

Building The Android Wrapper

Version 3.0.1+

[Introduction](#)

[Get The Project](#)

[Updating The Omnis Interface Framework](#)

[Set Up The Development Environment](#)

[Ensure You Have The Correct SDK Version Installed](#)

[Configuring The App](#)

[Server Settings](#)

[Offline Mode Settings](#)

[Database Settings](#)

[Push Notifications](#)

[Push Notification Channels](#)

[Behaviour Settings](#) (NEW)

[Menu Options](#)

[Accessing Settings From Omnis](#)

[Customizing Settings](#)

[Hide Settings](#)

[Add Custom Settings](#)

[Setting Groups](#)

[Settings](#)

[Customizing The App](#)

[Creating A Unique Package Name](#)

[Changing The App's Display Name](#)

[Changing The App Icon](#)

[Changing The Splash Screen](#)

[Removing Unnecessary Permissions](#)

[Localize App](#)

[Edit The Offline Template HTML File](#)

[Bundle SCAFs \(offline apps only\)](#)

[Bundle Local Database](#)

[Edit The About Screen](#)

[Localize The About Screen](#)

[Edit The Credits Screen](#)

[Change The Color Scheme](#)

[Building The App](#)

[Deploying The App](#)

[Manual Deployment](#)
[Google Play Deployment](#)

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.

Most of the implementation of the device functionality is contained within a library - ***omnisInterface*** which the wrapper is built around. This means that you can build your own native app around the *omnisInterface* library if you wish. Or otherwise, you should be able to update the library independently of the rest of your wrapper project.

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.

NOTE: If you are using Windows, there is a path length limitation of 240 characters when building the app. This includes all generated files, so we recommend you store the project at a relatively short file path location.

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.

Updating The Omnis Interface Framework

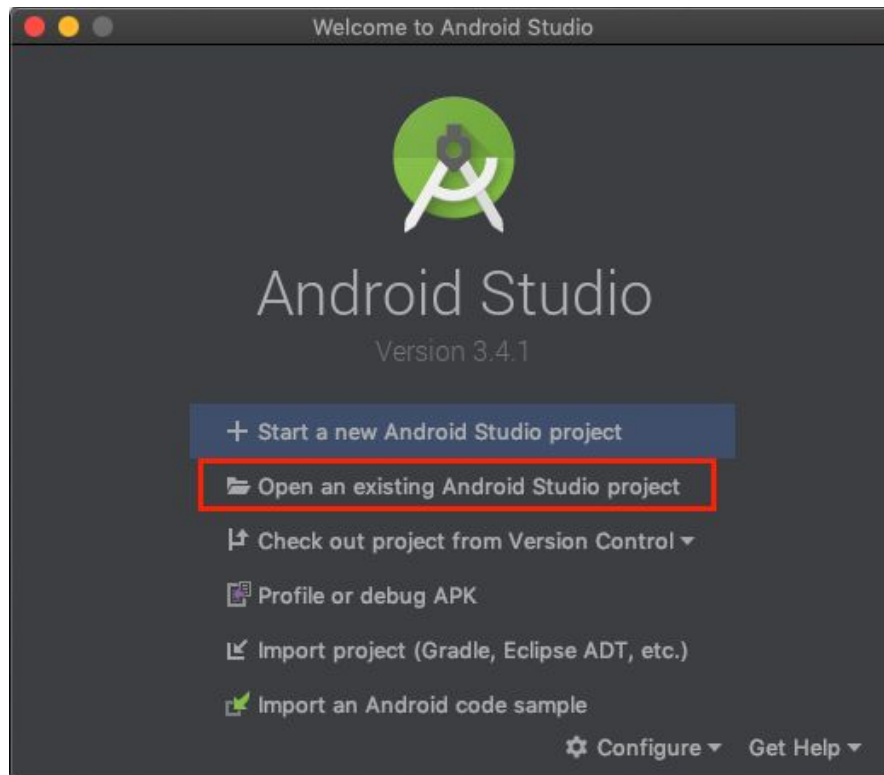
Updates to the App Framework (containing the bulk of the features used by the wrapper) can be applied without replacing the wrapper project.

- Download an updated Android **Omnis Interface Framework** from the wrapper [download page](#).
- Extract the files contained in the zip into the **omnisinterface** folder at the root of your wrapper project.
- Clean and rebuild your app.

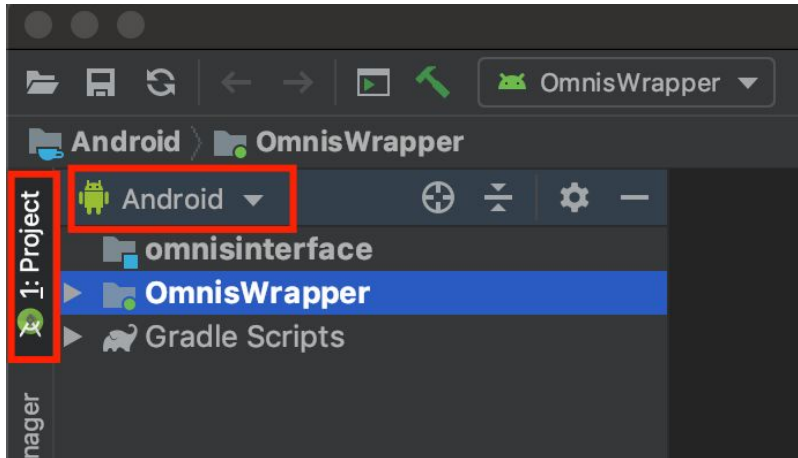
Set Up The Development Environment

The Android wrapper is built using [Android Studio](#).

