Master Essay: App Development

Claus Børnich
clausbo@ifi.uio.no

May, 2014

Contents

1 Introduction

This paper will give a brief introduction to app development for mobile devices based on research, primarily in the form of self study and development of simple apps.

The focus is on the first stage of writing an app(1) for the Galaxy(2) system. An open, web-based platform for running analyses jobs in biology and medicine. Widely used in bioinformatics the software makes it possible to run tools and workflows without programming experience and captures the information to make it possible to repeat and share analyses via the web. The ultimate aim is to develop an app which provides quick and convenient access to increase usability and improve productivity, but the extent of this paper is to explore some of the current tools, frameworks and programming languages. The advantages offered, limitations imposed and design and implementation methodologies used will be considered from the viewpoint of a software developer with no previous experience in app development.

2 Mobile Phone Apps

Increasingly web access is done through mobile devices, such as smartphones and tablets, because of their portability which makes it convenient, quick and easy to get online from almost anywhere at any time. However, web access on such devices can be slow and using the normal web interface can be time consuming, frustrating or impossible. For example, logging in to services such as the Galaxy system can be tricky, frustrating and take an unnecessarily long time, especially when done under less than ideal circumstances, such as on a train with intermittent Internet connection. Using a website interface, such as that provided for the Galaxy system to browse jobs or get an overview of analyses results, can also be a challenge on a small screen. Especially if the interface has not been designed for the low precision of a touch based user interface.

2.1 Modern Mobile Platforms

While the first smart phone dates back 90s(3), and Nokia's Symbian was the first real operating system, the leading mobile software platforms of today only date back to 2007 when Apple launched iOS on the iPhone and the Open Handset Alliance led by Google announced Android(4). According to published numbers from Gartner, Inc the Symbian platform dominated the smartphone market in 2007 with a 63.5% share and worldwide sale of more than 77 million smarthpones (5), but in 2013 Android had a 78.4% market share with 758 million smartphones sold, followed by iOS at 15.6% and 150 million units(6). There are also new mobile platforms emerging like Firefox OS(7) by Mozilla which features a HTML5, JavaScript and CSS application layer and web API built on top of the Linux kernel and Ubuntu Touch(8) which is a mobile version of the Linux based operating system Ubuntu.

This development is important to note as it shows how rapidly the mobile industry is moving. At the end of 2010 Symbian was still the market leader(9), but during 2011 both Android and iOS raced past(10). This illustrates the importance of considering the intended lifetime of an app and strategies for cross-platform deployment. While Android is currently dominating the market and still growing, iOS still has a huge market share, and Microsoft's Windows Phone has grow from a 2.5% share in 2012 to 3.2% in 2013 with a respectable 30 million units sold. The tablet market is a similar story with Android dominating the market and increasing their share of the growing market, followed by iOS(11). However, according to Gartner sales numbers iOS still has 36% of the tablet market in 2013 with over 70 million tablets sold, while Android has climbed from 45.8% in 2012 to 61.9% and over 120 million tablets sold in 2013. While Android has the largest user base it is also a very fragmented platform as there are hundreds of devices from different manufacturers running different versions of Android. It can therefore be difficult to provide support and if older versions are excluded the app will not be available to large portions of the user base.

Due to limited time and to reflect the above situation this paper will focus a lot on cross-platform methods of deploying apps, but with a focus on Android apps.

2.2 Cross-Platform Development

The first challenge in writing an app is to select the right technology for the target device platforms and desired functionality. An app compiled for a specific platform is referred to as a “native app”, while server-side websites designed for use on a mobile device browser is typically referred to as a “mobile web app”(12).

Native apps are built for a specific device, although as we shall see later it is possible to use cross-platform tools and frameworks to write an app once and then build it for multiple target platforms. Both free and commercial native apps are primarily distributed through marketplaces, such as Google Play and Amazon Appstore for Android devices, the Mac App Store for iOS devices and the Windows Store for Windows devices.

Native apps are much more common for mobile devices than web apps and perform better with a native user interface that is fully integrated with the platform style

and can take full advantage of the platform's capabilities, such as camera, GPS, accelerometer and other device features. The most obvious downside for native apps is the limitation to one specific platform and the high maintenance of potentially supporting several versions for different platforms.

For mobile web apps the benefit is that it will be cross-platform and available on any computer or device with a compatible browser and access to the server, and maintenance and updates need only be applied to the server. For experienced web developers it is also likely to be faster than developing a native app. The drawbacks of this approach are reduced performance, loss of function when offline, greater security risks and that mobile web apps are not installed but must be navigated to in a user's browser making them less convenient and potentially harder to find. Furthermore, they do not have the same look and feel as native apps making them potentially harder to learn and cannot utilise all the mobile device platform functionality. HTML5 has come a long way in providing the missing functionality for web apps, but has not entirely closed the gap with native apps.

