
Qt Programming Course Documentation

Release 1.0

Miguel Menéndez Carrión

Oct 12, 2021

1	Introduction to Clickable	1
2	Introduction	13
3	Web App - Basic	19
4	QML App - Basic	35
5	QML App - Format	45
6	QML App - Events	65
7	QML App - Database access	79
8	QML App - Charts	91
9	Compilation of documentation	103
10	Ubuntu SDK (Legacy)	105
11	Indices and tables	129

Introduction to Clickable

The source code is the DNA of an application. It includes a series of instructions that when executed allow multiple functions to be performed. For example, when we open a web browser like Firefox, the source code tells you how to download a web page or how to display it. In order for these instructions, which are defined in a text file, to be executed in an application, they must be compiled. With the compilation, the instructions are translated into machine code that can be executed directly by the computer. The machine code varies depending on the processor running the application. We can therefore distinguish various architectures. On the desktop we have i386 / AMD64. With mobile devices we have, in most cases, ARM architecture. To do the whole process we will use Clickable.

We have seen that the compiler translates the source code into machine code. A compiler running on a computer will generate an application with the same architecture. You cannot directly generate an application for another architecture such as ARM. Under these conditions, how can we compile an application for ARM if we use a desktop computer? The solution lies in the use of containers. For practical purposes, the container allows applications of another architecture to be run on the computer.

Therefore, we will have a container with all the necessary tools to compile an application. We will pass you the source code and it will generate an application prepared for the ARM architecture. The next step is to pass that application to Ubuntu Touch and use it.

1.1 Clickable

There are many ways to create a container for cross-compiling (compiling an application from another architecture on the desktop). Canonical had it integrated into the Ubuntu Touch SDK. Since that SDK is no longer supported, the way to work now is to use Clickable. If you want to program an application for Ubuntu Touch you have to use this tool. It is the one that has official support and receives updates with bug fixes and new features.

Next we will see how to create an application with Clickable as well as compile the typical “Hello world”. Initially we will see the steps in console. Later we will see how to integrate Clickable with the programming environment (Qt). The only requirement is to have GNU / Linux on the computer. It can be run on a real machine or in a virtual machine. You have all the links with the information of the chapter in the References section.

There are several ways to install Clickable. I will take the recommended form as a basis.

1.2 Dependencies

The container that is used with Clickable is docker. As auxiliary tools we have Git and Python. After installing docker it is recommended to restart the computer.

```
$ sudo apt install docker.io adb git python3 python3-pip
```

```
mimcar@rhudean:~$ sudo apt install docker.io adb git python3 python3-pip
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
python3 ya está en su versión más reciente (3.6.7-1~18.04).
fijado python3 como instalado manualmente.
Starting pkgProblemResolver with broken count: 0
Starting 2 pkgProblemResolver with broken count: 0
Done
Se instalarán los siguientes paquetes adicionales:
android-libadb android-libbase android-libboringssl android-libcrypto-utils android-libcutils android-liblog
android-sdk-platform-tools-common bridge-utils build-essential cgroups-mount containerd dh-python dpkg-dev fakeroot g++ g++-7 gcc gcc-7
git-man libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl libasan4 libatomic1 libc-dev-bin libc6-dev libcilkrts5
liberror-perl libexpat1-dev libfakeroot libgcc-7-dev libitm1 liblsan0 libmpx2 libpython3-dev libpython3.6-dev libstdc++-7-dev libtsan0
libubsan0 linux-libc-dev make manpages-dev pigz python-pip-whl python3-asn1crypto python3-crypto python3-cryptography python3-dev
python3-distutils python3-keyring python3-keyrings.alt python3-lib2to3 python3-secretstorage python3-setuptools python3-wheel python3.6-dev
runc ubuntu-fan
Paquetes sugeridos:
ifupdown aufs-tools btrfs-progs debotstrap docker-doc rinse zfs-fuse | zfsutils debian-keyring g++-multilib g++-7-multilib gcc-7-doc
libstdc++6-7-dbg gcc-multilib autoconf automake libtool flex bison gcc-doc gcc-7-multilib gcc-7-locales libgcc1-dbg libgomp1-dbg
libitm1-dbg libatomic1-dbg libasan4-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg libcilkrts5-dbg libmpx2-dbg libquadmath0-dbg git-daemon-run
| git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn glibc-doc libstdc++-7-doc make-doc
python-crypto-doc python-cryptography-doc python3-cryptography-vectors gir1.2-gnomekeyring-1.0 python-secretstorage-doc
python-setuptools-doc
Se instalarán los siguientes paquetes NUEVOS:
adb android-libadb android-libbase android-libboringssl android-libcrypto-utils android-libcutils android-liblog
android-sdk-platform-tools-common bridge-utils build-essential cgroups-mount containerd dh-python docker.io dpkg-dev fakeroot g++ g++-7
gcc gcc-7 git git-man libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl libasan4 libatomic1 libc-dev-bin libc6-dev
libcilkrts5 liberror-perl libexpat1-dev libfakeroot libgcc-7-dev libitm1 liblsan0 libmpx2 libpython3-dev libpython3.6-dev libstdc++-7-dev
libtsan0 libubsan0 linux-libc-dev make manpages-dev pigz python-pip-whl python3-asn1crypto python3-crypto python3-cryptography python3-dev
```

Clickable

dependencies

PIP is a Python tool that allows you to easily install Python modules. It is important to use the command with python3, if we do not use the same name we can install the version of Python by mistake.

```
$ pip3 install --user --upgrade clickable-ut
```

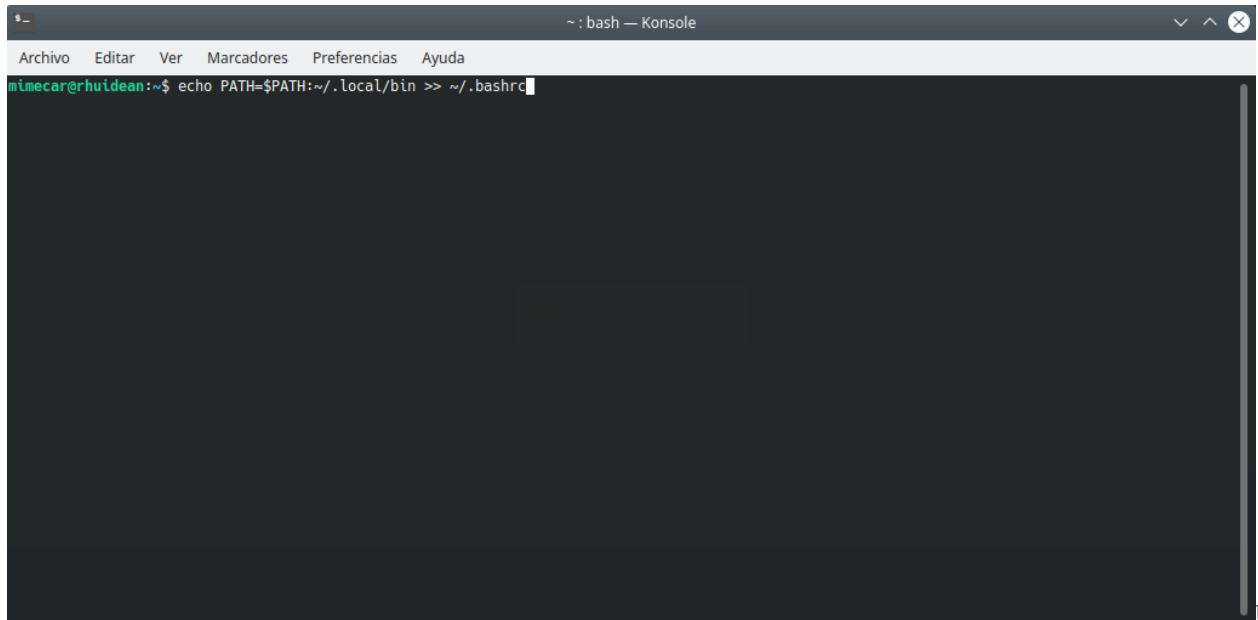
```
mimcar@rhudean:~$ pip3 install --user --upgrade clickable-ut
Collecting clickable-ut
  Downloading https://files.pythonhosted.org/packages/5f/1f/d01b6f043f6c8eb24a344a68a511b43e417f5190e75a6d9e41778e7c702/clickable_ut-6.13.1-py3-none-any.whl (76kB)
    100% |#####| 81kB 1.3MB/s
Collecting jsonschema (from clickable-ut)
  Downloading https://files.pythonhosted.org/packages/c5/8f/51e89ce52a085483359217bc72c0df6e75ee595d5b1d4b5ade40c7e018b8/jsonschema-3.2.0-py2.py3-none-any.whl (56kB)
    100% |#####| 61kB 4.9MB/s
Collecting requests (from clickable-ut)
  Downloading https://files.pythonhosted.org/packages/1a/70/1935c770cb3be63a8b78ced23d7e0f3b187f5cbfab4749523ed65d7c9b1/requests-2.23.0-py2.py3-none-any.whl (58kB)
    100% |#####| 61kB 1.8MB/s
Collecting cookiecutter (from clickable-ut)
  Downloading https://files.pythonhosted.org/packages/95/83/83ebf950ec99b02c61719ccb116462844ba2e873df7c4d40afc962494312/cookiecutter-1.7.2-py2.py3-none-any.whl
Collecting importlib-metadata; python_version < "3.8" (from jsonschema->clickable-ut)
  Downloading https://files.pythonhosted.org/packages/ad/e4/891bfca868ccabc619942f7940c77a8a4b45fd8367098955bb7e152fb1/importlib_metadata-1.6.0-py2.py3-none-any.whl
Collecting pyrsistent>=0.14.0 (from jsonschema->clickable-ut)
  Downloading https://files.pythonhosted.org/packages/9f/0d/cbca4d0bbc5671822a59f270e4ce3f2195f8a899c97d0d5abb81b191efb5/pyrsistent-0.16.0.tar.gz (108kB)
    100% |#####| 112kB 3.0MB/s
Collecting attrs>=17.4.0 (from jsonschema->clickable-ut)
  Downloading https://files.pythonhosted.org/packages/a2/db/4313ab3be961f7a763066401fb77f7748373b6094076ae2bda2806988af6/attrs-19.3.0-py2.py3-none-any.whl
Collecting six>=1.11.0 (from jsonschema->clickable-ut)
  Downloading https://files.pythonhosted.org/packages/ee/ff/48bde5c0f013094d729fe4b0316ba2a24774b3ff1c52d924a8a4cb04078a/six-1.15.0-py2.py3-none-any.whl
Collecting setuptools (from jsonschema->clickable-ut)
```

