entre las acciones CRUD sobre los datos y los métodos HTTP correspondientes:

Acción sobre los datos	Protocolo HTTP equivalente
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

En su forma más simple, los servicios web RESTful son aplicaciones cliente-servidor a través de la red que manipulan el estado de los recursos. En este contexto, la manipulación de los recursos significa creación de recursos, recuperación, modificación y borrado. Sin embargo, los servicios web RESTful no están limitados solamente a estos cuatro conceptos básicos de manipulación de datos. Por el contrario, los servicios RESTful pueden ejecutar lógica en el lado del servidor, pero recordando que cada respuesta debe ser una representación del recurso del dominio en cuestión. Debemos determinar qué operación HTTP se ajusta mejor a la manipulación que deseamos realizar sobre los datos. Mención especial merece el método PUT, ya que no se trata simplemente de una actualización de los datos, sino de establecer el estado del recurso, exista previamente o no. A continuación trataremos cada uno de estos métodos con más detalle.

Nota

Una interfaz uniforme centra la atención en los conceptos abstractos que hemos visto: recursos, representaciones y URIs. Por lo tanto, si consideramos todos estos conceptos en su conjunto, podemos describir el desarrollo RESTful en una frase: utilizamos URIs para conectar clientes y servidores para intercambiar recursos en forma de sus representaciones. O también: en una arquitectura con estilo REST, los clientes y servidores intercambian representaciones de los recursos utilizando un protocolo e interfaces estandarizados.

3.2. Tipos de peticiones HTTP

A continuación vamos a ver los cuatro tipos de peticiones HTTP con detalle, y veremos cómo se utiliza cada una de ellas para intercambiar representaciones para modificar el estado de los recursos.

3.2.1. GET/RETRIEVE

El método GET se utiliza para **RECUPERAR** recursos. Antes de indicar la mecánica de

la petición GET, vamos a determinar cuál es el recurso que vamos a manejar y el tipo de representación que vamos a utilizar. Para ello vamos a seguir un ejemplo de un servicio web que gestiona alumnos en una clase, con la URI: <http://restfuljava.com>. Para dicho servicio, asumiremos una representación como la siguiente:

```
<alumno>
  <nombre>Esther</nombre>
  <edad>10</edad>
  <link>/alumnos/Jane</link>
</alumno>
```

Una lista de alumnos tendrá el siguiente aspecto:

```
<alumnos>
  <alumno>
    <nombre>Esther</nombre>
    <edad>10</edad>
    <link>/alumnos/Esther</link>
  <alumno>
    <nombre>Pedro</nombre>
    <edad>11</edad>
    <link>/alumnos/Pedro</link>
  <alumno>
  </alumnos>
```

Una vez definida nuestra representación, asumimos que las URIs tienen la forma: <http://restfuljava.com/alumnos> para acceder a la lista de alumnos, y <http://restfuljava.com/alumnos/{nombre}> para acceder a un alumno específico con el identificador con el valor *nombre*.

Ahora hagamos peticiones sobre nuestro servicio. Por ejemplo, si queremos recuperar la información de una alumna con el nombre *Esther*, realizamos una petición a la URI: <http://restfuljava.com/alumnos/Esther>.

Una representación de *Esther* en el momento de la petición, puede ser ésta:

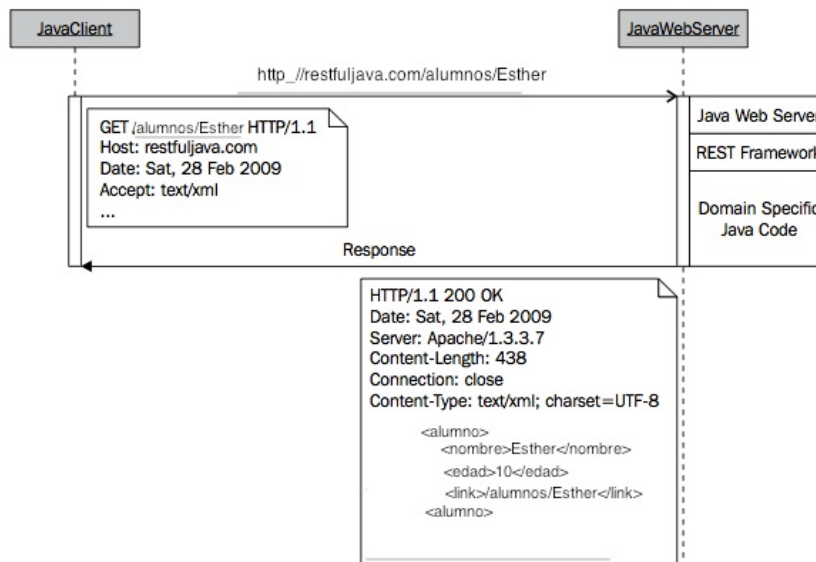
```
<alumno>
  <nombre>Esther</nombre>
  <edad>10</edad>
  <link>/alumnos/Esther</link>
</alumno>
```

También podríamos acceder a una lista de estudiantes a través de la URI: y <http://restfuljava.com/alumnos> y la respuesta del servicio sería algo similar a ésta (asumiendo que solamente hay dos alumnos):

```
<alumnos>
  <alumno>
    <nombre>Esther</nombre>
    <edad>10</edad>
    <link>/alumnos/Esther</link>
  <alumno>
    <nombre>Pedro</nombre>
    <edad>11</edad>
    <link>/alumnos/Pedro</link>
  <alumno>
</alumnos>
```

```
</alumnos>
```

Echemos un vistazo a los detalles de la petición. Una petición para recuperar un recurso *Esther* usa el método GET con la URI: `http://restfuljava.com/alumnos/Esther`. Un diagrama de secuencia de dicha petición sería como el que mostramos a continuación:



Escenario de petición GET/RETRIEVE

¿Qué está ocurriendo aquí?:

1. Un cliente Java realiza una petición HTTP con el método GET y Esther es el identificador del alumno
2. El cliente establece la representación solicitada a través del campo de cabecera `Accept`
3. El servidor web recibe e interpreta la petición GET como una acción RETRIEVE. En este momento, el servidor web cede el control al *framework* RESTful para gestionar la petición. Remarquemos que los *frameworks* RESTful no recuperan de forma automática los recursos, ése no es su trabajo. La función del *framework* es facilitar la implementación de las restricciones REST. La lógica de negocio y la implementación del almacenamiento es el papel del código Java específico del dominio.
4. El programa del lado del servidor busca el recurso *Esther*. Encontrar el recurso podría significar buscarlo en una base de datos, un sistema de ficheros, o una llamada a otro servicio web.
5. Una vez que el programa encuentra a *Esther*, convierte el dato binario del recurso a la representación solicitada por el cliente.
6. Con la representación convertida a XML, el servidor envía de vuelta una respuesta HTTP con un código numérico de 200 (OK) junto con la representación solicitada. Si hay algún error, el servidor HTTP devuelve el código numérico correspondiente, pero es el cliente el que debe tratar de forma adecuada el fallo. El fallo más común es que

el recurso no exista, en cuyo caso se devolvería el código 404 (Not Found).

Todos los mensajes entre el cliente y el servidor son llamadas del protocolo estándar HTTP. Para cada acción de recuperación, enviamos una petición GET y obtenemos una respuesta HTTP con la representación del recurso solicitada, o bien, si hay un fallo, el correspondiente código de error (por ejemplo, 404 Not Found si un recurso no se encuentra; 500 Internal Server Error si hay un problema con el código Java en forma de una excepción).

En las peticiones de recuperación de datos resulta recomendable también implementar un sistema de caché. Para hacer esto utilizaremos el código de respuesta 304 Not Modified en caso de que los datos no hubiesen cambiado desde la última petición que realizamos (se podría pasar un parámetro con la fecha en la que obtuvimos la representación por última vez). De esta forma, si un cliente recibe ese código como respuesta, sabe que puede seguir trabajando con la representación de la que ya dispone, sin tener que descargar una nueva.

Solicitar una representación para todos los alumnos funciona de forma similar.

Nota

El método HTTP GET solamente debería utilizarse para recuperar representaciones. Podríamos utilizar una petición GET para actualizar el estado de los datos en el servidor, pero no es recomendable. Una operación GET debe ser segura e idempotente (para más información ver <http://www.w3.org/DesingIssues/Axioms>). Para que una petición sea **segura**, múltiples peticiones al mismo recurso no deben cambiar el estado de los datos en el servidor. Por ejemplo, supongamos una petición en el instante t1 para un recurso R devuelve R1; a continuación, una petición en el instante t2 para R devuelve R2; suponiendo que no hay más acciones de modificación entre t1 y t2, entonces R1 = R2 = R. Para que una petición sea **idempotente** tiene que ocurrir que múltiples llamadas a la misma acción dejan siempre el mismo estado en el recurso. Por ejemplo, múltiples llamadas para crear un recurso R en los instantes t1, t2, y t3, darían como resultado que el recurso R existe sólo como R, y que las llamadas en los instantes t2 y t3 son ignoradas.

3.2.2. POST/CREATE

El método POST se utiliza para **CREAR** recursos. Vamos a utilizar el método HTTP POST para crear un nuevo alumno. De nuevo, la URI para añadir un nuevo alumno a nuestra lista será: <http://restfuljava.com/alumnos>. El tipo de método para la petición lo determina el cliente.

Asumamos que el alumno con nombre *Ricardo* no existe en nuestra lista y queremos añadirlo. Nuestra nueva representación XML de *Ricardo* es:

```
<alumno>
  <nombre>Ricardo</nombre>
  <edad>10</edad>
  <link></link>
</alumno>
```

El elemento *link* forma parte de la representación, pero está vacía debido a que éste valor se genera en tiempo de ejecución y no es creado por el cliente cuando envía la petición POST. Esto es solamente una convención para nuestro ejemplo; sin embargo, los clientes que utilizan el servicio web pueden especificar la estructura de las URIs.

En este caso, no mostraremos el escenario, pero los pasos que se siguen cuando se realiza la petición son los siguientes:

1. Un cliente Java realiza una petición HTTP a la URI *http://restfuljava.com/alumnos*, con el método HTTP POST
2. La petición POST incluye una representación en forma de XML de *Ricardo*
3. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para almacenar dicha representación (de nuevo, el dispositivo de almacenamiento puede ser cualquiera).
4. Una vez que se ha completado el almacenamiento del nuevo recurso, se envía una respuesta de vuelta: si no ha habido ningún error, enviaremos el código 201 (Created); si se produce un fallo, enviaremos el código de error adecuado. Además, podemos devolver en la cabecera *Location* la URL que nos dará acceso al recurso recién creado.

```
Location: http://restfuljava.com/alumnos/Ricardo
```

Las peticiones POST no son idempotentes, por lo que si invocamos una misma llamada varias veces sobre un mismo recurso, el estado del recurso puede verse alterado en cada una de ellas. Por ejemplo, si ejecutamos varias veces la acción POST con los datos del ejemplo anterior, podríamos estar creando cada vez un nuevo alumno de nombre Ricardo, teniendo así varios alumnos con el mismo nombre y edad (pero asociados a IDs distintos, por ejemplo: /Ricardo, /Ricardo1, /Ricardo2, etc).

3.2.3. PUT/UPDATE

El método PUT se utiliza para **ACTUALIZAR** (modificar) recursos, o para crearlos si el recurso en la URI especificada no existiese previamente. Es decir, PUT se utiliza para establecer un determinado recurso, dada su URI, a la representación que proporcionemos, independientemente de que existiese o no. Para actualizar un recurso, primero necesitamos su representación en el cliente; en segundo lugar, en el lado del cliente actualizaremos el recurso con los nuevos valores deseados; y finalmente, actualizaremos el recurso mediante una petición PUT, adjuntando la representación correspondiente.

Para nuestro ejemplo, omitiremos la petición GET para recuperar a *Esther* del servicio web, ya que es el mismo que acabamos de indicar en la sección anterior. Supongamos que queremos modificar la edad, y cambiarla de 10 a 12. La nueva representación será:

```
<alumno>
  <nombre>Esther</nombre>
  <edad>12</edad>
  <link>/alumnos/Esther</link>
```

```
</alumno>
```

La secuencia de pasos necesarios para enviar/procesar la petición PUT es:

1. Un cliente Java realiza una petición HTTP PUT a la URI `http://restfuljava.com/alumnos/Esther`, incluyendo la nueva definición XML
2. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para actualizar la representación de *Esther*.
3. Una vez que se ha completado la actualización, se envía una respuesta al cliente. Si el recurso que hemos enviado no existía previamente, se devolverá el código 201 (Created). En caso de que ya existiese, se podría devolver 200 (OK) con el recurso actualizado como contenido, o simplemente 204 (No Content) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Muchas veces se confunden los métodos PUT y POST. El significado de estos métodos es el siguiente:

- **POST:** Publica datos en un determinado recurso. El recurso debe existir previamente, y los datos enviados son añadidos a él. Por ejemplo, para añadir nuevos alumnos con POST hemos visto que debíamos hacerlo con el recurso lista de alumnos (`/alumnos`), ya que la URI del nuevo alumno todavía no existe. La operación **no es idempotente**, es decir, si añadimos varias veces el mismo alumno aparecerá repetido en nuestra lista de alumnos con URIs distintas.
- **PUT:** Hace que el recurso indicado tome como contenido los datos enviados. El recurso podría no existir previamente, y en caso de que existiese sería sobrescrito con la nueva información. A diferencia de POST, PUT **es idempotente**. Múltiples llamadas idénticas a la misma acción PUT siempre dejarán el recurso en el mismo estado. La acción se realiza sobre la URI concreta que queremos establecer (por ejemplo, `/alumnos/Esther`), de forma que varias llamadas consecutivas con los mismos datos tendrán el mismo efecto que realizar sólo una de ellas.

Podríamos añadir nuevos alumnos de dos formas diferentes. La primera de ellas es haciendo POST sobre el recurso que contiene la lista de alumnos:

```
POST /alumnos HTTP/1.1
```

También podríamos hacer PUT sobre el recurso de un alumno concreto:

```
PUT /alumnos/Esther HTTP/1.1
```

Si *Esther* existía ya, sobrescribirá sus datos, en caso contrario, creará el nuevo recurso.

Si utilizamos POST de esta última forma, sobre un recurso concreto, si el recurso existiese podríamos realizar alguna operación que modifique sus datos, pero si no existiese nos daría un error, ya que no podemos hacer POST sobre un recurso inexistente.

```
POST /alumnos/Esther HTTP/1.1
```

El caso anterior sólo será correcto si *Esther* existe, en caso contrario obtendremos un

error. Para crear nuevos recursos con POST debemos recurrir al recurso del conjunto de alumnos. Una diferencia entre estas dos formas alternativas de crear nuevos recursos es que con PUT podemos indicar explícitamente el identificador del recurso creado, mientras que con POST será el servidor quien lo decida.

3.2.4. DELETE/DELETE

El método DELETE se utiliza para **BORRAR** representaciones. Para nuestro ejemplo, usaremos la misma URI de las secciones anteriores.

La secuencia de pasos necesarios para enviar/procesar la petición DELETE es:

1. Un cliente Java realiza una petición DELETE a la URI `http://restfuljava.com/alumnos/Esther`
2. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para borrar la representación de *Esther*.
3. Una vez que se ha completado la actualización, se envía una respuesta al cliente. Se podría devolver 200 (OK) con el recurso borrado como contenido, o simplemente 204 (No Content) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Hasta aquí hemos visto las principales acciones que podemos realizar con los recursos en un servicio web RESTful. No conocemos cómo el servicio web implementa el almacenamiento de los datos, y no conocemos qué tecnologías se utilizan para implementar el servicio. Todo lo que conocemos es que nuestro cliente y servidor se comunican a través de HTTP, que usamos dicho protocolo de comunicaciones para enviar peticiones, y que nuestras representaciones de los recursos se intercambian entre el cliente y el servidor a través del intercambio de URIs.