Another popular choice, supported by a number of frameworks, are “Hybrid apps” written using web technologies - typically HTML5, CSS and JavaScript - but run inside a native container that can use the device's browser engine to render the app along with access to the device's features, such as the camera. Different frameworks, such as Phonegap and Titanium, use different approaches to how this is achieved and thus how the app performs, integrates and which device features can be accessed(13).

Hybrid apps balance the strengths and weaknesses mentioned above for native and web apps and can be seen as a compromise between the two approaches. Specific benefits include a reusable web interface that can be integrated with native code for device specific features and making it easier for experienced web developers to write apps without having to learn the development languages and APIs of each target platform. However, hybrid apps have a higher knowledge requirement than pure web apps as the developer needs knowledge of both web technologies, such as HTML5, and the hybrid framework used to integrate it into a native container. The particular framework used will also introduce its own unique issues such as adding complexity in debugging, performance issues, limitations in device functionality supported and different ways of rendering the user interface on the various target platforms making the frameworks more or less suited for different types of apps.

2.3 App Design Considerations

Visual design is important on mobile devices as it needs to take into account the low precision touch interface, small form factor, wide range of device formats and the way mobile devices are typically used for quick and urgent tasks. The small screen combined with low precision interface methods also makes data input slow and prone to error. Users also typically use mobile devices while mobile, such as walking or even driving, and so a well designed app should require little effort to use and be easy to navigate with a consistent interface that follows platform standards. The importance of good visual design is reflected in the emphasis placed on visual design in the developer documentation and guidelines for platforms such as iOS(14) and Android(15).

Apps should be designed to minimise their use of power, network, data and system resources such as CPU and memory. Apps also need to be resilient and cache data as the network connection is low bandwidth and high latency, and may switch between using a wireless and cellular connection or shut down the connection when not used, to preserve battery. An app can also be suspended or closed at any time and must handle preserving its state and resuming to a previous state.

This section only briefly outlines some of the design aspects that are important to consider when developing a mobile app based on the platform specific developer documentation mentioned above and best practice guidelines from the App Quality Alliance(16). The security permissions and privacy settings for an app are also important, but are not discussed in this paper.

2.4 Android App Development

Developing a native app for any mobile platform requires the software development kit (SDK) for that particular platform. For Android the Android SDK has the libraries and tools needed to compile code, data and resources into an Android Package (APK) archive file that contains all the app contents and which is used to install the app.

The SDK can be downloaded from the official Android developer site(17) on its own as a 100 MB zip file for Windows, Linux and Mac in both 32 and 64-bit formats.

The SDK install process is just a matter of unpacking the zip file and then running the SDK Manager to update and add packages. By default the packages for the latest API level is installed, but packages for older versions can be downloaded using the SDK Manager. The API levels are a sequential list of numbers for the version of the framework API each version of the Android platform released supports a specific API level. So for example the Android platform version 4.4 'Kitkat' uses API level 19. The API levels are backward compatible with earlier versions.

The Android Virtual Device Manager is also a part of the SDK and with the system images downloaded with the SDK Manager can be used to configure and run emulators for Android devices. Device definitions can be added with different settings for screen size, resolution, pixel density, input methods, available RAM and whether the device has GPS, Accelerometer, Camera (front or rear) and so on. A list of known device definition allows easily setting up virtual devices for various common configurations and google Nexus devices. The device definitions can be used to create a virtual device for which the platform version, API level and CPU/ABI (e.g. ARM or Intel x86), internal storage and other settings can be selected as long as the appropriate system image packages have been downloaded.

The SDK can also be downloaded as part of a larger Android Developer Tool (ADT) package which contains a pre-configured Eclipse IDE with an ADT-plugin. This version also contains the Android platform tools which contains the Android Debug Bridge (adb) which makes it possible to debug both instances of running virtual devices and connected debug enabled Android devices(18).

In addition to the SDK a Java Development Kit (JDK) must be installed. The standard edition of the JDK is open source and either the Oracle(19) version or the OpenJDK(20) can be used.

An Android app must have a manifest file which is an xml file containing elements which define the app. The elements must declare all aspects of the app, such as permissions the app needs and activities the interface will perform. The name and version of the app should also be specified in the manifest, and optionally the minimum API level required by the app can also be specified. When a minimum API level is specified the app can only be installed on devices that support that API level.

