将应用程序部署到安卓设备

在安卓设备上，应用程序以特殊结构的 ZIP 包形式分发，称为应用包 （APK） 或安卓应用包 （AAB） 。APK文件可以下载到设备上并在设备上执行，而AAB旨在由谷歌Play商店解释并用于生成APK文件。

Qt 安卓版具有 armv7a、arm64-v8a、x86 和 x86-64 的二进制文件。若要在应用程序中支持多个不同的 ABI，请生成一个包含每个 ABI 的二进制文件的 AAB。应用商店使用 AAB 为发出下载请求的设备生成优化的 APK 程序包，并使用您的发布者密钥自动对其进行签名。

Qt 创建器支持以下安卓应用程序的部署方法：

- 作为独立的可分发应用程序包 （APK） 。
- 从 Qt 5.14.0 开始，作为应用程序捆绑包 （AAB） ，打算在谷歌 Play 商店中分发。

注意： 由于Qt Creator 4.12，因此不支持部。

要指定应用程序包的设置，请选择**项目**>**构建**>**构建** Android APK >**详细信息**。.

有关用于运行应用程序的选项的详细信息，请参阅为 Android 设备指定运行设置。

包装应用

由于将应用程序捆绑为 APK 软件包并非易事，因此 Qt 5 提供了一个名为 的部署工具。当您使用*适用于安卓的Qt套件*部署应用程序时，Qt Creator会使用该工具创建必要的文件并将其捆绑到APK中：`androiddeployqtandroiddeployqt`

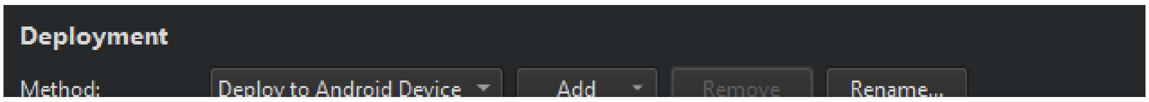
- Java 文件，它们充当应用程序的入口点，并自动加载 Qt 并在应用程序中执行本机代码。
- 安卓智能.xml，提供有关应用程序的元信息。
- 其他 XML 文件，这些文件指定应用程序的依赖项。
- 资源文件。
- 库和 QML 文件，这些文件可以包含在项目中，具体取决于您选择的部署方法。
- 下载和使用渐变所需的渐变包装器。
- Java IDE（如安卓工作室）所需的分级脚本。它允许用户扩展 Java 部分，而无需复制我们的 Java 源代码。它还允许 IDE 提供代码完成、语法突出显示等。

只有当您使用 Gradle 构建应用程序包时，才会捆绑 Gradle 包装器和脚本。有关更多信息，请参阅连接安卓设备。

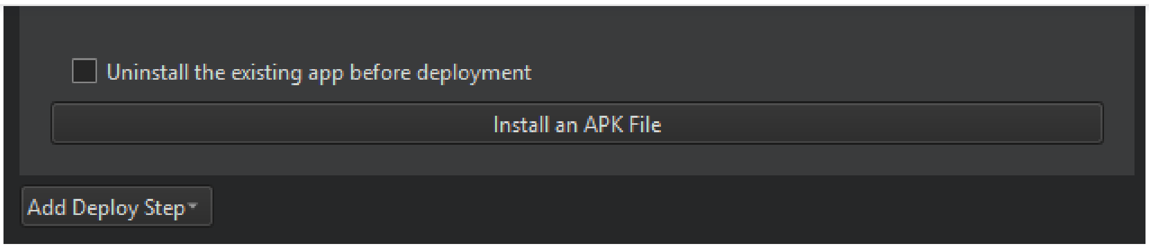
若要查看该工具创建的包，请选中“**生成后打开包位置**”复选框。`androiddeployqt`