PIP

dependencies

Finally we update the path. You need to close the console and open it again for the new path to be used.

```
$ echo PATH = $ PATH: ~ / .local / bin >> ~ / .bashrc
```



the path to use Clickable

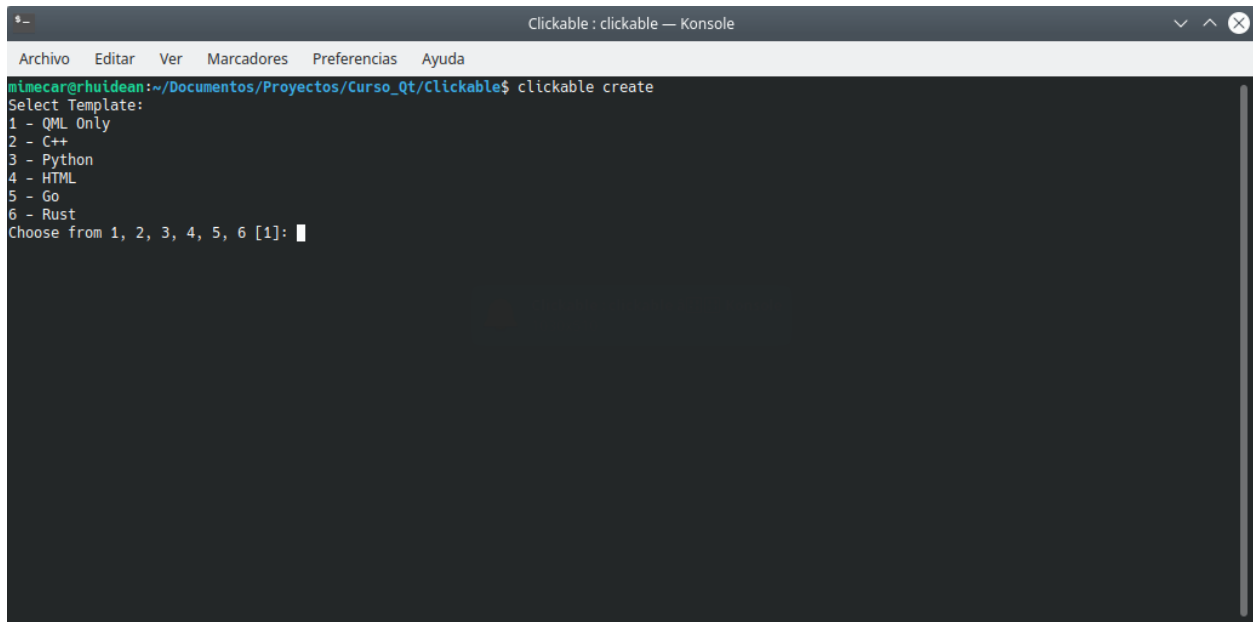
With these steps we already have everything ready to start working. It is possible to use nightly versions (they are daily compilations) of Clickable. Although it may have more functions than the stable version, it may also have errors. For development I recommend you always use the stable version. You will save a lot of time.

1.3 Creation of the first application

Before starting, I recommend creating a folder for the application. It is important that the route does not have spaces. For example, if you are going to program applications a possible route would be `~/Documents/Course_Qt/Clickable`. We create the route and open a console. You have to go to that folder using the command `cd`.

We start the wizard with the command: `clickable create`. Then we can choose the template we want to use. We select the first option. The options are:

- QML Only: QML + JavaScript
- C ++
- Python
- HTML5
- Go
- Rust

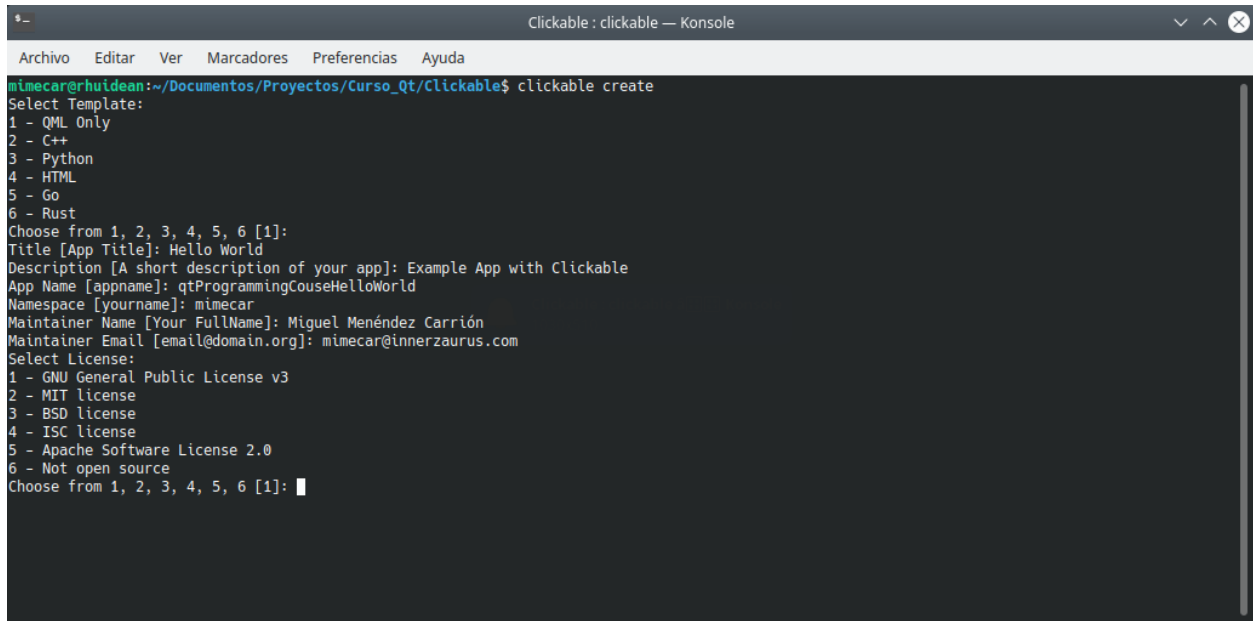


```
Clickable : clickable — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
mimecar@rhuidean:~/Documentos/Proyectos/Curso_Qt/Clickable$ clickable create
Select Template:
1 - QML Only
2 - C++
3 - Python
4 - HTML
5 - Go
6 - Rust
Choose from 1, 2, 3, 4, 5, 6 [1]:
```

Clickable

templates

The next step is to fill in the basic information of the application.



```
Clickable : clickable — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
mimecar@rhuidean:~/Documentos/Proyectos/Curso_Qt/Clickable$ clickable create
Select Template:
1 - QML Only
2 - C++
3 - Python
4 - HTML
5 - Go
6 - Rust
Choose from 1, 2, 3, 4, 5, 6 [1]:
Title [App Title]: Hello World
Description [A short description of your app]: Example App with Clickable
App Name [appname]: qtProgrammingCouseHelloWorld
Namespace [yourname]: mimecar
Maintainer Name [Your FullName]: Miguel Menéndez Carrión
Maintainer Email [email@domain.org]: mimecar@innerzaurus.com
Select License:
1 - GNU General Public License v3
2 - MIT license
3 - BSD license
4 - ISC license
5 - Apache Software License 2.0
6 - Not open source
Choose from 1, 2, 3, 4, 5, 6 [1]:
```

Application

data

Then we choose the license. By default, the GPL 3 license is used. We leave the rest of the questions with the default values.

```

2 - C++
3 - Python
4 - HTML
5 - Go
6 - Rust
Choose from 1, 2, 3, 4, 5, 6 [1]:
Title [App Title]: Hello World
Description [A short description of your app]: Example App with Clickable
App Name [appname]: qtProgrammingCouseHelloWorld
Namespace [yourname]: mimecar
Maintainer Name [Your FullName]: Miguel Menéndez Carrión
Maintainer Email [email@domain.org]: mimecar@innerzaurus.com
Select License:
1 - GNU General Public License v3
2 - MIT license
3 - BSD license
4 - ISC license
5 - Apache Software License 2.0
6 - Not open source
Choose from 1, 2, 3, 4, 5, 6 [1]:
Copyright Year [2020]:
Git Tag Versioning [n]:
Save as Default [n]:

Successfully created app "qtprogrammingcousehelloworld.mimecar"
Get started with the following commands:

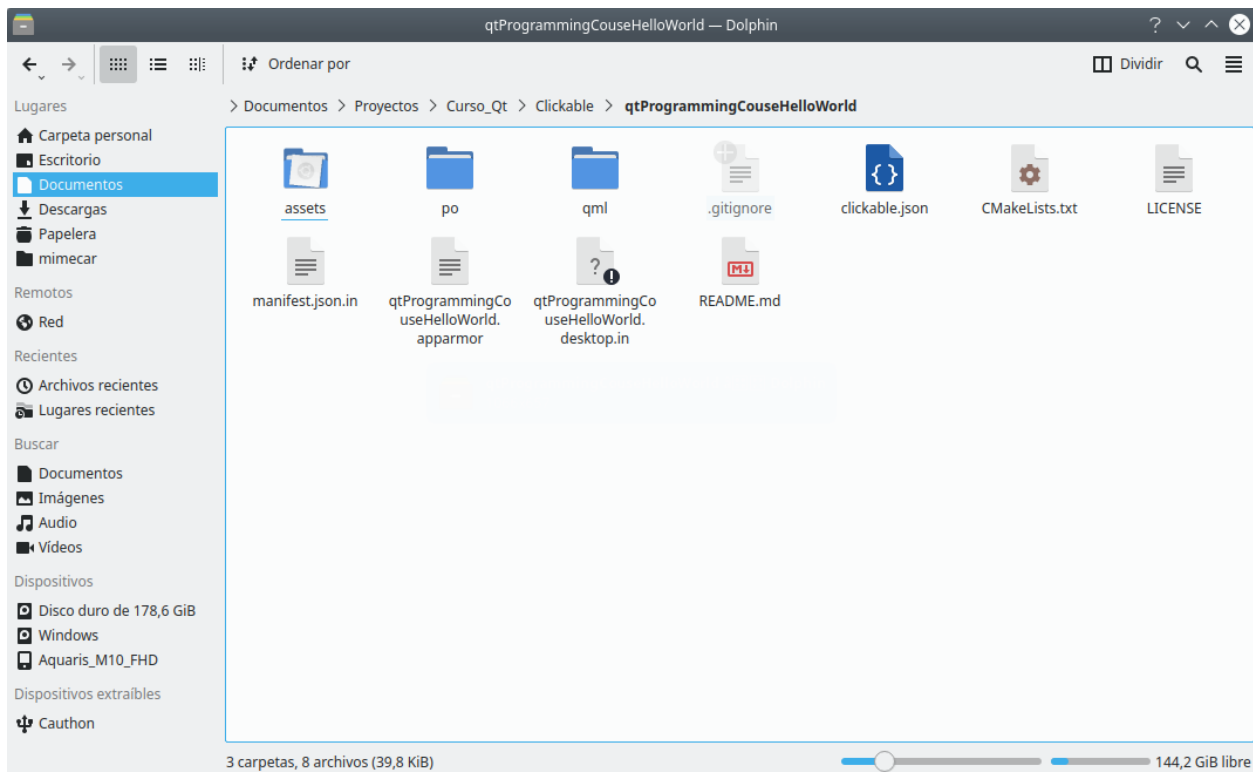
$ cd /home/mimecar/Documentos/Proyectos/Curso_Qt/Clickable/qtProgrammingCouseHelloWorld
$ clickable
mimecar@huidean:~/Documentos/Proyectos/Curso_Qt/Clickable$

```

Wizard

Summary

If we open the project folder, we will see the structure that the wizard has created.