2.5 Android App Deployment and Debugging

Code is compiled into an Android APK using a virtual Android device or an Android device connected via USB. The Apache ANT build tool is used to compile and build the project in either debug or release mode. When built the app is signed so that it can run in the emulator or on the device. Debug builds are automatically given a debug key, but for distribution the app must be manually signed with the developers private key.

Once the APK is created it can be installed on the virtual or connected device using the Android Debug Bridge (adb). To install the APK on an Android device over a USB connection, the USB debugging option must be enabled. This is done in the Android settings menu under developer options, but from Android version 4.2 onwards these options are hidden and to activate them the user must open the about menu and tap the build number seven times to make the developer options available.

Once apps are installed on the virtual or connected device they are available just like any other app. Android apps are interpreted into machine code at runtime using the Dalvik virtual machine interpreter which is a variant of the Java virtual machine optimised

for mobile devices. In the latest 4.4 release of Android (API level 19) a new Android Runtime (ART) that aims to improve Android app performance is included as a developer option. ART compiles apps into native code when installed, rather than at runtime, which makes them run faster, and make them more responsive with less drain on the CPU and battery.

The Android Debug Bridge can be used to debug apps running on a virtual or connected device.

3 Qt

"Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS & JavaScript like language. Qt Creator is the supporting Qt IDE."(21) Elaborating on this quote from the Qt website, Qt is an open source framework that makes it possible to write code once and then compile it for a range of target platforms. The same code can be compiled for both desktop platforms and mobile platforms, and Windows, Mac OS X, Linux, Android and iOS are all supported.

The Qt project(21) has a history that stretches back to 1991 and five major versions, and is offered under several licenses(22). Qt is owned by Digia Plc(23) who sells a commercial license, but Qt is also available under both a LPGL 2.1 and GPL 3.0 licenses which means that the Qt framework and its libraries can be used for free as long as the license conditions are met. There is extensive documentation and a large and active community supporting Qt development and Qt is widely used in software by many leading companies. There are also several books published and many online video tutorials which have been used to develop test apps and as a source for this section(24) (25) (26) (27).

3.1 Qt Setup

The Qt Libraries along with the Qt Creator IDE can be downloaded as source or binary installers. There are installers for developing on Linux, Mac, Windows, Android and iOS in both 32-bit and 64-bit versions. This means that the same IDE and libraries can be used to develop software from almost any platform.

Qt is based on C++, but developers can choose to use either C++ or Qt Quick with the declarative QML language based on JavaScript to build user interfaces. Python can also be used with PySide or PyQt providing the bindings. A utility tool called qmake is used to generate the Makefiles that are used to build the software from the source code. The advantage of qmake is that it can use the same build instructions regardless of the platform on which the project is being compiled.

To develop Android apps with Qt the location of the SDK and JDK must be specified in Qt Creator, but additional kits and tools must also be downloaded and installed. The Android Native Development Kit (NDK) which handles using native code such as C++ in the Android app by compiling it into shared libraries (JNI), the Java based Apache ANT build tool and a C++ compiler. The GNU Compiler Collection (GCC) and MinGW C++ compilers were used for Linux and Windows respectively.

Qt 5 handles multiple target platforms through the use of kits, where each kit can be configured for a particular target platforms. Multiple target platforms can be configured and a project can be configured to compile any number of the target sets each time. However, Mac OS X or iOS require the Apple Xcode development tools which means the code must be compiled on a mac platform.

3.2 Qt Creator

The Qt Creator integrated development environment (IDE) comes with an sophisticated editor for graphically designing the user interface as well as many other developer tools such as a code editor, debugger and integrated help system. The Android Virtual Device manager is also integrated into Qt Creator making it possible to create new virtual devices and set them or connected Android devices as targets for Qt Creator builds.

Qt projects are a useful way to group together code, resources and build configuration. Qt Creator wizards are provided to assist with the setup of new projects and generate files, and custom wizards can be added. The qmake utility tool is integrated with Qt Creator, but any Qt Project can also be built from the command line using qmake. A simple project text file specifies the Qt elements and files that need to be compiled when qmake is executed, and allows configuring the project. As an example the line “CONFIG += console” enables debug output on a Windows 8 machine.

3.3 Qt Library

The Qt class library is modular. A Qt library module is included by adding it to the project file, so for example to add the Qt Core module the line “QT += core” is added. Qt Core contains the core C++ classes not related to the user interface, with the Qt GUI module providing the user interface classes. Other modules rely on the core module which contains file IO, event and object handling classes, as well as multi-threading and more. In addition Qt Core contains implementations for data types such as QString, QChar, QDate, QTime and many more including QVariant which is a class that acts as a union for most Qt data types(28).