指定部署设置

可用的部署设置在“**方法**”字段中列出。若要为项目添加部署方法，请选择“**添加**”。


Deployment
Method: Deploy to Android Device

Qt DOCUMENTATION



若要重命名当前部署方法，请选择"**重命名**"。

若要删除当前部署方法，请选择"**删除**"。

程序包部署在工具包选择器中选择的 Android 设备上。若要添加设备，请选择"**管理**"。

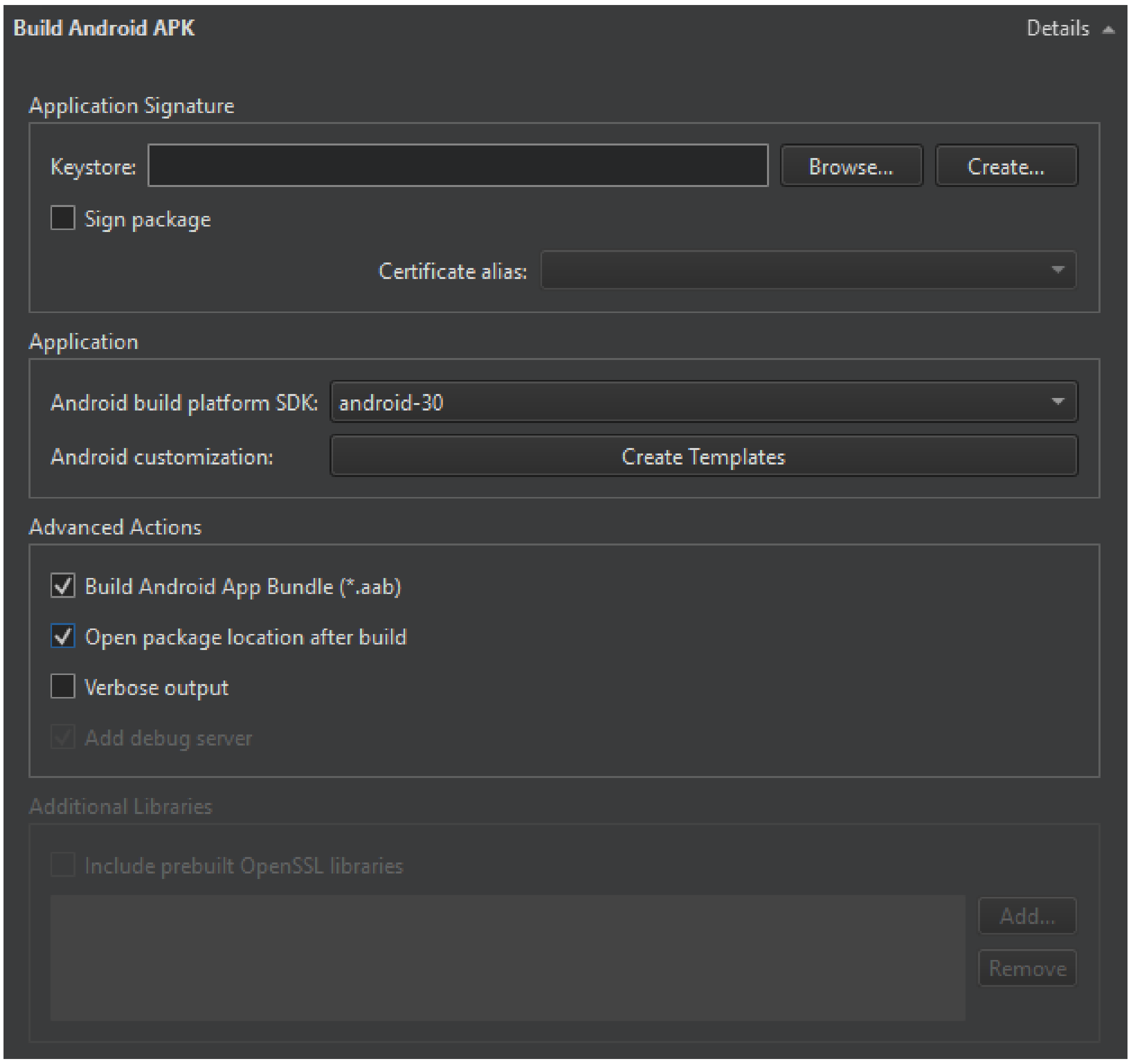
有关为应用程序指定其他启动选项的详细信息，请参阅为 Android 设备指定运行设置。

若要从设备中删除以前安装的文件，请选择"**在部署前卸载现有应用**"。

要将预构建的 APK（例如第三方应用）安装到设备，请选择**安装** APK **文件**。

指定包的设置

要指定该工具的设置，请选择**项目**>**生成 > 运行**>**生成** >**生成** Android APK >**详细信息**。.androiddeployqt



The tool uses the configuration information to create APKs. For more information about the available options, see androiddeployqt.androiddeployqt

You can view information about what the tool is doing in Compile Output. To view additional information, select the **Verbose output** check box.androiddeployqt

Selecting API Level

In the **Android build platform SDK** field, you can select the API level to use for building the application. Usually, you should select the highest API level available.

