Building The Android Wrapper

Version 1.3.0+

Contents

Introduction 2
Get The Project 2
Set Up The Development Environment 2
- Ensure You Have The Correct SDK Version Installed 4

Configuring The App (UPDATED) 5
- Behavior Settings (UPDATED) 5
- Connection Settings 7

Customizing The App 8
- Creating A Unique Package Name 8
- Changing The App's Display Name 8
- Changing The App Icon 9
- Changing The Splash Screen 9
- Removing Unnecessary Permissions 9
- Localize App 10
- Bundle SCAFs (offline apps only) 11
- Bundle Local Database 12
- Edit The About Screen (UPDATED) 12
  - Localize The About Screen 13
- Edit The Credits Screen (NEW) 14
- Change The Color Scheme (UPDATED) 14

Building The App 15
Deploying The App 15
- Manual Deployment 16
- Google Play Deployment 17

Introduction

In addition to using the Omnis JavaScript Client in the browser on any computer, tablet or mobile device, you can create standalone apps for Android that have your JavaScript remote form embedded. These can even operate completely offline (if you have a Serverless Client serial).

To do this, we provide a custom app, or "wrapper", project for Android. This project allows you to build custom apps, which create a thin layer around a simple Web Viewer which can load your JavaScript remote form. They also allow your form access to much of the device's native functionality, such as contacts, GPS, and camera.

This document describes the steps required in order to create and deploy your own customized wrapper app for Android. It should provide you with all of the information you need to create your own, self-contained, branded mobile app, and deploy it to users manually or through the Play Store.

Get The Project

- Download the Omnis Android JavaScript Wrapper project from our website.
- Extract the zip file to a location on your computer.

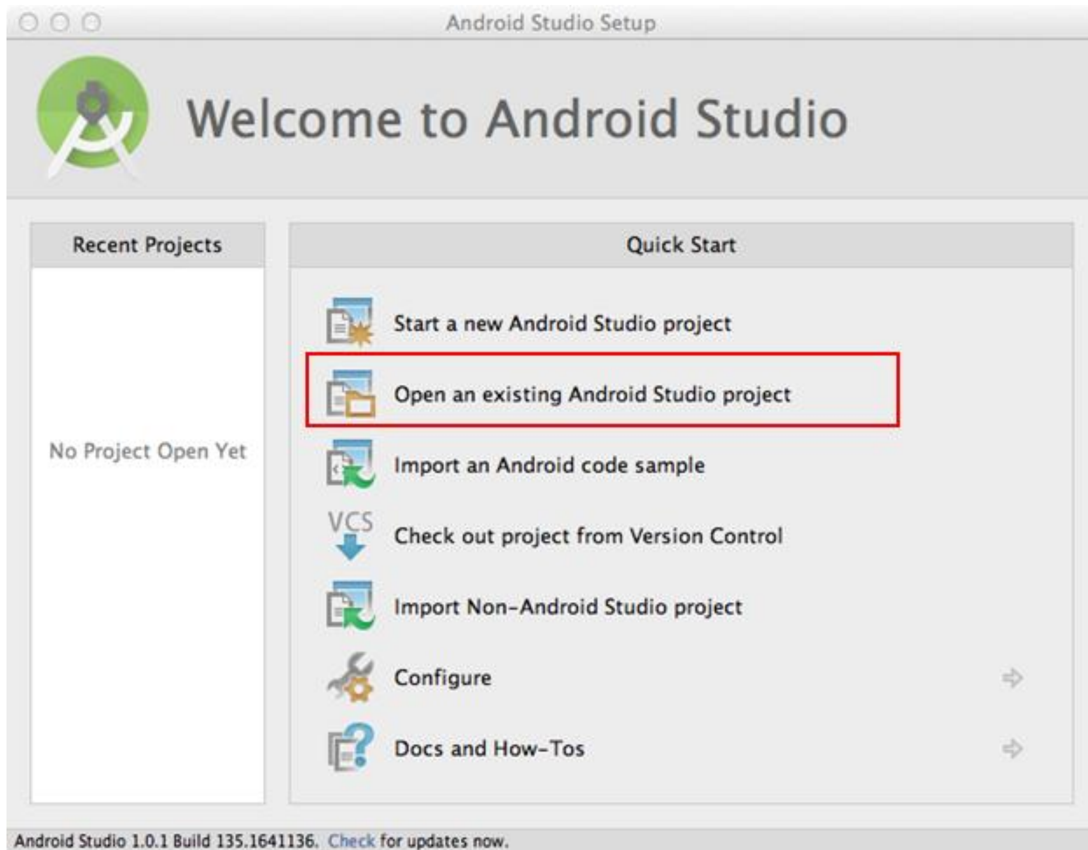
A bug in Android Studio (at the time of writing) means that if your project is located on a different drive to your Android Studio installation, the build process becomes very slow.

Set Up The Development Environment

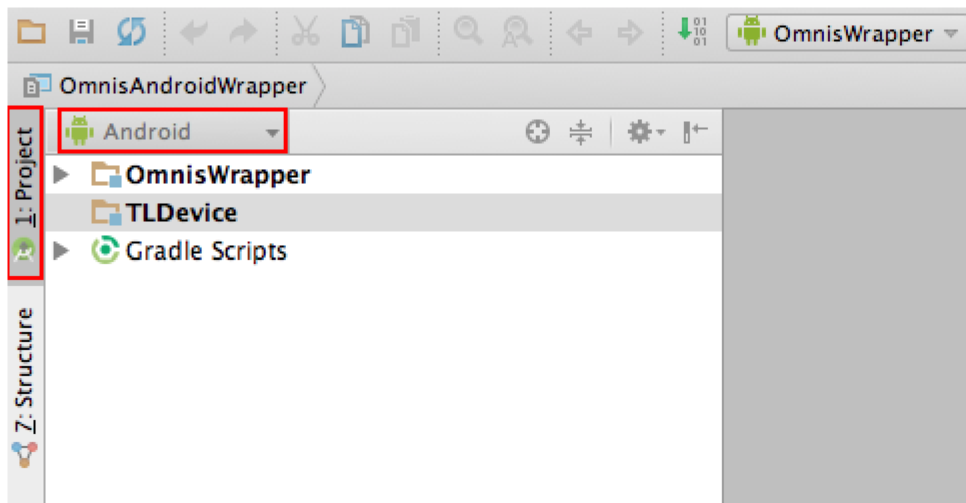
The Android wrapper is now built using **Android Studio**.