- Java **JDK 8** 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 directory you extracted from the .zip ('**Android**' by default). 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:



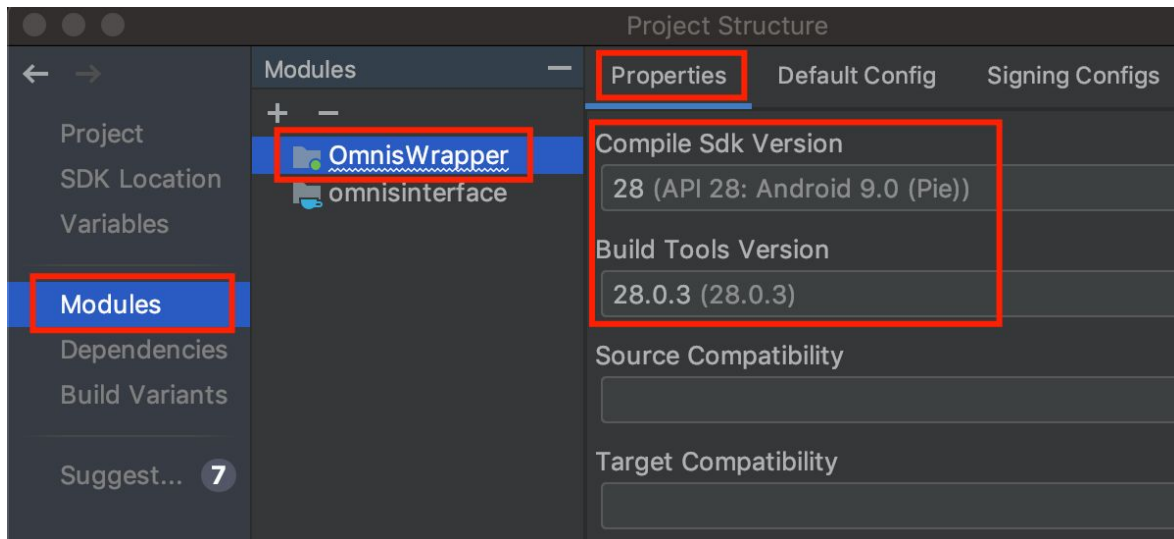
- Along the left side of the screen, select “**Project**”.
- In the droplist at the top of the Project view, select “**Android**”.
- Your view should now look like that shown in the screenshot above, and this is the view you will want for all of your work with the wrapper.

Ensure You Have The Correct SDK Version Installed

The Android app builds against a particular *Compile* Android SDK version.

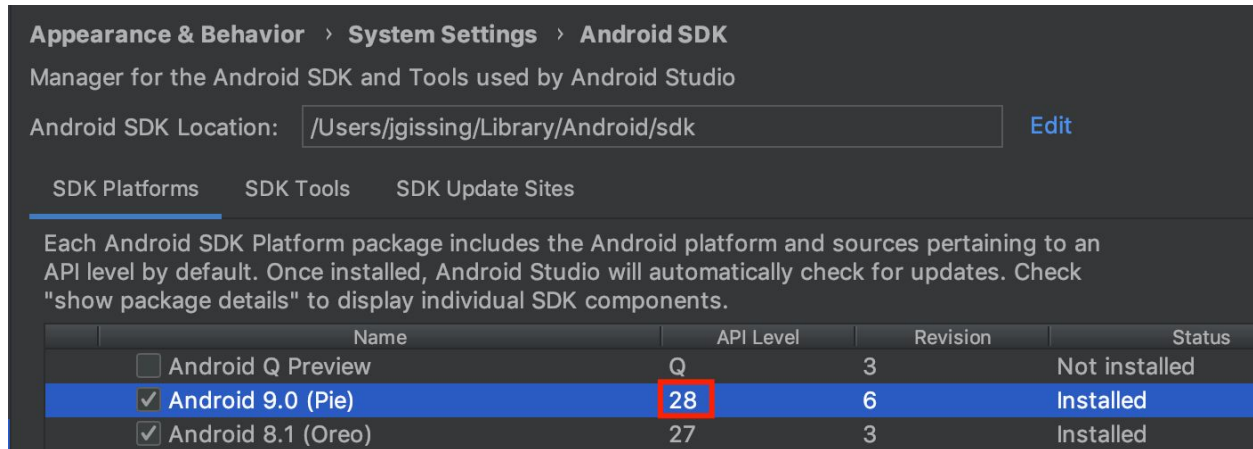
It's important to note that this is not the minimum version of Android the app will run on, but instead allows the app to take advantage of features and theming introduced in later SDK versions.

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



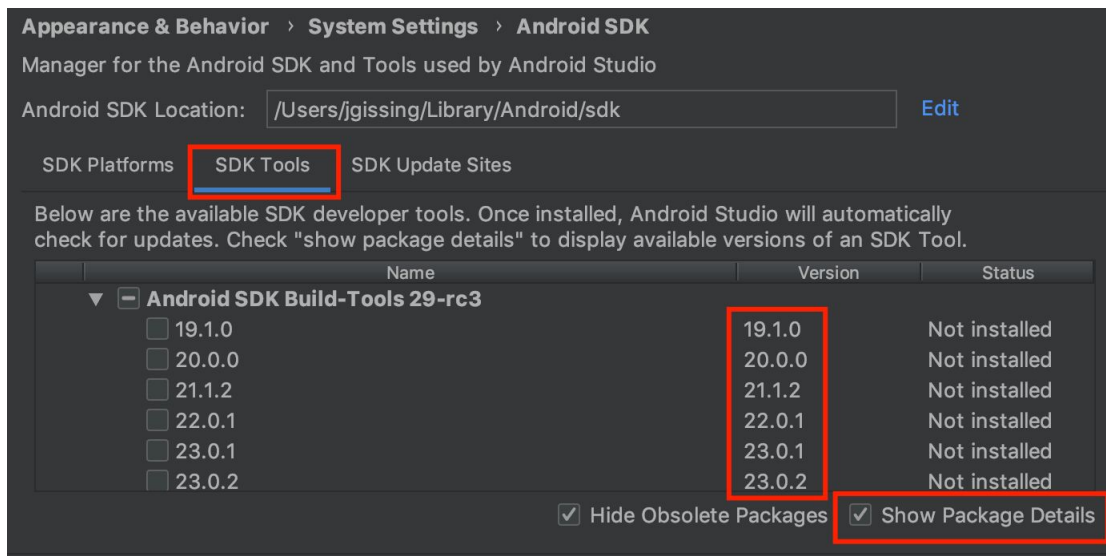
- Select the **Properties** tab, and here you can see the **Compile Sdk Version** and **Build Tools Version** - make a note of these, then close the window.
- Open the **SDK Manager** (from the main toolbar, or **Tools -> SDK Manager**)

- Use the SDK manager to check that you have the **SDK Platform** for the compile SDK version found above installed. If not, use the SDK Manager to install it.