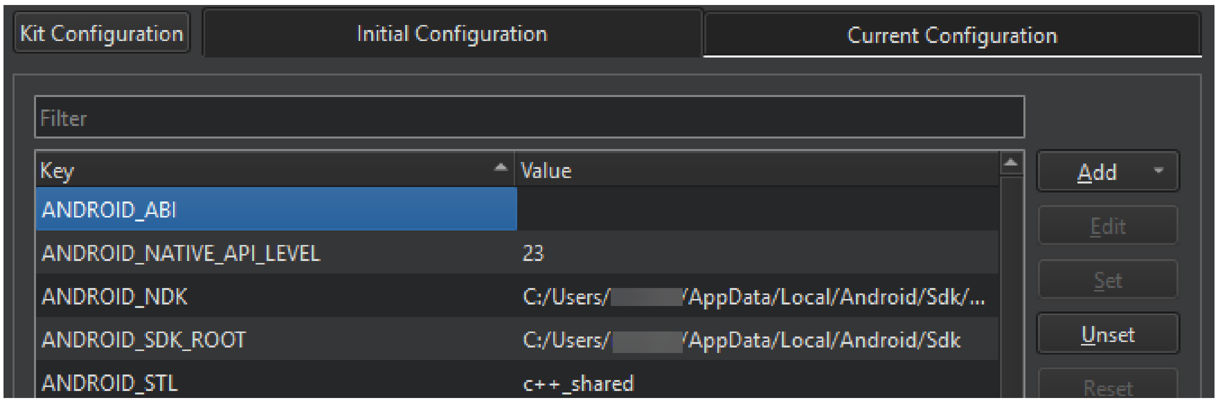
Note: For Qt 5.12.0 to 5.12.5 and Qt 5.13.0 to 5.13.1, Android build platform SDK 28 should be used. For more recent versions than Qt 5.13.1, build platform SDK 29 or the most recent one should be used.

This field does not specify the minimum supported API level nor the target API level, which you can specify in the Android manifest. See Editing Manifest Files. For more information about Android API levels, see What is API Level?.

Building AABs

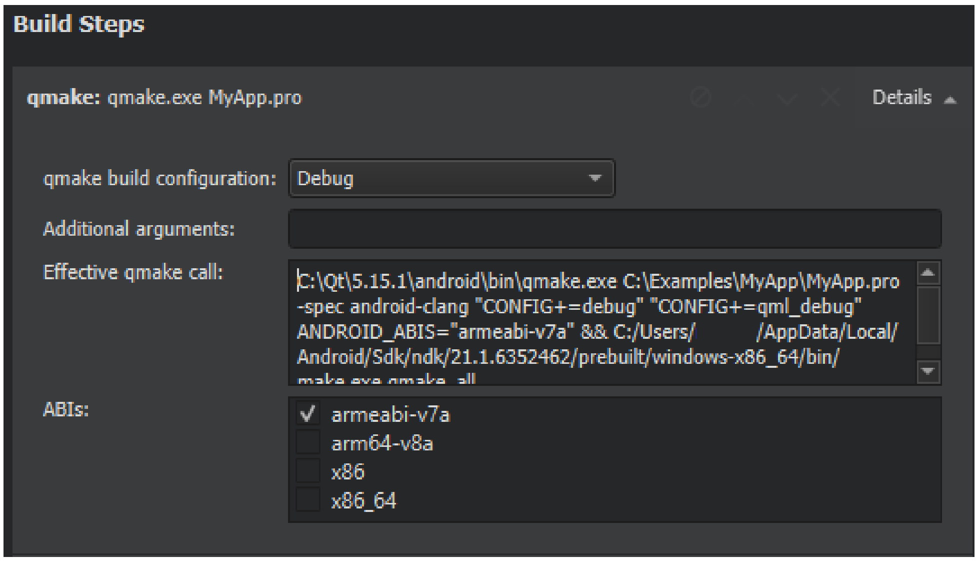
For testing the application locally, use the APK format because the package can be uploaded directly to the device and run. For distribution to the Google Play store, create an AAB by selecting the **Build Android App Bundle (*.aab)** check box.

When building with CMake, you can view the selected ABIs in **Initial Configuration** in the **CMake** section. You can set additional ABIs as values of the key:`ANDROID_ABI`



Key	Value
ANDROID_ABI	
ANDROID_NATIVE_API_LEVEL	23
ANDROID_NDK	C:/Users/[redacted]/AppData/Local/Android/Sdk/...
ANDROID_SDK_ROOT	C:/Users/[redacted]/AppData/Local/Android/Sdk
ANDROID_STL	c++_shared

When building with Qbs or qmake, you can select the ABIs in the **ABIs** field in the **Build Steps**:



qmake: qmake.exe MyApp.pro

qmake build configuration: Debug

Additional arguments:

Effective qmake call:

```
C:\Qt\5.15.1\android\bin\qmake.exe C:\Examples\MyApp\MyApp.pro
-spec android-clang "CONFIG+=debug" "CONFIG+=qml_debug"
ANDROID_ABIS="armeabi-v7a" && C:/Users/        /AppData/Local/
Android/Sdk/ndk/21.1.6352462/prebuilt/windows-x86_64/bin/
make.exe qmake_all
```

ABIs:

- [✓] armeabi-v7a
- [ ] arm64-v8a
- [ ] x86
- [ ] x86_64

Signing Android Packages

To publish your application, you must sign it by using a *public-private key pair* that consists of a *certificate* and a corresponding *private key* and is identified by an *alias*. The key pair is used to verify that the future versions of your application are actually created by you.