There are fourteen modules that define the essential foundation of the Qt framework with many more add on modules that provide special purpose classes that may also be platform specific. Qt Core and Qt Gui have already been mention, but a few other notable foundation modules include Qt Network for networking, Qt SQL for database handling, Qt Test for unit testing, Qt Multimedia for video and audio functionality and Qt WebKit which provides a web view API.

All Qt classes are prefixed by a capital Q making them easy to differentiate from third-party code. The benefit of using Qt classes is the additional functionality and handling inherited. All Qt classes are objects derived from the base class QObject. One of the distinguishing features of QObject is the signals and slots feature which allows communication between widgets without using callback functions. By declaring the “Q_OBJECT” macro at the top of the class definition the meta object compiler (moc) generates the necessary meta-object code for the class. The moc can be run manually, but will be done automatically as part of the qmake process and in addition to the signals and slots mechanism it also provides a dynamic property system and run-time type information.

QObjects are organised as a tree structure with a single parent and any number of children. When a QObject is deleted it automatically deletes all its children and this makes cleanup much simpler and helps prevent memory leaks. However, this also means that QObjects should be created on the heap using the "new" keyword. If they are created on the stack then it becomes important that children are created after the parent as the destructors of local objects are called in the reverse order of their constructors - so first-in-last-out. If a QObject is set as a parent to an object created before itself on the stack then this will cause a program crash as the child's destructor is first called from the parent and then again as the child goes out of scope.

3.4 Qt Widgets

The Qt Widgets module extends Qt GUI with widgets allowing the entire graphical interface to be built up of widgets that can be arranged in layouts, either using the click-and-drag interface of the designer or by writing C++ code. If the designer is used a user interface compiler (uic) reads the Qt Designer XML format and generates a corresponding C++ header file. Just like the meta object compiler the uic will run automatically when using qmake, but can also be executed manually.

Some examples of basic widgets are QPushButton for buttons, QLabel for text labels, QTextEdit to edit and display text, QCheckBox for check boxes and so on. There are also more advanced widgets such as QCalendarWidget, or QListView and QTreeView which displays data stored in model classes using the model-view pattern(29).

Qt widgets can emit signals and receive signals by connecting a signal to a slot

declared in the class. For example the clicked() signal emitted by a QPushButton can be received by a connected class to act on the action. Another feature of widgets is that they are drawn in the style of the target platform and to make cross-platform layout easier layouts can be used to organise the widgets in nested arrangements, for example arranging a subset of widgets in a horizontal or vertical layout inside a larger grid layout.

Qt is event driven which means that a main event loop is executed and runs for the lifetime of the application receiving events, such as user interaction with the GUI, from the window system. The loop is provided by the QCoreApplication class, but applications with a GUI will use the QApplication class which inherits from QCoreApplication. An application will have exactly one QApplication object which handles dispatching events to the widgets.

3.5 Qt Quick

Qt Widgets has been the standard way to implement user interfaces with Qt, but widgets were designed for traditional static user interfaces found on most desktops. With Qt 5 a new method for implementing modern animated user interfaces such as found on mobile device and in the Windows 8 metro user interface can be used instead of Qt Widgets. This is done using the Qt Quick module together with the Qt QML module to create user interfaces with the QML declarative language. QML is based on JavaScript, but can be integrated and extended with C++ libraries.

Qt Quick provides the standard QML library with among other things visual types and animations, as well as more sophisticated effects such as shader and particle effects. The Qt Quick Controls module provides controls such as buttons and menus that follow the native behaviour of the target platform.

Apps for mobile devices can be developed using either Qt Widgets or Qt Quick with good results. Qt Widgets have been around for longer and only C++ is required to code an interface with widgets. Qt Quick on the other hand requires the developer to learn the QML declaration language, but developers with JavaScript experience can benefit from this as no C++ is required to implement the interface. Furthermore, Qt Quick allows for faster prototyping and animated user interface states and transitions.

3.6 Qt Galaxy Portal

A number of tutorials were developed using both Qt Widgets and Qt Quick and deployed to an Android smartphone and tablet. Because everything the entire project can be managed within the Qt Creator IDE and build, deployed and debugged from within the IDE the process is quite fast.