*Android wrappers prior to version **1.2.0** were built using Eclipse. If you wish to build one of these older projects, please refer to the documentation here.*

- Java **JDK 7** or later is needed to build the wrappers, so please ensure this is installed.
- Download and install **Android Studio** (available for Windows, Mac & Linux).
- Run Android Studio, and when you reach the Welcome Screen, select “**Open an existing Android Studio project**”



- Browse to the project you downloaded, and select the **OmnisAndroidWrapper** directory. The project will be opened in Android Studio.
- Once opened, Android Studio will attempt to build the project in the background (you should see a little spinner at the bottom of the window while this is in progress).
- Make sure you have the Android Project view open:



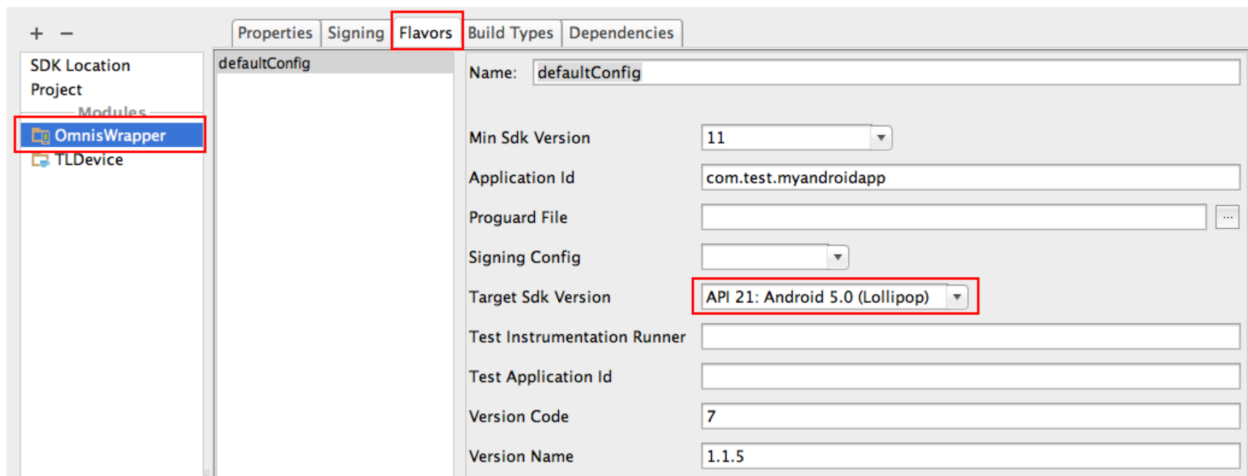
  - Along the left side of the screen, select **"Project"**.

- In the droplist at the top of the Project view, select **“Android”**.
- Your view should now look like that shown in the screenshot above, and this is the view you will want for all of your work with the wrapper.

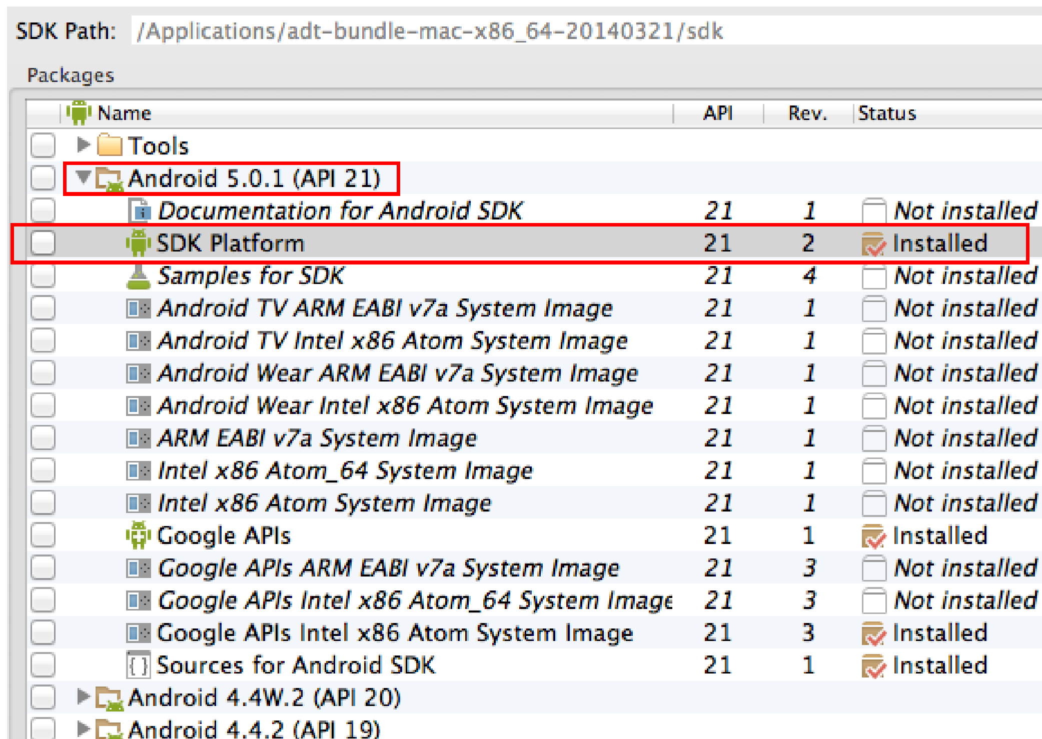
Ensure You Have The Correct SDK Version Installed

The Android app builds against a particular *Target* Android SDK version.
It’s important to note that this is not the minimum version of Android the app will run on, but instead allows the app to take advantage of features and theming introduced in later SDK versions.

