Building Web Apps in WebView

Quickview

- Use `WebView` to display web pages in your Android application layout
- You can create interfaces from your JavaScript to your client-side Android code

In this document

Key classes

- `WebView`
- `WebSettings`
- `WebViewClient`

If you want to deliver a web application (or just a web page) as a part of a client application, you can do it using `WebView`. The `WebView` class is an extension of Android's `View` class that allows you to display web pages as a part of your activity layout. It does *not* include any features of a fully developed web browser, such as navigation controls or an address bar. All that `WebView` does, by default, is show a web page.

A common scenario in which using `WebView` is helpful is when you want to provide information in your application that you might need to update, such as an end-user agreement or a user guide. Within your Android application, you can create an `Activity` that contains a `WebView`, then use that to display your document that's hosted online.

Another scenario in which `WebView` can help is if your application provides data to the user that always requires an Internet connection to retrieve data, such as email. In this case, you might find that it's easier to build a `WebView` in your Android application that shows a web page with all the user data, rather than performing a network request, then parsing the data and rendering it in an Android layout. Instead, you can design a web page that's tailored for Android devices and then implement a `WebView` in your Android application that loads the web page.

This document shows you how to get started with `WebView` and how to do some additional things, such as handle page navigation and bind JavaScript from your web page to client-side code in your Android application.

Adding a WebView to Your Application

To add a `WebView` to your Application, simply include the `<WebView>` element in your activity layout. For example, here's a layout file in which the `WebView` fills the screen:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView  xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

To load a web page in the `WebView`, use `loadUrl()`. For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.loadUrl("http://www.example.com");
```

Before this will work, however, your application must have access to the Internet. To get Internet access, request the `INTERNET` permission in your manifest file. For example:

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

That's all you need for a basic `WebView` that displays a web page.

Using JavaScript in WebView

If the web page you plan to load in your `WebView` use JavaScript, you must enable JavaScript for your `WebView`. Once JavaScript is enabled, you can also create interfaces between your application code and your JavaScript code.

Enabling JavaScript

JavaScript is disabled in a `WebView` by default. You can enable it through the `WebSettings` attached to your `WebView`. You can retrieve `WebSettings` with `getSettings()`, then enable JavaScript with `setJavaScriptEnabled()`.

For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

`WebSettings` provides access to a variety of other settings that you might find useful. For example, if you're developing a web application that's designed specifically for the `WebView` in your Android application, then you can define a custom user agent string with `setUserAgentString()`, then query the custom user agent in your web page to verify that the client requesting your web page is actually your Android application.

Binding JavaScript code to Android code

When developing a web application that's designed specifically for the `WebView` in your Android application, you can create interfaces between your JavaScript code and client-side Android code. For example, your JavaScript code can call a method in your Android code to display a `Dialog`, instead of using JavaScript's `alert()` function.

To bind a new interface between your JavaScript and Android code, call `addJavascriptInterface()`, passing it a class instance to bind to your JavaScript and an interface name that your JavaScript can call to access the class.

For example, you can include the following class in your Android application:

```
public class WebAppInterface {
    Context mContext;

    /** Instantiate the interface and set the context */
    WebAppInterface(Context c) {
        mContext = c;
    }

    /** Show a toast from the web page */
    @JavascriptInterface
    public void showToast(String toast) {
        Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
    }
}
```

> **Caution:** If you've set your `targetSdkVersion` to 17 or higher, **you must add the `@JavascriptInterface` annotation** to any method that you want available to your JavaScript (the method must also be public). If you do not provide the annotation, the method is not accessible by your web page when running on Android 4.2 or higher.

In this example, the `WebAppInterface` class allows the web page to create a `Toast` message, using the `showToast()` method.

You can bind this class to the JavaScript that runs in your `WebView` with `addJavascriptInterface()` and name the interface `Android`. For example:

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.addJavascriptInterface(new WebAppInterface(this), "android");
```

This creates an interface called `Android` for JavaScript running in the `WebView`. At this point, your web application has access to the `WebAppInterface` class. For example, here's some HTML and JavaScript that creates a toast message using the new interface when the user clicks a button:

```
<input type="button" value="Say hello" onClick="showAndroidToast('Hello Android!')" />

<script type="text/javascript">
    function showAndroidToast(toast) {
        Android.showToast(toast);
    }
</script>
```

There's no need to initialize the `Android` interface from JavaScript. The `WebView` automatically makes it available to your web page. So, at the click of the button, the `showAndroidToast()` function uses the `Android` interface to call the `WebAppInterface.showToast()` method.

> **Note:** The object that is bound to your JavaScript runs in another thread and not in the thread in which it was constructed.

> **Caution:** Using `addJavascriptInterface()` allows JavaScript to control your Android application. This can be a very useful feature or a dangerous security issue. When the HTML in the `WebView` is untrustworthy (for example, part or all of the HTML is provided by an unknown person or process), then an attacker can include HTML that executes your client-side code and possibly any code of the attacker's choosing. As such, you should not use `addJavascriptInterface()` unless you wrote all of the HTML and JavaScript that appears in your `WebView`. You should also not allow the user to navigate to other web pages that are not your own, within your `WebView` (instead, allow the user's default browser application to open foreign links—by default, the user's web browser opens all URL links, so be careful only if you handle page navigation as described in the following section).

Handling Page Navigation

When the user clicks a link from a web page in your `WebView`, the default behavior is for Android to launch an application that handles URLs. Usually, the default web browser opens and loads the destination URL. However, you can override this behavior for your `WebView`, so links open within your `WebView`. You can then allow the user to navigate backward and forward through their web page history that's maintained by your `WebView`.

To open links clicked by the user, simply provide a `WebViewClient` for your `WebView`, using `setWebViewClient()`. For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient());
```

That's it. Now all links the user clicks load in your `WebView`.

If you want more control over where a clicked link load, create your own `WebViewClient` that overrides the `shouldOverrideUrlLoading()` method. For example:

```
private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        if (Uri.parse(url).getHost().equals("www.example.com")) {
            // This is my web site, so do not override; let my WebView load the page
            return false;
        }
        // Otherwise, the link is not for a page on my site, so launch another Activity that handles URLs
        Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
        startActivity(intent);
        return true;
    }
}
```

Then create an instance of this new `WebViewClient` for the `WebView`:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new MyWebViewClient());
```

Now when the user clicks a link, the system calls `shouldOverrideUrlLoading()`, which checks whether the URL host matches a specific domain (as defined above). If it does match, then the method returns false in order to *not* override the URL loading (it allows the `WebView` to load the URL as usual). If the URL host does not match, then an `Intent` is created to launch the default Activity for handling URLs (which resolves to the user's default web browser).

Navigating web page history

When your `WebView` overrides URL loading, it automatically accumulates a history of visited web pages. You can navigate backward and forward through the history with `goBack()` and `goForward()`.

For example, here's how your `Activity` can use the device *Back* button to navigate backward:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    // Check if the key event was the Back button and if there's history
    if ((keyCode == KeyEvent.KEYCODE_BACK) && myWebView.canGoBack()) {
        myWebView.goBack();
        return true;
    }
    // If it wasn't the Back key or there's no web page history, bubble up to the default
    // system behavior (probably exit the activity)
    return super.onKeyDown(keyCode, event);
}
```

The `canGoBack()` method returns true if there is actually web page history for the user to visit. Likewise, you can use `canGoForward()` to check whether there is a forward history. If you don't perform this check, then once the user reaches the end of the history, `goBack()` or `goForward()` does nothing.