The aim of the Qt Galaxy Portal app was to create a basic proof-of-concept app to present a user with a simple login screen and deploy it to an Android device. The project was created using the Qt project wizard and choosing “Qt Widgets Application” as the template. Both desktop and Android kits were chosen for the project during the wizard setup to allow the app to be compiled and deployed on both Windows and Android testing. Finally, the QMainWindow class was chosen as the base class for the main GalaxyPortal class and the headers and source files for the project were generated, along with the project file GalaxyPortal.pro.

The details of the implementation will not be presented in this essay, but a basic main window was drawn with a simple menu and toolbar with a login dialog presented to the user at startup. The login dialog was based on QDialog and used a QGridLayout to place a username and password field, along with appropriate labels and using the standard QDialogButtonBox to present the user with a choice to “Login” or “Abort”. The password box was set to hide input simply by setting the QLineEdit control's echo mode with “QLineEdit::Password”. The buttons were connected to appropriate actions using the signals and slots mechanism described earlier. The login check simply tested against a hard coded user and password, but used QCryptographicHash to encrypt the password. A simple resource file was also added to package icons for the toolbar and again slots and signals

were used to connect the menu and toolbar items to actions such as closing the app or showing an about dialog.

The full source for this small test app is available on github(30) as well as a basic proof-of-concept unit test(31).

4 Kivy

The motivation for Kivy(32) was a lack of cross-platform toolkits for programming Natural User Interfaces (NUI), according to the paper "Kivy – A Framework for Rapid Creation of Innovative User Interfaces" authored by three of the core developers of the Kivy toolkit. The two key aspects of the NUI stated by the paper are input handling and output rendering. For input handling new event models have been explored focusing on touch capabilities, while for output rendering Kivy uses OpenGL to take advantage of modern graphics hardware and device independence. The paper written in 2011 presents an overview of the architecture and design choices made to make Kivy a powerful tool for developing the next generation user interface(33).

Kivy is a relatively young framework and the available online documentation is not always complete. For example the programming guide sections on "Best Practices" and "Advanced Graphics" were empty placeholders at the time this was written. Also, there is only a single short 138 page book for Kivy targeted at advanced developers who are already familiar with Python and assumes that the reader can use online documentation to install Kivy and get it working(34).

Kivy is distributed under a very liberal MIT license which means that Kivy can be used in both open source, free and commercial projects without any restrictions or requirements. However, there is no commercial support, only community support through user forums and online IRC chat.

Kivy packages can be created to run on Windows, Mac OS X, Android and iOS.

4.1 Kivy Setup

Kivy can be used for development on Windows, Mac and Linux. On Windows the compressed Kivy package is simply downloaded and unzipped and it is ready to be run. It even comes complete with a Python interpreter.

Since Kivy packages are not included with Ubuntu by default it must be downloaded or a Person Package Archive repository can be added using “sudo add-apt-repository ppa:kivy-team/kivy”. Then an update gets the packages “sudo apt-get update”, and finally “sudo apt-get install python-kivy” installs Kivy.

Regardless of platform Kivy depends on third party software which is downloaded and installed together with Kivy. Kivy supports Python 2.7 and upwards and can be downloaded with either Python 2.7 or Python 3.3. PyGame, PyEnchant, gs-python and Cython are also included as part of Kivy.

4.2 Kivy and Python

Kivy grew out of the PyMT project(35) - a python library for multi-touch applications, but the Kivy framework is not backward compatible due to fundamental changes to improve the design and performance.

As mentioned Kivy uses Python 2.7 or higher. The entire Python stdlib is supported. Kivy is a framework on top of Python, so all normal Python features exist, just with extra Kivy functionality added. All other external library imports are also supported (e.g. numpy, scipy).

Python is a high level language which makes it easier to implement applications quickly, but at the cost of execution speed. Python is significantly and often noticeably slower than C or C++ when it comes to tasks such as serious number crunching and drawing sophisticated graphics. Kivy solves this by using Cython to compile critical code such as event dispatching, matrix calculations and graphics rendering down to the C level

where the compiler can optimise performance. Custom code can also use Cython to speed execution if necessary. Kivy also uses its own custom drawing compiler to optimise drawing code automatically and shift the load to the GPU.

4.3 Kivy Architecture

An important feature of Kivy is the separation of OS and Kivy API, which together with an API abstraction of OpenGL graphics and the use of modular core and input providers makes Kivy code OS and device independent. These modular core tasks are abstracted by the API to make it easy to use different providers depending on the platform on which app is run. For example to display text or graphics Kivy calls into the API and the API acts as an intermediate communication layer to the libraries of the target operating system. This not only helps to keep Kivy cross-platform, but also reduces the distribution size and keeps Kivy flexible(36).