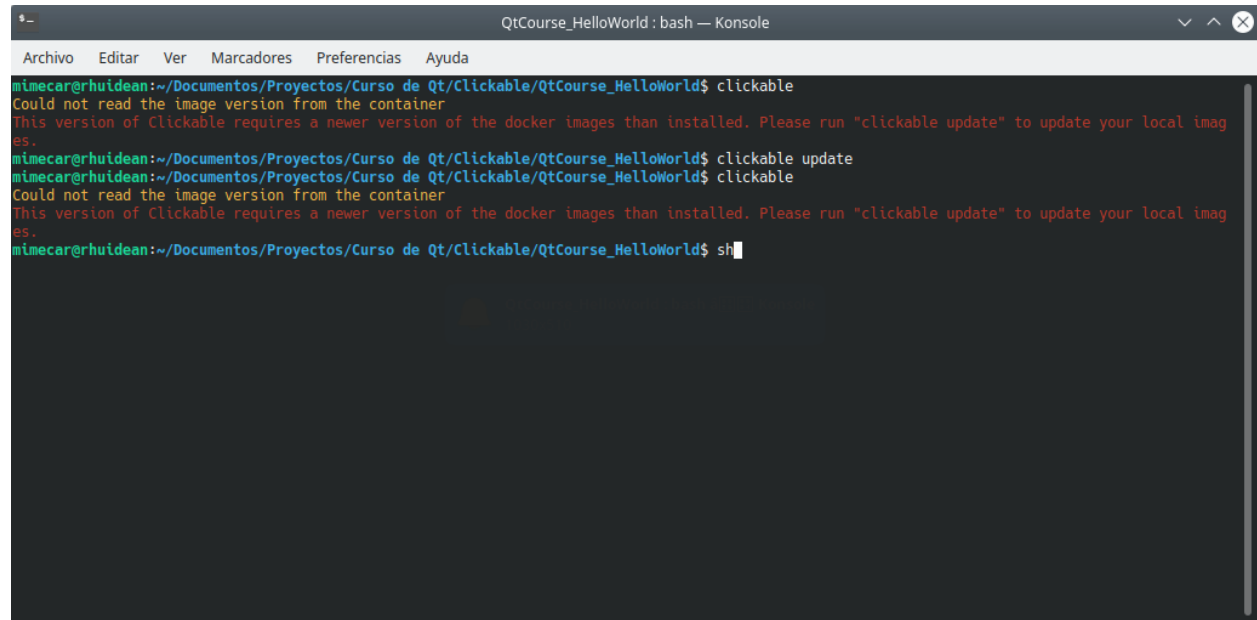


Files

in the project folder

1.4 Run the application on the desktop

To run the application on the desktop we enter the folder that has been created and run the command: `clickable desktop`. The first time the container is initialized. It is normal to ask for administrator permissions. If you get the message, you have to run `clickable update` and then `clickable desktop`. The message may continue to appear. If that happens, check that the name has no spaces.

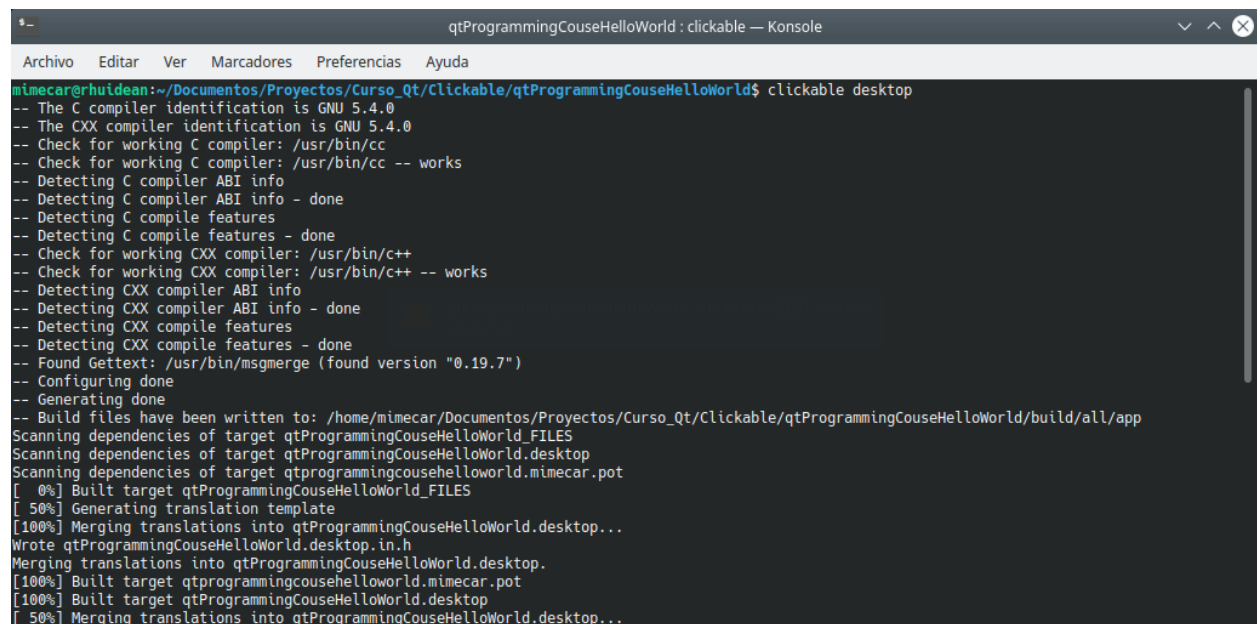


```
QtCourse_HelloWorld : bash — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
mimicar@rhuidean:~/Documentos/Proyectos/Curso de Qt/Clickable/QtCourse_HelloWorld$ clickable
Could not read the image version from the container
This version of Clickable requires a newer version of the docker images than installed. Please run "clickable update" to update your local images.
mimicar@rhuidean:~/Documentos/Proyectos/Curso de Qt/Clickable/QtCourse_HelloWorld$ clickable update
mimicar@rhuidean:~/Documentos/Proyectos/Curso de Qt/Clickable/QtCourse_HelloWorld$ clickable
Could not read the image version from the container
This version of Clickable requires a newer version of the docker images than installed. Please run "clickable update" to update your local images.
mimicar@rhuidean:~/Documentos/Proyectos/Curso de Qt/Clickable/QtCourse_HelloWorld$ sh
```

Container

error

All the elements necessary to compile the application will be downloaded.



```
qtProgrammingCouseHelloWorld : clickable — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
mimicar@rhuidean:~/Documentos/Proyectos/Curso_Qt/Clickable/qtProgrammingCouseHelloWorld$ clickable desktop
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Gettext: /usr/bin/msgmerge (found version "0.19.7")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mimicar/Documentos/Proyectos/Curso_Qt/Clickable/qtProgrammingCouseHelloWorld/build/all/app
Scanning dependencies of target qtProgrammingCouseHelloWorld_FILES
Scanning dependencies of target qtProgrammingCouseHelloWorld.desktop
Scanning dependencies of target qtprogrammingcousehelloworld.mimicar.pot
[ 0%] Built target qtProgrammingCouseHelloWorld_FILES
[ 50%] Generating translation template
[100%] Merging translations into qtProgrammingCouseHelloWorld.desktop...
Wrote qtProgrammingCouseHelloWorld.desktop.in.h
Merging translations into qtProgrammingCouseHelloWorld.desktop.
[100%] Built target qtprogrammingcousehelloworld.mimicar.pot
[100%] Built target qtProgrammingCouseHelloWorld.desktop
[ 50%] Merging translations into qtProgrammingCouseHelloWorld.desktop...
```

Starting

the desktop build

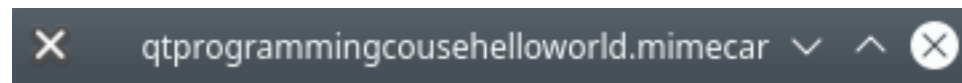
After a good cup of coffee ...

```
qtProgrammingCourseHelloWorld : clickable — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
[100%] Merging translations into qtProgrammingCourseHelloWorld.desktop...
Wrote qtProgrammingCourseHelloWorld.desktop.in.h
Merging translations into qtProgrammingCourseHelloWorld.desktop.
[100%] Built target qtProgrammingCourseHelloWorld.mimecar.pot
[100%] Built target qtProgrammingCourseHelloWorld.desktop
[ 50%] Merging translations into qtProgrammingCourseHelloWorld.desktop...
Merging translations into qtProgrammingCourseHelloWorld.desktop.
[ 50%] Built target qtProgrammingCourseHelloWorld.desktop
[ 50%] Built target qtProgrammingCourseHelloWorld_FILES
[100%] Generating translation template
Wrote qtProgrammingCourseHelloWorld.desktop.in.h
[100%] Built target qtprogrammingcousehelloworld.mimecar.pot
Install the project...
-- Install configuration: "Release"
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//manifest.json
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//qtProgrammingCourseHelloWorld.apparmor
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//assets
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//assets/logo.svg
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//qml
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//qml/Main.qml
-- Installing: /home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install//qtProgrammingCourseHelloWorld.desktop
Successfully built package in './qtprogrammingcousehelloworld.mimecar_1.0.0_all.click'.
non-network local connections being added to access control list
Mounting device home to /home/mimecar/.clickable/home
timedatectl: opción incorrecta -- «p»
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-mimecar'
"/home/mimecar/Documents/Proyectos/Curso_Qt/Clickable/qtProgrammingCourseHelloWorld/build/all/app/install/%U" does not exist.
```

Compilation

of the completed desktop application

The sample application will appear when the process is complete.