Warning: Keep the key pair in a safe place and take back up copies because you cannot update the application if you lose the key pair.

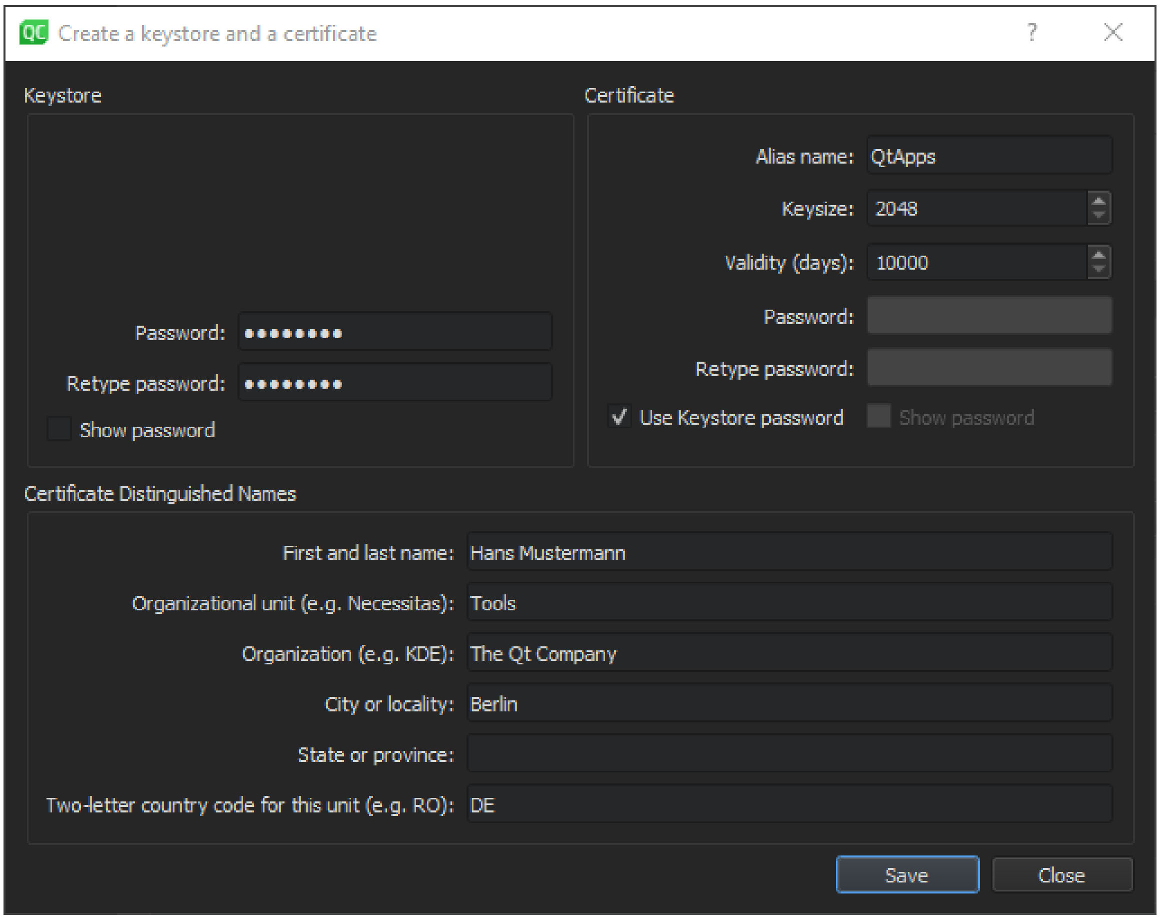
digital certificate. It contains information about the version, serial number, and validity period of the certificate, the ID of the algorithm that is used to encrypt the data, the organization that issued the certificate, and the *subject* (owner) of the certificate. In case of a self-signed certificate, the issuer and owner of the certificate are the same. In addition, the certificate contains information about the algorithm that is used to sign the certificate, as well as the signature of the certificate.

The keystore is protected by a password. In addition, you can protect each alias with its individual password.

When you sign an Android application, you must select a keystore that contains certificates and a certificate alias from the keystore. The public key (certificate) for the alias is embedded into the APK during signing.