3.3. Clientes de servicios REST

3.3.1. Invocación de servicios RESTful desde una clase Java

Vamos a ver como crear un cliente RESTful utilizando una sencilla clase Java, lo cual es aplicable a Android. Para ello vamos a utilizar el API de mensajes proporcionado por Twitter (<http://www.twitter.com>). No va a ser necesario disponer de una cuenta de Twitter ni conocer con detalle qué es Twitter para seguir el ejemplo.

Nota

Twitter es una plataforma de *micro-blogging* que permite que múltiples usuarios actualicen su estado utilizando 140 caracteres como máximo cada vez. Además, los usuarios pueden "seguirse" unos a otros formando redes de "amigos". Twitter almacena estas actualizaciones en sus servidores, y por defecto, están disponibles públicamente. Es por ésto por lo que utilizaremos Twitter para crear nuestros ejemplos de clientes REST.

```

try {
    URL twitter = new
        URL("http://twitter.com/statuses/public_timeline.xml");

    // Abrimos la conexión
    URLConnection tc = twitter.openConnection();

    // Obtenemos la respuesta del servidor
    BufferedReader in = new BufferedReader(new
        InputStreamReader( tc.getInputStream()));
    String line;

    // Leemos la respuesta del servidor y la imprimimos
    while ((line = in.readLine()) != null) {
        tvResult.append(line);
    }
    in.close();
} catch (MalformedURLException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}

```

Podemos ver que hemos utilizado el paquete estándar `java.net`. La URI del servicio web es: `http://twitter.com/statuses/public_timeline.xml`. Ésta será la URI de nuestro recurso y apuntará a las últimas 20 actualizaciones públicas.

Para conectar con el servicio web, primero tenemos que instanciar el objeto `URL` con la URI del servicio. A continuación, "abriremos" un objeto `URLConnection` para la instancia de Twitter. La llamada al método `twitter.openConnection()` ejecuta una petición HTTP GET.

Una vez que tenemos establecida la conexión, el servidor devuelve la respuesta HTTP. Dicha respuesta contiene una representación XML de las actualizaciones. Por simplicidad, volcaremos en la salida estándar la respuesta del servidor. Para ello, primero leemos el *stream* de respuesta en un objeto `BufferedReader`, y a continuación realizamos un bucle para cada línea del *stream*, asignándola a un objeto `String`. Finalmente hemos incluido nuestro código en una sentencia *try/catch*, y enviamos cualquier mensaje de excepción a la salida estándar.

Éste es el estado público de la última actualización de Twitter de la estructura XML obtenida (sólo mostramos parte de uno de los *tweets*).

```

<?xml version="1.0" encoding="UTF-8"?> <statuses type="array">
...
<status>
  <created_at>Tue Feb 22 11:43:25 +0000 2011</created_at>
  <id>40013788233216000</id>
  <text>Haar doen voor de cam. #ahahah</text>
  <source>web</source>
  <truncated>>false</truncated>
  <favorited>>false</favorited>
  <in_reply_to_status_id></in_reply_to_status_id>
  <in_reply_to_user_id></in_reply_to_user_id>
  <in_reply_to_screen_name></in_reply_to_screen_name>
  <retweet_count>0</retweet_count>
  <retweeted>>false</retweeted>

```

```

<user>
  <id>250090010</id>
  <name>Dani&#235;l van der wal</name>
  <screen_name>DanielvdWall</screen_name>
  <location>Nederland, Hoogezand</location>
  <description></description>
  <profile_image_url>
    http://a0.twimg.com/profile_images/1240171940/Picture0003_normal.JPG
  </profile_image_url>
  <url>http://daniel694.hyves.nl/</url>
  <protected>false</protected>
  <followers_count>50</followers_count>
  ...
  <friends_count>74</friends_count>
  ...
  <following>false</following>
  <statuses_count>288</statuses_count>
  <lang>en</lang>
  <contributors_enabled>false</contributors_enabled>
  <follow_request_sent>false</follow_request_sent>
  <listed_count>0</listed_count>
  <show_all_inline_media>false</show_all_inline_media>
  <is_translator>false</is_translator>
</user>
<geo/>
<coordinates/>
<place/>
<contributors/>
</status>
...
</statuses>

```

No entraremos en los detalles de la estructura XML, podemos encontrar la documentación del API en <http://dev.twitter.com/>

La documentación del API nos dice que si cambiamos la extensión *.xml* obtendremos diferentes representaciones del recurso. Por ejemplo, podemos cambiar *.xml*, por *.json*, *.rss* o *.atom*. Así, por ejemplo, si quisiéramos recibir la respuesta en formato JSON (JavaScript Object Notation), el único cambio que tendríamos que hacer es en la siguiente línea:

```

URL twitter =
  new URL("http://twitter.com/statuses/public_timeline.json");

```

En este caso, obtendríamos algo como esto:

```

[{"in_reply_to_status_id_str":null,"text":"THAT GAME SUCKED ASS.",
  "contributors":null,"retweeted":false,"in_reply_to_user_id_str":
  null,"retweet_count":0,"in_reply_to_user_id":null,"source":"web",
  "created_at":"Tue Feb 22 11:55:17 +0000 2011","place":null,
  "truncated":false,"id_str":"40016776221696000","geo":null,
  "favorited":false,"user":{"listed_count":0,"following":null,
  "favourites_count":0,"url":"http://www.youtube.com/user/
  haezelnut","profile_use_background_image":true,...

```

Los detalles sobre JSON se encuentran en la documentación del API.

Nota

Aunque Twitter referencia este servicio como servicio RESTful, esta API en particular no es completamente RESTful, debido a una elección de diseño. Analizando la documentación del API

vemos que el tipo de representación de la petición forma parte de la URI y no de la cabecera *accept* de HTTP. El API devuelve una representación que solamente depende de la propia URI: *http://twitter.com/statuses/public_timeline.FORMATO*, en donde *FORMATO* puede ser *.xml*, *.json*, *.rss*, o *.atom*. Sin embargo, esta cuestión no cambia la utilidad del API para utilizarlo como ejemplo para nuestro cliente REST.

Otra posibilidad de implementación de nuestro cliente Java para Android es utilizar la librería *Http Client*. Dicha librería ofrece una mayor facilidad para controlar y utilizar objetos de conexión HTTP.

El código de nuestra clase cliente utilizando la librería quedaría así:

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(
    "http://twitter.com/statuses/public_timeline.xml");
try {
    ResponseHandler<String> handler = new BasicResponseHandler();
    String contenido = client.execute(request, handler);

    tvResponse.setText(contenido);
} catch (ClientProtocolException e) {
} catch (IOException e) {
} finally {
    client.getConnectionManager().shutdown();
}
```

Observamos que primero instanciamos el cliente HTTP y procedemos a crear un objeto que representa el método HTTP GET. Con el cliente y el método instanciados, necesitamos ejecutar la petición con el método *execute*. Si queremos tener un mayor control sobre la respuesta, en lugar de utilizar un *BasicResponseHandler* podríamos ejecutar directamente la petición sobre el cliente, y obtener así la respuesta completa, tal como vimos en la sesión anterior.

En este caso, hemos visto cómo realizar una petición GET, tal como se vio en sesiones anteriores, para acceder a servicios REST. Sin embargo, para determinadas operaciones hemos visto que REST utiliza métodos HTTP distintos, como POST, PUT o DELETE. Para cambiar el método simplemente tendremos que cambiar el objeto *HttpGet* por el del método que corresponda. Cada tipo incorporará los métodos necesarios para el tipo de petición HTTP que represente. Por ejemplo, a continuación vemos un ejemplo de petición POST. Creamos un objeto *HttpPost* al que le deberemos pasar una entidad que represente el bloque de contenido a enviar (en una petición POST ya no sólo tenemos un bloque de contenido en la respuesta, sino que también lo tenemos en la petición). Podemos crear diferentes tipos de entidades, que serán clases que hereden de *HttpEntity*. La más habitual para los servicios que estamos utilizando será *StringEntity*, que nos facilitará incluir en la petición contenido XML o JSON como una cadena de texto. Además, deberemos especificar el tipo MIME de la entidad de la petición mediante *setContentType* (en el siguiente ejemplo consideramos que es XML). Por otro lado, también debemos especificar el tipo de representación que queremos obtener como respuesta, y como hemos visto anteriormente, esto debe hacerse mediante

la cabecera Accept. Esta cabecera la deberemos establecer en el objeto que representa la petición POST (HttpPost).

```
HttpClient client = new DefaultHttpClient();

HttpPost post = new HttpPost("http://jtech.ua.es/recursos/peliculas");
StringEntity s = new StringEntity("[contenido xml]");
s.setContentType("application/xml");
post.setEntity(s);
post.setHeader("Accept", "application/xml");

ResponseHandler<String> handler = new BasicResponseHandler();
String respuesta = client.execute(post, handler);
```

Como podemos ver en el ejemplo anterior, una vez configurada la petición POST la forma de ejecutar la petición es la misma que la vista anteriormente para peticiones GET. Para el resto de métodos HTTP el funcionamiento será similar, simplemente cambiando el tipo del objeto de la petición por el que corresponda (por ejemplo HttpPut o HttpDelete).

3.3.2. Acceso a servicios REST desde iOS

En iOS podemos acceder a servicios REST utilizando las clases para conectar con URLs vistas en anteriores sesiones. Por ejemplo, para hacer una consulta al *public timeline* de Twitter podríamos utilizar un código como el siguiente para iniciar la conexión (recordemos que este método de conexión es asíncrono, y los datos los recibirá posteriormente el objeto delegado especificado):

```
NSURL *url = [NSURL URLWithString:
    @"http://twitter.com/statuses/public_timeline.json"];
NSURLRequest *theRequest = [NSURLRequest requestWithURL: url];
NSURLConnection *theConnection =
    [NSURLConnection connectionWithRequest: theRequest delegate: self];
```

En el caso de Objective-C, si queremos realizar peticiones con métodos distintos a GET, deberemos utilizar la clase NSMutableURLRequest, en lugar de NSURLRequest, ya que esta última no nos permite modificar los datos de la petición, incluyendo el método HTTP. Podremos de esta forma establecer todos los datos necesarios para la petición al servicio: método HTTP, mensaje a enviar (por ejemplo XML o JSON), y cabeceras (para indicar el tipo de contenido enviado, o los tipos de representaciones que aceptamos). Por ejemplo:

```
NSURL *url =
    [NSURL URLWithString:@"http://jtech.ua.es/resources/peliculas"];
NSData *datosPelicula = ... // Componer mensaje XML con datos de la
    // película a crear

NSMutableURLRequest *theRequest =
    [NSMutableURLRequest requestWithURL: url];
[theRequest setHTTPMethod: @"POST"];
[theRequest setHTTPBody: datosPelicula];
[theRequest setValue:@"application/xml" forHTTPHeaderField:@"Accept"];
[theRequest setValue:@"application/xml"
    forHTTPHeaderField:@"Content-Type"];
```

```

NSURLConnection *theConnection =
    [NSURLConnection connectionWithRequest: theRequest
                                     delegate: self];

```

Podemos ver que en la petición POST hemos establecido todos los datos necesarios. Por un lado su bloque de contenido, con los datos del recurso que queremos añadir en la representación que consideremos adecuada. En este caso suponemos que utilizamos XML como representación. En tal caso hay que avisar de que el contenido lo enviamos con este formato, mediante la cabecera `Content-Type`, y de que la respuesta también queremos obtenerla en XML, mediante la cabecera `Accept`.

3.4. Parsing de estructuras XML

En las comunicaciones por red es muy común transmitir información en formato XML, el ejemplo más conocido, después del HTML, son las noticias RSS. En este último caso, al delimitar cada campo de la noticia por tags de XML se permite a los diferentes clientes lectores de RSS obtener sólo aquellos campos que les interese mostrar.

3.4.1. Parsing de XML en Android

Android nos ofrece dos maneras de trocear o "parsear" XML. El `SAXParser` y el `XmlPullParser`. El parser SAX requiere la implementación de manejadores que reaccionan a eventos tales como encontrar la apertura o cierre de una etiqueta, o encontrar atributos. Menos implementación requiere el uso del parser Pull que consiste en iterar sobre el árbol de XML (sin tenerlo completo en memoria) conforme el código lo va requiriendo, indicándole al parser que tome la siguiente etiqueta (método `next()`) o texto (método `nextText()`).

A continuación mostramos un ejemplo sencillo de uso del `XmlPullParser`. Préstese atención a las sentencias y constantes resaltadas, para observar cómo se identifican los distintos tipos de etiqueta, y si son de apertura o cierre. También se puede ver cómo encontrar atributos y cómo obtener su valor.

```

try {
    URL text = new URL("http://www.ua.es");

    XmlPullParserFactory parserCreator = XmlPullParserFactory.newInstance();
    XmlPullParser parser = parserCreator.newPullParser();
    parser.setInput(text.openStream(), null);
    int parserEvent = parser.getEventType();
    while (parserEvent != XmlPullParser.END_DOCUMENT) {

        switch (parserEvent) {
            case XmlPullParser.START_DOCUMENT:
                break;
            case XmlPullParser.END_DOCUMENT:
                break;
            case XmlPullParser.START_TAG:
                String tag = parser.getName();
                if (tag.equalsIgnoreCase("title")) {
                    Log.i("XML", "El título es: " + parser.nextText());
                }
            }
        }
    }
}

```

```

    } else if(tag.equalsIgnoreCase("meta")) {
        String name = parser.getAttributeValue(null, "name");
        if(name.equalsIgnoreCase("description")) {
            Log.i("XML", "La descripción es:" +
                parser.getAttributeValue(null, "content"));
        }
    }
    break;
case XmlPullParser.END_TAG:
    break;
}

parserEvent = parser.next();
} catch (Exception e) {
    Log.e("Net", "Error en la conexión de red", e);
}

```

El ejemplo anterior serviría para imprimir en el LogCat el título del siguiente fragmento de página web, que en este caso sería "Universidad de Alicante", y para encontrar el meta cuyo atributo name sea "Description" y mostrar el valor de su atributo content:

```

<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<head>
<title>Universidad de Alicante<strong>&lt;/title>
<meta name="Description" content="Informacion Universidad Alicante.
    Estudios, masteres, diplomaturas, ingenierias, facultades, escuelas"/>
<![CDATA[<meta http-equiv="pragma" content="no-cache" />
<meta name="Author" content="Universidad de Alicante" />
<meta name="Copyright" content="&copy; Universidad de Alicante" />
<meta name="robots" content="index, follow" />

```

3.4.2. Parsing de XML en iOS

Para análisis de XML en iOS contamos en el SDK con `NSXMLParser`. Con esta librería el análisis se realiza de forma parecida a los parsers SAX de Java. Este es el parser principal incluido en el SDK y está escrito en Objective-C, aunque también contamos dentro del SDK con `libxml2`, escrito en C, que incluye tanto un parser SAX como DOM. También encontramos otras librerías que podemos incluir en nuestro proyecto como parsers DOM XML:

Parser	URL
TBXML	http://www.tbxml.co.uk/
TouchXML	https://github.com/TouchCode/TouchXML
KissXML	http://code.google.com/p/kissxml
TinyXML	http://www.grinninglizard.com/tinyxml/
GDataXML	http://code.google.com/p/gdata-objectivec-client

Nos vamos a centrar en el estudio de `NSXMLParser`, por ser el parser principal incluido en la API de Cocoa Touch.

Para implementar un parser con esta librería deberemos crear una clase que adopte el protocolo `NSXMLParserDelegate`, el cual define, entre otros, los siguientes métodos:

```
- (void) parser:(NSXMLParser *)parser
didStartElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qualifiedName
 attributes:(NSDictionary *)attributeDict;

- (void) parser:(NSXMLParser *)parser
didEndElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName;

- (void) parser:(NSXMLParser *)parser
foundCharacters:(NSString *)string;
```

Podemos observar que nos informa de tres tipos de eventos: `didStartElement`, `foundCharacters`, y `didEndElement`. El análisis del XML será secuencial, el parser irá leyendo el documento y nos irá notificando los elementos que encuentre. Cuando se abra una etiqueta, llamará al método `didStartElement` de nuestro parser, cuando encuentre texto llamará a `foundCharacters`, y cuando se cierra la etiqueta llamará a `didEndElement`. Será responsabilidad nuestra implementar de forma correcta estos tres eventos, y guardar la información de estado que necesitemos durante el análisis.

Por ejemplo, imaginemos un documento XML sencillo como el siguiente:

```
<mensajes>
  <mensaje usuario="pepe">Hola, ¿qué tal?</mensaje>
  <mensaje usuario="ana">Fetén</mensaje>
</mensajes>
```

Podemos analizarlo mediante un parser `NSXMLParser` como el siguiente:

```
- (void) parser:(NSXMLParser *)parser
didStartElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qualifiedName
 attributes:(NSDictionary *)attributeDict {

    if([[elementName lowercaseString] isEqualToString:@"mensajes"]) {
        self.listaMensajes = [NSMutableArray arrayWithCapacity: 100];
    } else if([[elementName lowercaseString]
                isEqualToString:@"mensaje"]) {
        self.currentMensaje = [UAMensaje mensaje];
        self.currentMensaje.usuario =
            [attributeDict objectForKey:@"usuario"];
    } else {
        // No puede haber etiquetas distintas a mensajes o mensaje
        [parser abortParsing];
    }
}

- (void) parser:(NSXMLParser *)parser
didEndElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName {

    if([[elementName lowercaseString] isEqualToString:@"mensaje"]) {
```

```

        [self.listaMensajes addObject: self.currentMensaje];
        self.currentMensaje = nil;
    }
}

- (void) parser:(NSXMLParser *)parser
  foundCharacters:(NSString *)string {

    NSString* value = [string stringByTrimmingCharactersInSet:
                      [NSCharacterSet whitespaceAndNewlineCharacterSet]];

    if([value length] != 0 && self.currentMensaje != nil) {
        self.currentMensaje.texto = value;
    }
}

```

Podemos observar que cada vez que encuentra una etiqueta de apertura, podemos obtener tanto la etiqueta como sus atributos. Cada vez que se abre un nuevo mensaje, se crea un objeto de tipo `UAMensaje` en una variable de instancia, y se van introduciendo en él los datos que se encuentran en el XML, hasta encontrar la etiqueta de cierre (en nuestro caso el texto, aunque podríamos tener etiquetas anidadas).

Para que se ejecute el *parser* que hemos implementado mediante el delegado, deberemos crear un objeto `NSXMLParser` y proporcionarle dicho delegado (en el siguiente ejemplo suponemos que nuestro objeto `self` hace de delegado). El parser se debe inicializar proporcionando el contenido XML a analizar (encapsulado en un objeto `NSData`):

```

NSXMLParser *parser = [[NSXMLParser alloc] initWithData: self.content];
parser.delegate = self;
BOOL result = [parser parse];

```

Tras inicializar el *parser*, lo ejecutamos llamando al método `parse`, que realizará el análisis de forma síncrona, y nos devolverá `YES` si todo ha ido bien, o `NO` si ha habido algún error al procesar la información (se producirá un error si en el delegado durante el *parsing* llamamos a `[parser abortParsing]`).

3.5. Parsing de estructuras JSON

JSON es una representación muy utilizada para formatear los recursos solicitados a un servicio web RESTful. Se trata de ficheros con texto plano que pueden ser manipulados muy fácilmente utilizando JavaScript.

La gramática de los objetos JSON es simple y requiere la agrupación de la definición de los datos y valores de los mismos. En primer lugar, los elementos están contenidos dentro de llaves `{` y `}`; los valores de los diferentes elementos se organizan en pares, con la estructura: `"nombre": "valor"`, y están separados por comas; y finalmente, las secuencias de elementos están contenidas entre corchetes `[` y `]`. Y esto es todo :)! Para una descripción detallada de la gramática, podéis consultar <http://www.json.org/fatfree.html>

Con la definición de agrupaciones anterior, podemos combinar múltiples conjuntos para crear cualquier tipo de estructura requerido. El siguiente ejemplo muestra una descripción

JSON de un objeto con información sobre una lista de mensajes:

```
[ { "texto": "Hola, ¿qué tal?", "usuario": "Pepe" },
  { "texto": "Fetén", "usuario": "Ana" } ]
```

Antes de visualizar cualquiera de los valores de una respuesta JSON, necesitamos convertirla en una estructura que nos resulte familiar, por ejemplo, sabemos cómo trabajar con las jerarquías de objetos Javascript.

Para convertir una cadena JSON en código "usable" utilizaremos la función Javascript nativa *eval()*. En el caso de un *stream* JSON, *eval()* transforma dicho *stream* en un objeto junto con propiedades que son accesibles sin necesidad de manipular ninguna cadenas de caracteres.

Nota

Los *streams* JSON son fragmentos de código Javascript y deben evaluarse utilizando la función *eval()* antes de poder utilizarse como objetos en tiempo de ejecución. En general, ejecutar Javascript a través de *eval()* desde fuentes no confiables introducen riesgos de seguridad debido al valor de los parámetros de las funciones ejecutadas como Javascript. Sin embargo, para nuestra aplicación de ejemplo, confiamos en que el JavaScript enviado desde Twitter es una estructura JSON segura.

Debido a que un objeto JSON evaluado es similar a un objeto DOM, podremos atravesar el árbol del objeto utilizando el carácter "punto". Por ejemplo, un elemento raíz denominado `Root` con un sub-elemento denominado `Element` puede ser accedido mediante `Root.Element`

Nota

Si estamos ante una cadena JSON sin ninguna documentación del API correspondiente, tendremos que buscar llaves de apertura y cierre (`{` y `}`). Esto nos llevará de forma inmediata a la definición del objeto. A continuación buscaremos pares de nombre/valor entre llaves.

El análisis de JSON en Android e iOS es algo más complejo que en Javascript, pero en ambas plataformas encontramos tanto librerías integradas en el SDK, como librerías proporcionadas por terceros.

3.5.1. Parsing de JSON en Android

Dentro de la API de Android encontramos una serie de clases que nos permiten analizar y componer mensajes JSON. Las dos clases fundamentales son `JSONArray` y `JSONObject`. La primera de ellas representa una lista de elementos, mientras que la segunda representa un objeto con una serie de propiedades. Podemos combinar estos dos tipos de objetos para crear cualquier estructura JSON. Cuando en el JSON encontremos una lista de elementos (`[...]`) se representará mediante `JSONArray`, mientras que cuando encontremos un conjunto de propiedades *clave-valor* encerrado entre llaves (`{ ... }`) se representará con `JSONObject`. Encontraremos estos dos tipos de elementos anidados

según la estructura JSON que estemos leyendo.

```
JSONArray mensajes = new JSONArray(contenido);

for(int i=0;i<mensajes.length();i++) {
    JSONObject mensaje = mensajes.getJSONObject(i);

    String texto = mensaje.getString("texto");
    String usuario = mensaje.getString("usuario");

    ...
}
```

El objeto `JSONArray` nos permite conocer el número de elementos que contiene (`length`), y obtenerlos a partir de su índice, con una serie de métodos `get-`. Los elementos pueden ser de tipos básicos (`boolean`, `double`, `int`, `long`, `String`), o bien ser objetos o listas de objetos (`JSONObject`, `JSONArray`). En el caso del ejemplo anterior cada elemento de la lista es un objeto *mensaje*.

Los objetos (`JSONObject`) tienen una serie de campos, a los que también se accede mediante una serie de métodos `get-` que pueden ser de los mismos tipos que en el caso de las listas. En el ejemplo anterior son cadenas (*texto*, y *usuario*), pero podrían ser listas u objetos anidados.

Esta librería no sólo nos permite analizar JSON, sino que también podemos componer mensajes con este formato. Los objetos `JSONObject` y `JSONArray` tienen para cada método `get-`, un método `put-` asociado que nos permite añadir campos o elementos. Una vez añadida la información necesaria, podemos obtener el texto JSON mediante el método `toString()` de los objetos anteriores.

Esta librería es sencilla y fácil de utilizar, pero puede generar demasiado código para parsear estructuras de complejidad media. Existen otras librerías que podemos utilizar como **GSON** (<http://sites.google.com/site/gson/gson-user-guide>) o **Jackson** (<http://wiki.fasterxml.com/JacksonInFiveMinutes>) que nos facilitarán notablemente el trabajo, ya que nos permiten mapear el JSON directamente con nuestros objetos Java, con lo que podremos acceder al contenido JSON de forma similar a como se hace en Javascript.

3.5.2. Parsing de JSON en iOS

El parsing de JSON no se ha incorporado al SDK de iOS hasta la versión 5.0. Anteriormente contábamos con diferentes librerías que podíamos incluir para realizar esta tarea, como **JSONKit** (<https://github.com/johnnezang/JSONKit>) o **json-framework** (<https://github.com/stig/json-framework/>). Sin embargo, con la aparición de iOS 5.0 podemos trabajar con JSON directamente con las clases de Cocoa Touch, sin necesidad de incluir ninguna librería adicional. Vamos a centrarnos en esta forma de trabajar.

Simplemente necesitaremos la clase `NSJSONSerialization`. A partir de ella obtendremos el contenido del JSON en una jerarquía de objetos `NSArray` y `NSDictionary`


```
NSError *error = nil;
NSData *data = ... // Contenido JSON obtenido de la red
NSArray *mensajes = [NSJSONSerialization JSONObjectWithData: data
                                                            options:0
                                                            error:&error];

if(error==nil) {
    for(NSDictionary *mensaje in mensajes) {
        NSString *texto = [mensaje valueForKey:@"texto"];
        NSString *usuario = [mensaje valueForKey:@"usuario"];
        ...
    }
}
```

El método `JSONObjectWithData:options:error:` de la clase `NSJSONSerialization` nos devolverá un `NSDictionary` o `NSArray` según si el elemento principal del JSON es un objeto o una lista, respectivamente.

Al igual que en el caso de Android, el objeto `NSJSONSerialization` también nos permite realizar la transformación en el sentido inverso, permitiendo transformar una jerarquía de objetos `NSArray` y `NSDictionary` en una representación JSON. Para eso contaremos con el método `dataWithJSONObject:`

```
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Hola!", @"texto",
    @"Pepe", @"usuario", nil];

NSData *json = [NSJSONSerialization dataWithJSONObject: dict
                                                  options:0
                                                  error:&error];
```

4. Servicios de acceso a servicios REST

Ayuda

Entre las plantillas de la sesión se incluye la aplicación `RESTfulSwingClient`. Se trata de una aplicación Java de escritorio que nos permite probar servicios web RESTful de forma sencilla. Simplemente tendremos que importarla en Eclipse y ejecutar su clase `restfulswingclient.RESTfulSwingClientApp`. Se abrirá una ventana en la que podremos introducir la URL del servicio a probar y realizar una petición, proporcionando el contenido y cabeceras que especifiquemos. En la misma ventana podremos visualizar la respuesta recibida. Esta aplicación puede resultar de utilidad para entender mejor el funcionamiento de los servicios que utilizaremos en esta sesión.

4.1. Consulta del libros

En primer lugar vamos a implementar una aplicación que consulte el listado de libros disponible en una biblioteca online. Partiremos de la plantilla `ListadoLibros`, donde encontramos una tabla o lista que deberemos rellenar con los datos obtenidos del servicio REST de la biblioteca. Se pide:

a) Realizar desde la aplicación una llamada al servicio, solicitando que nos devuelva la representación XML del recurso lista de libros. La URL a la que debemos acceder para obtener dicho listado es:

```
http://server.jtech.ua.es:8080/jbib-rest/resources/libros
```

Recoge el resultado como una cadena, y provisionalmente muéstralo como *log*. Comprueba que hemos obtenido los datos de forma correcta en formato XML (es importante enviar en la petición la cabecera `Accept` para asegurarnos de que se devuelva este formato).

Ayuda iOS

En el controlador `UAListadoLibrosViewController`, en `viewWillAppear` configura la petición para que pida representación `application/xml`.

Ayuda Android

En la actividad `ListadoLibrosActivity`, y dentro de ella en la tarea asíncrona `CargarTweetsTask`, configura la petición para obtener representación `application/xml`.

b) Implementa el *parsing* del XML anterior, y obtén una lista de libros (objetos `Libro` o `UALibro`). Tras esto se deberá actualizar la interfaz para mostrar en ella la lista de libros obtenida.

Ayuda iOS

Implementa el método del delegado del parser `parser: didStartElement: namespaceURI: qualifiedName: attributes:` para que lea los datos de los libros del XML y los añada a la lista de libros del controlador encapsulados en objetos `UALibro`. En `actualizarLibrosConDatos:` deberemos ejecutar el parser utilizando `self` como delegado.

Ayuda Android

En `parseLibros` utiliza `XmlPullParser` para obtener la lista de libros a partir del XML.

4.2. Timeline público de Twitter

Vamos a continuar trabajando con el cliente de Twitter que comenzamos en la sesión anterior. Hasta ahora esta aplicación está obteniendo la lista de *tweets* de memoria. Vamos ahora a modificar este comportamiento para que obtenga los *tweets* de los servicios REST publicados por Twitter. Concretamente, accedemos al servicio de *timeline* público y utilizaremos la representación JSON:

```
http://twitter.com/statuses/public_timeline.json
```

Ayuda iOS

En el controlador `UAListadoTweetsViewController`, en `viewDidLoad` elimina la llamada a `inicializaTweets`, que es quien carga una lista predefinida de *tweets* en memoria, y en su lugar realiza una conexión asíncrona al servicio mostrando el indicador de actividad de red cuando sea oportuno. En el método `actualizarTweetsConDatos`, lee el JSON obtenido en forma de `NSArray`. De cada elemento del *array* (se trata de diccionarios), obtén las propiedades `text`, `user.screen_name` y `user.profile_image_url` (realmente `user` es una propiedad que contiene otras propiedades, es decir, es un diccionario) y encapsúlalas en objetos `UATweet`.

Ayuda Android

En la actividad `ClienteTwitterActivity`, y dentro de ella en la tarea `CargarTweetsTask`, cambia la inicialización de la lista de *tweets* predefinida por una llamada al servicio de Twitter anterior, y obtén la representación JSON de la respuesta.

5. Autenticación en servicios remotos

Los servicios REST están fuertemente vinculados al protocolo HTTP, por lo que los mecanismos de seguridad utilizados también deberían ser los que define dicho protocolo. Pueden utilizar los diferentes tipos de autenticación definidos en HTTP: *Basic*, *Digest*, y *X.509*. Sin embargo, cuando se trata de servicios que se dejan disponibles para que cualquier desarrollador pueda acceder a ellos, utilizar directamente estos mecanismos básicos de seguridad puede resultar peligroso. En estos casos la autenticación suele realizarse mediante el protocolo OAuth.

5.1. Seguridad HTTP básica

Vamos a ver en primer lugar cómo acceder a servicios protegidos con seguridad HTTP estándar. En estos casos, deberemos proporcionar en la llamada al servicio las cabeceras de autenticación al servidor, con los credenciales que nos den acceso a las operaciones solicitadas.

Por ejemplo, desde un cliente Android en el que utilicemos la API de red estándar de Java SE deberemos definir un `Authenticator` que proporcione estos datos:

```
Authenticator.setDefault(new Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication (
            "usuario", "password".toCharArray());
    }
});
```

En caso de que utilicemos `HttpClient` de Apache, se especificará de la siguiente forma:

```
DefaultHttpClient client = new DefaultHttpClient();
client.getCredentialsProvider().setCredentials(
    new AuthScope("jtech.ua.es", 80),
    new UsernamePasswordCredentials("usuario", "password")
);
```

Aquí además de las credenciales, hay que indicar el ámbito al que se aplican (host y puerto).

Para quienes no estén muy familiarizados con la seguridad en HTTP, conviene mencionar el funcionamiento del protocolo a grandes rasgos. Cuando realizamos una petición HTTP a un recurso protegido con seguridad básica, HTTP nos devuelve una respuesta indicándonos que necesitamos autenticarnos para acceder. Es entonces cuando el cliente solicita al usuario las credenciales (usuario y password), y entonces se realiza una nueva petición con dichas credenciales incluidas en una cabecera *Authorization*. Si las credenciales son válidas, el servidor nos dará acceso al contenido solicitado.

Este es el funcionamiento habitual de la autenticación. En el caso del acceso mediante `HttpClient` que hemos visto anteriormente, el funcionamiento es el mismo, cuando el

servidor nos pida autentificarnos la librería lanzará una nueva petición con las credenciales especificadas en el proveedor de credenciales.

Sin embargo, si sabemos de antemano que un recurso va a necesitar autenticación, podemos también autentificarnos de forma preventiva. La autenticación preventiva consiste en mandar las credenciales en la primera petición, antes de que el servidor nos las solicite. Con esto ahorramos una petición, pero podríamos estar mandando las credenciales en casos en los que no resulta necesario.

Con `HttpClient` podemos activar o desactivar la autenticación preventiva con el siguiente método:

```
client.getParams().setAuthenticationPreemptive(true);
```

En iOS, la autenticación la deberá hacer el delegado de la conexión. Para ello deberemos implementar el método `connection:didReceiveAuthenticationChallenge:`. Este método será invocado cuando el servidor nos pida autentificarnos. En ese caso deberemos crear un objeto `NSURLCredential` a partir de nuestras credenciales (usuario y password).

A continuación vemos un ejemplo típico de implementación:

```
- (void)connection:(NSURLConnection *)connection
    didReceiveAuthenticationChallenge:
        (NSURLAuthenticationChallenge *)challenge
{
    if(challenge.previousFailureCount == 0) {
        NSURLCredential *cred =
            [NSURLCredential credentialWithUser:@"mi_login"
                                           password:@"mi_password"
                                           persistence:NSURLCredentialPersistenceNone];
        [[challenge sender] useCredential: cred
                               forAuthenticationChallenge:challenge];
    } else {
        [[challenge sender] cancelAuthenticationChallenge: challenge];
    }
}
```

Podemos observar que comprobamos los fallos previos de autenticación que hemos tenido. Es decir, si con los credenciales que tenemos en el código ha fallado la autenticación, será mejor que cancelemos el acceso, ya que si volvemos a intentar acceder con los mismos credenciales vamos a tener el mismo error. En caso de que sea el primer intento, creamos los credenciales (podemos ver que se puede indicar que se guarden de forma persistente para los próximos accesos), y los utilizamos para responder al reto de autenticación (`NSURLAuthenticationChallenge`).

Este método nos permite autentificarnos siempre que realicemos una conexión asíncrona. Sin embargo, si queremos conectar de forma síncrona, o queremos autentificarnos de forma preventiva, esto resultará bastante más complejo, ya que deberemos añadir las cabeceras de autenticación manualmente a la petición. La cabecera que necesitamos es `Authorization`, a la que le proporcionaremos el tipo de autenticación y los

credenciales (login y password). Por ejemplo, para utilizar seguridad de tipo BASIC como en los casos anteriores, esta cabecera tendrá la siguiente forma:

```
Authorization: Basic login_base64:password_base64
```

La mayor dificultad radica en que el login y el password no se incluyen directamente, sino que deben estar codificados en base64. Necesitaremos por lo tanto una función que se encargue de realizar esta codificación, y no hay ninguna implementada en el SDK de iOS. Esto no es demasiado complejo de implementar, pero si no estamos familiarizados con este formato podemos encontrar diferentes implementaciones. Una de ellas es la implementación de Matt Gallagher:

<http://cocoawithlove.com/2009/06/base64-encoding-options-on-mac-and.html>

Esta implementación es bastante elegante, ya que se introduce como una categoría de NSData, lo cual añade a esta clase la capacidad de realizar la codificación y decodificación. Con ella podemos implementar la codificación de la siguiente forma:

```
NSString *cred = [NSString stringWithFormat:@"%s:%s", login, password];
NSString *credBase64 = [[cred dataUsingEncoding:NSUTF8StringEncoding]
                        base64EncodedString];
NSString *authHeader = [NSString stringWithFormat:@"Basic %@",
                                                credBase64];

NSMutableURLRequest *theRequest = [NSMutableURLRequest
                                requestWithURL: url];
[theRequest addValue:authHeader
 forHTTPHeaderField:@"Authorization"];
```

Donde login y password son variables de tipo NSString donde tenemos almacenados nuestros credenciales, y url es de tipo NSURL y contiene la URL a la que conectar. En el código podemos ver que primero se genera la cadena de credenciales, concatenando login:password. A continuación esta cadena se codifica en base64, y con esto ya podemos añadir una cabecera Authorization cuyo valor sea la cadena Basic <credenciales_base64>.

Si en lugar de acceder a estos servicios desde una aplicación nativa, lo hacemos desde Javascript, deberemos asegurarnos de que el usuario se ha autenticado previamente en el sitio web que está haciendo la llamada AJAX al servicio REST seguro.

5.2. Protocolo OAuth

En la anterior sección hemos visto cómo acceder a servicios web que requieren autenticación, proporcionando las credenciales necesarias en la petición HTTP. Esto es apropiado cuando accedemos a nuestros propios servicios, sin embargo, si queremos permitir que terceros puedan acceder a ellos esto puede resultar problemático. Esto se debe a que la aplicación que acceda a nuestro servicio estaría recogiendo el login y el password del usuario, y enviándolas a nuestro servicio. Cuando el servicio responda la aplicación sabrá si las credenciales eran válidas o no. Es decir, dicha aplicación contará con las credenciales validadas de un usuario, y podría tratar dicha información de forma

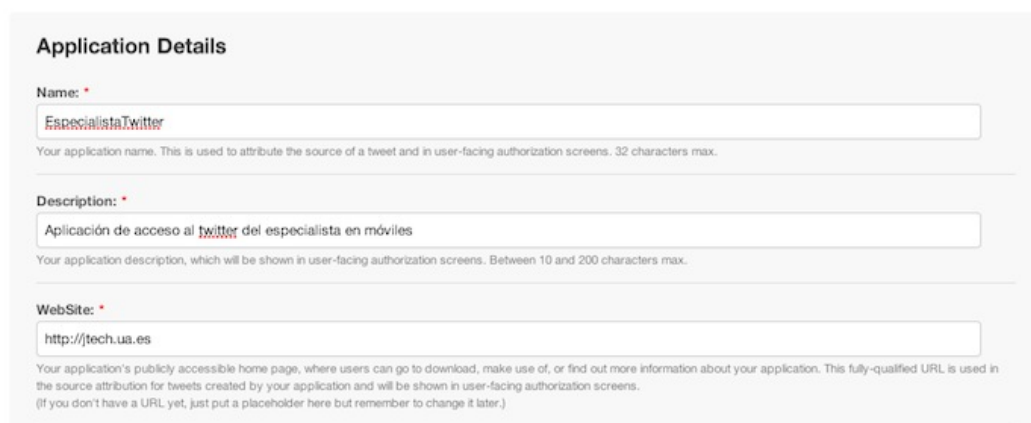
inadecuada o fraudulenta.

Por lo tanto, deberíamos evitar que las credenciales del usuario pasen por una aplicación que no controlemos nosotros. Para ello contamos con el protocolo *OAuth* (*Open Authorization*), que se encarga de autenticar usuarios en aplicaciones de terceros de forma segura. Este protocolo lo encontraremos en gran cantidad de servicios, como Twitter o Facebook. Vamos a ver a continuación el funcionamiento de este protocolo y algún ejemplo de utilización.

Supongamos que queremos que nuestra aplicación acceda a Twitter en nombre del usuario que esté utilizándola. Este es el típico escenario en el que se utiliza *OAuth*. En este caso nuestra aplicación se comportará como cliente de Twitter, y deberá tener unos credenciales que la identifiquen como tal. Para ello deberemos dar de alta nuestra aplicación en Twitter mediante el siguiente formulario:

<https://dev.twitter.com/apps/new>

Create an application



The screenshot shows the 'Create an application' form on the Twitter developer portal. It is titled 'Application Details' and contains three main sections, each with a text input field and a description:

- Name:** The input field contains 'EspecialistaTwitter'. Below it, a note states: 'Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.'
- Description:** The input field contains 'Aplicación de acceso al twitter del especialista en móviles'. Below it, a note states: 'Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.'
- WebSite:** The input field contains 'http://tech.ua.es'. Below it, a note states: 'Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens. (If you don't have a URL yet, just put a placeholder here but remember to change it later.)'

Alta como consumidor Twitter

Importante

Para poder acceder al servicio mediante OAuth de la forma que vamos a ver a continuación, será imprescindible haber rellenado en este formulario el campo *Callback URL*. Será suficiente con poner cualquier URL válida (puede coincidir con *WebSite*).

Tras registrarnos, nos da una serie de datos para nuestra aplicación:

OAuth settings

Your application's OAuth settings. Keep the "Consumer secret" a secret. This key should never be human-readable in your application.

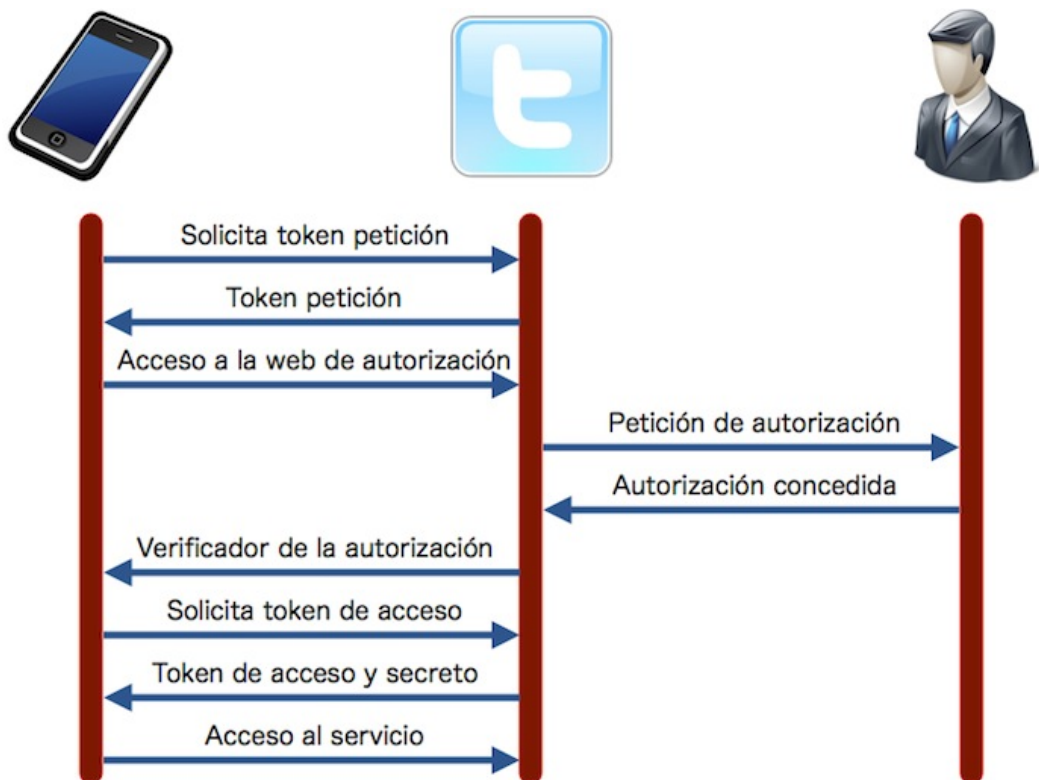
Access level	Read and write About the application permission model
Consumer key	sQivwNeLYgId62DHpcQuw
Consumer secret	vz7CtgY0jpY1GnDccM9nb3lW3KgFCIWft0V9uEglg
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token
Callback URL	None

Datos para el consumidor OAuth

Cuando una aplicación accede en nombre de un usuario a un servicio mediante *OAuth*, no sólo el usuario final debe autenticarse en el servicio, sino que la aplicación también debe hacerlo. Para esto se nos proporcionan unas credenciales al registrar nuestra aplicación en el servicio al que vamos a acceder (en el ejemplo anterior Twitter), conocidas como `oauth_consumer_key` y `oauth_consumer_secret`. Estas credenciales identifican a nuestra aplicación en el servicio, por lo que éste último tendrá un control sobre las aplicaciones que están accediendo en nombre de usuarios, y ninguna de ellas podrá acceder sin haberse registrado antes. Cuando accedamos al servicio desde nuestra aplicación, deberemos incluir estas credenciales en la petición para que el servicio conozca la identidad de la aplicación que está accediendo. Podemos pensar en `oauth_consumer_key` como el nombre de usuario único con el que nuestra aplicación accede al servicio, y en `oauth_consumer_secret` como el password asociado a dicho usuario.

Una vez contamos con estos datos, vamos a ver cómo se realizaría el proceso de autenticación. Vemos que el servicio nos proporciona diferentes URLs para realizar este proceso:

URL	Descripción
Request token URL	Solicita un token de petición acceso para nuestra aplicación, dadas nuestras credenciales.
Authorize URL	Solicita al usuario que autorice el acceso al servicio a través de nuestra aplicación. Se proporciona el token generado por la URL anterior, y si el usuario autoriza la petición, nos devuelve un código de autorización.
Access token URL	Genera el token que da acceso al servicio a partir del código de autorización obtenido mediante la URL anterior.



Ejemplo de escenario del protocolo OAuth

Vamos a ver paso a paso cómo se realiza este proceso:

1. Lo primero que debemos hacer es solicitar un token de petición de acceso para nuestra aplicación accediendo a *Request token URL*. A dicha URL le pasaremos nuestras credenciales, incluyendo `oauth_consumer_key` como parámetro de la petición, y firmando la petición entera mediante `oauth_consumer_secret` (este código se utilizará siempre para firmar las peticiones, no se enviará directamente ya que debe ser secreto). Como respuesta nos dará un token de petición (`oauth_token`) y una clave secreta asociada (`oauth_token_secret`):