- Go to **File -> Project Structure** and select the **OmnisWrapper** module in the window which opens.



- Select the **Flavors** tab, and here you can see the **Target Sdk Version** - make a note of this, then close the window.
- Open the **SDK Manager** (from the main toolbar, or **Tools -> Android -> SDK Manager**)
- Use the SDK manager to check that you have the **SDK Platform** for the target SDK version found above installed. If not, use the SDK Manager to install it.



Before going any further, make sure that you can build the project by selecting **Rebuild Project** from the build menu.

Configuring The App (UPDATED)

Configuration of the app is done through the **config.xml** file, which is situated in **OmnisWrapper/assets.** You should set the values in this config file to point the app to your Omnis server, and to configure how the app behaves.

The properties within the config file are as follows:

Behavior Settings (UPDATED)

- **AppTitle**: Whether the app should have an Action Bar title at the top. (1 for yes, 0 for no).

- **AppStandardMenu**: Whether the menu should be added to the Action Bar. Only applies if AppTitle=1. (1 for yes, 0 for no)

> **NOTE:** The menu provides a way for the user to access the **Credits** page. If you do not enable the menu, the onus is on you to ensure that the credits page is accessed in some way through your application.
> Licensing stipulations of various libraries used the wrapper mean that you must make this page accessible in your application.

- **AppTimeout**: If the app is sent to the background, this is the number of milliseconds it will wait before killing the app and freeing the connection to the server. If you set this to a negative number, it will never timeout (although the system may kill the app if it requires the resources).

It's worth noting that if you are making use of some of the device functionality (e.g. Camera, SMS etc), the app will be sent to the background while you are using the Camera/SMS apps etc. So you should be careful not to set this value too low, so that your app is not killed while the user is taking a photo, for example.

- **HardwareAccelerated (NEW)**: Determines whether the webview which runs the forms is hardware accelerated. When enabled, an issue with the Android webview means there may be issues with redrawing controls such as maps and graphs. This is the case with the latest Lollipop webview at the time of writing.
  However, hardware acceleration is required for the video control.

Unfortunately, due to the webview's issue with hardware acceleration, this becomes something of a juggling act. If you do not use video controls, we advise you to disable hardware acceleration. If you do use video controls, as well as maps/graphs, you may need to enable acceleration and live with the redraw issues until Google fix the webview.

- **MenuIncludeSettings**: If the menu is enabled, whether the Settings option is shown. (1 for yes, 0 for no). You will probably want to disable this for your released app.

- **MenuIncludeOffline**: If the menu is enabled, whether the option to switch between online & offline modes is shown. (1 for yes, 0 for no).

- **MenuIncludeAbout**: If the menu is enabled, whether the About option is shown. (1 for yes, 0 for no).
  - If the **About** option is enabled, the **Credits** screen will be accessible from the **About** screen.
  - If the **About** option is disabled, the **Credits** screen will be accessed as an option from the main menu.

- **SettingsFloatControls**: Whether controls on the form should float (using their designed $edgefloat property) to adapt to the device's full screen size. Does not apply if *SettingsScaleForm* is enabled. (1 for yes, 0 for no). **Recommended set to 1.**

- **SettingsScaleForm:** Whether the designed form should be scaled to fit the current device's screen. (1 for yes, 0 for no). **Recommended set to 0**.

- **SettingsAllowHScroll & SettingsAllowVScroll:** Set these to 1 if you want to allow horizontal or vertical scrolling of the form respectively, or 0 if not.

- **SettingsMaintainAspectRatio**: If *SettingsScaleForm* is set to 1, this controls whether the scaling maintains the design form's aspect ratio. 1 for yes, 0 for no.

- **SettingsOnlineMode**: Whether the app should initially start in online mode (set to 1), or offline mode (set to 0).

- **ServerLocalDatabaseName**: The name (including .db extension) of the local sqlite database to use. If you are bundling a prepopulated database with your app, its name should match what you set here.

- **UseLocalTime**: If 0, dates & times are converted to/from UTC, as default. Setting this to 1 will disable this conversion. (Offline only - online mode reads from remote task's $localtime property).

Connection Settings

- **ServerOmnisWebUrl**: URL to the Omnis or Web Server. If using the Omnis Server it should be *http://<ipaddress>:<omnis port>*. If using a web server it should be a URL to the root of your Web server. E.g. *http://myserver.com*

- **ServerOnlineFormName**: Route to the form's .htm file from ServerOmnisWebUrl. So if you're using the built in Omnis server, it will be of the form */jschtml/myform*. If you are using a web server, it will be the remainder of the URL to get to the form, e.g. */omnisapps/myform*. (**Do not add the .htm extension!**)

Only ServerOmnisWebUrl & ServerOnlineFormName are needed for Online forms. The other Server… properties are needed in addition to ServerOmnisWebUrl for Offline mode.

- **ServerOmnisServer**: The Omnis Server <IP Address>:<Port>. Only necessary if you are using a web server with the Omnis Web Server Plugin. If the Omnis App Server is running on the same machine as the web server, you can just supply a port here.
E.g. 194.168.1.49:5912

- **ServerOmnisPlugin**: If you are using a web server plug-in to talk to Omnis, the route to this from *ServerOmnisWebUrl*.
E.g. /cgi-bin/omnisapi.dll

- **ServerOfflineFormName:** Name of the offline form. (**Do not add .htm extension!**)

- **ServerAppScafName**: Name of the App SCAF. This will be the same as your library name.
Note: this is case-sensitive and must match the App Scaf (by default this is generally all lower-case).

- **TestModeEnabled**: Enable test mode (Ctrl-M on form from Studio to test on device).
Make sure to disable before publishing your release app. (1 to enable, 0 to disable)

- **TestModeServerAndPort**: The *<ipaddress>:<port>* of the Omnis Studio Dev version you wish to use test mode with.