Hello World



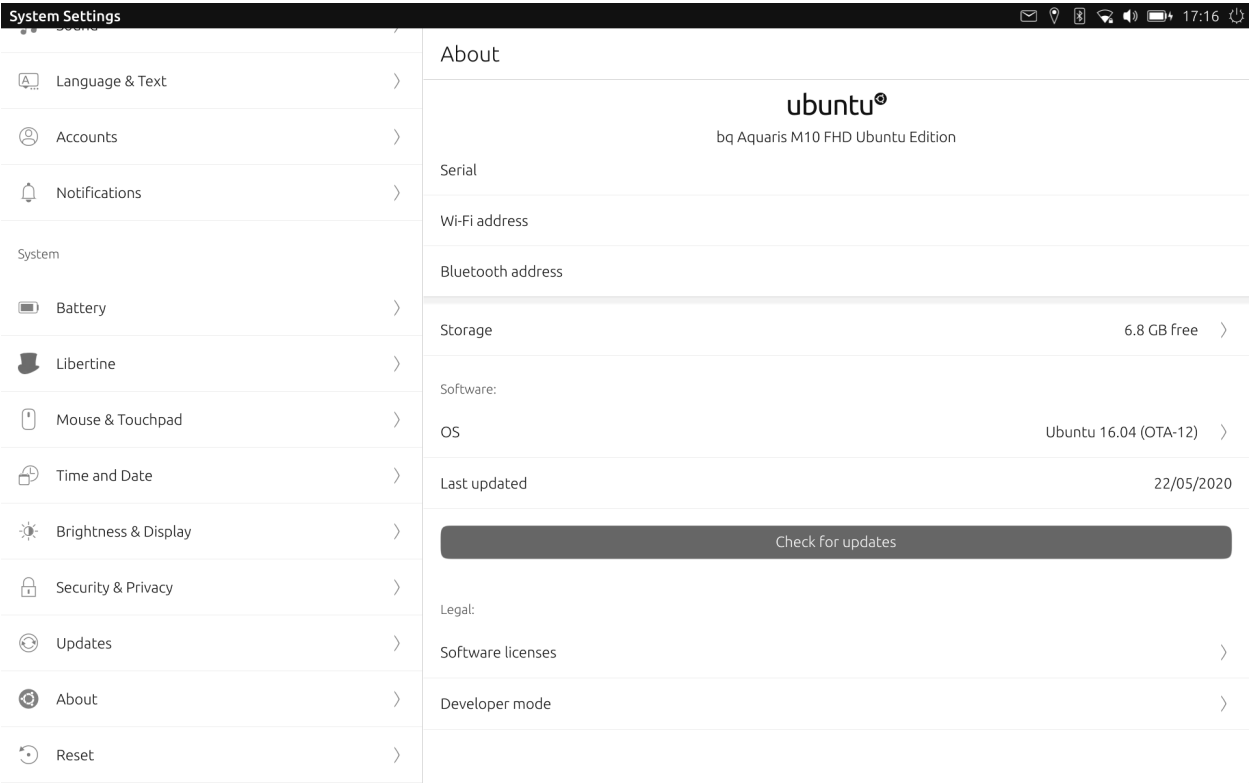
Hello world example

1.5 Run the app on the tablet

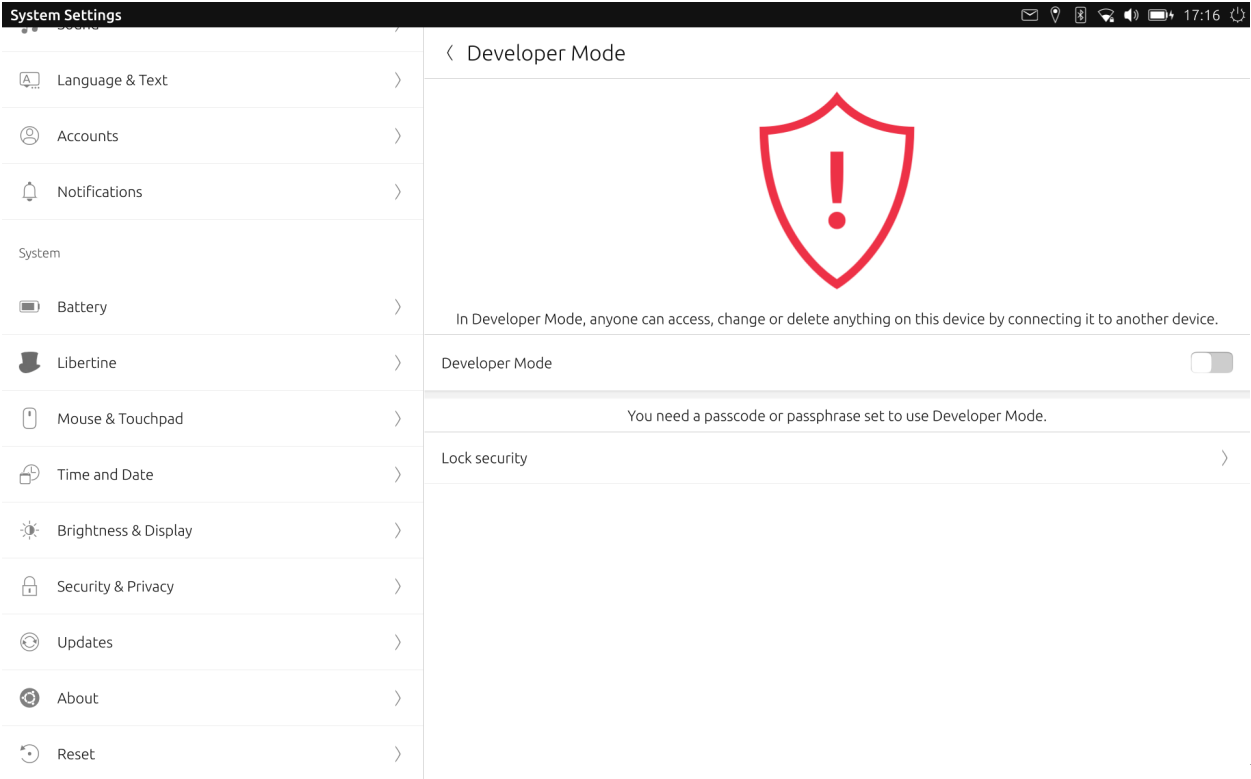
The steps are the same as before, the difference is the command we use. To compile for Ubuntu Touch just type `clickable`. The command launches the application directly in Ubuntu Touch. At this point we have to have the device connected to the computer and enable the development options.

For an M10 FHD with Ubuntu Touch (OTA-12), the steps would be:

- We enter About ...

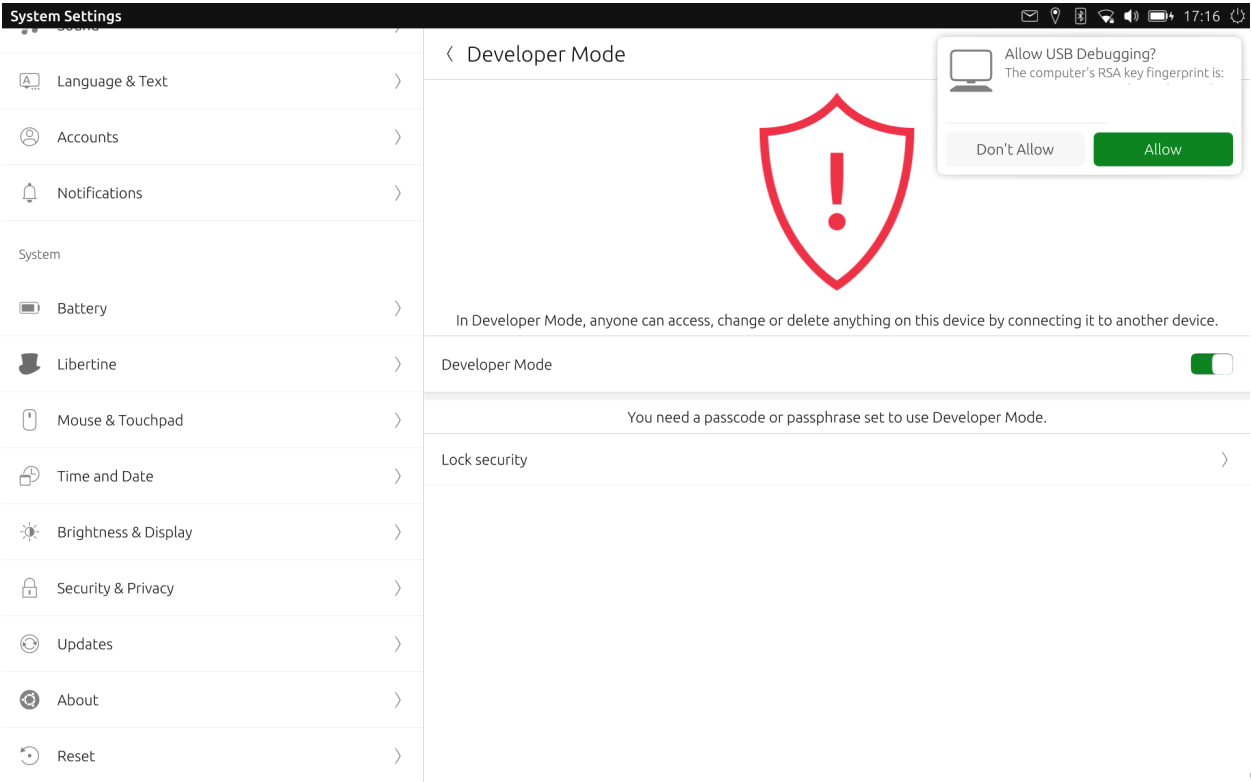


- Development options.



options

- We activate the development options (we will have to have a password to access the tablet). A connection request notice has to come out.



from PC

When everything is configured, we launch clickable and the application will appear on the device.



Hello World!

Demo

on Ubuntu Touch

1.6 Conclusions

In this section we have seen the basic concepts to use Clickable from the console. An important detail is that we do not depend on the Ubuntu Touch SDK. For this reason it should work on current desktop distributions. It may be that some package is called different but if we install Clickable using PIP, it should work without problems. There are many things to see in Clickable. To make it more friendly we will see them little by little.