```
oauth_token=ab3cd9j4ks73hf7g  
oauth_token_secret=ZXhnbXBsZS5jb20
```

Estos datos no se pueden utilizar directamente para acceder al servicio, antes deberemos pedirle autorización al usuario para que nos permita acceder en su nombre. Deberemos convertir el token de petición actual (no autorizado), en un token de petición autorizado por el usuario. Esto lo haremos en el siguiente paso.

2. Una vez tengamos el token de petición de acceso, deberemos solicitar que el usuario autorice que nuestra aplicación pueda acceder al servicio, pero esto no podemos hacerlo directamente desde nuestra aplicación, sino que el usuario deberá darnos la autorización a través del servicio final al que accederemos. Para ello tenemos la *Authorize URL*. Deberemos acceder a dicha URL en un navegador, proporcionando el token obtenido en el paso anterior. Al hacer esto, nos redirigirá a la página de login

del servicio (por ejemplo la página de login de Twitter), y le pedirá al usuario introducir sus credenciales en el servicio y le preguntará si desea autorizarnos a acceder en su nombre. Todo esto se estará haciendo fuera de nuestra aplicación. En caso de una aplicación web nos habrá redirigido a la URL de un sitio web distinto (por ejemplo `www.twitter.com`), pero en una aplicación móvil nativa tendremos que abrir un navegador. La cuestión es, ¿cómo volvemos a nuestra aplicación tras recibir la autorización del usuario? Para que esto sea posible, en la petición podemos proporcionar también una URL de *callback* como parámetro. Esta es la URL a la que el servicio al que accedemos nos llevará tras autorizarnos el usuario, pasándonos como parámetro de la query un token de autorización (`oauth_token`) y un verificador (`oauth_token_verifier`). Pero en el caso de los móviles es más complejo, ya que tenemos una aplicación nativa, no un navegador, y queremos que el token sea devuelto a nuestra aplicación, y no que el navegador nos redirija a otra página. Podemos conseguir esto poniendo como esquema de la URL de *callback* un esquema propio creado por nosotros, y haciendo que nuestra actividad "escuche" la peticiones a dicho tipo de URLs, como por ejemplo podría ser

`oauth-jtech-twitter://callback:`

```
<activity android:name=".AutenticacionOAuthActivity"
    android:launchMode="singleTask">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="oauth-jtech-twitter" android:host="callback" />
    </intent-filter>
</activity>
```

Cuando se produzca un nuevo *intent* en `AutenticacionOAuthActivity` (`onNewIntent`), comprobaremos si la URL de la que viene es la del *callback*, y de ser así obtendremos de los parámetros de la query los datos `oauth_token` y `oauth_token_verifier`.

3. El paso final consiste en, dado el token autorizado de petición de acceso obtenido en el paso anterior, solicitar un token de acceso válido para que nuestra aplicación pueda acceder al servicio en nombre del usuario. Para ello se utiliza la URL *Access token URL*. Recibirá como parámetros de entrada nuestra clave de autenticación (`oauth_consumer_key`), y el token de petición autorizado en el paso anterior junto a su verificador (`oauth_token` y `oauth_token_verifier`). Como resultado obtendremos el token de acceso (`oauth_token`) y su clave secreta asociada (`oauth_token_secret`). Con estos datos ya podremos acceder a las operaciones del servicio.
4. Para acceder al servicio deberemos proporcionar en los mensajes de petición una cabecera *Authorization* con los tokens obtenidos anteriormente: las credenciales de nuestra aplicación obtenidas en el registro (`oauth_consumer_key`), y el token de acceso obtenido en el paso anterior (`oauth_token`). Además, dicha petición deberá estar firmada utilizando como clave de firmado una combinación de las claves secretas asociadas a los tokens anteriores: `oauth_consumer_secret` y `oauth_token_secret`.

Importante

La clave secreta no debe compartirse con nadie. Si pudiésemos pensar que ha sido comprometida, deberemos invalidarla y generar una nueva.

Encontramos diferentes librerías para trabajar con *OAuth* tanto en Android como en iOS. En Android podemos encontrar por ejemplo *Signpost* (<http://code.google.com/p/oauth-signpost/>) o *Scribe* (<http://github.com/fernandezpablo85/scribe-java>). Para iOS encontramos librerías como *RSOAuth* (<https://github.com/rsieiro/RSOAuthEngine>) o *MPOAuth* (<https://github.com/thekarladam/MPOAuth>).

A continuación veremos en un ejemplo cómo utilizar *Signpost* para acceder a Twitter desde una aplicación Android.

5.3. Acceso a servicios de terceros

Un ejemplo práctico de acceso a servicios proporcionados por terceros es Twitter. Anteriormente hemos visto cómo acceder a sus servicios para obtener el *public timeline*. Sin embargo, nos ofrece muchas más operaciones, y muchas de ellas requieren que estemos autenticados como usuarios, como por ejemplo publicar nuevos mensajes.

La autenticación se realiza utilizando *OAuth*, por lo que veremos un ejemplo práctico de uso de este protocolo.

Podemos recurrir a la documentación de la API REST publicada por diferentes servicios e implementar las comunicaciones de forma manual como hemos visto anteriormente.

Servicio	URL desarrolladores
Twitter	https://dev.twitter.com/
LinkedIn	https://developer.linkedin.com/rest
Facebook	https://developers.facebook.com/
Dropbox	https://www.dropbox.com/developers

En algunas de estas páginas podemos encontrar también SDKs para las distintas plataformas móviles, como ocurre por ejemplo en el caso de Facebook o Dropbox. Si no contamos con un SDK oficial, podemos implementar nosotros el acceso a los servicios REST o bien buscar una librería proporcionada por terceros.

Tanto en Android como en iOS hay diferentes librerías que nos facilitan el acceso a los servicios de Twitter y de otras redes. Por ejemplo, en iOS contamos con **ShareKit** (<http://getsharekit.com/>) que nos permite acceder a diferentes servicios, como Facebook, Twitter o Google Reader. De forma similar, en Android encontramos **AndroidLibs** (<http://androidlibs.com/sociallib.html>), que soporta Facebook,

Twitter, LinkedIn y Google Buzz. También encontramos librerías específicas para servicios concretos, como **JTwitter** (<http://www.winterwell.com/software/jtwitter.php>) o **Twitter4J** (<http://twitter4j.org/en/index.jsp>).

A modo de ejemplo, vamos a ver a continuación cómo podemos implementar el acceso a Twitter desde una aplicación Android utilizando **Signpost** y **Twitter4J**. Tras esto, veremos la forma de acceder a Twitter desde iOS. En este caso será más sencillo, ya que desde iOS 5.0 el acceso a Twitter se encuentra integrado en el SDK, por lo que no será necesario incluir ninguna librería adicional.

5.3.1. Acceso a Twitter desde Android

El primer paso para hacer que nuestra aplicación pueda acceder a Twitter será darla de alta en este servicio, tal como hemos visto anteriormente, para así poder realizar la autenticación mediante *OAuth*. Una vez hecho esto tendremos nuestra clave de consumidor y las URLs de acceso. Podemos introducir esta información como constantes en nuestra actividad (vamos a suponer que estamos creando una actividad cuyo nombre es `TwitterActivity`):

```
private final static String OAUTH_CONSUMER_KEY = "sQivwNeLYgId62DHpcQuw";
private final static String OAUTH_CONSUMER_SECRET =
    "vz7CtgYOjpYlGnDccM9nb3lW3KgFCIWftOV9uEglg";

private final static String REQUEST_TOKEN_URL =
    "https://api.twitter.com/oauth/request_token";
private final static String AUTHORIZE_URL =
    "https://api.twitter.com/oauth/authorize";
private final static String ACCESS_TOKEN_URL =
    "https://api.twitter.com/oauth/access_token";
```

Primer paso. Para obtener el token de acceso *OAuth*, lo primero que deberemos hacer es obtener el token de petición. Crearemos un objeto de tipo `OAuthConsumer` y otro de tipo `OAuthProvider`. El primero de ellos representa al consumidor (nuestra aplicación), y se crea a partir de nuestras credenciales (clave y secreto del consumidor). El segundo representa el proveedor (Twitter en este caso), y se crea a partir de las URLs para la autenticación mediante *OAuth*.

```
OAuthConsumer consumer;
OAuthProvider provider;

...

consumer = new CommonsHttpOAuthConsumer(OAUTH_CONSUMER_KEY,
    OAUTH_CONSUMER_SECRET);
provider = new CommonsHttpOAuthProvider(REQUEST_TOKEN_URL,
    ACCESS_TOKEN_URL,
    AUTHORIZE_URL);

String url = provider.retrieveRequestToken(consumer, CALLBACK_URL);
```

Una vez creados el consumidor y proveedor, solicitamos el token de petición *OAuth*. Para ello llamamos sobre el proveedor a `retrieveRequestToken`, proporcionando la

información sobre el consumidor y la URL de *callback* que utilizaremos. Nos devuelve una URL que llevará ya incluido como parámetro el token de petición. Es decir, al llamar a este método se estará conectando a la *Request token URL*, y una vez obtenga el token de petición, creará la URL de autorización adjuntando dicho token como parámetro. Esta URL tendrá la siguiente forma:

```
https://api.twitter.com/oauth/authorize?
oauth_token=vF7W9KKYwXgOYjFhpoXyRN66d89AxSXxLL9W0hB18
```

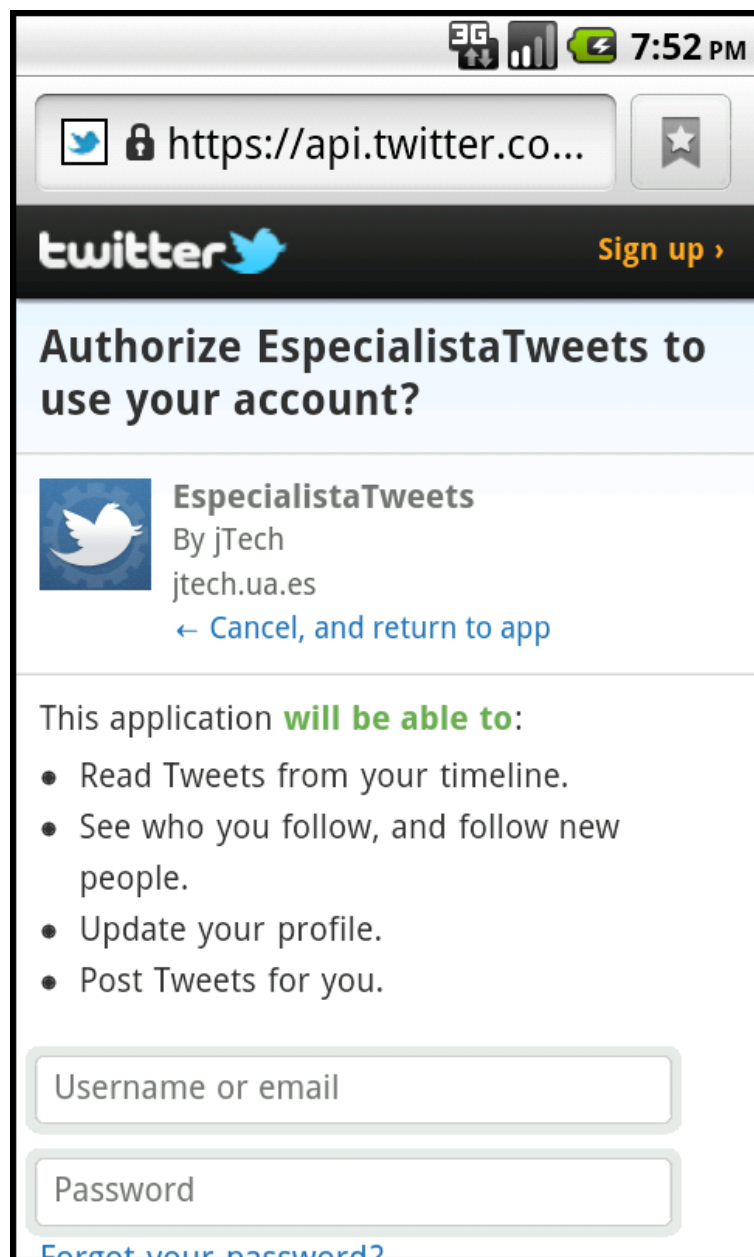
Segundo paso. A continuación debemos solicitar autorización para nuestro token de acceso. La autorización nos la debe dar el usuario conectando directamente a través de Twitter. Es decir, tendremos que abrir un navegador y acceder en él a la URL anterior:

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
```

Recomendación

Es recomendable añadir algunos *flags* al intent anterior, por ejemplo para evitar que se guarde en la historia, ya que no queremos que al volver atrás vuelva a aparecer la pantalla de login.

Veremos una página como la siguiente:



Autorización de acceso a Twitter

Cuando introduzcamos los datos y pulsemos el botón *Authorize app*, si los credenciales son correctos nos llevará a la URL de *callback*, a la que se le pasará como parámetro la clave autorizada. Aquí es donde viene la parte más compleja, ya que queremos que la URL de *callback* no nos lleve a una página en el navegador, sino que nos envíe de vuelta a la aplicación Android. Para ello podemos hacer que nuestra actividad tenga un filtro que haga que se invoque al visualizar un determinado tipo de URL, tal como hemos visto anteriormente. Podemos declararla de la siguiente forma en el `AndroidManifest.xml`:

```
<activity android:name=".TwitterActivity"
    android:label="Acceso a Twitter"
    android:launchMode="singleTask">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="oauth-jtech-twitter" android:host="callback" />
    </intent-filter>
</activity>
```

Podemos ver que la actividad la hemos declarado como `singleTask`, ya que queremos que cuando se acceda a la URL de *callback*, se vuelva a la actividad que ya teníamos abierta, y no se cree una nueva en la pila. En la actividad podemos especificar también la dirección de *callback* como constante:

```
private final static String CALLBACK_SCHEME = "oauth-jtech-twitter";
private final static String CALLBACK_URL = CALLBACK_SCHEME +
    "://callback";
```

Hemos de recordar que esta dirección de *callback* se ha utilizado en el paso anterior al llamar a `retrieveRequestToken` (es decir, no se utiliza la dirección de *callback* por defecto que se indicó al dar de alta nuestra aplicación en Twitter). Ahora cuando confirmemos nuestros datos en la pantalla del navegador, y autoricemos a la aplicación, al cargar la URL de *callback* nos llevará de nuevo a la actividad `TwitterActivity` y se ejecutará el método `onNewIntent` de la misma. En este método deberemos comprobar si la `Uri` del intent que nos ha llegado es la de *callback*, y en tal caso continuaremos con la autenticación:

```
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);

    Uri uri = intent.getData();
    if (uri != null && uri.getScheme().equals(CALLBACK_SCHEME)) {
        // Solicitar token de acceso
        ...
    }
}
```

La `Uri` de *callback* obtenida tendrá la siguiente forma:

```
oauth-jtech-twitter://callback?
  oauth_token=vF7W9KKYwXgOYjFhpoXyRN66d89AxSXxLL9W0hB18&
  oauth_verifier=2pXSOjjwJ48Ues51iQm2K5PDTI6WS8xkWJzcgABoQ
```

Tercer paso. Para finalizar la autenticación deberemos solicitar el token de acceso. Para ello extraeremos de la URL anterior el verificador *OAuth*, y solicitaremos al proveedor el token de acceso proporcionándole como entrada dicho verificador de la autorización del usuario:

```
String oauthVerifier = uri.getQueryParameter(OAuth.OAUTH_VERIFIER);
provider.retrieveAccessToken(consumer, oauthVerifier);

String oauthToken = consumer.getToken();
String oauthTokenSecret = consumer.getTokenSecret();
```

El token de acceso se solicita llamando al método `retrieveAccessToken` del proveedor, pasándole como parámetro la información del consumidor y el verificador obtenido en el paso anterior. Eso llamará a la *Access token URL*, y nos devolverá el token de acceso autorizado junto a su secreto asociado. Ahora ya podremos acceder a Twitter utilizando estos datos. La información de autenticación que deberemos proporcionar a Twitter es:

- Clave del consumidor y secreto asociado
- Token de acceso y secreto asociado

Esta información se añadirá al cliente de Twitter4J de la siguiente forma:

```
Twitter twitter = TwitterFactory.getSingleton();
twitter.setOAuthConsumer(OAUTH_CONSUMER_KEY, OAUTH_CONSUMER_SECRET);
AccessToken token = new AccessToken(oauthToken, oauthTokenSecret);
twitter.setOAuthAccessToken(token);
```

Esto sólo es necesario hacerlo una vez, a partir de este momento podremos utilizar el *singleton* Twitter todas las veces que queramos sin necesidad de volver a especificar la información de acceso. Es importante destacar que el token de acceso no tiene caducidad, por lo que será recomendable guardarlo en las *shared preferences* para así tenerlo disponible para futuras ejecuciones de la aplicación, sin tener que volver a solicitarlo. Sólo dejará de funcionar si el usuario revoca el acceso a Twitter para nuestra aplicación.

