Visor web

Índice

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 **appMovi** (`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.

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.

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 principar 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];
}
```

1.2. WebView en Android

Al igual que en iOS, en Android el visor web es un tipo de vista (`View`) que podemos

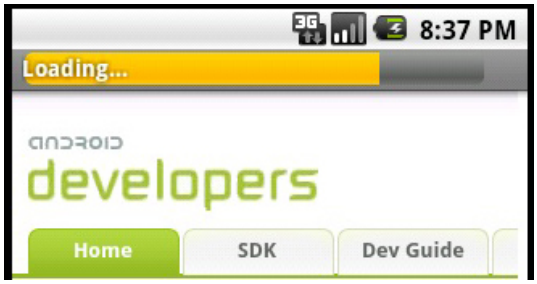
incluir en nuestro *layout*. En este caso se representa mediante la clase `WebView`, y en nuestro código podemos cargar una página llamando a su método `loadUrl` (un lugar típico para hacer esto es el método `onCreate` de la actividad que lo gestiona):

```
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.

2. Propiedades del visor

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 establecer 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 flanja 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);
```

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 pertencen 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];
}
```

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.

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.

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:(UIWebViewNavigationType)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.

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

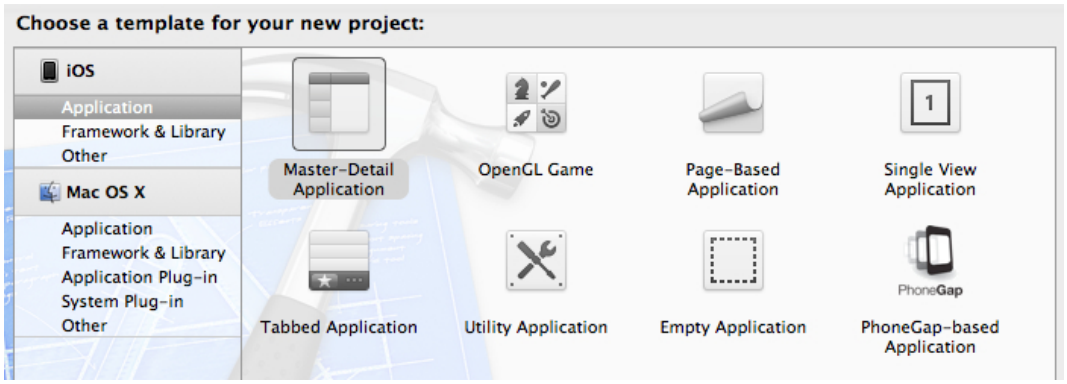
PhoneGap nos ofrece una API Javascript que nos da acceso a características del dispositivo como las comentadas anteriormente, de forma que utilizando HTML 5 y dicha API podremos tener aplicaciones que funcionen en todos los dispositivos soportados por PhoneGap.

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.

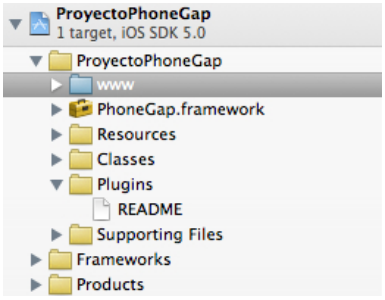
Si descargamos PhoneGap (`http://phonegap.com/`) y descomprimimos el fichero, veremos que habrá creado un subdirectorio para cada plataforma. En estos directorios encontraremos los recursos necesarios para realizar la instalación de cada una de ellas.

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. Deberemos 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.

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 si 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: `baterystatus`, `batterylow` y `batterycritital`.
- 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`.

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
        '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**.