Kivy focuses on touch-based interface with mouse events translated to simulated touch events, and is based on widgets ordered in layouts much like in Qt. These widgets take the form of text labels, images, input fields and so on which make up the user interface. Kivy is based on the Python coding language and so code is written in Python, but also using the Kivy language for widgets. The Kivy language is similar to Qt's QML language, but has its own unique traits such as CSS style rule definitions and templating. It should also be noted that the Kivy language uses indentation similar to that used by Python. Until recently the indentation had to be four spaces, but in the latest versions any number of spaces is allowed as long as it is a multiple of the spaces used on the first indented line(37).

4.4 Kivy Development

There is no IDE for Kivy or plugins to integrate Kivy with existing IDEs, but in the github documentation for Kivy there are some informal instructions for how to configure most IDEs, such as Visual Studio or Eclipse, to work with Kivy. The apps developed for this essay were coded using GEdit on Linux and Notepad++ on Windows. These text editors have syntax highlighting for Python (and other languages) and for Notepad++ there is an auto-indent feature and plugins can be downloaded with additional features. Even so there is no project management, auto-complete, debug functionality, user interface designer and other IDE features that provide development support.

Apps can be tested on Linux from the command line with “kivy <filename.py>”, while on Windows the easiest way to run a Kivy app is to drag the main python file to the “kivy.bat” file in the portable Kivy directory, which sets up the environment for Kivy and starts the Python interpreter.

Writing Kivy apps will be familiar to Python developers and typically consists of writing a collection of classes using the Kivy modules such as kivy.app and kivy.uix.widget in python files with the .py extension. Widgets, such as labels and buttons, can be defined in the Kivy language in files with the .kv extension and Kivy automatically looks for Kv files that are named the same as the app class, but without “App” if that is what the name ends with. Alternatively, Kv files can be manually loaded using the Kivy Builder class. Positioning and layout, colouring, sizing, text, actions and so on can be defined in the Kv files, including reusable widget styles.

4.5 Kivy Android Development

A Linux platform is required to deploy Android apps, so although the code could be written and tested on Windows or Mac, the Android packages must be created on Linux. As described in the section on Qt Setup the Android SDK, Android NDK, Apache ANT and Java Development Kit must be installed, but in addition the Jinja2 python module and Python for Android must be installed to deploy Kivy projects to Android. Python for Android takes care of the APK packaging and contains a small API for bootstrapping the Android app. It is downloaded from github and then has to be built for Kivy using

“./distribute.sh -m "kivy” which creates the distribution.

On Ubuntu 13.10 a number of additional packages also had to be installed to create the minimal build environment for building Python Kivy apps for Android. The build-essential, patch, git-core, ccache, ant, python-pip, python-dev, ia32-libs, libc6-dev-i386, lib32stdc++6 and lib32z1 packages were installed using the “apt-get install” command and then Cython had to be upgraded to the latest version with “pip install –upgrade cython”. Installation paths for the SDK, NDK along with version and API must be exported, and the Android binary must be added to the PATH variable. Knowing which packages need to be installed can be challenging and when writing the Kivy tests apps the instructions for setting up the prerequisites were found in the source documentation for the Python For Android github project(38).

To build the APK for a Kivy project from the Android for Python dist/default directory the build.py script is run with the required parameters. In the case of the Kivy Galaxy Portal app the following build instructions were used: ./build.py --dir ~/Projects/Kivy/KivyGalaxyPortal --package org.bmi.kivygalaxyportal --name "Kivy Galaxy Portal" --version 1.1.0 debug installd.

The Android Debug Bridge is used to install the APK on the device and for the above project the command “adb install -r bin/KivyGalaxyPortal-1.0.0-debug.apk” was used. However, the first time this was run it failed with an error “waiting for device” and the command had to be repeated for the app to be successfully installed.

The Kivy user interface is very customisable, but has a very distinct look that differs from the standard native Android UI. This makes Kivy a great choice for custom styled apps and games but may not always be suitable for utility apps that need to integrate closely with the Android UI and other Android apps.

4.6 Kivy Galaxy Portal

A Pong game was developed as part of a Kivy tutorial and deployed to an Android device with success and it was relatively easy to develop the game with a touch interface, animated ball and simple textures applied to the objects.

The goal of the Kivy Galaxy Portal was to create the same proof-of-concept app as for Qt to present a basic app with a login request to the user. This was accomplished by writing a Kivy app to invoke the login screen with a reference back to the app to allow the the login screen to call back into the main app with the result of the login check. A simple Kivy grid layout was used to draw label, text input and button widgets. The password field was set to hide the text by passing the parameter “password=True” in the Label constructor. A Kivy language file was used to define the widgets, but the buttons were bound to actions using Python code to invoke a function to login or close the app depending on the button pressed.