To create a keystore and a self-signed certificate:

1. In the **Keystore** field, select **Create** to create a new keystore that contains one key pair in the **Create a Keystore and a Certificate** dialog:



Create a keystore and a certificate

Keystore

Password:	••••••••
Retype password:	••••••••
Show password	

Certificate

Alias name:	QtApps
Keysize:	2048
Validity (days):	10000
Password:	
Retype password:	
✓ Use Keystore password	Show password

Certificate Distinguished Names

First and last name:	Hans Mustermann
Organizational unit (e.g. Necessitas):	Tools
Organization (e.g. KDE):	The Qt Company
City or locality:	Berlin
State or province:	
Two-letter country code for this unit (e.g. RO):	DE

2. In the **Keystore** group, enter a password to protect the keystore.
3. In the **Certificate** group, specify the key size and validity period of the certificate. You can specify a separate password to protect the key pair or use the keystore password.
4. In the **Certificate Distinguished Names** group, enter information about yourself and your company or organization that identifies the issuer and the owner of the key pair.
5. Select **Save**.
6. In the **Keystore File Name** dialog, enter a name for the keystore and select a location for it.
7. In the **Keystore** dialog, enter the keystore password to create the key pair in the keystore.

To sign an Android package by using a key pair, set the **Sign package** group settings described in Specifying Settings for Packages:

1. In the **Keystore** field, select **Choose** to select an existing keystore.
2. In the **Certificate alias** field, select an alias from the list of key pairs that the keystore contains.
3. Select the **Sign package** check box to use the alias to sign the Android package.

Adding External Libraries

application's library folder and loaded on startup.

To add OpenSSL libraries, select **Include prebuilt OpenSSL libraries** in the **Additional Libraries** group. This will add the OpenSSL include project defined in device settings in **Android OpenSSL** group. This can be used for QMake and CMake projects.

Otherwise, you can manually add the paths to the required and libraries to the **Additional Libraries** field.`libssl.solibcrypto.so`

Editing Manifest Files

You can use the configuration options to specify all the settings you need for the tool. You only need an Adroid manifest file to specify Android-specific settings, such as the application icon. However, the manifest file is needed when you want to publish the package in the Play Store.`androiddeployqt`

Package Names

Android application packages are usually named by using a hierarchical pattern with the levels in the hierarchy separated by periods (.). In general, a package name begins with the top level domain name of the organization followed by the organization's domain name and any subdomain names listed in reverse order. The organization can then choose a specific name for their package. Package names should be written in all lowercase characters whenever possible. For example, .`org.qtproject.example`

Complete conventions for disambiguating package names and rules for naming packages when the Internet domain name cannot be directly used as a package name are described in section 7.7 of the Java Language Specification.

For more information about package names, see Android Application Fundamentals.

Styling

Qt uses different methods to determine how Qt Widgets and Qt Quick Controls should be styled:

- Select **Default** or **Full** when using Qt Widgets or Qt Quick Controls 1 in your project.

  Note: This method uses some Android non-SDK interfaces, that are being restricted by Google starting from Android 9.0 (API 28).

- Select **Minimal** when using Qt Quick Controls 2 but no Qt Widgets or Qt Quick Controls 1. This is faster than using the default or full options.
- Select **None** when using neither Qt Widgets nor Qt Quick Controls 1 or 2.

Screen Orientation

You can specify different options for determining screen orientation, depending on sensor readings or user preferences. The following table lists the options available.

Orientation	Description
Unspecified	The system chooses the orientation. The policy it uses, and therefore the choices made in specific contexts, may differ from device to device.
Behind	Use the same orientation as the activity that's immediately beneath it in the activity stack.
Landscape	Landscape orientation, where the display width is larger than its height.
Portrait	Portrait orientation, where the display height is larger than its width.
Reverse landscape	Landscape orientation in the opposite direction of normal landscape.
Reverse portrait	Portrait orientation in the opposite direction of normal portrait.