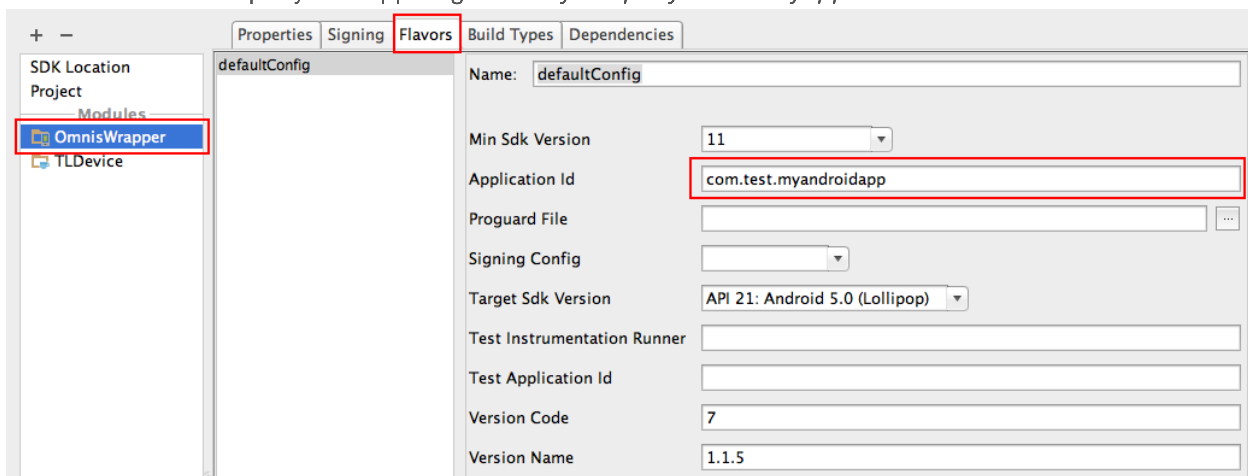
NOTE: The values in the config.xml file are currently read only on first launch of the app. They are then saved to, and read from, local storage to improve performance and allow in-app configuration from the Settings screen.
As such, if you make changes to the config.xml, you will need to uninstall the old app from your device before running the new version.

Customizing The App

Creating A Unique Package Name

The first step of customizing your app is to change the *Package Name*.
This is the unique ID for your app, and is what is used to identify it within the OS.
Two apps with the same package name will be seen by the device as the same app, so this is an important step.
As such you should use a reverse-domain-name style identifier to ensure you do not conflict with other apps.

- Go to **File -> Project Structure** and select the **OmnisWrapper** module in the window which opens.
- Select the **Flavors** tab and change the **Application Id** to your own unique package name.
    - You should use a reverse-domain-name qualifier, to ensure the id is unique to your company and app. E.g: *com.mycompany.omnis.myapp*.

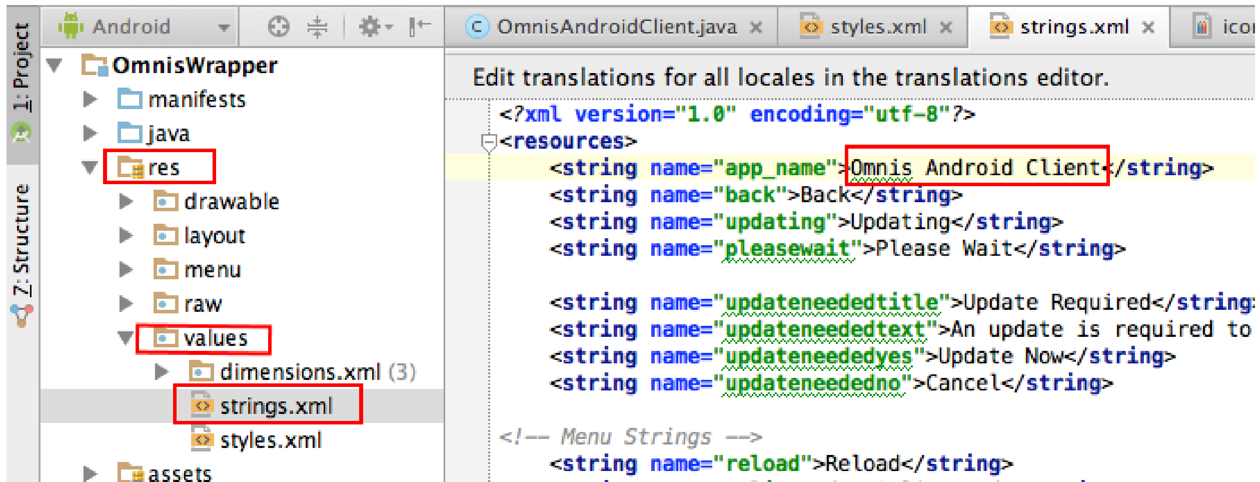


Changing The App's Display Name

The app's display name is defined in a strings resource file.

- In the Project View pane, drill down to **OmnisWrapper/res/values** and open **strings.xml** by double-clicking it.
- Within this file look for the xml tag with **name="app_name"**
- Change the *value* of this string, by changing the text inside the xml tags to whatever you wish your app to be named.

You can use a different app name for different locales - see the Localization section for more details on this.



Changing The App Icon

As Android devices are so wide-ranging in their displays, it is necessary to create several different resolutions of icons. The OS will then use the appropriate icon for the user's device.

- Browse to the Android Wrapper project's location on your file system.
- Drill down into **src/main/res** and note the **drawable-...dpi** folders.
- Each of these folders contains an **icon.png** file, sized correctly for devices classed as part of that dpi group.
- Edit or replace these files, making sure to keep the image sizes and file name the same.

Changing The Splash Screen

The Android wrapper displays a splash screen while it is loading (or reloading) the form.

This needs to be an image named *splashscreen* in the **res/drawable-...dpi** folders. The file extension can be just a standard .png file, or it could be a **9-Patch** (.9.png).