(Note, this version may not be the same as in the screenshot above)

- Also make sure you have the correct Build Tools Version as found above.
 - Using the SDK manager, select **SDK Tools**, check the **Show Package Details** checkbox, and make sure the correct version is installed, under **Android SDK Build-Tools**:



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

Configuring The App

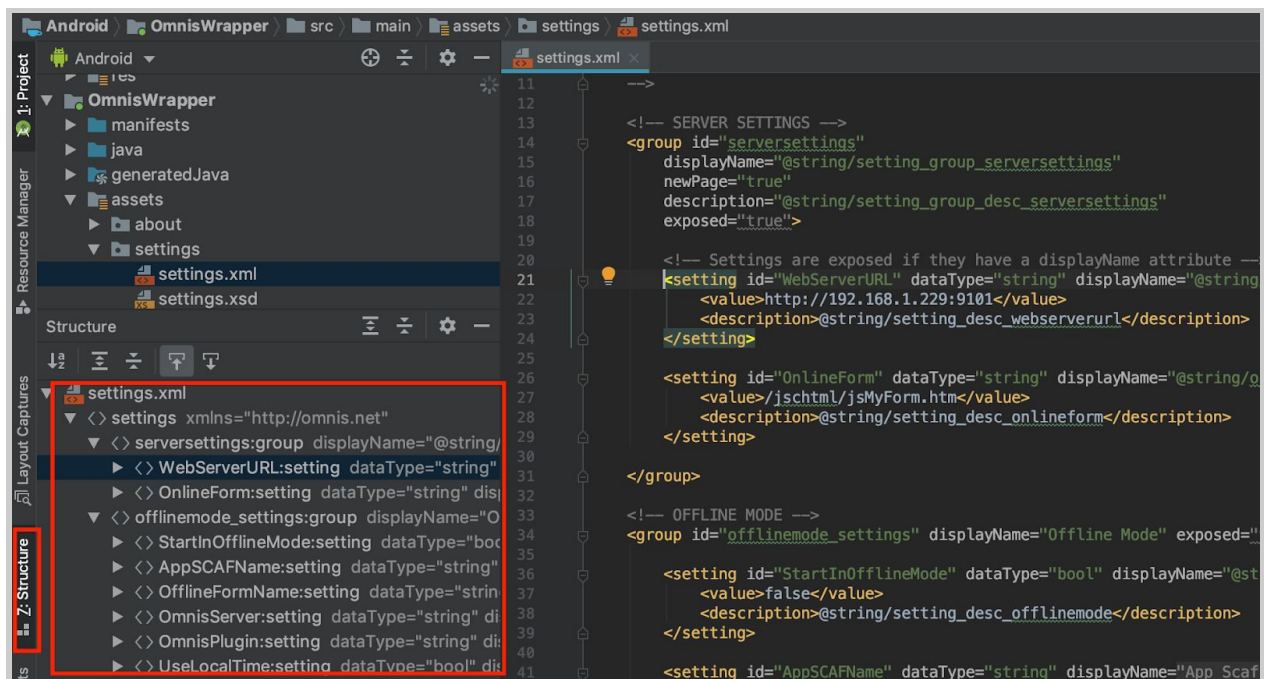
Configuration of the app is done through the **settings.xml** file, which is situated in **OmnisWrapper/assets/settings**.

This file defines both *exposed* (shown in the in-app Settings menu) and *hidden* settings, and comprises **groups** and **settings**. Groups are just groupings of settings, and these can be nested.

You should set the settings' **values** in this file to point the app to your Omnis server, and to configure how the app behaves.

This file contains a lot of XML data, and so can be a little difficult to navigate directly. It is recommended to use Android Studio's **Structure** view to jump to the setting you are interested in.

The Structure view is usually exposed as a tab along the left edge of the window (if you have **View > Tool Buttons** enabled). If you can't see it, try via the menu option: **View > Tool Windows > Structure**.



The properties within the settings file are as follows:

Server Settings

- **WebServerURL**: 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*
- **OnlineForm**: 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.htm*. If you are using a web server, it will be the remainder of the URL to get to the form, e.g. */omnisapps/myform.htm*.

Only WebServerURL & OnlineForm are needed for Online forms. The following properties are needed in addition to webserverurl for Offline mode.

Offline Mode Settings

- **StartInOfflineMode:** Whether the app should start up in offline mode (*true*) or online mode (*false*).
- **OfflineFormName:** Name of the offline form. (**Do not add .htm extension!**)
- **AppSCAFName:** The name of the App SCAF, which is generated in Omnis' **html/sc** folder. This will generally be a lower-cased (*case-sensitive!*) version of your library name, with certain characters escaped to underscores.
- **OmnisServer:** 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
- **OmnisPlugin:** 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
- **UseLocalTime:** If *false*, dates & times are converted to/from UTC, as default. Setting this to *true* will disable this conversion. (Offline only - online mode reads from remote task's *\$localtime* property).

Database Settings

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

Push Notifications

- **PushNotificationServer:** URL to the RESTful interface to your Omnis server used for handling push notifications.
 - For direct to Omnis connections, this will be of the form: **http://<ip address>:<\$serverport>**
 - When using a Web Server, this will be a URL to your RESTful web server plugin, suffixed with **"/ws/xxx"**
Where xxx is either:
 - 1) nnnn (a port number)
 - 2) ipaddress_nnnn (IP address and port number)
 - 3) serverpool_ipaddress_nnnn

- **PushNotificationGroupMask:** An integer bitmask to specify the initial notification groups the app should be a member of.

Push Notification Channels

There is a group with the id “**pushnotification_channels**”.

This contains *settings* which define specific ‘**Channels**’ for your push notifications, which the user can opt in/out of individually (from Android Oreo (8.0)).

Note: There is a [separate document](#) detailing the use of Push Notifications. Please refer to that for more details.

Behaviour Settings (NEW)

- **ImmersiveMode:** If set to **true**, the app will run in full-screen ‘[Immersive Mode](#)’, hiding the status bar and navigation bar. Changes to this at runtime will require an app restart in order to take effect.