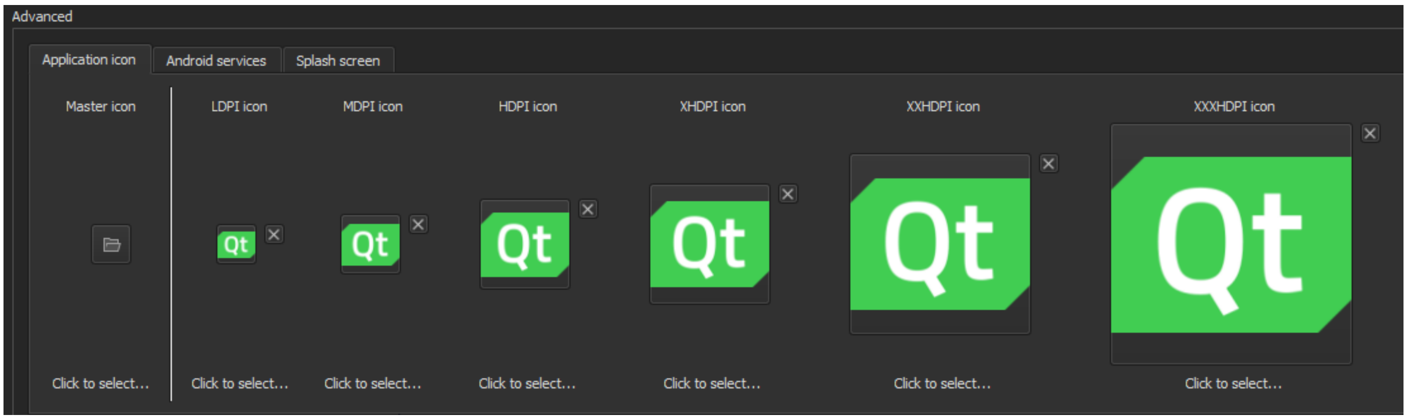
Sensor portrait	Portrait orientation, but it can be either normal or reverse portrait based on the device sensor. The sensor is used even if the user has locked sensor-based rotation.
User landscape	Landscape orientation, but it can be either normal or reverse landscape based on the device sensor and the user's preference.
User portrait	Portrait orientation, but it can be either normal or reverse portrait based on the device sensor and the user's preference.
Sensor	The orientation is determined by the device orientation sensor. The orientation of the display depends on how the user is holding the device. It changes when the user rotates the device. Some devices, however, will not rotate to all four possible orientations by default. To allow all four orientations, select the full sensor option. The sensor is used even if the user locked sensor-based rotation.
Full sensor	The orientation is determined by the device orientation sensor for any of the four orientations. This is similar to the sensor option, except that it allows any of the four possible screen orientations, regardless of what the device will normally do. For example, some devices won't normally use reverse portrait or reverse landscape, but this option enables them.
No sensor	The orientation is determined without reference to a physical orientation sensor. The sensor is ignored, so the display will not rotate based on how the user moves the device.
User	The user's current preferred orientation.
Full user	If the user has locked sensor-based rotation, this option behaves in the same way as the user option. Otherwise, it behaves the same as the full sensor option, and allows any of the four possible screen orientations.
Locked	Locks the orientation to its current rotation, whatever that is.

Icons and Splash Screens

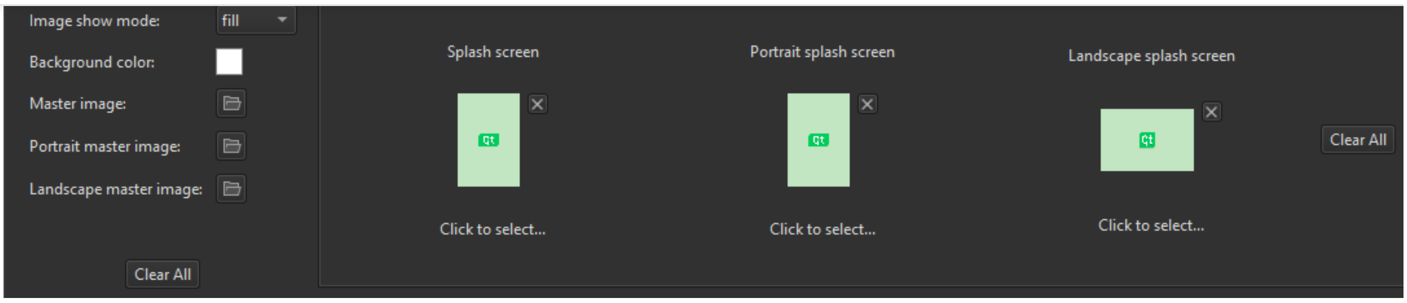
You can set different images to be shown as application icons and splash screens on low, medium, high, and extra high DPI displays. The following list summarizes the DPI values typically associated with each category:

- Low-density (LDPI): ~120dpi
- Medium-density (MDPI): ~160dpi
- High-density (HDPI): ~240dpi
- Extra-high-density (XHDPI): ~320dpi
- Extra-extra-high-density (XXHDPI): ~480dpi
- Extra-extra-extra-high-density (XXXHDPI): ~640dpi

Specify settings for icons in the **Application icon** tab. Select the image with the highest resolution as the **Master icon**. Qt Creator resizes the icon and sets versions of it to be shown on low, medium, high, and extra high DPI displays, as needed. Alternatively, set the icons for each resolution separately.



Specify settings for splash screens in the **Splash screen** tab. Select images to display as splash screens depending on the device orientation and screen resolution.



By default, the splash screen is hidden automatically when an activity is drawn. To keep it visible until QNativeInterface::QAndroidApplication::hideSplashScreen() is called, select the **Sticky splash screen** check box.

In **Image show mode**, select whether to center the splash screen on the device display or scale it to fill the display.

Set a background color in **Background color**.