Clickable can be integrated with Qt Creator (the version that is in the distribution repositories). In the next installment we will see how to do it and on that basis the rest of the course deliveries will be developed. The revised chapters will not depend on the virtual machine and will be easier to follow.

1.7 References

- [Clickable](#)

1.8 Credits

- Writer: Miguel Menéndez Carrión
- English translator: Milan Korecky
- English translator: lionelb

CHAPTER 2

Introduction

This course is a new project to learn how to program applications that work on Ubuntu Touch. In the process I will generate documentation with all the phases of development of an application: requirements acquisition, implementation and distribution in the Ubuntu store. One of the problems found in Ubuntu Touch is the lack of applications, both, in number and in functionality. I do not expect to change this situation in the short term, but one way to change it, is by programming applications and helping other users do the same. Only in this way it will (it) be possible to change this situation.

This course is not a master class in which I explain something and others repeat them automatically. The idea, is to publish chapters and complete the course documentation with the feedback of the other users. If a particular block gets more interest, it can be extend later. There is no problem in asking questions in the resources associated with the course.

The documentation will be structured as a book. Its access is free and any user can read it in the browser or as PDF, ePub or Mobi files. You can [add comments to the book](#) in the GitHub repository. The examples source code is hosted in GitHub and uses Git as version control.

Finally I want to thank the users who have encouraged me to start this madness, among them are kain_X_X and LarreaMikel. A course of this type can not be done by a single user. Evolving and achieving larger goals is only possible when many people contribute to.

2.1 Previous Knowledge

Due to the subject of the course itself, some programming skills are requested. In this course we will mainly use QML for the user interface and JavaScript or C/C++ for the logic. It helps to know either of the two languages, although it will not be critical. Each chapter will explain the basic elements and a bibliography will be included for the user to consult if in doubt.

The Ubuntu Touch Software Development Kit (SDK) is being released for the Ubuntu distribution. It will therefore be necessary to use Ubuntu or any of the distributions that take it as a base. Not meeting this requirement is not a serious problem either, because you can do the same in a virtual machine or using a Live USB.

To make it easier to follow the course, I will only put the most important parts of the source code. The rest of the files will be in a source repository in Launchpad. It would be helpful knowing the basic commands of Bazaar.

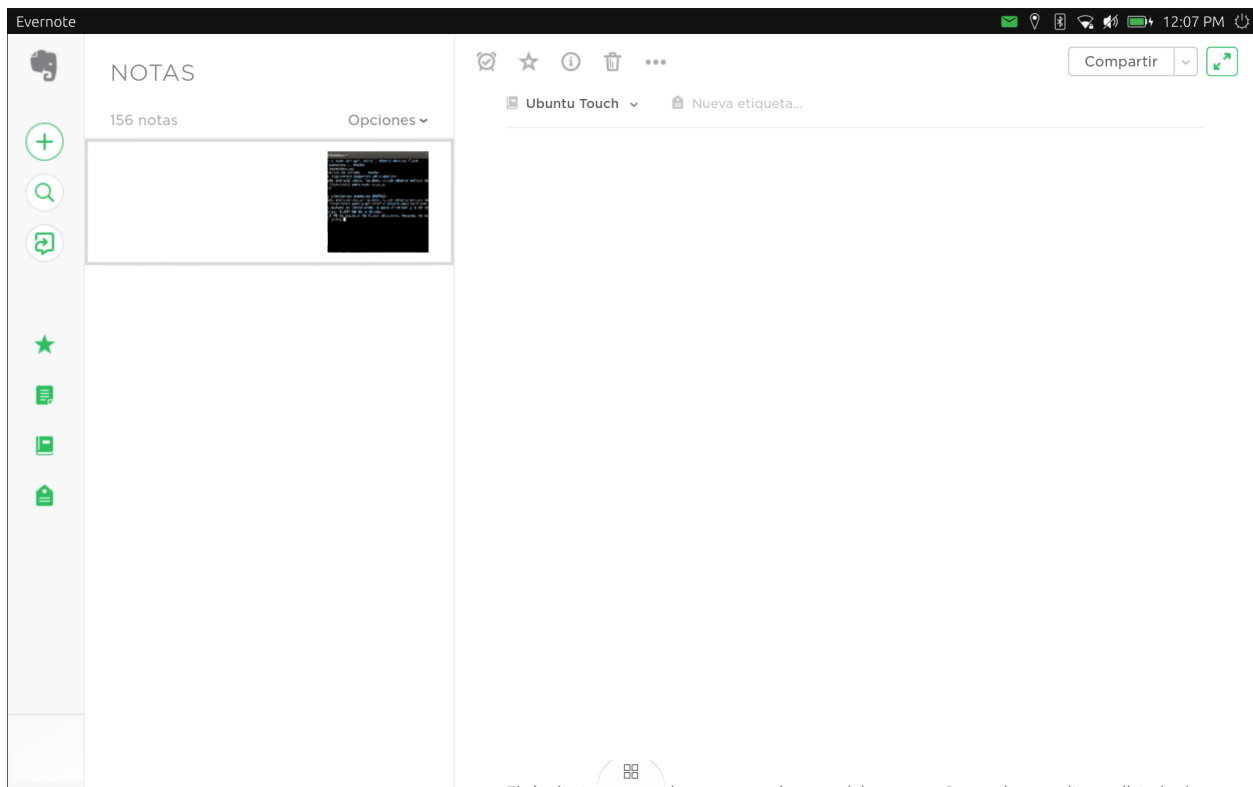
2.2 Objectives of the Course

The main objective of the course is to learn how to program applications for Ubuntu Touch while having fun. Programming is a time-consuming task and you have to like what you are doing. Applications can be simple or complex, the important thing is that they solve a need that we have. For example, an application that has a list of plants in the garden and let us know when we have to water them.

A good design in the logic of the application can greatly reduce development time. In the same way a bad design can cause to end up throwing the code to the recycle bin and starting over, because it is easier to start with a different approach.

2.3 Types of Applications

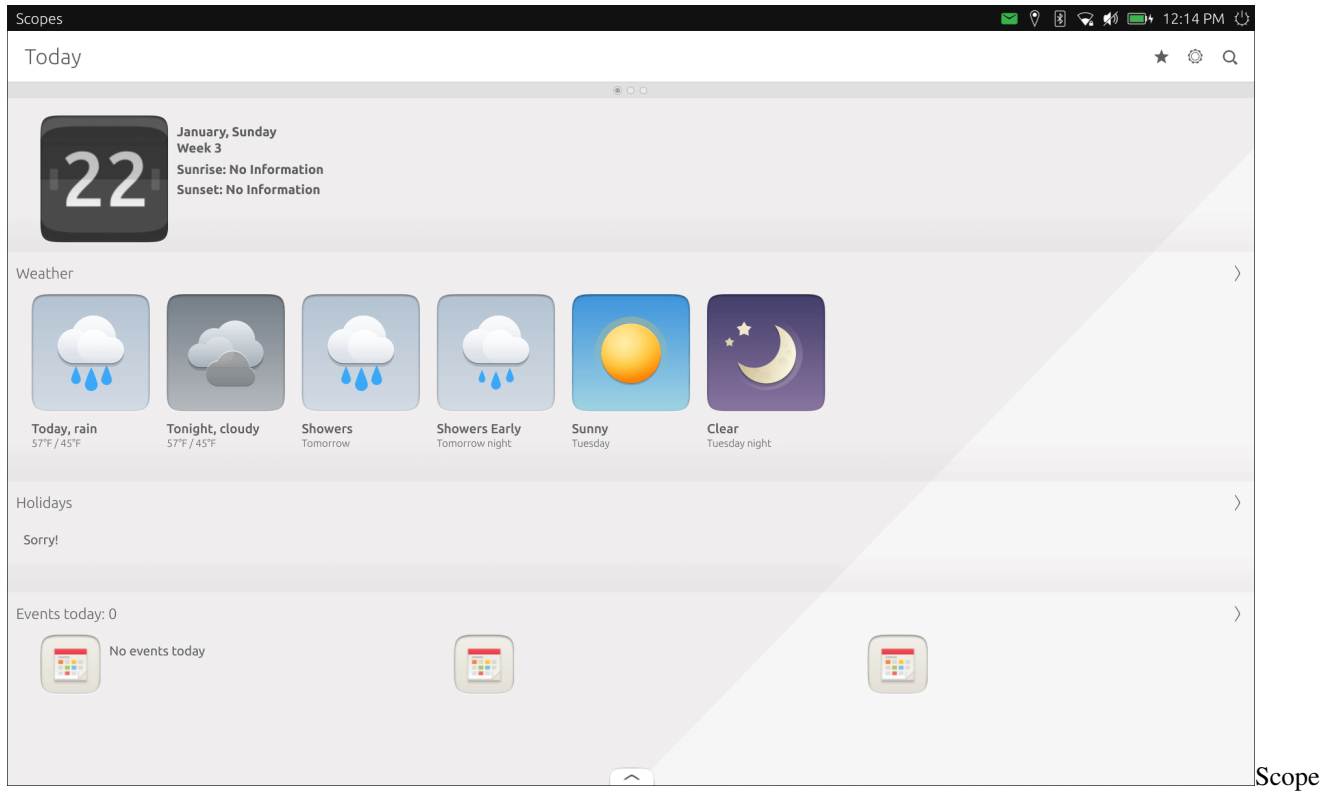
Ubuntu Touch has three types of applications. We can find Web Applications (WebApps), Scopes and Native Applications. A web application is basically a web browser tab that runs independently. It has its own icon in Unity (the application launcher) and can contain remote information of any type. For security reasons a Web application does not have access to the local content of the terminal.



