Integrating server-generated pages in hybrid applications

Overview

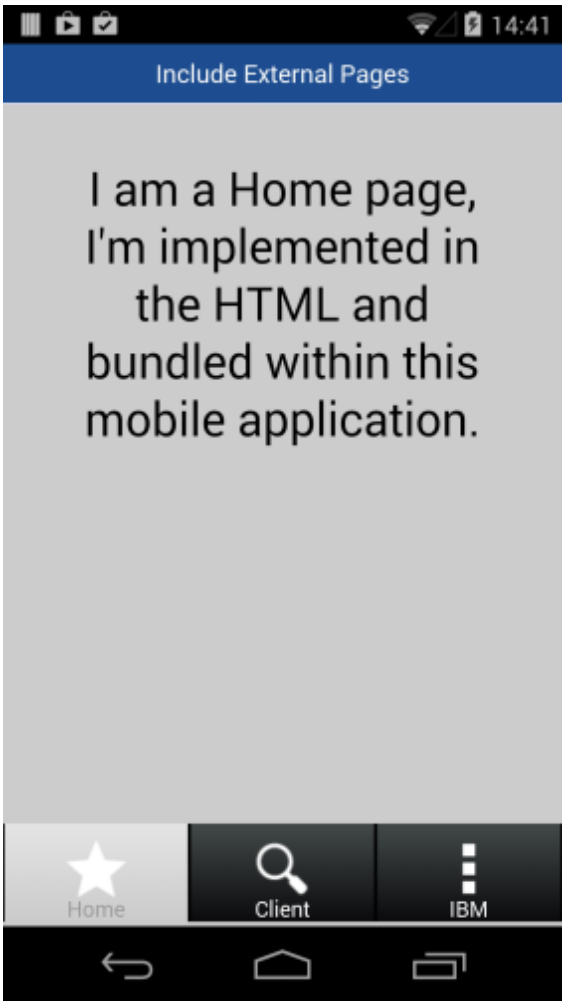
Many enterprises today decide to develop their own customer or employee-facing mobile applications. Some of those companies already have mobile-facing websites (mobile web).

An important decision must be made then by the companies:

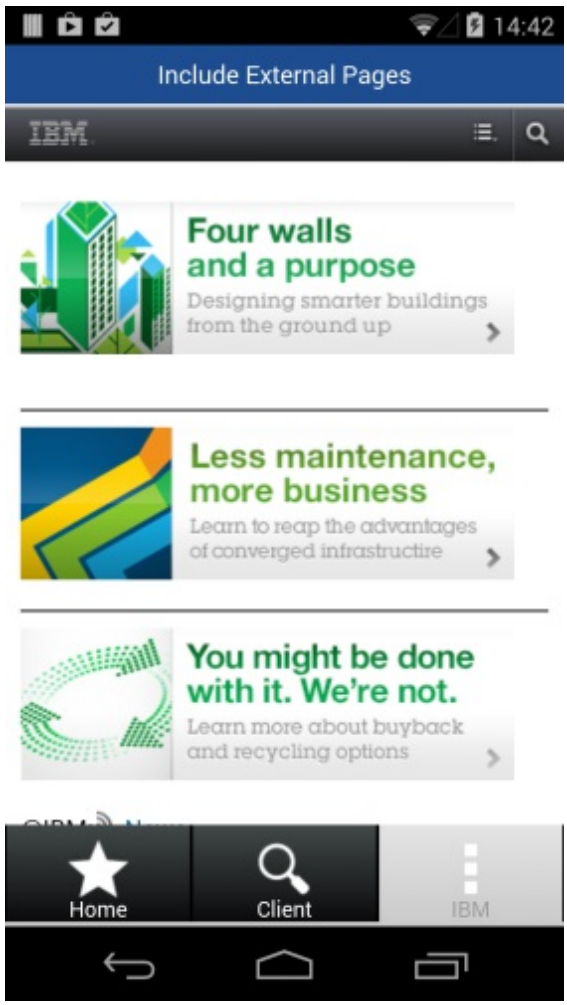
- Should all the mobile web features be implemented from scratch in the mobile application, which is great from a user experience perspective, but very time- and money-consuming?
- Should the mobile application contain only new features with old ones still accessible by a mobile browser, which is easier to implement but not great in terms of user experience?

The **WebViewOverlay** approach allows the re-use and integration of existing mobile websites within a mobile application.
Navigation is smooth and seamless between components that are internal in the mobile application, and the external content on the external mobile website.

Web resources that are bundled inside the application



External web content



In this tutorial, an application implementation is demonstrated for the Android environment.
The application contains three tab items.

The first two tabs contain internal content. The third tab shows an external IBM mobile website.

First and second tabs contain internal web resources

Third tab looks like another application page

But technically, it contains an extra WebView component on top of it

WebViewOverlay is implemented by using the Apache Cordova plug-ins functionality.
Developers can easily create their own protocol between internal web components and the **WebViewOverlay** control. The provided MobileFirst project can be to understand the concepts of **WebViewOverlay**.

Before proceeding, proficient with implementing Cordova plug-ins (../../adding-native-functionality/) is advised.

Java implementation

Implementing the webViewOverlay

First, in the application main class, a **webViewOverlay** object is declared, to be used to display the external content. Static references are used for simplicity.

```
public class IncludeExternalPages extends CordovaActivity implements WLInitWebFrameworkListener {
    private static WebView webViewOverlay;
    public static Activity thisapp;
```

The object is created and its layout properties are set, and it is added as a view to the root element. It is invisible initially.
Note the *setMargins* API. This API can be used to position your **webViewOverlay** component in the screen.