*There are issues with using immersive mode in conjunction with split screen mode.
It's recommended to disable split screen support if using immersive mode, by setting **android:resizeableActivity** to 'false' for the 'MainActivity' activity in **AndroidManifest.xml***

Menu Options

- **MenuIncludeOffline:** Whether the in-app menu includes the option to switch between online & offline modes.
- **MenuIncludeUpdate:** Whether the in-app menu includes the option to update when in offline mode.
- **MenuIncludeAbout:** Whether the in-app menu includes the option to show the About screen.

The About screen includes a link to the *Credits* (which it is necessary to expose, to meet licensing criteria of the wrapper's dependencies).

If you disable the *About* menu option, a menu option to open the *Credits* screen will show instead.

- **MenuIncludeSettings:** Whether the in-app menu includes the option to show the Settings screen.

NOTE: The values in the settings.xml for ‘exposed’ settings (those which are exposed through the in-app Settings menu) are read only on **first launch** of the app. They are then saved to, and read from, local storage.

Settings which are not exposed (they, or their group, have an **exposed="false"** attribute) are read from the settings.xml each time the app runs.

As such, if you make changes to *exposed* settings in settings.xml, you will need to uninstall the old app from your device before running the new version, for changes to take effect.

Accessing Settings From Omnis

You can use the **savepreference** & **loadpreference** *\$clientcommands* to read or set these native app settings at runtime, from your Omnis code.

To do so, you just need to prefix the setting name (the **id** of the `<setting>` element in *settings.xml*) with “_SETTING_”, and pass as the preference name when calling the above *\$clientcommands*.

E.g: Do `$cinst.$clientcommand("loadpreference",row("_SETTING_OmnisForm","iPref"))`

Customizing Settings

[Hide Settings](#)

When you come to release your app, you will find that it would make sense to hide many of the settings which are exposed through the in-app settings, so that end-users can't change connection settings, for example.

You can hide individual settings or entire groups by adding the **exposed="false"** attribute to their tags in *settings.xml*. (If the *exposed* attribute is not provided, it defaults to true for Groups, and true for Settings if the setting has a *displayName*).

```
<!-- OFFLINE MODE -->
<group id="offlinemode_settings" displayName="Offline Mode" exposed="false" newPage="true">
  <setting id="startinofflinemode" dataType="bool" displayName="Start Offline">
    <value>false</value>
    <description>@string/setting_desc_offlinemode</description>
  </setting>
```

NOTE: If you ever *set* a setting which is **not exposed** (using the **savepreference** *\$clientcommand*), that setting will no longer read its value from *settings.xml* on launch (it will use the saved value).

Add Custom Settings

You can add your own custom settings to *settings.xml*, which can be exposed through the in-app Settings menu if you wish.

A *settings.xsd* is included, so that you get auto-complete help when editing *settings.xml* in Android Studio.

Setting Groups

You can add new **<group>** elements to the settings xml document in order to group a collection of settings together.

They have the following *attributes*:

- **id**: A unique ID for the group.
- **displayName**: The name to show for the group within the Settings screen.
 - Can be a string literal, or refer to a (localizable) string resource using the format: **"@string/<string_res_name>"** (probably defined in *res/values/strings_settings.xml*)
 - To make a group visible you must have a display name.
- **newPage**: **"true"** if the group should open in a new page (the default), **"false"** if the group should be shown in a section on the current page.
- **description**: A description for the group.
 - Can be a string literal, or refer to a (localizable) string resource in the same way as *displayName*.
 - You can use [Markdown](#) syntax to add simple styling (not all syntax is supported).
- **exposed**: **"true"** to show this group in the settings screen (the default), **"false"** to hide the group and all of its settings (regardless of their individual *exposed* attribute) on the setting screen.

It is possible to nest groups within other groups, to create a complex hierarchy of settings.

Settings

You can add *Settings* to *Groups* by adding a **<setting>** element inside a **<group>** element.

Each Setting element can have the following attributes:

- **id**: A unique ID for the setting.
 - This will be how you refer to the setting from Omnis (with a **"_SETTING_"** prefix).
- **displayName**: The name to show for the group within the Settings screen.
 - Can be a string literal, or refer to a (localizable) string resource using the format: **"@string/<string_res_name>"** (probably defined in *res/values/strings_settings.xml*)
- **dataType**: The data type of the setting. Determines the type of data that is stored with the setting, and the UI exposed to change it in the Settings screen:
 - **"string"**: (The default)

- **“bool”**: The *<value>* (see below) should be **true** or **false**.
 - **“int”**
 - **“float”**
 - **“button”**: (Advanced) A button which will call custom (native Kotlin/Java) code when clicked. You will need to extend **OMPrefButton**’s **onButtonClicked** function to add your own custom handling.
- **exposed**: “true” to show this setting in the settings screen (the default, if a *displayName* is provided), “false” to hide it (e.g. it is some configuration constant).
 - If it is not exposed, its value will be read from *settings.xml* on each launch of the app, so can be changed in updated builds of your app.

Settings may also contain the following **child elements**:

- **<value>**: **(Required)** Its content is the initial value for the setting.
 - For bool settings, this should be **true** or **false**.
- **<description>**: Its content provides a description for the setting.
 - The content can be raw text, or refer to a string resource using the standard format: **@string/<string_res_name>**
 - You can use [Markdown](#) syntax to add simple styling (not all syntax is supported).
 - You can use the “%s” placeholder to insert the current value of the setting.
- **<options>**: Contains a series of **<option>** tags containing possible values for this setting.
 - The setting will open a pop-up list of options for the user to choose from.
 - **<option>** children have a **displayName** attribute, and their content forms the value that will be set if it is selected.

```
<group id="others" displayName="Other Settings" description="@string/setting_dec_group_others">
  <setting id="maxthings" dataType="int" displayName="Max Things" exposed="true">
    <value>4</value>
    <description>The maximum number of things.
      That could possibly be used.

      Currently: %s
    </description>
    <options>
      <option displayName="One">1</option>
      <option displayName="Two">2</option>
      <option displayName="@string/three">3</option>
      <option displayName="Four!">4</option>
    </options>
  </setting>
</group>
```