Ahora por ejemplo podremos publicar un nuevo *tweet* de la siguiente forma:

```
Twitter twitter = TwitterFactory.getSingleton();
twitter.updateStatus(mensaje);
```

Importante

Debemos tener en cuenta que los métodos `retrieveRequestToken` y `retrieveAccessToken` establecen una conexión de red para obtener los tokens de petición y acceso respectivamente. Por lo tanto, deberemos llamar a estos métodos siempre desde hilos en segundo plano (podemos utilizar una `AsyncTask` como hemos visto anteriormente). Lo mismo se aplica a las llamadas a operaciones de Twitter, como es el caso de `updateStatus`.

5.3.2. Acceso a Twitter desde iOS

Como hemos visto, en iOS también encontramos diferentes librerías para acceder a Twitter. Sin embargo, a partir de iOS 5.0 el acceso a Twitter se incorpora al SDK, por lo que no será necesario introducir ninguna librería adicional, y simplificará enormemente el acceso a este servicio, ya que el propio sistema operativo se ocupará de la gestión de las cuentas de usuario para acceder a Twitter, lo cual evitará que tengamos que preocuparnos nosotros de implementar el acceso mediante *OAuth*.

Con la API de iOS 5.0 SDK tenemos dos formas de utilizar Twitter: utilizar un formulario predefinido para componer *tweets* (`TWTweetComposeViewController`), o enviar los *tweets* directamente desde el código (`TWRequest`). Las dos clases anteriores

pertenecen el *Twitter framework*, que deberemos incluir en nuestra aplicación para acceder a Twitter utilizando esta API.

La forma más sencilla de integrar Twitter en nuestra aplicación es utilizar el formulario predefinido. Hacer esto es tan sencillo como añadir las siguientes líneas:

```
TWTweetComposeViewController *controller =
    [[TWTweetComposeViewController alloc] init];
[controller setInitialText: @"Texto por defecto"];
[self presentViewController:controller animated: YES];
```

Como podemos ver, podemos añadir un texto inicial por defecto, pero el usuario podrá modificarlo y no se nos permitirá alterar lo que haya introducido finalmente el usuario. Es decir, el usuario tendrá siempre la última palabra sobre lo que se va a publicar en Twitter.

Si necesitamos conocer cuándo se ha enviado el *tweet* o cuando se ha cancelado, podemos añadir un *completion handler* de la siguiente forma:

```
TWTweetComposeViewController *controller =
    [[TWTweetComposeViewController alloc] init];
[controller setInitialText: @"Texto por defecto"];
[controller setCompletionHandler:
    ^(TWTweetComposeViewControllerResult result) {
        switch(result) {
            case TWTweetComposeViewControllerResultDone:
                NSLog(@"Tweet enviado");
                break;
            case TWTweetComposeViewControllerResultCancelled:
                NSLog(@"Tweet cancelado");
                break;
        }
        [self dismissModalViewControllerAnimated:YES];
    }];
[self presentViewController:controller animated: YES];
```

Si queremos enviar un *tweet* desde el código de la aplicación, sin necesitar la interacción del usuario, deberemos utilizar la clase *TWRequest*. Esta clase nos permitirá acceder a toda la API REST de forma sencilla, no estando limitada únicamente al envío de *tweets* como en el caso simplificado anterior.

Antes de acceder, necesitaremos seleccionar la cuenta de usuario que utilizaremos. Para ello necesitaremos añadir a nuestro proyecto el *framework Accounts*. Las cuentas de usuario de Twitter se gestionan a nivel de sistema operativo, así que lo único que tendremos que hacer es seleccionar una de las disponibles. Para ello podemos acceder al almacén de cuentas (*ACAccountStore*), y solicitar la lista de cuentas de tipo Twitter (*ACAccountTypeIdentifierTwitter*). Puede que haya más de una cuenta, en el siguiente ejemplo por simplicidad cogemos siempre la primera de la lista, pero podríamos por ejemplo presentar una interfaz al usuario que le permita seleccionar la cuenta que desea utilizar.

```
ACAccountStore *accountStore = [[ACAccountStore alloc] init];
ACAccountType *accountType = [accountStore
    accountTypeWithIdentifier:ACAccountTypeIdentifierTwitter];
[accountStore requestAccessToAccountsWithType:accountType
    withCompletionHandler:^(BOOL granted, NSError *error) {
```

```

if(granted) {
    NSArray *accountsArray =
        [accountStore accountsWithType:accountType];

    if ([accountsArray count] > 0) {
        ACAccount *twitterAccount = [accountsArray objectAtIndex:0];
        ...
    }
}
};

```

Una vez obtenida la cuenta a utilizar (ACAccount), ya podemos acceder a Twitter mediante `TWRequest`. Este objeto se inicializa a partir de la URL de los servicios de Twitter a la que queramos acceder, los parámetros que queramos enviar, y el método HTTP a utilizar (consultar la documentación de los servicios REST de Twitter, <https://dev.twitter.com/>, para ver la lista completa de operaciones, junto con su URL, lista de parámetros admitidos, y método HTTP a utilizar). Tras inicializarlo, deberemos establecer la cuenta de Twitter a utilizar. A continuación vemos la forma de enviar un nuevo *tweet*:

```

TWRequest *postRequest = [[TWRequest alloc]
    initWithURL:
        [NSURL URLWithString:
            @"http://api.twitter.com/1/statuses/update.json"]
    parameters:
        [NSDictionary dictionaryWithObject:@"Tweet predefinido"
            forKey:@"status"]
    requestMethod:TWRequestMethodPOST];
[postRequest setAccount:twitterAccount];

```

Una vez tenemos este objeto, podemos enviar la petición HTTP al servicio firmada correctamente con nuestro token de acceso *OAuth* llamando al siguiente método:

```

[postRequest performRequestWithHandler:^(NSData *responseData,
    NSHTTPURLResponse *urlResponse, NSError *error) {
    NSLog(@"Tweet enviado");
}];

```

Tenemos que proporcionar también un *handler* al que se le notificará el resultado del envío.

Hemos visto que en este caso no nos ha hecho falta utilizar nuestra clave de consumidor, ya que la autenticación se está realizando a nivel de sistema operativo. Es decir, el consumidor en este caso es el propio sistema iOS, que incorporará su propia clave de consumidor con la que se accede a Twitter. El usuario configurará su cuenta en los ajustes del teléfono, y desde el código lo único que deberemos hacer es seleccionar una de las cuentas disponibles.

6. Ejercicios de autenticación en servicios remotos

6.1. Publicación de libros

Vamos a continuar trabajando con el ejercicio del listado de libros de la sesión anterior. Hasta ahora tenemos únicamente un listado de libros. Vamos ahora a permitir publicar nuevos libros en la biblioteca mediante nuestra aplicación. Para publicar libros utilizaremos la misma URL que para la consulta, pero en este caso conectaremos mediante POST en lugar de GET, y deberemos enviar como contenido la representación `application/json` del libro a añadir. Además, esta operación deben poder realizarla únicamente los bibliotecarios, por lo que el servicio nos pedirá autenticación de tipo BASIC. Para poder realizar la operación deberemos proporcionar las credenciales `bibliotecario:bibliotecario`.

Ayuda iOS

Trabajaremos con el controlador `UANuevoLibroViewController`. En `pulsadoAgregarLibro`: deberemos componer el mensaje JSON y configurar la petición como POST. ¡Cuidado!, los atributos del JSON llevan una `@` delante del nombre, es decir, son `"@isbn"`, `"@titulo"`, `"@autor"`, y `"@numPaginas"`. Configura la autenticación en `connection: didReceiveAuthenticationChallenge:`. Si al realizar la petición POST se obtiene un error `NSURLErrorDomain -1012`, lo más probable es que se trate de un fallo de autenticación.

Ayuda Android

Trabajaremos con la actividad `NuevoLibroActivity`. En su método `enviarLibro` establece las credenciales y configura la petición como POST.

6.2. Acceso a Twitter

En este caso vamos a continuar con el cliente de Twitter con el que hemos trabajado en la sesiones anteriores. Hasta ahora tenemos implementado el acceso al listado de *tweets* del *timeline* público. Vamos a añadir ahora una opción para publicar un nuevo *tweet*. En iOS utilizaremos la interfaz predefinida que se incluye en el SDK a partir de su versión 5.0. En Android será bastante más complejo, ya que deberemos implementar nosotros la autenticación mediante OAuth.

Ayuda iOS

Mostraremos la interfaz predefinida de iOS 5 para envío de *tweets* en el método `nuevoTweet` del controlador `UAListadoTweetsViewController`.

Ayuda Android

Trabajaremos en la actividad `NuevoTweetActivity`. En ella debemos implementar la autenticación mediante OAuth con la librería **Signpost** y el envío de *tweets* con **Twitter4J**. Estas librerías ya están incluidas en el proyecto. Sustituye en la actividad todos los comentarios *TODOs* según las instrucciones indicadas, ayudándote de los apuntes de teoría.

7. Visor web

Un componente muy importante en las aplicaciones para dispositivos móviles es el visor web. Se trata de una vista más, pero nos permite introducir en ella contenido web, personalizar la forma en la que éste se muestra, e incluso comunicarnos con él de forma bidireccional, haciendo llamadas a Javascript o recibiendo *callbacks* desde la web hacia el código nativo de nuestra aplicación.

De este modo podremos integrar de forma muy sencilla contenido rico en nuestras aplicaciones nativas, e incluso acceder a características hardware de nuestro dispositivo desde los componentes web. Existen *frameworks* que nos proporcionan mecanismos para acceder al hardware del dispositivo desde los componentes web, permitiendo así implementar aplicaciones para móvil independientes de la plataforma. Vamos a ver el caso de **PhoneGap** (<http://phonegap.com/>), que es uno de los más conocidos. Podemos también encontrar otros como **appMobi** (<http://www.appmobi.com/>), **Rhodes** (<http://rhomobile.com/>) o **Appcelerator Titanium** (<http://appcelerator.com/>). En este último caso las aplicaciones se programan con una serie de librerías Javascript, pero el *framework* lo que hace es generar una aplicación nativa a partir de dicho código, en lugar de empaquetar los documentos web directamente en la aplicación. Es decir, ya no se trata de una aplicación web empaquetada como aplicación nativa, sino que lo que hace es definir un lenguaje y una API únicos que nos servirán para generar aplicaciones nativas para diferentes plataformas. De forma similar, encontramos también **Adobe Air Mobile** (<http://www.adobe.com/devnet/devices.html>), que nos permite crear aplicaciones multiplataforma escritas en *ActionScript*. Para utilizar dicha tecnología podemos utilizar tanto la herramienta de pago **Flash Builder** (<http://www.adobe.com/es/products/flash-builder.html>), como la alternativa **open source Flex SDK** (<http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK>). Otra alternativa similar es **Corona** (<http://www.anscamobile.com/corona/>), que nos permite crear aplicaciones escritas en Lua. Por último, hacemos mención también de **J2ME Polish** (<http://www.j2mepolish.org>), ya que una de sus herramientas nos permite portar aplicaciones escritas en JavaME a diferentes plataformas móviles, como Android o iOS.

7.1. Creación de un visor web

El visor web es un tipo de vista más, que se define de forma muy parecida tanto en Android como en iOS. Vamos a ver cada uno de los casos: *UIWebView* en iOS y *WebView* en Android. En ambos casos el componente nos permitirá cargar de una URL el contenido que queramos mostrar en él, que puede ser tanto remoto como local (ficheros web alojados en el dispositivo). Este contenido se mostrará utilizando el motor del navegador nativo de cada dispositivo.

7.1.1. UIWebView en iOS

En iOS podemos introducir este componente bien en el fichero NIB, y hacer referencia a él como *outlet*, o bien instanciarlo de forma programática:

```
UIWebView *theWebView = [[UIWebView alloc] initWithFrame:
    [self.view bounds]];

self.webView = theWebView;
[self.view addSubview: theWebView];
[theWebView release];
```

Tanto si se ha creado de forma visual como programática, lo principal que deberemos hacer con el `UIWebView` es especificar la URL que queremos que muestre. Un lugar apropiado para hacer esto es en el método `viewDidLoad`, en el que podemos establecer esta URL de la siguiente forma:

```
[self.webView loadRequest: [NSURLRequest requestWithURL:
    [NSURL URLWithString: @"http://www.ua.es"]]];
```

Si queremos mostrar un documento web empaquetado con la misma aplicación, deberemos obtener la URL que nos da acceso a dicho documento. Para ello utilizaremos la clase `NSBundle` que representa el paquete de la aplicación, y obtendremos a partir del *bundle* principal la URL que nos da acceso al recurso deseado. En el caso del siguiente ejemplo, accedemos al recurso `index.html` empaquetado junto a la aplicación (en el directorio raíz):

```
[self.webView loadRequest: [NSURLRequest requestWithURL:
    [[NSBundle mainBundle] URLForResource:@"index"
    withExtension:@"html"]]];
```

Cuando estemos accediendo a una URL remota puede que tarde un tiempo en cargar, por lo que sería buena idea activar el indicador de actividad de red mientras la web está cargando. Para ello podemos crearnos un delegado que adopte el protocolo `UIWebViewDelegate` (podemos utilizar nuestro propio controlador para ello). Los métodos que deberemos implementar del delegado para saber cuándo comienza y termina la carga del contenido web son:

```
- (void) webView:(UIWebView *)webView
  didFailLoadWithError:(NSError *)error {
    [[UIApplication sharedApplication]
      setNetworkActivityIndicatorVisible: NO];
}

- (void)webViewDidStartLoad:(UIWebView *)webView {
    [[UIApplication sharedApplication]
      setNetworkActivityIndicatorVisible: YES];
}

- (void)webViewDidFinishLoad:(UIWebView *)webView {
    [[UIApplication sharedApplication]
      setNetworkActivityIndicatorVisible: NO];
}
```

7.1.2. WebView en Android

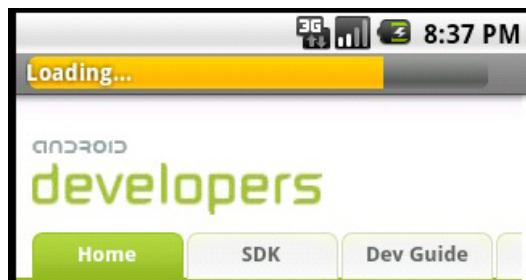
Al igual que en iOS, en Android el visor web es un tipo de vista (*view*) que podemos incluir en nuestro *layout*. En este caso se representa mediante la clase *WebView*, y en nuestro código podemos cargar una página llamando a su método *loadUrl* (un lugar típico para hacer esto es el método *onCreate* de la actividad que lo gestiona):

```
final WebView wbVisor = (WebView)this.findViewById(R.id.wvVisor);
wbVisor.loadUrl("http://www.ua.es");
```

Al igual que ocurría en el caso de iOS, puede que nos interese mostrar contenido web empaquetado con la aplicación. Normalmente este contenido lo guardaremos como *assets* del proyecto (en un directorio de nombre *assets*), que son aquellos recursos que queremos incluir en la aplicación, pero que no queremos que se procesen en el proceso de construcción, sino que queremos que se incluyan tal cual los hemos añadido al directorio. La URL para acceder a estos *assets* es la siguiente:

```
wbVisor.loadUrl("file:///android_asset/index.html");
```

En el caso de Android, en la barra de estado no sólo tenemos un indicador de actividad indeterminado, sino que además tenemos una barra de progreso. Vamos a ver cómo podemos vincular dicha barra al avance en la carga de la web, para así informar al usuario del punto exacto en el que se encuentra el proceso de carga.