The limited documentation and lack of experience with the Kivy language meant that it took a good deal of trial-and-error to get the app working to validate the username and password and to correctly terminate the app. The result was satisfactory given the limited time, and although less functional than the equivalent Qt app it was successfully packaged using Python-for-Android and deployed to an Android Samsung Galaxy II.

The Kivy Pong tutorial consisted of just one Python file and one Kivy language file with three textures, while the Kivy Galaxy Portal app implementation consisted of three python files and one Kivy language file. Both projects are available on github(39) (40).

5 HTML5

Next...

6 Final Word

A wide selection of technologies have been investigated to learn about what is generally possible and the state of the art for cross platform app development. Because of

the learning process and broad exploration of frameworks, programming languages and tools the focus has been on developing only a few tutorials and a basic login app for the Galaxy interface for each technology. Together with research of app development in general this has provided insight not only what is possible with each approach, but also the benefits and drawbacks of working with the various technologies. Such as available documentation, the practical challenges of deploying apps to various platforms and which provide the most efficient and flexible development environment for my particular software development background.

Although initially considered, Android Developer Tools for Eclipse and Android Studio were skipped due to time limitations, but also because they are restricted to developing apps targeting only the Android platform. However, as the Android market is currently dominating the mobile device market these technologies may still be considered for the next stage after further review.

With the experience gained from this first stage that the next phase can be spent focusing on a more narrow set of technologies to gain a deeper knowledge about more advanced user interface features and networking. This next phase will focus on connecting with the Galaxy system to log in and retrieve data on jobs and manage them.

References

1. Mobile application software [Internet]. Wikipedia, the free encyclopedia. 2014 [cited 2014 Apr 5]. Available from: http://en.wikipedia.org/w/index.php?title=Mobile_application_software&oldid=602798480

2. FrontPage - Galaxy Wiki [Internet]. [cited 2014 Apr 6]. Available from: https://wiki.galaxyproject.org/

3. Smartphone [Internet]. Wikipedia, the free encyclopedia. 2014 [cited 2014 Apr 6]. Available from: http://en.wikipedia.org/w/index.php?title=Smartphone&oldid=602886822

4. Helft M, Markoff J. Google Enters the Wireless World. The New York Times [Internet]. 2007 Nov 5 [cited 2014 Apr 6]; Available from: http://www.nytimes.com/2007/11/05/technology/05cnd-gphone.html

5. Gartner Says Worldwide Smartphone Sales Reached Its Lowest Growth Rate With 3.7 Per Cent Increase in Fourth Quarter of 2008 [Internet]. [cited 2014 Apr 6]. Available from: http://www.gartner.com/newsroom/id/910112

6. Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013 [Internet]. [cited 2014 Apr 6]. Available from: http://www.gartner.com/newsroom/id/2665715

7. Firefox OS - Mozilla | MDN [Internet]. [cited 2014 Apr 16]. Available from: https://developer.mozilla.org/en-US/Firefox_OS

8. Ubuntu on phones | Ubuntu [Internet]. [cited 2014 Apr 16]. Available from: http://www.ubuntu.com/phone

9. Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010 [Internet]. [cited 2014 Apr 6]. Available from: http://www.gartner.com/newsroom/id/1543014

10. Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth [Internet]. [cited 2014 Apr 6]. Available from: http://www.gartner.com/newsroom/id/1924314

11. Study: Android Tablets Surpass iPads In Q1, Tablet Usage Up 282% Since 2011 [Internet]. Marketing Land. [cited 2014 Apr 6]. Available from: http://marketingland.com/study-android-tablet-users-surpass-ipad-users-by-34m-in-q1-with-tablet-usage-up-282-since-2011-2-47740

12. 7 MBA, 2013, 221 1:00 Pm, 4 253. The Future Of Mobile Development: HTML5 Vs. Native Apps [SLIDE DECK] [Internet]. Business Insider. [cited 2014 Apr 6]. Available from: http://www.businessinsider.com/html5-vs-native-apps-for-mobile-2013-6

13. Comparing Titanium and PhoneGap [Internet]. [cited 2014 Apr 8]. Available from: http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap/

14. iOS Human Interface Guidelines: Designing for iOS 7 [Internet]. [cited 2014 Apr 8]. Available from: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/index.html#//apple_ref/doc/uid/TP40006556

15. Design | Android Developers [Internet]. [cited 2014 Apr 8]. Available from: https://developer.android.com/design/index.html