Customizing The App

Creating A Unique Package Name

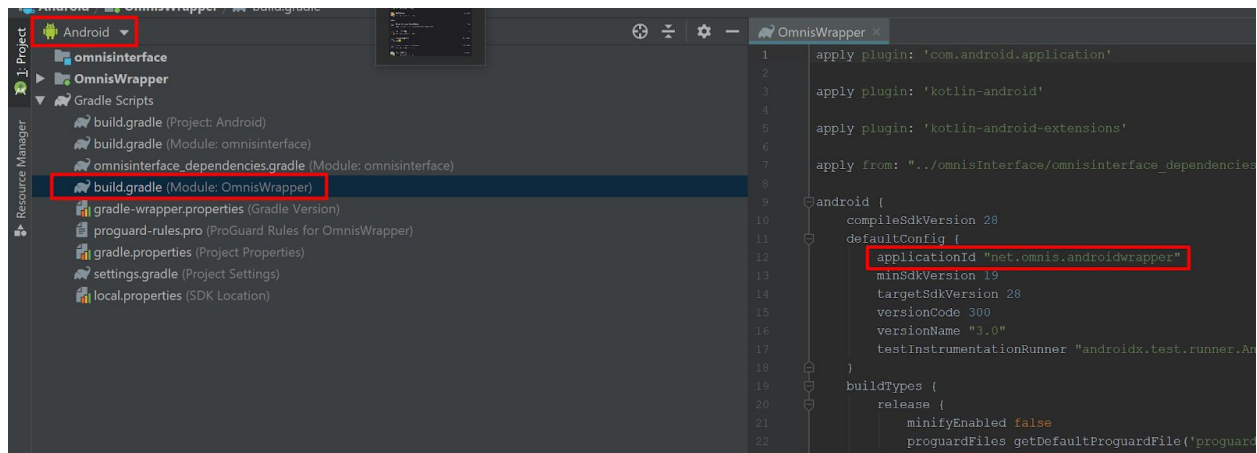
The first step of customizing your app is to change the *Package Name*.

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.

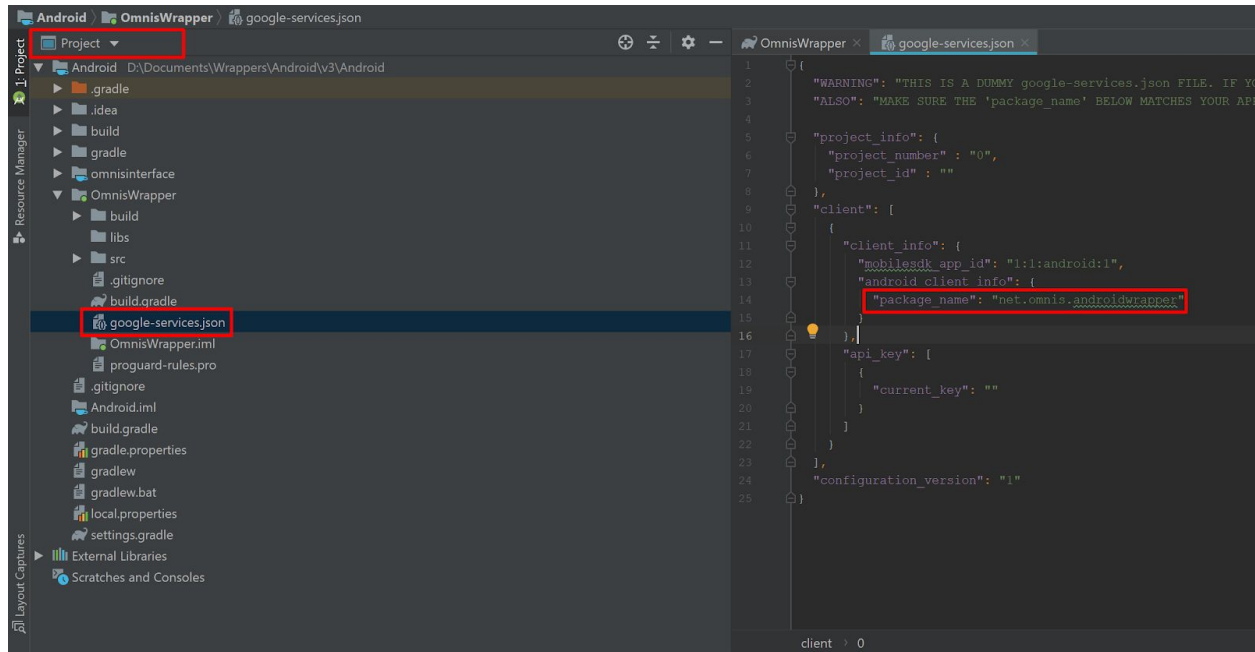
As such you should use a reverse-domain-name style identifier to ensure you do not conflict with other apps.

- With the drop down selection set to **Android**
- Navigate to **Gradle Scripts > build.gradle (Module: OmnisWrapper)**
- Edit the line **applicationId** with your chosen package name
 - e.g. `applicationId "net.omnis.androidwrapper"`
 - 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`.
- Note that here are some restrictions to this value:
 - It must have at least two segments (one or more dots).
 - Each segment must start with a letter.
 - All characters must be alphanumeric or an underscore [a-zA-Z0-9_].



If you are not going to be replacing the dummy google-services.json file (to support push notifications etc) then you will need to go into the file to edit the package name.