Barra de progreso en Android

El navegador integrado en el *webView* de Android es Google Chrome. Si queremos recibir notificaciones sobre los eventos producidos en dicho navegador deberemos crear una subclase suya y sobrescribir los eventos que nos interesen. Por ejemplo, en nuestro caso nos interesa *onProgressChanged*, que nos informará de los avances en el progreso de carga. Cada vez que recibamos una actualización del progreso estableceremos dicho avance en la barra de progreso de nuestra actividad:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().requestFeature(Window.FEATURE_PROGRESS);
    getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
    super.onCreate(savedInstanceState);

    WebView wv = new WebView(this);
    this.setContentView(wv);
```

```

wv.setWebChromeClient(new WebChromeClient() {
    public void onProgressChanged(WebView view, int progress) {
        setProgress(progress * 100);
    }
});

wv.loadUrl("http://www.ua.es");
}

```

Debemos multiplicar el progreso por 100, ya que el cliente Google Chrome nos da un valor de progreso de 0 a 100, mientras que la barra de progreso por defecto se mueve entre 0 y 10000. Hay que destacar además que al crear la clase, antes de realizar cualquier otra operación, debemos solicitar que se habilite la característica `Window.FEATURE_PROGRESS`.

Sobrescribir el `WebChromeClient` como hemos hecho en el caso anterior nos permitirá recibir eventos que afecten a lo que se vaya a mostrar en la interfaz (actualización de progreso, recepción del título e icono de la página, etc), y eventos de Javascript (alertas, mensajes en consola, peticiones de confirmación, etc). Por otro lado, podemos sobrescribir también `WebClient`, que contiene eventos relacionados con la carga del contenido web. Por ejemplo podemos utilizarlo para saber cuándo ha fallado la carga de la página solicitada:

```

wv.setWebViewClient(new WebViewClient() {
    public void onReceivedError(WebView view, int errorCode,
        String description, String failingUrl) {
        Toast.makeText(activity, getString(R.string.problemas_red),
            Toast.LENGTH_LONG).show();
    }
});

```

Sobrescribiendo este último componente podremos saber también, entre otras cosas, cuándo comienza o finaliza la carga de una página.

7.2. Propiedades del visor

7.2.1. Renderizado de la página

El visor web, tanto de Android como de iOS, tiene una serie de propiedades con las que podemos cambiar la forma en la que muestra el contenido. Una propiedad importante que encontramos en ambos dispositivos es la que nos permite especificar cómo se ajustará la web al tamaño de la pantalla. Tenemos dos alternativas:

- **Escalar la web al tamaño de la pantalla.** En este caso se renderiza la web, de forma que se visualice correctamente, y después se escala para que coincida con el tamaño de la pantalla. El usuario podrá hacer *zoom* para ver con más detalle determinadas secciones de la página. Esta estrategia es la adecuada cuando tenemos contenido web que no está optimizado para dispositivos móviles, ya que se genera con el tamaño que sea necesario para que se visualice correctamente y luego se escala al tamaño de la pantalla.

- **Mostrar la web con su tamaño original.** Muestra los elementos de la web con su tamaño original para que se puedan visualizar claramente en pantalla, y no permite que haya escalados en la web. Esta estrategia está pensada para páginas adaptadas a las pantallas de dispositivos móviles, en las que no nos interesa que el usuario pueda hacer *zoom*, para así darle un aspecto más "nativo". El problema está en que si la web no está preparada y ocupa más del ancho disponible en la pantalla, podremos tener descuadres y no se visualizará correctamente. En estos casos deberíamos recurrir a la estrategia anterior.

A continuación vemos un ejemplo de visor web sin escalado (izquierda) y con escalado (derecha):



Visor web sin (izquierda) y con escalado (derecha)

En iOS este comportamiento se puede modificar mediante la propiedad `scalesPageToFit`. Si su valor es `YES` escalará la página para que se visualice completa en pantalla, independientemente de su anchura, y nos permitirá hacer *zoom* en ella. Si su

valor es NO, el *zoom* no estará permitido, y la web se renderizará sujeta al ancho de la pantalla, por lo que si no está preparada no se visualizará correctamente.

En Android no hay una única propiedad para controlar esto, sino que hay varias que nos permitirán tener un control más afinado sobre el comportamiento del visor. Para conseguir el mismo efecto que se conseguía poniendo `scalesPageToFit = YES` en iOS tendremos que establecer las siguientes propiedades:

```
wbVisor.getSettings().setSupportZoom(false);
wbVisor.getSettings().setUseWideViewPort(true);
wbVisor.getSettings().setLoadWithOverviewMode(true);
```

En primer lugar podemos ver que muchas de las propiedades del visor web se establecen a través de un objeto de tipo `WebSettings` asociado a él, que obtendremos mediante el método `getSettings`. Las propiedades que utilizamos en este caso son:

- `supportZoom`: Indica si se permite al usuario hacer *zoom* en la página o no.
- `useWideViewPort`: Indica si la página se renderiza sobre un lienzo del tamaño de la pantalla (`false`), o uno con la anchura suficiente para que el contenido no aparezca descuadrado (`true`). Si una página no está adaptada para visualizarse en la pantalla de un móvil, deberemos poner esta propiedad a `true`.
- `loadWithOverviewMode`: Con las dos propiedades anteriores habilitadas podemos conseguir que una página no adaptada a móviles se renderice correctamente, pero lo hará a una anchura mayor que la de la pantalla, por lo que será necesario hacer *scroll* horizontal para movernos por ella. Si queremos que el *zoom* se adapte al tamaño de la pantalla para así poder ver la página completa en pantalla, deberemos poner esta tercera propiedad a `true`.

Nota

Cuando mostremos una web realizada con jQuery Mobile en dispositivos Android, deberemos establecer el estilo de las barras de *scroll* a `WebView.SCROLLBARS_OUTSIDE_OVERLAY`, ya que si no hacemos esto veremos una antiestética franja blanca en el lateral derecho de la pantalla. Estableciendo dicho tipo de barras de *scroll* conseguimos que se superpongan al contenido del visor, por lo que no tendrá que reservar un hueco para ellas. Estableceremos dicho estilo con:

```
webView.setScrollBarStyle(WebView.SCROLLBARS_OUTSIDE_OVERLAY);
```

7.2.2. Movimientos en el historial

En los visores web el usuario puede navegar entre distintas páginas pulsando sobre los enlaces que encuentre, por lo que puede ser interesante permitir que se mueva adelante o atrás en el historial de páginas visitadas (como ocurre en cualquier navegador web). Para ello, tanto en Android como en iOS tenemos los métodos `goBack` y `goForward`. Estos métodos pertenecen a las clases `WebView` y `UIWebView` respectivamente.

Es posible que en un momento dado nos encontramos en la primera o última página del historial de navegación, por lo que no podremos movernos atrás o adelante

respectivamente. Para saber si estos movimientos están permitidos tenemos los métodos `canGoBack` y `canGoForward`. Estos métodos también están presentes con el mismo nombre en el visor web de Android y de iOS.

```
if(wbVisor.canGoBack()) {  
    wbVisor.goBack();  
}  
  
if([self.webView canGoBack]) {  
    [self.webView goBack];  
}
```

7.3. Comunicación entre web y aplicación

Una característica muy interesante de los visores web es su capacidad de comunicar de forma bidireccional aplicación y contenido web. La comunicación se hará entre el código nativo de nuestra aplicación (Objective-C o Java) y el código de la web (Javascript). Primero veremos cómo hacer una llamada desde nuestra aplicación a una función Javascript de la web, y después veremos cómo desde Javascript también podemos ejecutar código Java u Objective-C de nuestra aplicación.

7.3.1. Comunicación aplicación-web

La comunicación en este sentido se realiza invocando desde nuestra aplicación una función Javascript que haya definida en el documento que estemos mostrando actualmente en el visor web. Supongamos que tenemos la siguiente función definida:

```
<script type="text/javascript">  
    function miFuncionJavascript() {  
        alert("Hola mundo!");  
    }  
</script>
```

Realizar esto con iOS es inmediato, ya que el visor web incorpora el método `stringByEvaluatingJavaScriptFromString`: que nos permite ejecutar un bloque de código Javascript. Como comentamos, se puede tratar de cualquier bloque de código, y aunque lo habitual es realizar una única llamada, podríamos utilizarlo incluso para inyectar código Javascript en la web que se esté visualizando. A continuación vemos cómo podríamos realizar una llamada a una función Javascript con este método:

```
[self.webView stringByEvaluatingJavaScriptFromString:  
    @"miFuncionJavascript()"];
```

En Android no es tan directo, aunque no por ello deja de ser sencillo. En esta plataforma no contamos con ningún método específico para ejecutar Javascript, pero podemos utilizar el mismo `loadUrl` pasando como parámetro una URL con la forma `javascript:`

```
wbVisor.loadUrl("javascript:miFuncionJavascript()");
```

Con esto podríamos por ejemplo incluir un botón en nuestra aplicación, que al pulsarlo

produzca la ejecución de un *script* en el visor web.

Importante

En Android para poder utilizar Javascript en el visor web deberemos activarlo mediante las propiedades del visor, concretamente mediante `wbVisor.getSettings().setJavaScriptEnabled(true);`. De no hacer esto, por defecto Javascript se encuentra deshabilitado.

7.3.2. Comunicación web-aplicación

Vamos a ver el caso de la comunicación en sentido contrario. Este caso resulta ligeramente más complejo, y aquí se intercambian los papeles: para este tipo de comunicación la API de Android presenta un mecanismo más elegante que la de iOS.

En iOS la comunicación web-aplicación se implementa mediante uno de los métodos del delegado del visor web (`UIWebViewDelegate`). A parte de los métodos que hemos visto anteriormente, este delegado presenta el método `webView:shouldStartLoadWithRequest:navigationType:`. Este método se ejecuta cuando en el visor web se intenta cargar una nueva URL (por ejemplo cuando el usuario pulsa sobre un enlace). Podemos aprovechar esto, y crear nuestro propio esquema para las URLs (por ejemplo `webview:`), de forma que cuando se intercepte la solicitud de carga de una URL de este tipo esto se interprete como una llamada desde el Javascript hacia nuestra aplicación. Imaginemos que en nuestra función Javascript queremos notificar a nuestra aplicación que un temporizador ha finalizado. Podríamos hacerlo de la siguiente forma, indicando desde Javascript al navegador que debe modificar su URL, con la propiedad `document.location`:

```
<script type="text/javascript">
  function finTemporizador() {
    document.location = "webview:temporizadorFinalizado";
  }
</script>
```

En nuestro delegado, interceptaremos la petición de carga de URL, y si el esquema es `webview` y a continuación se indica `temporizadorFinalizado`, entonces ejecutamos el código necesario para tratar dicho evento y devolvemos `NO` para que no intente cargar dicha URL en el navegador.

```
- (BOOL) webView:(UIWebView *)webView
  shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UINavigationController *)navigationType {

  NSString *url = [[request URL] absoluteString];
  NSArray *componentes = [url componentsSeparatedByString:@":"];

  if([componentes count] > 1) {
    if([componentes objectAtIndex: 0] isEqualToString:@"webview"]) {
      if([componentes objectAtIndex: 1] isEqualToString:
        @"temporizadorFinalizado"]) {
```

```

        [self.buttonReset setEnabled: YES];
    }
    return NO;
}
return YES;
}

```

Como hemos comentado, en Android esto se puede hacer de una forma bastante más elegante, ya que dicha plataforma nos permite vincular objetos Java a la web como objetos Javascript. Primero crearemos un objeto Java con los métodos y propiedades que queramos exponer en la web, por ejemplo:

```

class JavascriptCallbackInterface {
    public void temporizadorFinalizado() {
        runOnUiThread(new Runnable() {
            public void run() {
                final Button bReset = (Button)findViewById(R.id.bReset);
                bReset.setEnabled(true);
            }
        });
    }
}

```

Nota

En el código anterior utilizamos `runOnUiThread` porque cuando dicho código sea ejecutado desde Javascript no nos encontraremos en el hilo de la UI, de forma que no podremos modificarla directamente.

Una vez tenemos el objeto Java, podemos vincularlo como interfaz Javascript, para así poder acceder a él desde la web. Esto lo haremos mediante el siguiente método:

```

wbVisor.addJavascriptInterface(
    new JavascriptCallbackInterface(), "webview");

```

Con esto estamos creando en la web un objeto Javascript de nombre `webview` que nos dará acceso al objeto Java que acabamos de definir. Ahora desde Javascript será muy sencillo llamar a un método de nuestra aplicación, simplemente llamando `window.webview.temporizadorFinalizado()`:

```

<script type="text/javascript">
    function finTemporizador() {
        window.webview.temporizadorFinalizado();
    }
</script>

```

De esta forma podemos acceder de forma muy sencilla desde los documentos web a objetos de nuestra aplicación.

Cuidado

Si dejamos el visor web abierto a que pueda navegar por páginas que no hemos creado nosotros, exponer estas interfaces puede resultar peligroso, ya que estaríamos dando a terceros acceso a

nuestra aplicación. Por lo tanto, deberemos utilizar esta característica sólo cuando tengamos controlado el contenido web que se vaya a mostrar.

7.4. PhoneGap

Con los métodos de comunicación anteriores podemos hacer llamadas desde componentes web a la aplicación nativa, lo cual nos permite que la web pueda acceder a características de los dispositivos a las que no tendríamos acceso si sólo contásemos con las características incluidas en HTML 5, como son el acelerómetro, la cámara, la base de datos local, etc. Esto lo han aprovechado *frameworks* como PhoneGap para permitirnos construir aplicaciones mediante HTML 5 y Javascript independientes de la plataforma, pero que se instalan como aplicaciones nativas y nos permiten acceder al hardware del dispositivo.

	 iOS iPhone / iPhone 3G	 iOS iPhone 3GS and newer	 Android	 OS 4.6-4.7	 OS 5.x	 OS 6.0+	 WebOS	 WP7	 Symbian	 Bada
ACCELEROMETER	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
CAMERA	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
COMPASS	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓
CONTACTS	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
FILE	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗
GEOLOCATION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MEDIA	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗
NETWORK	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (ALERT)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (SOUND)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (VIBRATION)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STORAGE	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗

Características soportadas por PhoneGap

PhoneGap nos ofrece una API Javascript que nos da acceso a características del dispositivo como las comentadas anteriormente, de forma que utilizando HTML 5 y dicha API podremos tener aplicaciones que funcionen en todos los dispositivos soportados por

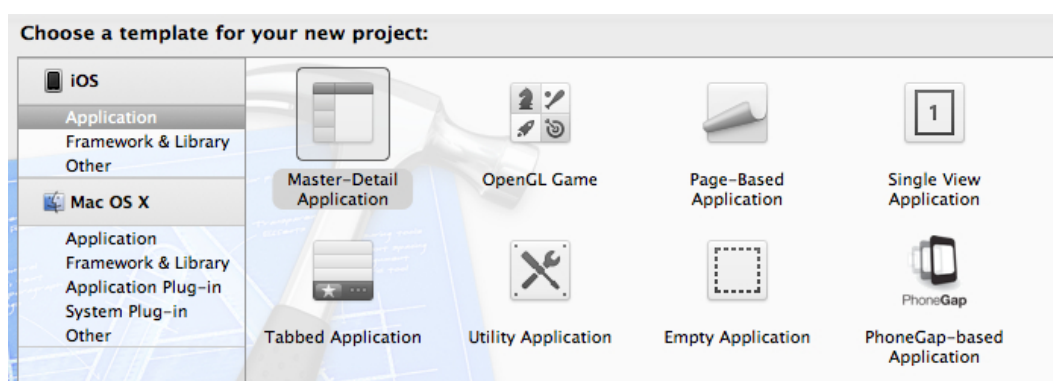
PhoneGap.

7.4.1. Instalación

Como hemos comentado, con PhoneGap podemos utilizar HTML 5 y Javascript para crear aplicaciones independientes de la plataforma. Sin embargo, para crear el paquete de la aplicación para cada plataforma que queramos soportar deberemos utilizar el entorno de desarrollo de dicha plataforma. Por ejemplo, para generar el paquete iOS deberemos utilizar Xcode, mientras que para generar el paquete para Android deberemos crear un proyecto de aplicación Android en Eclipse. Vamos a ver ahora cómo crear estos proyectos contenedores para las plataformas Android e iOS.

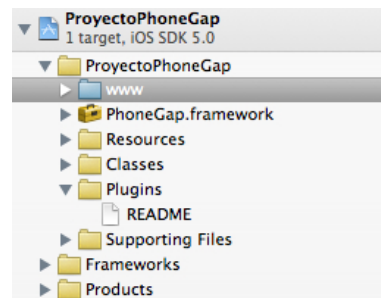
Si descargamos PhoneGap (<http://phonegap.com/>) y descomprimos el fichero, veremos que habrá creado un subdirectorio para cada plataforma. En estos directorios encontraremos los recursos necesarios para realizar la instalación de cada una de ellas.

En el caso de iOS, contamos con un instalador que nos añade un nuevo tipo de plantilla al asistente para crear un nuevo proyecto. En el directorio `ios` encontramos un fichero DMG para la instalación en MacOS de PhoneGap. Tras ejecutar el instalador veremos que en el asistente para crear un nuevo proyecto de Xcode aparece la opción *PhoneGap-based Application* en la sección *iOS > Applications*:



Asistente para crear aplicación PhoneGap

Para crear una aplicación PhoneGap en iOS deberemos crear un proyecto de ese tipo. Una vez creado, para terminar de configurarlo deberemos abrir con el Finder el directorio en el que se ha creado el proyecto, y arrastrar el directorio `www` que encontraremos ahí sobre nuestro proyecto en Xcode, y añadirlo como carpeta (no como grupo). Ahora podremos ejecutar el proyecto en el simulador y deberá funcionar correctamente.



Estructura del proyecto PhoneGap

Todo el contenido de la aplicación lo deberemos crear en el directorio `www` que hemos añadido al proyecto.

En el caso de Android deberemos crear un proyecto de aplicación Android manualmente e incluir en él los componentes necesarios de PhoneGap. Estos componentes los encontraremos en el subdirectorio `Android` del fichero que hemos descargado de PhoneGap (los componentes que nos interesan son `phonegap.js`, `phonegap.jar`, y el directorio `xml`). Deberemos seguir los siguientes pasos:

- El proyecto deberá estar dirigido a la versión 2.2 de Android.
- Crearemos en el proyecto los directorios `/assets/www` y `libs`.
- Copiar el fichero `phonegap.js` a `/assets/www`.
- Copiar el fichero `phonegap.jar` a `libs`. Añadiremos el JAR al *build path* del proyecto, pulsando sobre el JAR con el botón derecho, y seleccionando *Build Path > Add to Build Path*.
- Copiar el directorio `xml` al directorio `res` del proyecto.
- Hacer que la actividad principal herede de `DroidGap` en lugar de `Activity`.
- Sustituir en dicha actividad la llamada a `setContentView()` por `super.loadUrl("file:///android_asset/www/index.html");`
- Organizar los *imports* de la actividad, para añadir los que falten y eliminar los sobrantes.
- En el `AndroidManifest.xml`, añadir los siguientes permisos:

```
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:resizeable="true"
    android:anyDensity="true"
/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
```



```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.BROADCAST_STICKY" />
<uses-permission
    android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
```

- Añadir el atributo `android:configChanges="orientation|keyboardHidden"` a la actividad principal en `AndroidManifest.xml`
- Añadir la declaración de una nueva actividad a `AndroidManifest.xml`:

```
<activity
    android:name="com.phonegap.DroidGap"
    android:label="@string/app_name"
    android:configChanges="orientation|keyboardHidden">
    <intent-filter></intent-filter>
</activity>
```

- Crear en `assets/www` un fichero `index.html` como página principal de la aplicación. Debemos importar la librería Javascript de PhoneGap:

```
<!DOCTYPE HTML>
<html>
<head>
<title>PhoneGap</title>
<script type="text/javascript" charset="utf-8" src="phonegap.js"></script>
</head>
<body>
<h1>Hola Mundo</h1>
</body>
</html>
```

Una vez hecho esto, podremos ejecutar el proyecto en el simulador y veremos la página creada en el último paso. En esta página podremos utilizar la API de PhoneGap para acceder al hardware del dispositivo.

7.4.2. Eventos

Para comenzar a utilizar la API de PhoneGap, lo fundamental es conocer su modelo de eventos. Los eventos se tratan mediante un *listener*, que se implementará mediante una función *callback* (será una función Javascript que deberemos crear nosotros con el código que dé respuesta al evento). Para registrar esta función *callback* como *listener* deberemos llamar a la siguiente función, proporcionando como primer parámetro el nombre del evento que queremos escuchar, y como segundo parámetro nuestra función *callback*:

```
document.addEventListener("nombre_evento", funcionCallback, false);
```

La función anterior no forma parte de PhoneGap, esta es la forma estándar de tratar eventos en Javascript. Lo que sí que formará parte de PhoneGap son los tipos de eventos que podemos registrar. El evento básico que siempre deberemos tratar es "deviceready":

```
document.addEventListener("deviceready", onDeviceReady, false);
function onDeviceReady() {
```

```
// Inicializar aplicación PhoneGap
}
```

El evento anterior es el punto en el que inicializaremos la aplicación PhoneGap. Esto se hace así porque es posible que el código Javascript de PhoneGap se inicialice antes que la parte nativa de la librería. Por ese motivo nunca deberemos utilizar la API de PhoneGap antes de que se produzca este evento, ya que los componentes nativos necesarios podrían no estar inicializados todavía.

Este evento es básico para el funcionamiento de cualquier aplicación PhoneGap. También encontramos otros eventos que nos pueden resultar de utilidad para recibir notificaciones de cambios producidos en el dispositivo:

- Eventos para controlar el ciclo de vida de la aplicación, que nos indiquen cuando pasa a segundo plano y cuando se reanuda: `pause` y `resume`.
- Eventos que nos indican el estado de la conexión, para así saber cuándo perdemos la conexión y cuando la recuperamos: `online` y `offline`.
- Eventos sobre el estado de la batería, que nos informan sobre cualquier cambio de estado de la misma, o cuándo el nivel es bajo o crítico: `batterystatus`, `batterylow` y `batterycritical`.
- Eventos para recibir notificaciones de pulsaciones en los distintos botones que incorporan los dispositivos. Dependiendo de la plataforma tendremos disponibles unos u otros, por lo que deberemos dar alternativas para que cualquier funcionalidad pueda ejecutarse en cualquier dispositivo: `backbutton`, `menubutton`, `searchbutton`, `startcallbutton`, `endcallbutton`, `volumeupbutton`, y `volumedownbutton`.

7.4.3. Uso de la API

Vamos a ver a continuación algunas de las funcionalidades que nos ofrece PhoneGap a modo de ejemplo, y la forma de utilizarlas.

Por ejemplo, una funcionalidad sencilla es la que nos da acceso a las notificaciones, que nos permite mostrar alertas en pantalla, hacer sonar una alarma, o que vibre el dispositivo. Todo esto lo encontramos en el objeto `navigator.notification`:

```
function mostrarAviso() {
    navigator.notification.alert(
        'Reunión a las 10:30', // Mensaje
        callback,              // Funcion de callback al pulsar
        botón,                  // Botón
        'Aviso',                // Título
        'Cerrar'                // Botón
    );
    navigator.notification.beep(3);
    navigator.notification.vibrate(2000);
}
```

Otra funcionalidad a la que podemos acceder es a la agenda de contactos. Si queremos mostrar nuestros contactos en la aplicación, deberemos hacerlo en el evento `deviceready`, ya que como hemos comentado anteriormente no podemos confiar en que

los componentes necesarios se hayan inicializado antes de producirse dicho evento. Por ejemplo, podemos mostrar la lista de contactos de la siguiente forma:

```
document.addEventListener("deviceready", onDeviceReady, false);

function onDeviceReady() {
    var options = new ContactFindOptions();
    var fields = ["displayName", "name"];
    navigator.contacts.find(fields, onSuccess, onError, options);
}

function onSuccess(contacts) {
    for (var i=0; i<contacts.length; i++) {
        // Mostrar contacto en el documento
        ...
    }
}

function onError(contactError) {
    // Mostrar mensaje de error
}
```

Para ver la documentación completa de la API podemos dirigirnos a la página de documentación de PhoneGap, donde encontramos todas las clases, funciones, y ejemplos de uso de cada una de ellas: <http://docs.phonegap.com>.

Nota

Hemos de destacar que la API de PhoneGap se encarga de proporcionarnos acceso al hardware del dispositivo, pero no de la creación de la interfaz. Para crear la interfaz de nuestra aplicación PhoneGap podemos utilizar HTML 5 o cualquier *framework* construido sobre dicha tecnología, como **Sencha Touch**, **jQueryMobile**, o **GWT**.

8. Ejercicios de visor web

8.1. Integración de contenido web

Vamos a crear una aplicación que integre contenido web. Debemos decidir, en función del contenido con el que contamos, la forma más adecuada de integrarlo. Partiremos de la plantilla `TituloSEPS`, que nos dará información sobre las titulaciones ofrecidas por la politécnica. En la aplicación tenemos varias pestañas, una por cada titulación que ofrece dicha escuela. Vamos a mostrar en ellas una serie de visores web con la ficha de cada título (en Android en lugar de pestañas tenemos un menú de opciones). Se pide:

a) Las páginas que debemos mostrar son las siguientes:

Estudio		URL
Ingeniería en Informática		<code>http://cv1.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C203</code>
Ingeniería Telecomunicaciones	en	<code>http://cv1.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C201</code>
Ingeniería Multimedia		<code>http://cv1.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C205</code>

¿Qué tipo de escalado será más adecuado para mostrar estas páginas en un dispositivo móvil? Implementalo en la aplicación y comprueba que se visualizan correctamente.

Ayuda iOS

Cargaremos la URL (`self.url`) en el visor web en el método `viewDidLoad` del controlador `UINavigationController`. Establece ahí el tipo de escalado del visor.

Ayuda Android

Trabajaremos con la actividad `TituloSEPSActivity`. En `onCreate` inicializaremos el visor: estableceremos el tipo de escalado, activaremos Javascript (muy importante para las páginas de la UA), y vincularemos el progreso de carga del visor web con la barra de progreso de la actividad. Tras esta configuración cargaremos la URL de Informática en el visor. En `onOptionsItemSelected` cargaremos en el visor la URL asociada a la opción del menú que se haya pulsado.

b) Tenemos una cuarta pestaña con ayuda sobre la aplicación. En este caso la URL donde tenemos el documento a mostrar no es remota, sino que debemos hacer referencia a un fichero dentro de nuestra aplicación. Debemos acceder al fichero `/www/index.html`, que en Android estará en el directorio de `assets`, y en iOS estará en el raíz del `bundle` principal.

Ayuda iOS

En el método `application: didFinishLaunchingWithOptions:` de `UAppDelegate`, asignar a la propiedad `controllerAcerca.url` una URL que dé acceso al recurso local `/www/index.html`.

Ayuda Android

En `TitulosEPSActivity` hay que dar un valor a la constante `URL_ABOUT`, para que haga referencia al recurso `/www/index.html` ubicado en el directorio de `assets` de la aplicación.

Este documento ha sido creado específicamente para mostrarlo como ayuda de nuestra aplicación móvil. ¿Qué escalado resultará más apropiado en este caso? Implementalo de forma adecuada y comprueba que funciona correctamente.

8.2. Temporizador web

Vamos a implementar un temporizador web integrado en nuestra aplicación. En nuestra aplicación nativa tendremos una pantalla que constará de un `webView` y un botón con el que poner en marcha el temporizador. Mientras el temporizador esté funcionando dicho botón debe quedar deshabilitado, y cuando el temporizador termine volveremos a habilitarlo. Partiremos de la plantilla `TemporizadorWeb`, que contiene la pantalla nativa y el código Javascript del temporizador. Se pide:

a) Hacer que al pulsar el botón nativo se ponga en marcha el temporizador. Para ello, en el evento del botón se deberá llamar a la función Javascript `inicializaTemporizador` de nuestro visor. También haremos que el botón se deshabilite al producirse dicho evento.

Ayuda iOS

En el método `inicializarTemporizador:` del controlador `UAViewController`, llamaremos a la función Javascript `inicializaTemporizador()`.

Ayuda Android

En la actividad `VisorWebActivity`, en el evento del botón llamaremos a la función Javascript `inicializaTemporizador()`.

b) Hacer que el botón vuelva a habilitarse al finalizar el temporizador. Para ello, en la función Javascript `actualizaContador` deberemos notificar a la aplicación nativa que el temporizador ha terminado, y cuando la aplicación nativa reciba dicha notificación deberá volver a habilitar el botón.

Ayuda iOS

Desde la función `actualizaContador` del Javascript enviaremos una notificación de temporizador finalizado a la aplicación, y dicha notificación deberá recibirse en el método

```
webView: shouldStartLoadWithRequest: navigationType: del controlador  
UINavigationController.
```

Ayuda Android

En `onCreate` crearemos una interfaz Javascript con un objeto nativo de tipo `JavascriptCallbackInterface`, al que se referenciará en Javascript mediante una variable de nombre `webview`. En la función `actualizaContador` de Javascript llamaremos a la función `temporizadorFinalizado()` de dicha interfaz cuando termine el contador.

8.3. Alarma con PhoneGap

Con un código Javascript similar al del ejercicio anterior, vamos a implementar una aplicación PhoneGap que nos permita programar un temporizador y que cuando el tiempo llegue a su fin suene la alarma y vibre el dispositivo. Podemos partir de la plantilla `AlarmaPhoneGap`, que en este caso no contiene ningún proyecto Android ni iOS, sino que simplemente contiene un directorio `www` que podremos agregar a cualquier proyecto PhoneGap. Se pide:

- a) Crea un proyecto iOS de tipo PhoneGap, e incluye en él el directorio `www`. Es importante que dicho directorio quede incluido como carpeta, no como grupo. Es decir, `www` debe aparecer con un icono de carpeta azul, no amarilla.
- b) Implementa en la función Javascript `temporizadorFinalizado` el código PhoneGap necesario para que muestre una alerta en pantalla, suene una alarma, y vibre el dispositivo.
- c) Porta la aplicación PhoneGap a Android y comprueba que funciona correctamente.