A 9-patch image is a png with special markers which control how the image is scaled. This allows you to avoid any horrible stretching as the image is scaled. This is our recommended format for splash screens on Android. Info on 9-Patches can be found here.

You may notice that the default project only contains splash screen images in two of the dpi folders. This is OK (especially if using 9-patch images) - the device should pick the closest available image to its dpi. We did this to keep the size of the app/project down.

Removing Unnecessary Permissions

Each Android App must request specific permissions to access various areas of the device - e.g. Contacts, Camera, Location etc.

It is bad practice to include unnecessary permissions for your app - especially if you are distributing through Google Play.
When downloading/installing your app, the user can see which permissions your app has requested access to. Unnecessary permissions may give the user the impression that your app is malicious.

- Open the project's ***AndroidManifest.xml*** file (from *OmnisWrapper/manifests*).
- This file lists the currently requested permissions in a group of **<uses-permission …>** tags.
- By default, all permissions which an Omnis app may use are present.
- Remove those permissions not needed by your app, by selecting the permission in the list, and deleting the line, or commenting it out using Ctrl-/ (or Cmd-/ for Mac).

MANDATORY PERMISSIONS:
- **INTERNET**

All other permissions may be removed if your particular app does not make use of their functionality.
The optional permissions you may require, depending on the functionality you use in your app, are:

- **CAMERA** - necessary to use barcode reader (*kJSDeviceActionGetBarcode*).
- **READ_CONTACTS** - necessary if you use the *kJSDeviceActionGetContacts* device action to access the device's contacts.
- **ACCESS_FINE_LOCATION** - provides fine grain (provided by GPS sensors) location data to the *kJSDeviceActionGetGps* device action.
- **ACCESS_COARSE_LOCATION** - provides coarse location (provided by network) data to the *kJSDeviceActionGetGps* device action.
- **WRITE_EXTERNAL_STORAGE** - necessary if you are obtaining images from the camera (*kJSDeviceActionTakePhoto*).
- **READ_EXTERNAL_STORAGE** - necessary if you are obtaining images from the camera or the device's saved images (*kJSDeviceActionTakePhoto* or *kJSDeviceActionGetImage*).
  Only enforced after Android 4.3
- **CALL_PHONE** - necessary in order to make phone calls from the app (kJSDeviceActionMakeCall).
- **VIBRATE** - necessary in order to make the device vibrate (kJSDeviceActionVibrate).

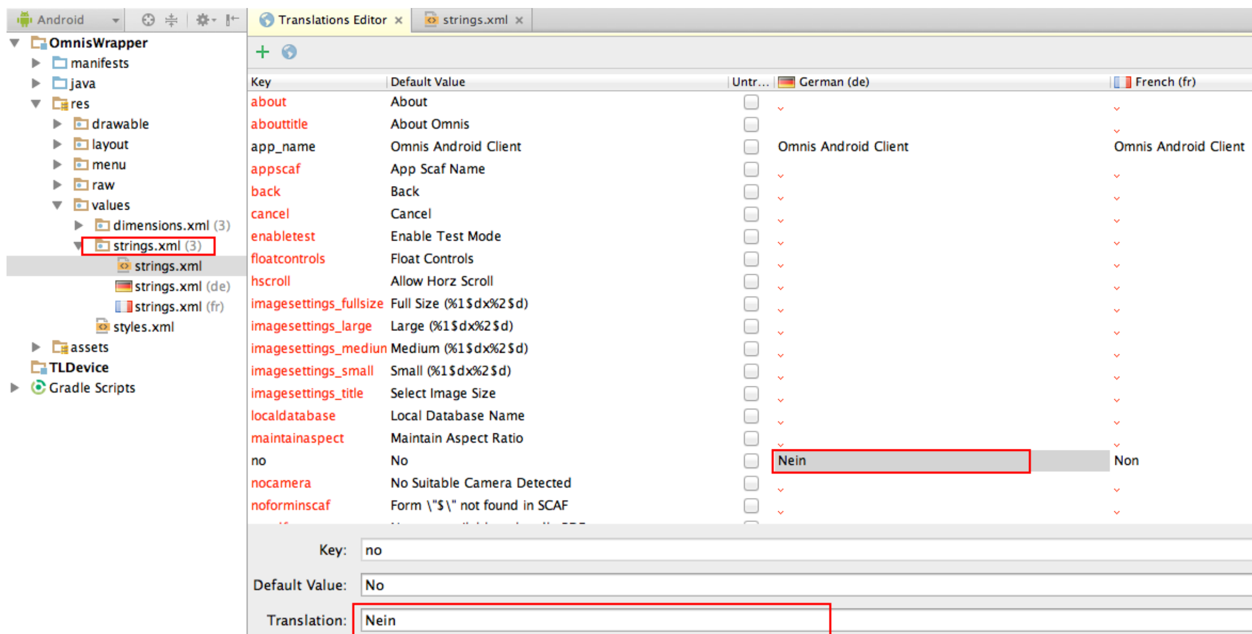
Localize App

If you wish to translate text used by the wrapper app, you can do so as described here. If the user's device is set to one of the supported languages, it will use any alternative translated text strings which have been specified.

- Locate your project’s **res/values/strings.xml** file in Android Studio.
- Right-click **strings.xml** and select **Open Translation Editor**.
- At the top of the Translations Editor window, click the Add Locale button (the globe icon), and select the locale(s) you wish to add a translation for.

At the time of writing, the Translations Editor does not refresh after adding a new locale, so close and re-open the Translations Editor window.

- Select the string you wish to translate for a particular locale in the table, and apply the translation, as shown in the image below.



- Any strings which haven't been translated to every locale will be highlighted in red. If a particular locale does not have a translation for a string, it will fall back to the (English) Default Value.

Some text strings contain placeholder sequences (e.g. "%1$d"). These should generally be maintained in your translations. Comments on most of these strings can be seen by viewing the original strings.xml source.