- Use the drop down menu to switch into the **Project** view
- Navigate to **OmnisAndroidWrapper->OmnisWrapper->google-services.json**
- Edit the **package_name** field to match your new **Application ID**

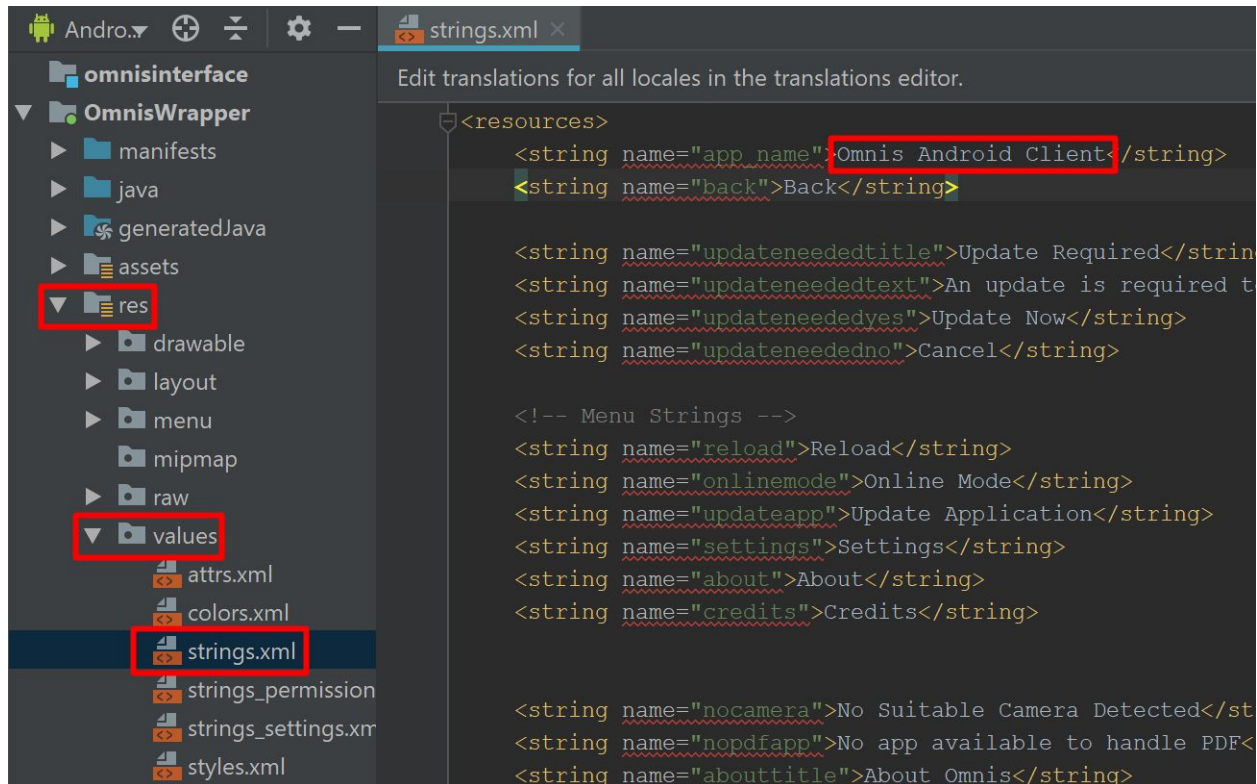


Changing The App's Display Name

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.
- **READ_EXTERNAL_STORAGE** - necessary if you are obtaining images from the camera or the device's saved images (*kJSDeviceActionTakePhoto* or *kJSDeviceActionGetImage*).
- **CALL_PHONE** - necessary in order to make phone calls from the app (*kJSDeviceActionMakeCall*).
- **VIBRATE** - necessary in order to make the device vibrate (*kJSDeviceActionVibrate*).

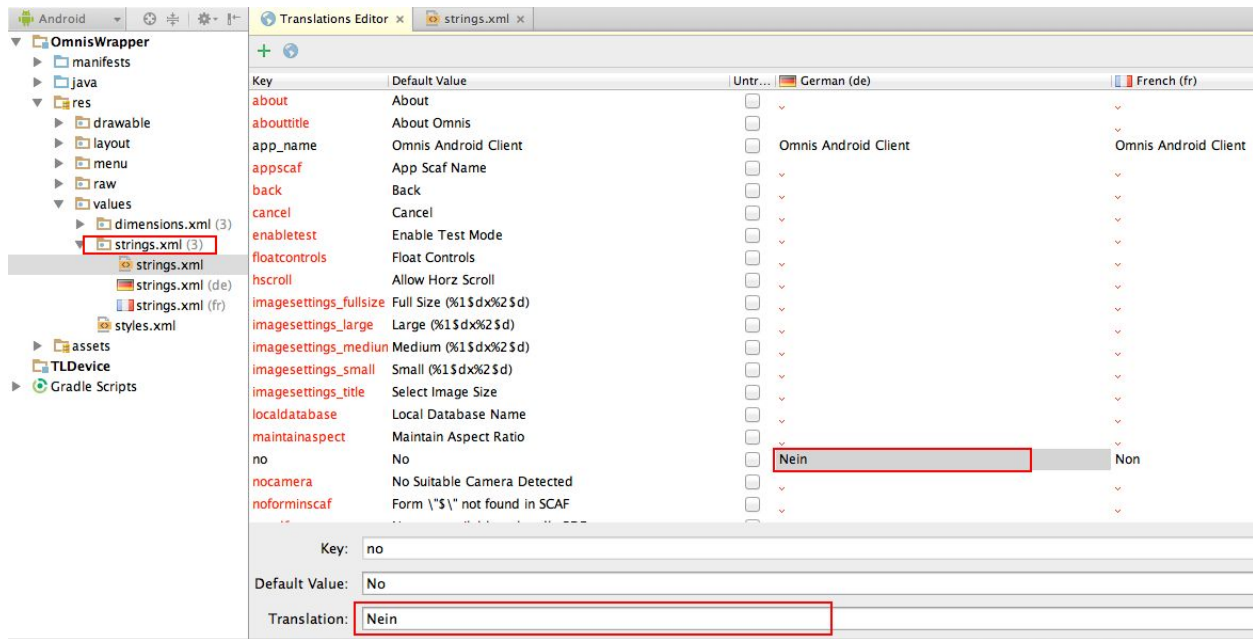
Localize App

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.

There are also other localizable string files, which work in the same way:

- **strings_permissions.xml**: Contains strings used in permission request dialogs, when the app needs to request permission from the user for certain actions.
- **strings_settings.xml**: Contains strings used in the Settings screens.

Edit The Offline Template HTML File

The offline forms are created from the included **jsctempl.htm** template HTML file in your project's **assets** folder. If for any reason you want to change the HTML structure of the page, you can edit this file.

The default jsctempl.htm file will not work with versions of Omnis prior to 10.0.1.