Note: Android 4.4 introduces a new version of WebView that is based on Chromium and that affects the WebView margins. More information about this issue can be found here: http://developer.android.com/guide/webapps/migrating.html (http://developer.android.com/guide/webapps/migrating.html)

```
public void onInitWebFrameworkComplete(WLInitWebFrameworkResult result){
    if (result.getStatusCode() == WLInitWebFrameworkResult.SUCCESS) {
        super.loadUrl(WL.getInstance().getMainHtmlFilePath());
        thisapp = this;
        WebViewClient webViewClient = new WebViewClient() {
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                view.loadUrl(url);
                return true;
            }
        };
        webViewOverlay = new WebView(this);
        webViewOverlay.setVisibility(View.INVISIBLE);
        webViewOverlay.setWebViewClient(webViewClient);
        RelativeLayout.LayoutParams webViewOverlayLayoutParams = new
        RelativeLayout.LayoutParams(
            RelativeLayout.LayoutParams.MATCH_PARENT,
            RelativeLayout.LayoutParams.MATCH_PARENT);
            webViewOverlayLayoutParams.setMargins(0, 120, 0, 196);
            webViewOverlay.setLayoutParams(webViewOverlayLayoutParams);
            webViewOverlay.getSettings().setJavaScriptEnabled(true);
        …
        …
```

Next, setting up the content view.

- A **RelativeLayout** object is created that functions as a root layout.
- The current root view is removed from its original parent. Instead, the **root** and **webViewOverlay** are added to the **rootRelativeLayout**.
- The content view is set to **rootRelativeLayout**.

```
public void onInitWebFrameworkComplete(WLInitWebFrameworkResult result){
    …
   webViewOverlay.getSettings().setJavaScriptEnabled(true);
   RelativeLayout rootRelativeLayout = new RelativeLayout(this);
   ((FrameLayout)root.getParent()).removeAllViews();
   rootRelativeLayout.addView(root);
   rootRelativeLayout.addView(webViewOverlay);
   setContentView(rootRelativeLayout);
    …
```

Implementing the Java code of the Cordova plug-in

In a new class, **WebViewOverlayPlugin.java**, the actions the plug-in supports are declared.

```
public class WebViewOverlayPlugin extends CordovaPlugin {
   private final String ACTION_OPEN_URL = "open";
   private final String ACTION_CLOSE_WEBVIEWOVERLAY = "close"
;
   …
```

If an “open” request was received from the web part of the application, load the external content and render the **webViewOverlay** visible.

```
public class WebViewOverlayPlugin extends CordovaPlugin {
   ...
@Override
public boolean execute(String action, JSONArray args, CallbackContext callbackContext) {
if (action.equals(ACTION_OPEN_URL)) {
IncludeExternalPages.thisapp.runOnUiThread(new Runnable() {
public void run() {
IncludeExternalPages.clearWebViewOverlayHistory();
IncludeExternalPages.loadWebViewOverlay("http://m.ibm.com/");
IncludeExternalPages.setWebViewOverlayVisibility(View.VISIBLE);
IncludeExternalPages.requestWebViewOverlayFocus();
IncludeExternalPages.clearWebViewOverlayHistory();
}
});
return true;
```

If receive a “close” request from the web part of the application, clean the **webViewOverlay** and hide it. UI-related actions are performed on a dedicated UI thread.

```
public class WebViewOverlayPlugin extends CordovaPlugin {
    …
    …
    } else if (action.equals(ACTION_CLOSE_WEBVIEWOVERLAY)) {
        IncludeExternalPages.thisapp.runOnUiThread(new Runnable() {
            public void run() {
                IncludeExternalPages.loadWebViewOverlayContent("");
                IncludeExternalPages.setWebViewOverlayVisibility(View.INVISIBLE
);
                IncludeExternalPages.clearWebViewOverlayHistory();
            }
        });
        return true;
    } else
        return false;
    }
}
```

JavaScript implementation

In JavaScript, a **WebViewOverlayPlugin** object is created and populated with the required methods.

```
function WebViewOverlayPlugin() {};
WebViewOverlayPlugin.prototype.open = function() {
    cordova.exec(null, null, 'WebViewOverlayPlugin', 'open', [])
;
};
WebViewOverlayPlugin.prototype.close = function() {
    cordova.exec(null, null, 'WebViewOverlayPlugin', 'close', [])
;
};
window.webViewOverlay = new WebViewOverlayPlugin();
```

The clicked tab ID is analyzed and either shows or hides the **webViewOverlay** accordingly.
Loading external content can take time, so adding a Busy Indicator to improve the user experience should be considered.

```
function tabClicked(id){
    WL.Logger.debug("tabClicked >> id :: " + id)
;
    $(".tab").addClass('hidden');
    if (id=="3"){
        openWebViewOverlay();
    } else {
        $("#tab" + id).removeClass('hidden');
        closeWebViewOverlay();
    }
}
function openWebViewOverlay(){
    WL.Logger.debug("openWebViewOverlay");
    window.webViewOverlay.open();
}
function closeWebViewOverlay(){
    WL.Logger.debug("closeWebViewOverlay");
    window.webViewOverlay.close();
}
```

Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v630/IncludeExternalPagesProject.zip) the Studio project.