Accessing Web-based content in Mobile App

Android offers a variety of ways to present content to a user. To provide a user experience that's consistent with the rest of the platform, it's usually best to build a native app that incorporates framework-provided experiences, such as Android App Links or search. Additionally, you can use Google Play-based experiences, such as App Actions, where Google Play services is available. However, some apps might need increased control over the UI. In this case, a WebView is a good option for displaying trusted first-party content.

Figure 1 illustrates how you can provide access to your web pages from a browser or your own Android app. The WebView framework lets you specify viewport and style properties that make your web pages appear at the proper size and scale on all screen configurations for all major web browsers. You can define an interface between your Android app and your web pages that lets JavaScript in the web pages call APIs in your app, providing Android APIs to your web-based application.

However, don't develop an Android app as a means to view your website. Rather, the web pages you embed in your app must be designed specifically for that environment.

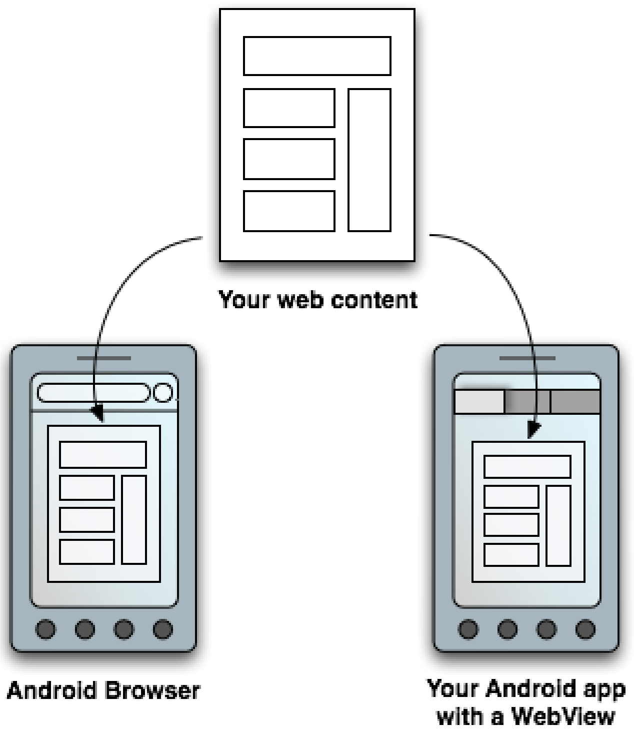

Figure 1. You can make your web content available to users in two ways: in a traditional web browser or in an Android application that includes a WebView in the layout.

Alternatives to WebView

Although WebView objects provide increased control over the UI, there are alternatives that might provide similar functionality with less configuration, faster loading and performance, improved privacy protections, and access to the browser's cookies.

Consider using these alternatives to WebView if your app falls into the following use cases:

- If you want to send users to a mobile site, build a progressive web app (PWA).
- If you want to display third-party web content, send an intent to installed web browsers.

- If you want to avoid leaving your app to open the browser, or you want to customize the browser's UI, use Custom Tabs.

Figure 2. Comparison of Chrome, Chrome Custom Tabs, and WebView.

Additional resources

To develop web pages for Android-powered devices using WebView objects, see the following documents:

- Build web apps in WebView
- Manage WebView objects
- Support different screens in web apps
- Debugg web apps
- Best practices for web apps

simillar functionality with less configuration, faster loading and performance, improved privacy protections, and access to the browser's cookies.

Consider using these alternatives to WebView if your app falls into the following use cases:

- If you want to send users to a mobile site, build a progressive web app (PWA).
- If you want to display third-party web content, send an intent to installed web browsers.
- If you want to avoid leaving your app to open the browser, or you want to customize the browser's UI, use Custom Tabs.

Figure 2. Comparison of Chrome, Chrome Custom Tabs, and WebView.

Additional resources

To develop web pages for Android-powered devices using WebView objects, see the following documents:

- Build web apps in WebView
- Manage WebView objects
- Support different screens in web apps
- Debugg web apps
- Best practices for web apps

Build web apps in WebView

bookmark_border

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 app that you might need to update, such as an end-user agreement or a user guide. Within your Android app, 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 app 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 app 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 app 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 app.

Add a WebView to your app

To add a WebView to your app, you can either include the <WebView> element in your activity layout, or set the entire Activity window as a WebView in onCreate().