*If you wish to use the wrapper in offline mode with an earlier version of Omnis, replace the contents of **jsctempl.htm** with the included **jsctempl_pre10.0.1.htm** file.*

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.
`C:\Users\<username>\AppData\Local\Omnis Software\OS10.0`
 - On macOS, this will be in the Application Support area.
Macintosh HD ▸ ▢Users ▸ <username> ▸ ▢Library ▸ ▢Application Support ▢ ▸ ▢Omnis ▢ ▸ ▢OS10.0
- 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.

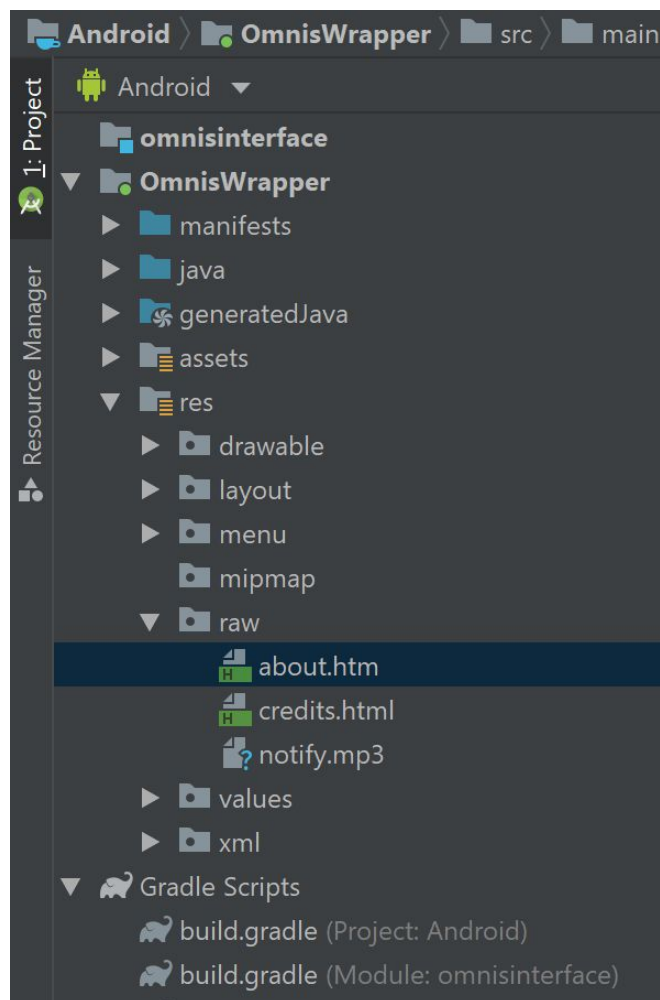
If you need to access data from a large database, it may make sense for you to hold the whole database on your Omnis server, and use the Sync Server functionality to synchronize a subset of this data with your device.

Details on the Sync Server can be found [here](#).

Edit The About Screen

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.

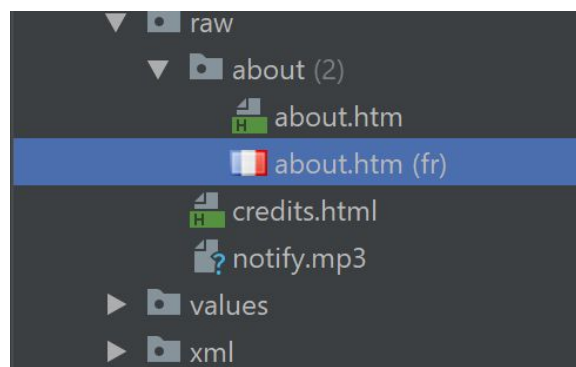
- To enable(/disable) the About menu option, edit your project's **config.xml** file, setting the value of **<MenuIncludeAbout>** to **1** (or **0** to disable it).
- Open your project's **res/raw** folder. This needs to contain a file named **about.htm**, which is the html page which will be loaded 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

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

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.

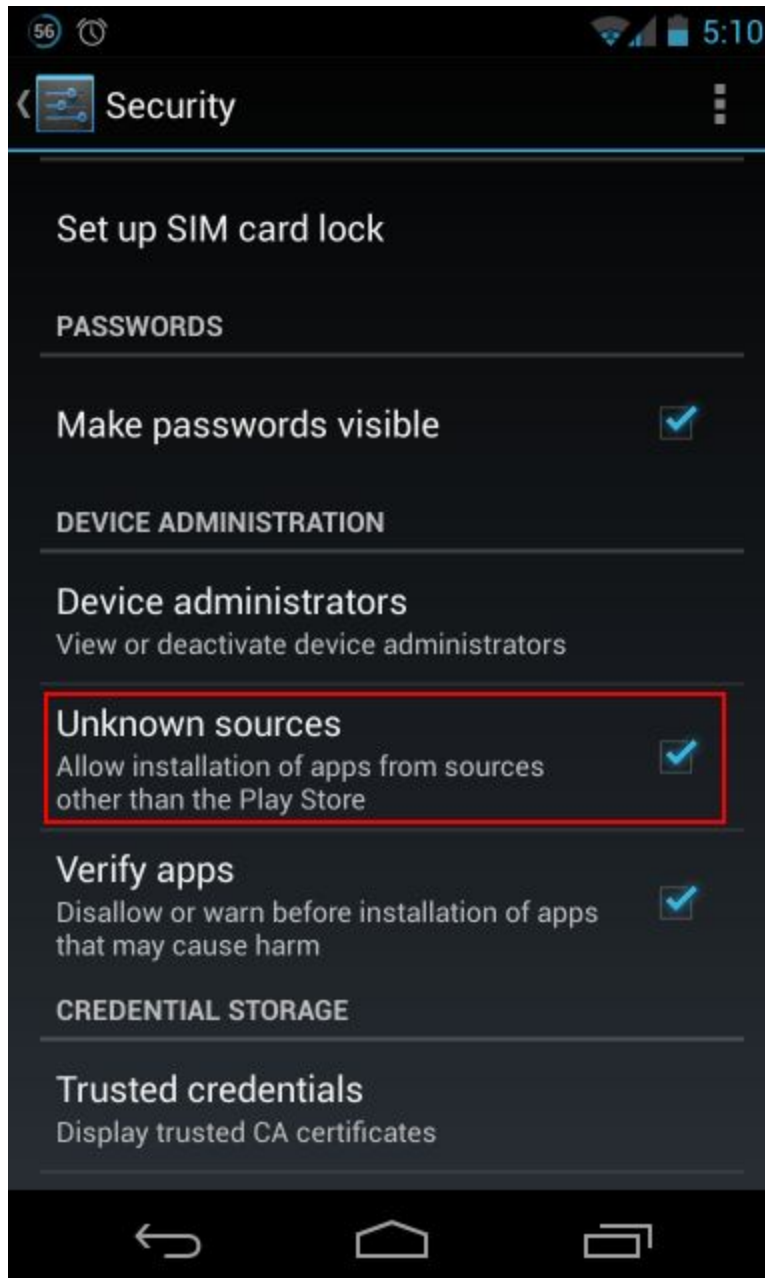
[Google Play](#) is Android's app store. Getting your app onto this platform is a bit more involved, but the benefits are very worthwhile. There is no manual verification of your app by Google, so this process is much quicker than deploying to the iOS App Store.