WebApp

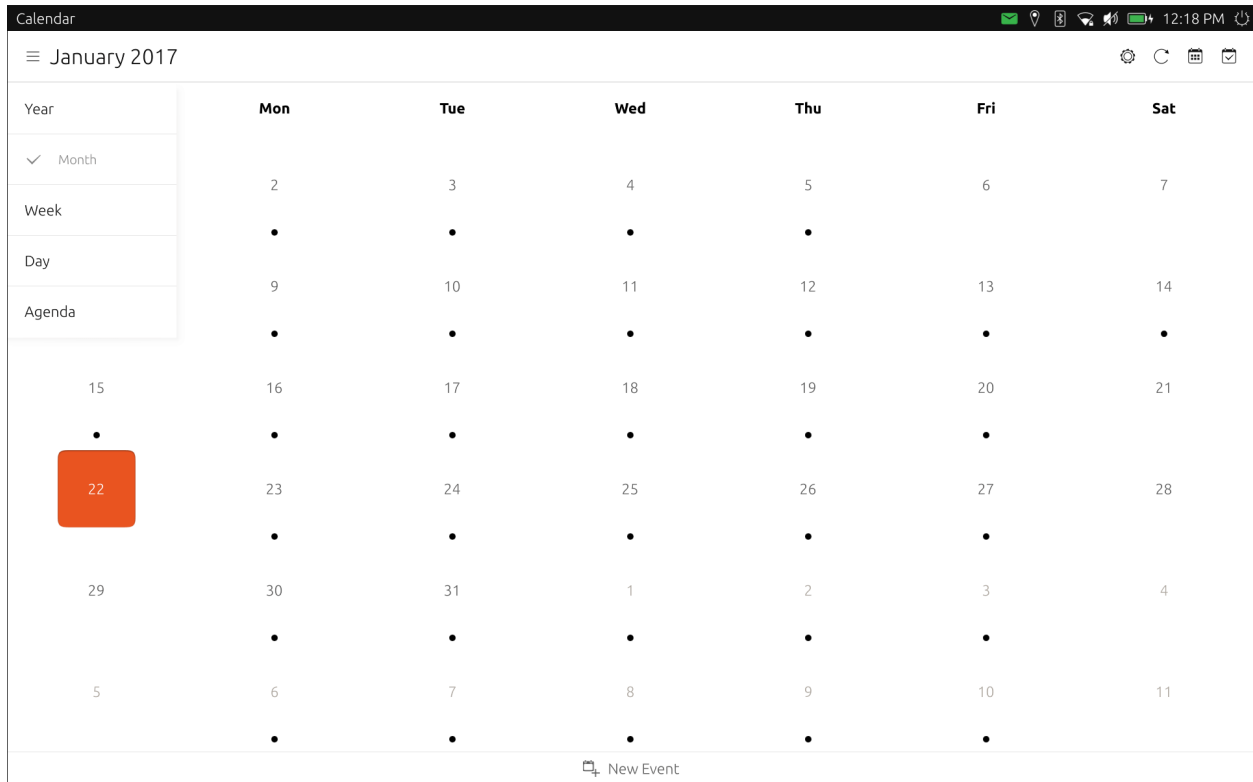
Example

The scope is the second type of application found in Ubuntu Touch. To some extent it behaves like a screen that shows information to the user. The information can be external, for example the weather forecast, or internal in the form of information aggregator. An example of this case would be the “Today” scope. This scope shows information from different applications.



Example

Finally we have native applications. In this case the applications can access all the resources of the phone and are, eventually, more complex than the Web applications and the scopes. Applications are confined in Ubuntu Touch and can only access their own information. If we want to access information from other applications, we need to pass it through the content-hub. As an example of native application we have the calendar.



Native

Application Example

2.4 Evolution of the Course

One detail that I want to point out (and that you will get tired of reading it throughout the course) is that this course is not a master class. It is important that you participate with either questions, suggestions or errors. The order of the chapters can vary and chapters that were already closed can be opened to add new content. This course is alive and can only be improved if we all get involved. It doesn't matter if questions are very basic or what the other users would say. The main reason of following this course is learning. Remember: The only stupid questions is the one that you don't ask.

Access to the mailing list is open and only a Launchpad account is needed. There is no censorship except several cases of common sense:

- The questions must be related to the course.
- Spam of any kind is not allowed.
- Attacking other users is not allowed.

If any of these rules is broken, I will warn the person first. If the user continues with its attitude, will be asked to leave the mailing list. I hope I never have to get to that end.

This course is not set as a whole and therefore I will write it week after week. For this reason, it is possible that some error appears. If this is the case, do not hesitate to warn me in order to correct it. This course is an opportunity to create content and give a boost to Ubuntu Touch.

2.5 Resources

- Mailing List: <https://launchpad.net/~ubuntu-touch-programming-course>
- Trello Board: <https://trello.com/b/gQEXHP3v/ubuntu-touch-programming-course>

2.6 People who have collaborated

- Larrea Mikel: revision of the chapter in Spanish.
- Cesar Herrera: revision of the English translation.
- Joan CiberSheep: revision of the English translation.

In this chapter we'll study the creation of a Web App and will go through all phases of the process, ending with the creation of a click package for Ubuntu Touch. To complete this chapter it will be required to have the Ubuntu SDK installed. It is also advisable to have completed the sample applications from the previous chapter.

3.1 What is a Web App?

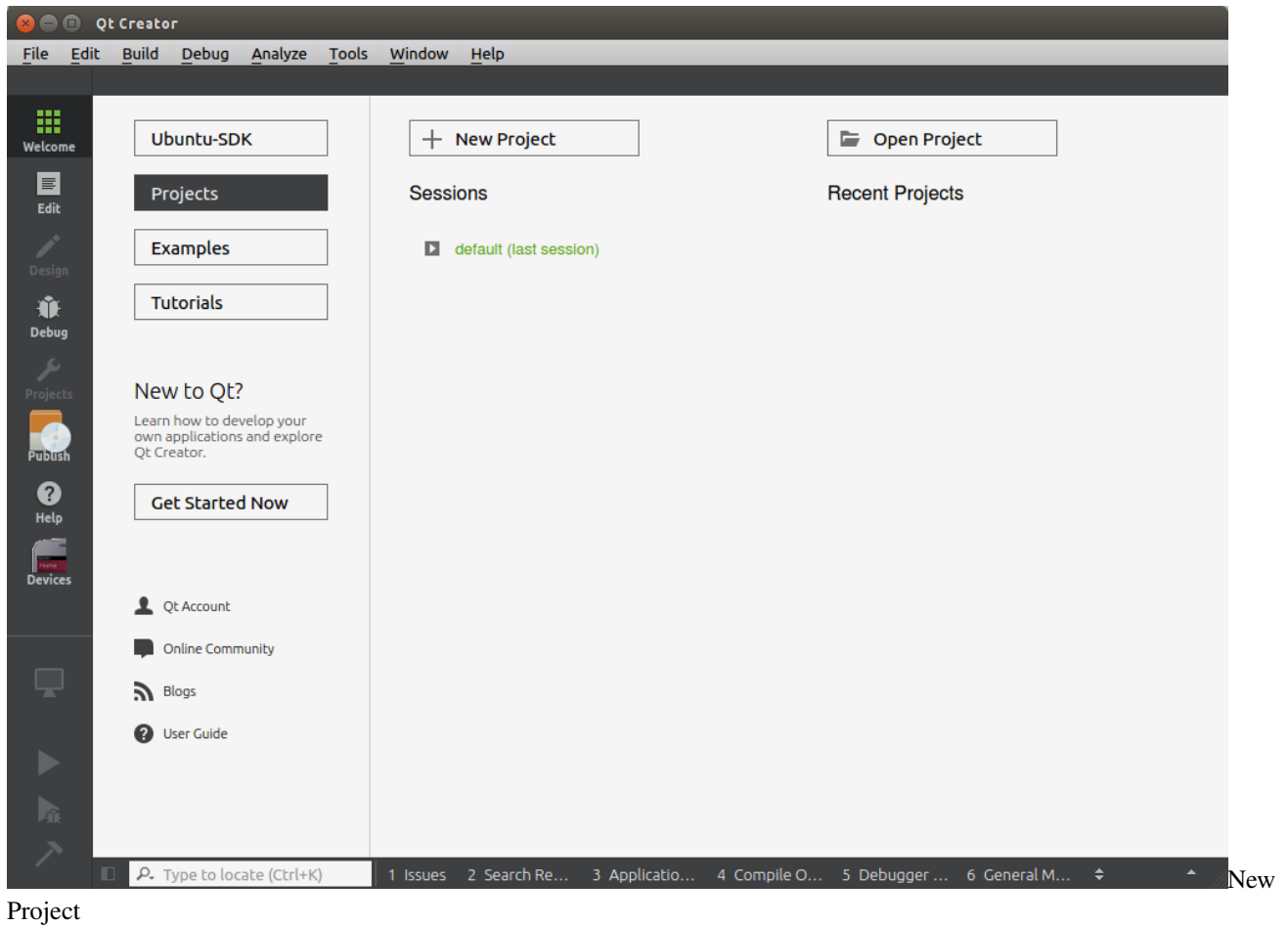
A Web App is a type of Ubuntu Touch application. It works by displaying a web page as if it were a standalone application. Like any other application, it will appear in the scope of applications and can be managed from the Ubuntu Store. You may wonder why display a web page as an app if the browser already does that natively.

Internally, the browser and a Web App share the same code base. The difference is that the Web App is isolated, so it does not share information with the browser. Also the user can only open URLs that match a filter that is previously defined. However, this type of applications has a limited access to device resources. If you need to access these resources you must write a native application.