Add a WebView in the activity layout

To add a WebViewto your app in the layout, add the following code to your activity's layout XML file:

```
<WebView
   android:id="@+id/webview"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
/>
```

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

KotlinJava

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

Add a WebView in onCreate()

To add a WebView to your app in an activity's onCreate() method instead, use logic similar to the following:

KotlinJava

```
val myWebView = WebView(activityContext)
setContentView(myWebView)
```

Then load the page with:

KotlinJava

```
myWebView.loadUrl("http://www.example.com")
```

Or load the URL from an HTML string:

KotlinJava

```
// Create an unencoded HTML string
// then convert the unencoded HTML string into bytes, encode
// it with Base64, and load the data.
val unencodedHtml =
    "<html><body>'%23' is the percent code for ‘#‘ </body></html>";
val encodedHtml = Base64.encodeToString(unencodedHtml.toByteArray(),
Base64.NO_PADDING)
myWebView.loadData(encodedHtml, "text/html", "base64")
```

Note: There are restrictions on what this HTML can do. See **loadData()** and **loadDataWithBaseURL()** for more info about encoding options.

Before this works, however, your app 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. Additionally, you can customize your WebViewby modifying the following:

- Enabling fullscreen support with WebChromeClient. This class is also called when a WebView needs permission to alter the host app's UI, such as creating or closing windows and sending JavaScript dialogs to the user. To learn more about debugging in this context, read Debugging Web Apps.
- Handling events that impact content rendering, such as errors on form submissions or navigation with WebViewClient. You can also use this subclass to intercept URL loading.
- Enabling JavaScript by modifying WebSettings.
- Using JavaScript to access Android framework objects that you have injected into a WebView.

Work with WebView on older versions of Android

To safely use more-recent WebView capabilities on the device your app is running on, add AndroidX Webkit. The androidx.webkit library is a static library you can add to your application in order to use android.webkit APIs that are not available for older platform versions.

Use JavaScript in WebView

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

Enable 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:

KotlinJava

```
val myWebView: WebView = findViewById(R.id.webview)
myWebView.settings.javaScriptEnabled = 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 app, 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 app.

Bind JavaScript code to Android code

When developing a web application that's designed specifically for the WebView in your Android app, 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 app:

KotlinJava

```
/** Instantiate the interface and set the context  */
class WebAppInterface(private val mContext: Context) {

    /** Show a toast from the web page  */
    @JavascriptInterface
    fun showToast(toast: String) {
        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, and the method must 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:

KotlinJava

```
val webView: WebView = findViewById(R.id.webview)
webView.addJavascriptInterface(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, when a user clicks 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 app. 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).

Handle page navigation

When the user clicks a link from a web page in your WebView, the default behavior is for Android to launch an app 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.

Note: For security reasons, the system's browser app doesn't share its application data with your app.

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

KotlinJava

```
val myWebView: WebView = findViewById(R.id.webview)
myWebView.webViewClient = WebViewClient()
```

All links the user clicks load in your WebView.

If you want more control over where a clicked link loads, create your own WebViewClient that overrides the shouldOverrideUrlLoading() method. The following example assumes that MyWebViewClient is an inner class of Activity.

KotlinJava

```
private class MyWebViewClient : WebViewClient() {

    override fun shouldOverrideUrlLoading(view: WebView?, url: String?): Boolean {
        if (Uri.parse(url).host == "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.ACTION_VIEW, Uri.parse(url)).apply {
            startActivity(this)
        }
        return true
    }
}
```

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

KotlinJava

```
val myWebView: WebView = findViewById(R.id.webview)
myWebView.webViewClient = 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).

Handle custom URLs

WebView applies restrictions when requesting resources and resolving links that use a custom URL scheme. For example, if you implement callbacks such as shouldOverrideUrlLoading() or shouldInterceptRequest(),
then WebView invokes them only for valid URLs.

For example, WebView may not call your shouldOverrideUrlLoading() method for links like this:

<a href="showProfile">Show Profile</a>

Invalid URLs like above are handled inconsistently in WebView, so we recommend using a well-formed URL instead, such as using a custom scheme or using an HTTPS URL for a domain that your organization controls.

Instead of using a simple string in a link as shown earlier, you can use a custom scheme such as the following:

<a href="example-app:showProfile">Show Profile</a>