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.*



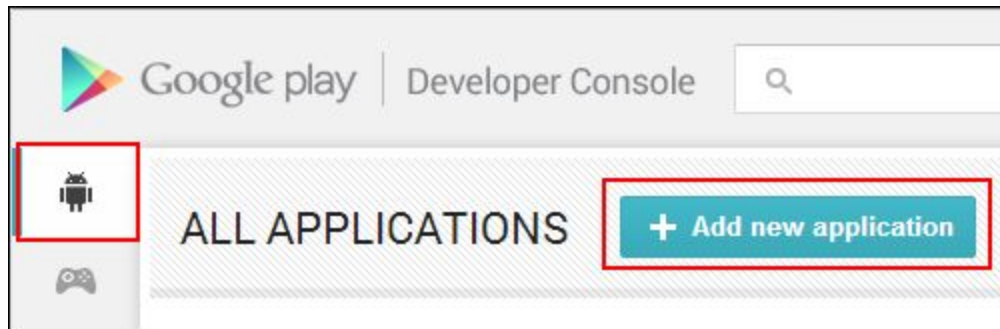
- Locate the **.apk** file on the device, and click it - you should then be prompted to install the app.

Google Play Deployment

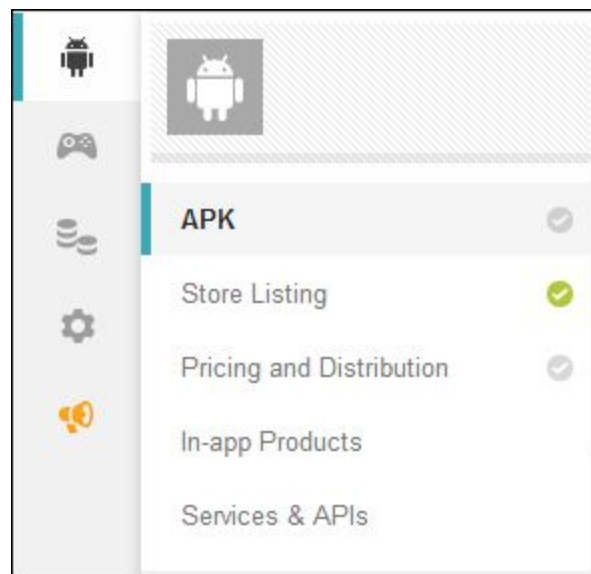
DISCLAIMER: Before embarking down this route, you should read Google's requirements and guidelines on app submission.
Omnis Software takes no responsibility for any content of your app.

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.



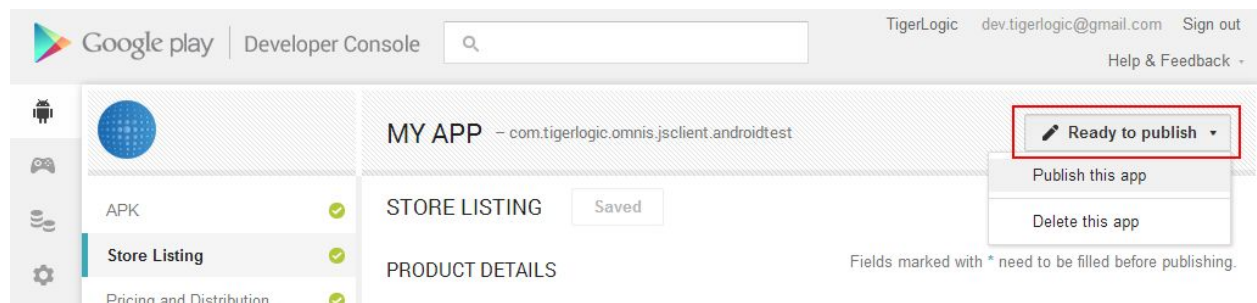
- This will begin a wizard to take you through the process of uploading your APK and preparing your store listing. It is up to you which of these you do first, but you need to do both.
 - The wizard will take you through everything required to get your app and store page ready. It gives descriptions 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**.

ALL APPLICATIONS + Add new application						
Page 1 of 1						
APP NAME	PRICE	ACTIVE / TOTAL INSTALLS ?	AVG. RATING / TOTAL #	CRASHES & ANRS ?	LAST UPDATE	STATUS
 My App 1.0	Free				—	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**.



The screenshot shows the Google Play Developer Console interface. At the top, there's a header with the Google Play logo, 'Developer Console', a search bar, and user information for 'TigerLogic' with email 'dev.tigerlogic@gmail.com' and a 'Sign out' link. Below the header, the left sidebar contains navigation icons for Android, Games, and a settings gear. The main content area is titled 'MY APP - com.tigerlogic.omnis.jsclient.androidtest'. It features a 'Ready to publish' button with a dropdown arrow, which is highlighted with a red rectangle. Below this button, a dropdown menu is visible with options: 'Publish this app' and 'Delete this app'. To the left of the main content, there's a list of app management tasks: 'APK' (checked), 'Store Listing' (checked), and 'Pricing and Distribution' (checked). Below these tasks, there are buttons for 'STORE LISTING' (with a 'Saved' status) and 'PRODUCT DETAILS'. A note at the bottom right states: 'Fields marked with * need to be filled before publishing.'

- 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.