16. Best Practice Guidelines [Internet]. App Quality Alliance; 2013 [cited 2014 Apr 8]. Available from: http://www.appqualityalliance.org/files/AQuA_best_practices_doc_v2_3_final_june_2013.pdf

17. Android SDK | Android Developers [Internet]. [cited 2014 Apr 9]. Available from: http://developer.android.com/sdk/index.html

18. Android Debug Bridge | Android Developers [Internet]. [cited 2014 Apr 9]. Available from: http://developer.android.com/tools/help/adb.html

19. Java SE | Oracle Technology Network | Oracle [Internet]. [cited 2014 Apr 12]. Available from: http://www.oracle.com/technetwork/java/javase/overview/index.html

20. OpenJDK [Internet]. [cited 2014 Apr 12]. Available from: http://openjdk.java.net/

21. Qt Project [Internet]. Digia Plc. 2013 [cited 2014 Feb 19]. Available from: http://qt-project.org/

22. C++ GUI Programming with Qt 4 > A Brief History of Qt - Pg. : Safari Books Online [Internet]. [cited 2014 Apr 12]. Available from: http://my.safaribooksonline.com/0131872494/pref04?portal=oreilly

23. Qt - Cross-platform application and UI development framework [Internet]. [cited 2014 Apr 12]. Available from: http://qt.digia.com/

24. Thelin J. Foundations of Qt development. Apress; 2007.

25. Ezust A, Ezust P. An Introduction to Design Patterns in C++ with Qt 4 (Bruce Perens Open Source). Prentice Hall PTR; 2006.

26. Summerfield M. Advanced Qt Programming: Creating Great Software with C++ and Qt 4. Prentice Hall Press; 2010.

27. C++ Qt Programming - YouTube [Internet]. [cited 2014 Apr 14]. Available from: https://www.youtube.com/playlist?list=PL2D1942A4688E9D63

28. QtCore Module | Documentation | Qt Project [Internet]. [cited 2014 Apr 14]. Available from: http://qt-project.org/doc/qt-4.8/qtcore.html

29. Widgets Classes | QtWidgets 5.2 | Documentation | Qt Project [Internet]. [cited 2014 Apr 14]. Available from: http://qt-project.org/doc/qt-5/widget-classes.html#advanced-widget-classes

30. Tarostar/QtGalaxyPortal [Internet]. GitHub. [cited 2014 Apr 14]. Available from: https://github.com/Tarostar/QtGalaxyPortal

31. Tarostar/GalaxyPortalUnitTest [Internet]. GitHub. [cited 2014 Apr 14]. Available from: https://github.com/Tarostar/GalaxyPortalUnitTest

32. Kivy: Crossplatform Framework for NUI [Internet]. [cited 2014 Apr 15]. Available from: http://kivy.org/

33. Virbel M, Hansen TE, Lobunets O. Kivy-A Framework for Rapid Creation of Innovative User Interfaces. 2011. p. 69–73.

34. Ulloa R. Kivy: Interactive Applications in Python [Internet]. Packt Publishing Ltd; 2013 [cited 2014 Apr 15]. Available from: http://www.google.com/books?hl=en&lr=&id=8Y79AAAAQBAJ&oi=fnd&pg=PT7&dq=+Kivy:+Interactive+Applications+in+Python&ots=fwjHUjcUIz&sig=z-FmZG3RlzACe7uYk8X8gpoMj3k

35. Hansen TE, Hourcade JP, Virbel M, Patali S, Serra T. PyMT: a post-WIMP multi-touch user interface toolkit. Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces [Internet]. ACM; 2009 [cited 2014 Apr 15]. p. 17–24. Available from: http://dl.acm.org/citation.cfm?id=1731907

36. Architectural Overview — Kivy 1.8.1-dev documentation [Internet]. [cited 2014 Apr 15]. Available from: http://kivy.org/docs/guide/architecture.html

37. Kivy Language — Kivy 1.8.1-dev documentation [Internet]. [cited 2014 Apr 15]. Available from: http://kivy.org/docs/api-kivy.lang.html

38. kivy/python-for-android [Internet]. GitHub. [cited 2014 Apr 15]. Available from: https://github.com/kivy/python-for-android

39. Tarostar/KivyPong [Internet]. GitHub. [cited 2014 Apr 15]. Available from: https://github.com/Tarostar/KivyPong

40. Tarostar/KivyGalaxyPortal [Internet]. GitHub. [cited 2014 Apr 15]. Available from: https://github.com/Tarostar/KivyGalaxyPortal