Bundle SCAFs (offline apps only)

If your app includes offline support, you need to decide whether or not to include the SCAFs inside your app. If you do so, the app will be larger, but it will run in offline mode immediately, with no need to first update from the server.

If you wish to include the SCAFs in your app, you should do the following:

- Browse to the **html/sc** folder of your Omnis Studio installation.
  - On Windows, this will be in the AppData area. e.g:*C:\Users\AppData\Local\TigerLogic\OS6.X*
- Locate your **App SCAF** (This will be a .db file in the root of the sc folder and will be named as your library).
- Also locate your **Omnis SCAF** (This will be the **omnis.db** file in *sc/omnis/*).
- Import both of these SCAF files into your Android project's **assets** directory.
  - The easiest way to do this is to copy them to your clipboard, then paste to the *assets* directory in Android Studio.

Bundle Local Database

It's possible to add a pre-populated SQLite database to use with your app. This will be used as the database which the form's $sqlobject connects to.

- Copy your SQLite .db file from your file system, into the **assets** folder of your project within Android Studio.
- Edit your project's config.xml file, and set the **<ServerLocalDatabaseName>** to the name of your local database (including the .db extension).

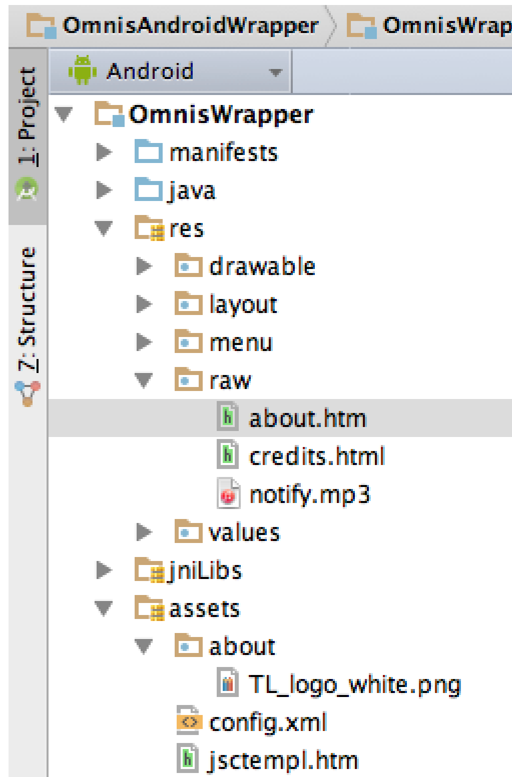
Bear in mind that you are creating a mobile application, and so should not be storing huge databases locally on the device.

To keep your data secure, the database is compiled into the apk. At the time of writing, Android .apk files submitted to the Play Store must be under 50MB, so this is another reason to keep the size of your local database down.
If you need to access data from a large database, it may make sense for you to hold the whole database on your Omnis server, and use the Sync Server functionality to synchronize a subset of this data with your device.
Details on the Sync Server can be found here.

Edit The About Screen (UPDATED)

If enabled in the config.xml, an **About** option is displayed in your app's menu. This will open a new screen which displays a html page which you can configure as you wish. It will also provide a link to the **Credits** screen.

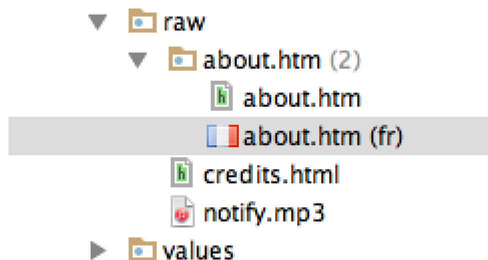
- To enable(/disable) the About menu option, edit your project's **config.xml** file, setting the value of **<MenuIncludeAbout>** to **1** (or **0** to disable it).
- Open your project's **res/raw** folder. This needs to contain a file named **about.htm**, which is the html page which will be loaded when opening the About screen.
  - This was previously stored in the *assets/about* folder, but was moved here to allow you to localize this page using **Resource Qualifiers**
- You can customize this however you wish, and add any resources it might need to the **assets/about** folder.



Localize The About Screen

In order to localize your About page for different languages etc, you need to make use of **Resource Qualifiers**.

- Locate the **res/raw** folder on your **File System**. You can right-click the **raw** folder and select **Reveal in Finder (/Show in Explorer)**.
- Create a new folder at the same level as the **raw** folder, and name it **raw-<language code>**. E.g. **raw-fr**.
    - This folder naming follows **Resource Qualifier** naming rules. See the **Language and region** section of the table on [this page]() for more information, and details on how to go further with region codes etc.
- Create an **about.htm** file in this folder, and populate it with your localized content for the particular language.
- Back in Android Studio, the **about.htm** file will now be represented as a folder in your Project view. If you open the folder, you will see the localized versions of the file, annotated with a flag and language code.



- To avoid polluting the raw directory, and as many of the resources you will need to link to in your About page will be shared across locales, any local resources loaded by your page should be put into **assets/about** (as shown by the TL_logo_white.png file in the screenshot in the *Edit The About Screen* section above).

*The **assets/about** folder is the working directory of the About page.*

Edit The Credits Screen (NEW)

If the **About** menu option is enabled in the config.xml, the About screen will have a link to the Credits screen in its Action Bar. Otherwise, a **Credits** option is displayed in your app's menu.

NOTE: The **Credits** page **MUST** be accessible from your app.

The **Credits** screen works in a similar way to the **About** screen - displaying the contents of the **credits.html** file from your project's **res/raw** folder. It can be localized in the same way as the About page.

You may add to or style this page if you like, but **you must include all of the included attributions**.
If you use any extra third-party libraries or resources, you should add your attributions to this page, otherwise, in most cases, it will be sufficient to leave this as it is.

Change The Color Scheme (UPDATED)

It's very simple to change the color scheme used by the native portions of the app (Action Bars, highlights etc), allowing you to drastically change the look and feel of the app.

- In Android Studio, open **res/values/colors.xml**
- This contains several **color…** items, whose values you can change to alter the colors used by the app.