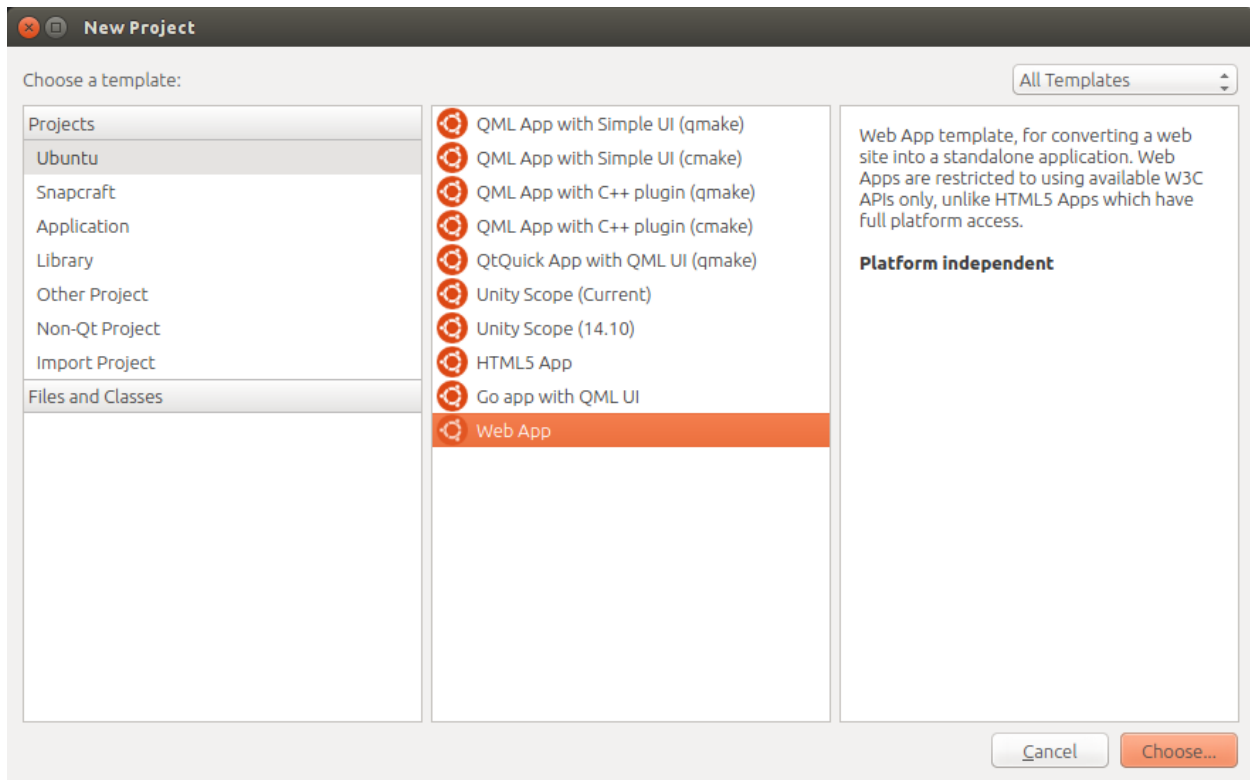
From the programming point of view, a Web App is the easiest type of application that can be found: just define the URLs that you are interested in. It's really simple to make and the SDK does all the work for you. To distribute a Web App it is necessary to package it as a click package that can be uploaded to the Ubuntu Store. Once uploaded to the Ubuntu Store, a Web App is automatically added to [uApp Explorer](#).

3.2 Creation of the project

To create a new project click on the button New Project. Another way to do this is from the File menu, New File or Project.



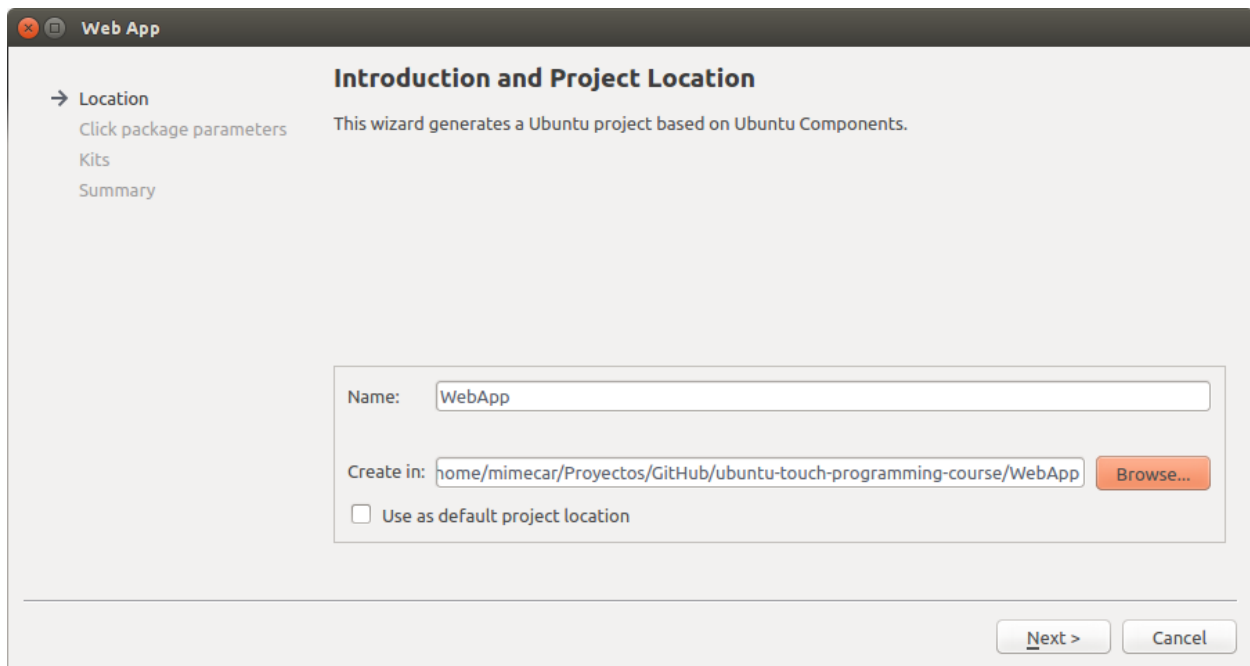
A window will appear with the project types that can be created. You have to select Web App. As shown in the description, it is a project independent of the platform and with limitations in the access to the resources.



Project

Web App

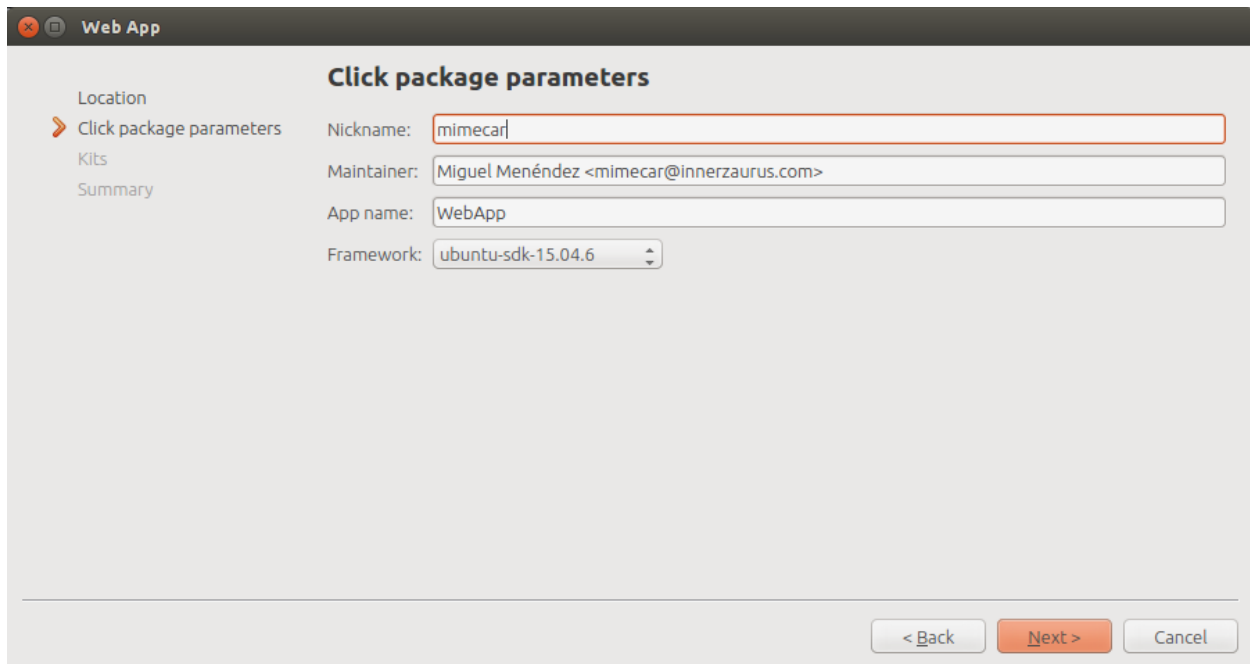
You must select the path in which the project will be saved. Remember that you can not use spaces or characters with accents.



Project

Folder

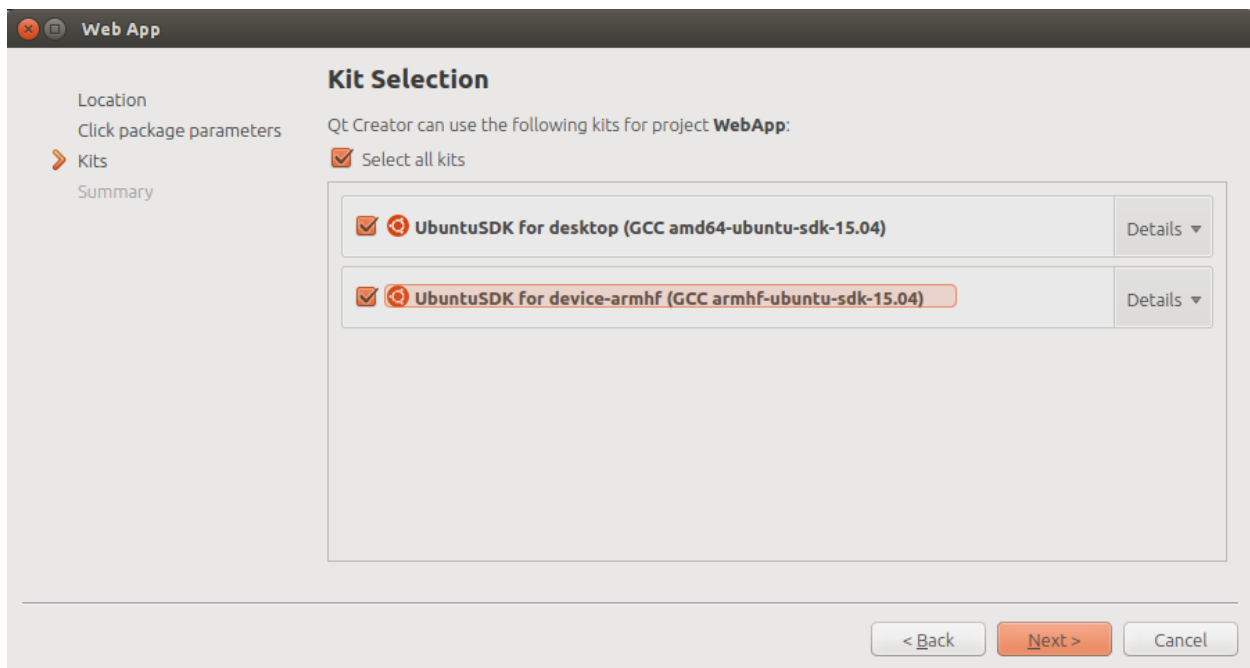
Fill in the application information.