Select the images with the highest resolution as the **Master image**, **Portrait master image**, and **Landscape master image**.

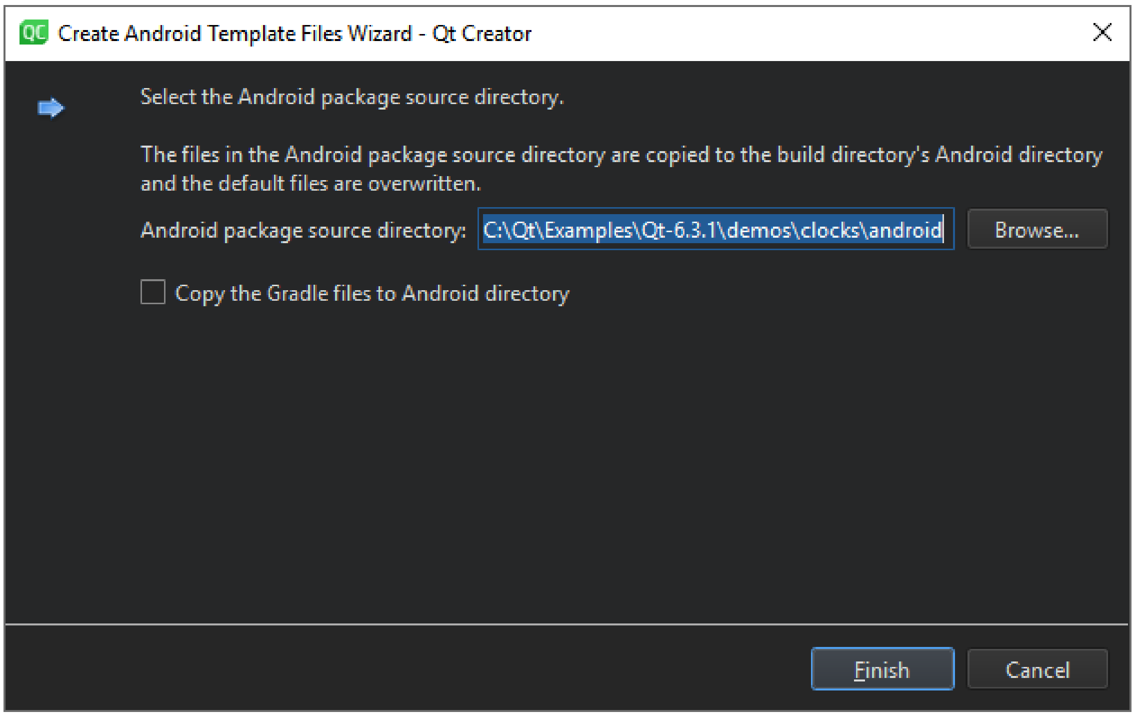
Select **Clear All** to reset all settings or remove all images.

Android Manifest Editor

If you use qmake as the build system, you can create an Android manifest file and edit it in Qt Creator.

To create an Android manifest file and to open it in the Android Manifest Editor:

1. Select **Projects** > **Build** > **Build Android APK** > **Create Templates**.
2. Check the path in **Android package source directory**.



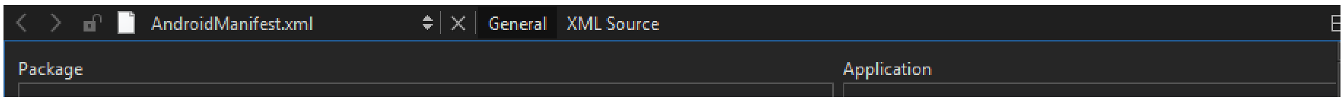
Select the Android package source directory.

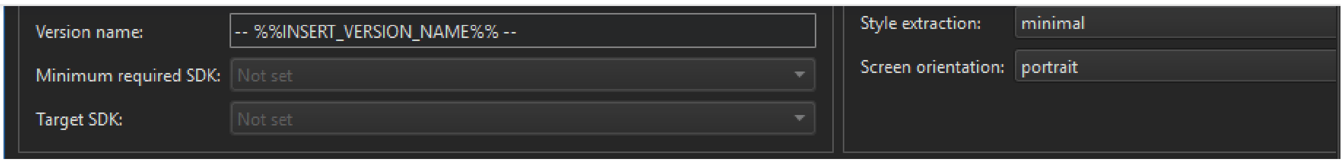
The files in the Android package source directory are copied to the build directory's Android directory and the default files are overwritten.