You can edit the values by changing them inline, or by clicking on the color preview swatch in the left margin.

- The colors you can set here are as follows:
    - **primary**: Will be used as the color for the Action Bars' background.
    - **primary_dark**: Will be used to color the status bar (only on Android 5.0+ devices).
    - **primary_accent**: Will be used to tint selected native Android fields (E.g. checkboxes and edit fields in Settings).
    - **fab_normal**: The color of Floating Action Buttons (FABs) (e.g. in Settings screen) in their normal state.
    - **fab_pressed**: The color of FABs when pressed.
    - **fab_ripple**: The color of the FAB ripple effect which occurs when you click (Lollipop and later only).

By default, the app uses a dark theme. This means that backgrounds of dialogs, popup menus, and the Settings screen etc will be dark.
If you would prefer your app to use a light theme, you can do so by changing the **<style**… tag to:

<style name="AppTheme" parent="Theme.AppCompat.Light">

This will also cause the color of the text on your Action Bar to become black. If you would rather use a light theme, but keep the white text on your Action Bars, set the **<style…** tag to:

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">

Building The App

Once you have customised the project for your application, creating a release build is very simple.

- Open Android Studio's **Build** menu, and select **Generate Signed APK**. This will open a wizard to take you through the process of compiling your app.
- Select the OmnisWrapper module.
- When prompted for the **Key Store**, if this is your first time building the app, you should use the button to create a new Key store. For later builds you can use this existing keystore.

This Key store is what identifies you as a developer, so it's important that you **back this up** after you create it. Any updates to your app must be signed with the same key. You can use the same key for multiple apps, if you wish.

If you are intending to deploy your app through Google Play, when creating your key you should ensure that you set its Validity to at least **25 years**.

*Guidance from Google on **Signing Strategies** can be found **here**.*

- When prompted for a **Build Type**, select **release**.

Once you finish this wizard, it will export an **.apk** file. This is your signed, release build of your app, ready to deploy to your users.

Deploying The App

Once you have built your app, you are ready to deploy it to devices.

Manual Deployment simply requires you to distribute the .apk file to your users manually. They then *sideload* the app to their device.

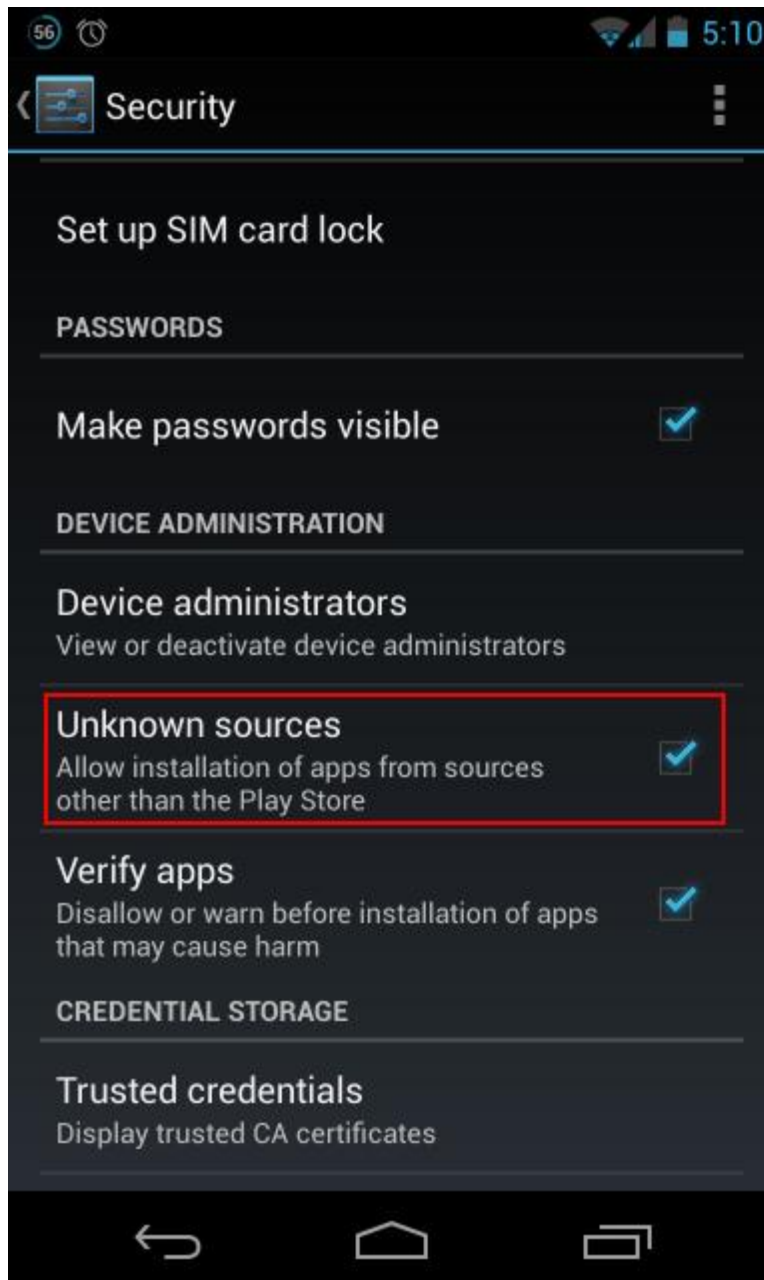
Google Play is Android's app store. Getting your app onto this platform is a bit more involved, but the benefits are very worthwhile. There is no manual verification of your app by Google, so this process is much quicker than deploying to the iOS App Store.

This does require a one-off registration fee of $25.

Manual Deployment

The first step is to distribute your **.apk** file to your users. It is up to you as to how you go about this. You could, for example, email the .apk file to their devices, or make it available as a download from a website. Once your users have the .apk file on their device:

- Open the device's **Settings**, select **Security**, and enable **Unknown Sources**.
  *On older Android devices, **Unknown Sources** may instead be under **Applications** in the device Settings.*



- Locate the **.apk** file on the device, and click it - you should then be prompted to install the app.