Android package source directory: C:\Qt\Examples\Qt-6.3.1\demos\clocks\android

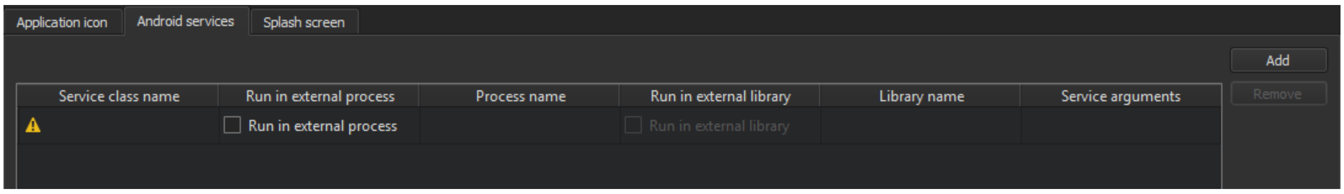
☐ Copy the Gradle files to Android directory

3. Select **Copy the Gradle files to Android directory** if you plan to extend the Java part of your Qt application.
4. Select **Finish** to copy the template files to the directory and to open the manifest file for editing.`android`
5. In the **Package name** field, enter a valid package name for the application. For example, . The application is launched by an automatically generated Java launcher that is packaged with the application into an Android package (.apk).`org.example.myapplication`



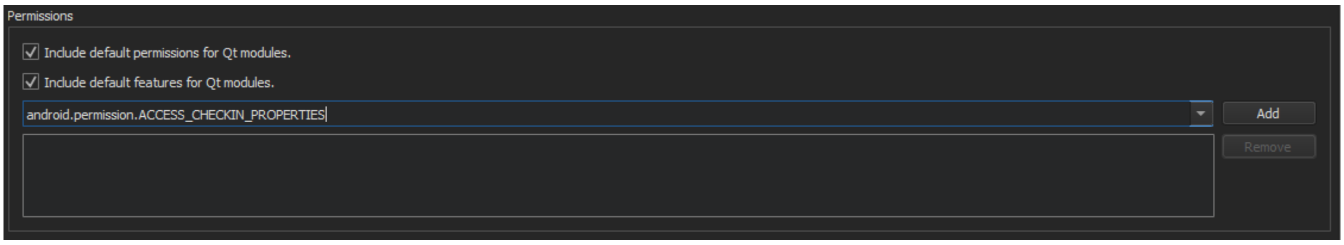

Version name: -- %%INSERT_VERSION_NAME%% --
Minimum required SDK: Not set
Target SDK: Not set
Style extraction: minimal
Screen orientation: portrait

6. You can specify an internal version number for the package in the **Version code** field. It is used to determine whether one version of the application is more recent than another. In the **Version name** field, specify the version number that is shown to users.
7. In the **Minimum required SDK** field, select the minimum API level required to run the application. The minimum supported API level for Qt Creator is android-9. However, Qt versions might have different minimum API levels, and therefore Qt Creator does not allow you to select an API level that the Qt version specified for the kit does not support.
8. In the **Target SDK** field, select the targeted API level of the application. This affects the activation of some compatibility features in the OS. The value used by the tool by default is 14, which means that the overflow button in the system navigation bar will not be enabled by default.`androiddeployqt`
9. In the **Application name** field, set the application's name.
10. In the **Activity name** field, set an activity name.
11. In the **Style extraction** field, set the method that Qt uses to determine which UI style to use.
12. In the **Screen orientation** field, select the option for determining screen orientation.
13. In **Application icon**, specify images to use as application icons depending on screen resolution.
14. In **Splash screen**, select images to display as splash screens depending on the screen orientation and resolution.
15. In **Android services**, select **Add** to add a service. You must enter at least a service class name for a new service. If you select **Run in external process**, you also need to enter a process name. If you select **Run in external library**, you need to enter a library name. Service arguments are mandatory for a service that is not run in an external library. For more information about writing service code and structure of services, see Android Services.



Service class name	Run in external process	Process name	Run in external library	Library name	Service arguments
⚠	☐ Run in external process		☐ Run in external library		

16. In the **Permissions** field, you can specify the permissions that your application needs. Starting from Android 6.0 (API 23), permissions have to be requested at runtime (see QtAndroidPrivate::requestPermission()). For lower Android API levels, users are asked to grant the permissions when they install the application. Android OS then grants the application access to the appropriate data and features.



☑ Include default permissions for Qt modules.
☑ Include default features for Qt modules.
android.permission.ACCESS_CHECKIN_PROPERTIES

17. Select the **Include default permissions for Qt modules** and **Include default features for Qt modules** check boxes to add the permissions needed by Qt libraries. This can be for Qt Core or for Qt Location.`android.permission.WRITE_EXTERNAL_STORAGE``android.permission.ACCESS_COARSE_LOCATION`
18. To add a permission, select it from the list, and then click **Add**.

On the top header, select the **XML Source** tab to edit the file in XML format.

< Deploying to Devices   Deploying Applications to Boot2Qt Devices >