Google Play Deployment

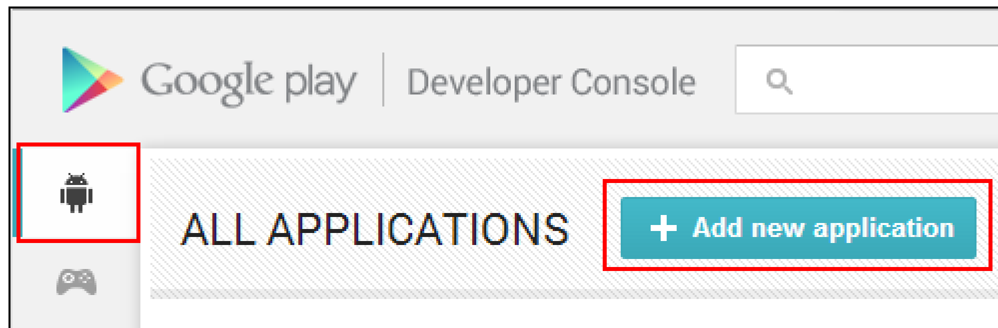
> **DISCLAIMER:** Before embarking down this route, you should read Google's requirements and guidelines on app submission.
> Tigerlogic takes no responsibility for any content of your app.

In order to deploy apps to Google Play you must register as a Google Play Developer.

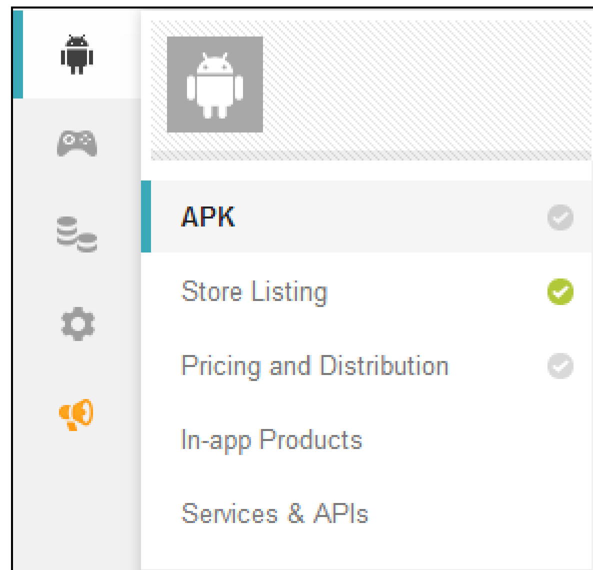
You can register **here**. (You will have to sign in with a Google account).
Once you have registered as a Google Play Developer:

- Sign in to your **Developer Console**.

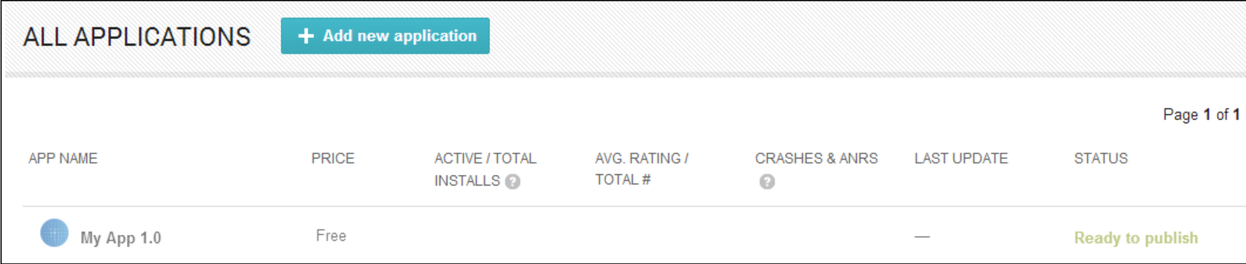
- On the **All Applications** page, push the **Add new application** button.



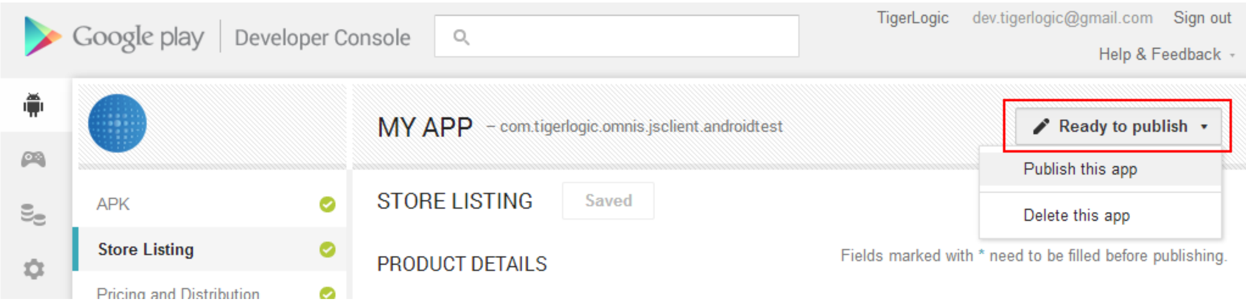
- This will begin a wizard to take you through the process of uploading your APK and preparing your store listing. It is up to you which of these you do first, but you need to do both.
    - The wizard will take you through everything required to get your app and store page ready. It gives descriptions of each of the fields you need to populate, and the size of images you need to upload.
    - You can save your details at any point, so there is no pressure to have everything ready before starting this process.
    - Make sure that you complete each of the sections shown in the sidebar when editing your app:



- Once you have uploaded your APK, and provided all of the necessary images/information which Google requires, your application will be marked as **Ready to publish**.



- If you are ready to publish your app; select your application, open the **Ready to publish** droplist, and select **Publish this app**.



- Your app will then be published to Google Play (it may take "several hours" until it becomes live on Google Play), whereupon it can be found by millions of potential users.