Click

Info

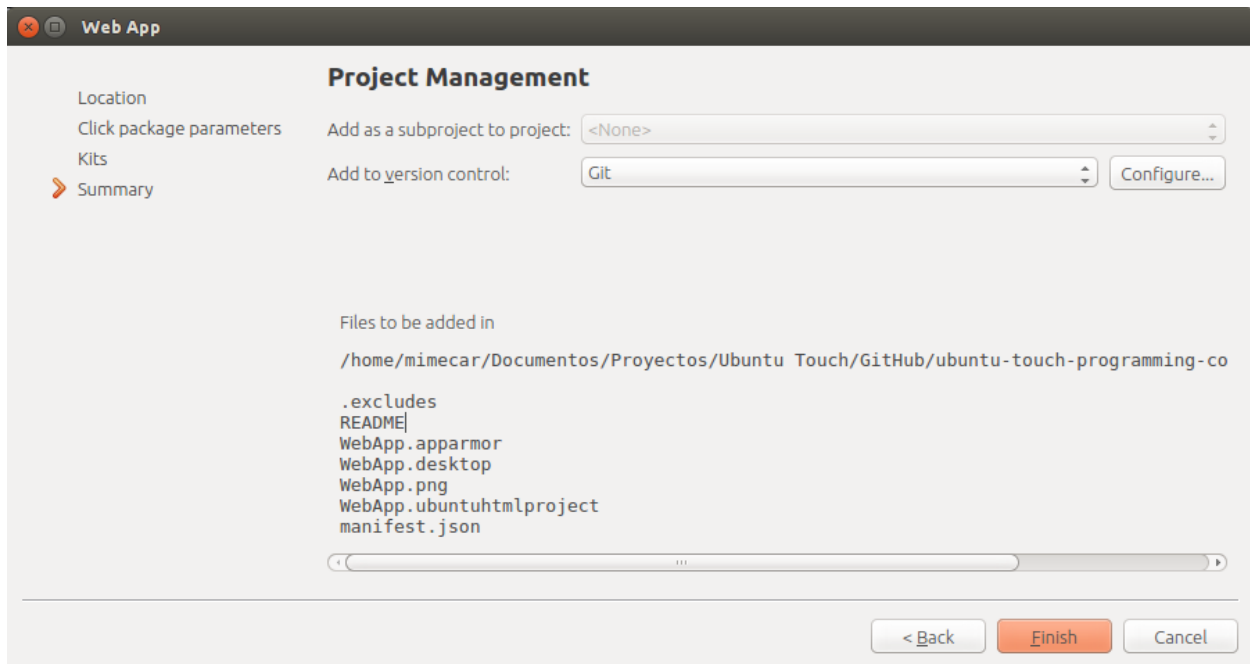
You have to select the Kits that will be used. As it is interesting to generate a computer executable (Desktop) and for the device (Ubuntu), you have to leave both ticked. The Desktop Kit always has to be selected by default otherwise, you will need to restart the ubuntu-sdk-ide configuration.



Project

Kits

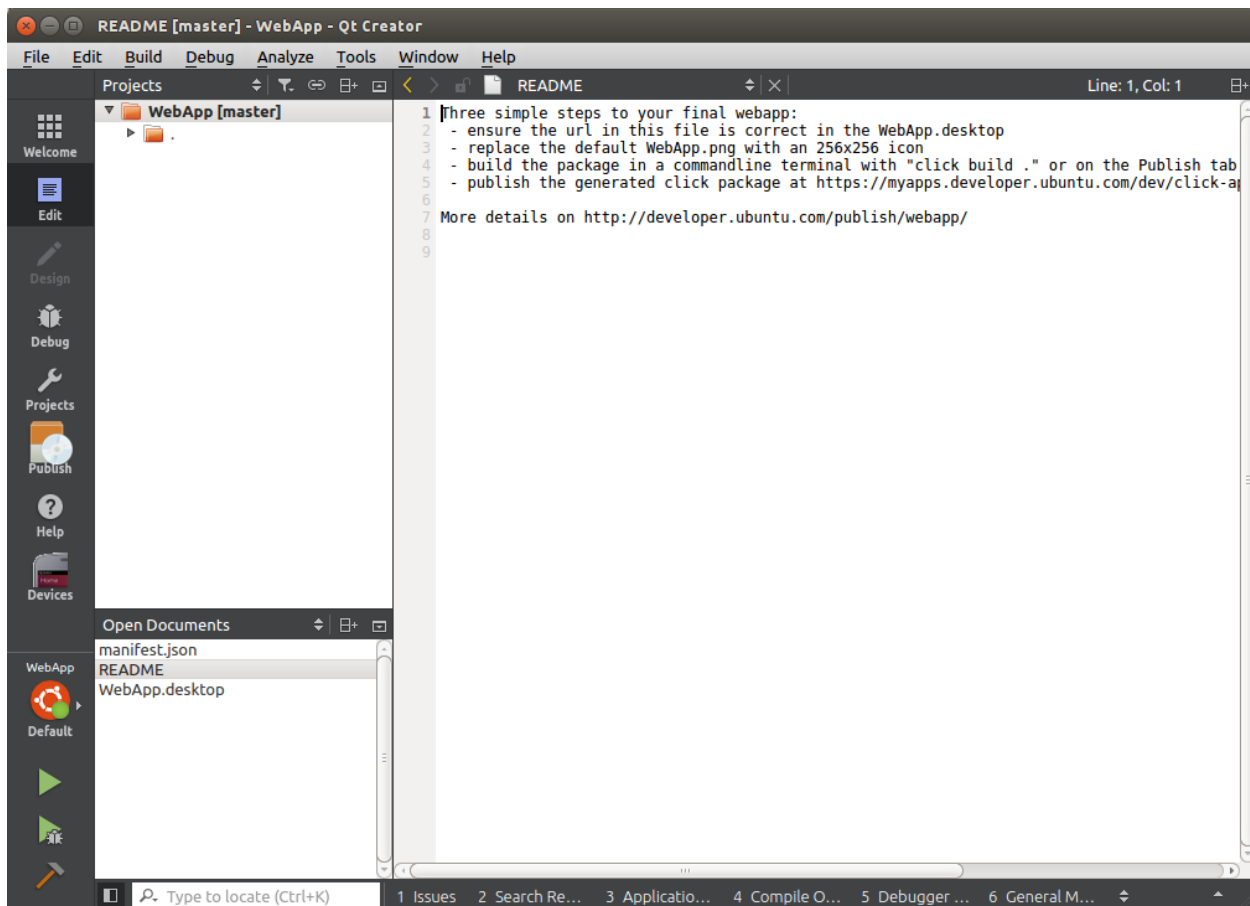
The last step shows a summary. I recommend using version control for the applications you write. It allows you to quickly return to specific points of development easily. This also controls changes between different versions of the code making it easier to fix bugs.



Project

Brief

The wizard is now complete. The project opens automatically.



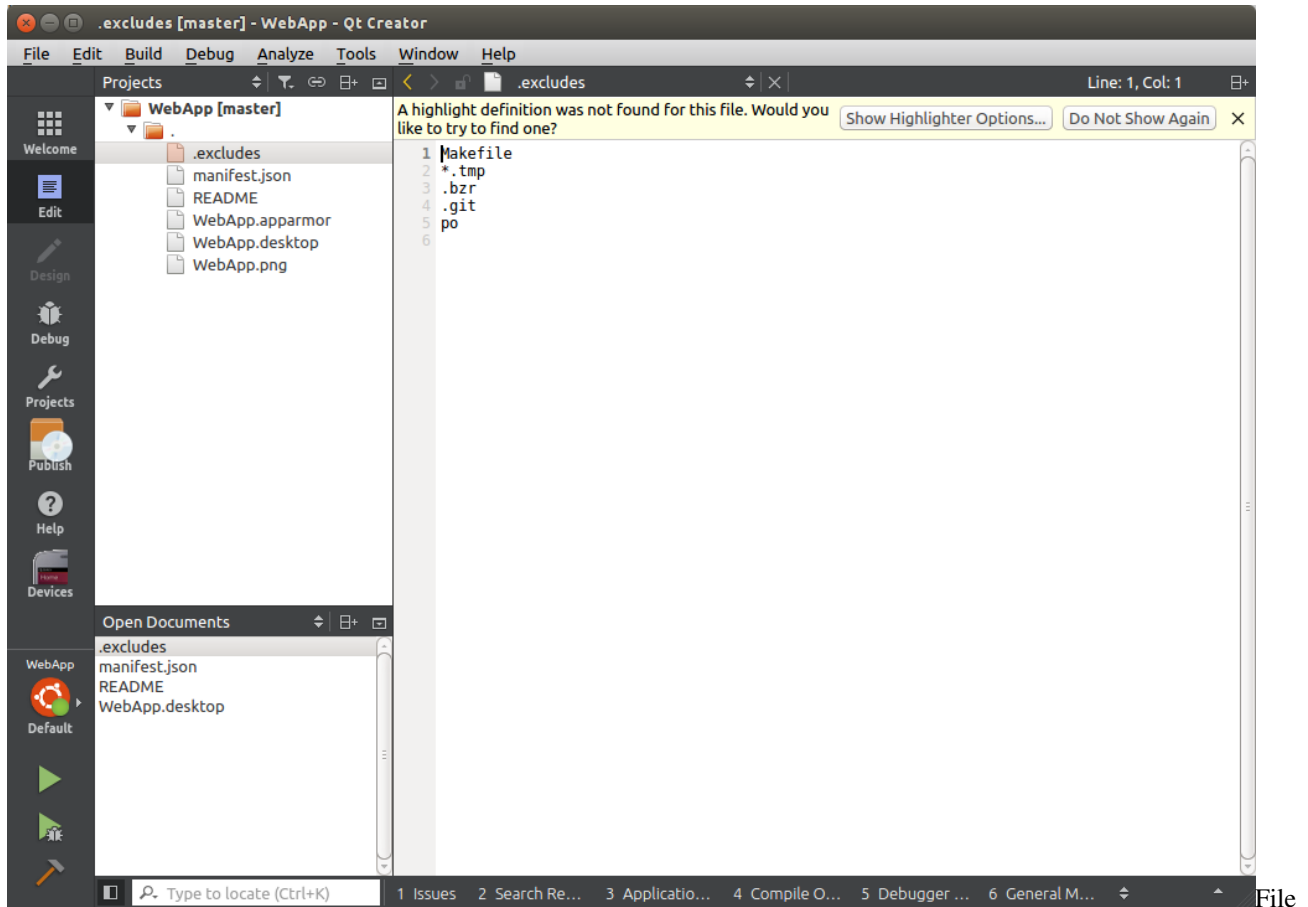
Project

Open

3.3 Structure of the project

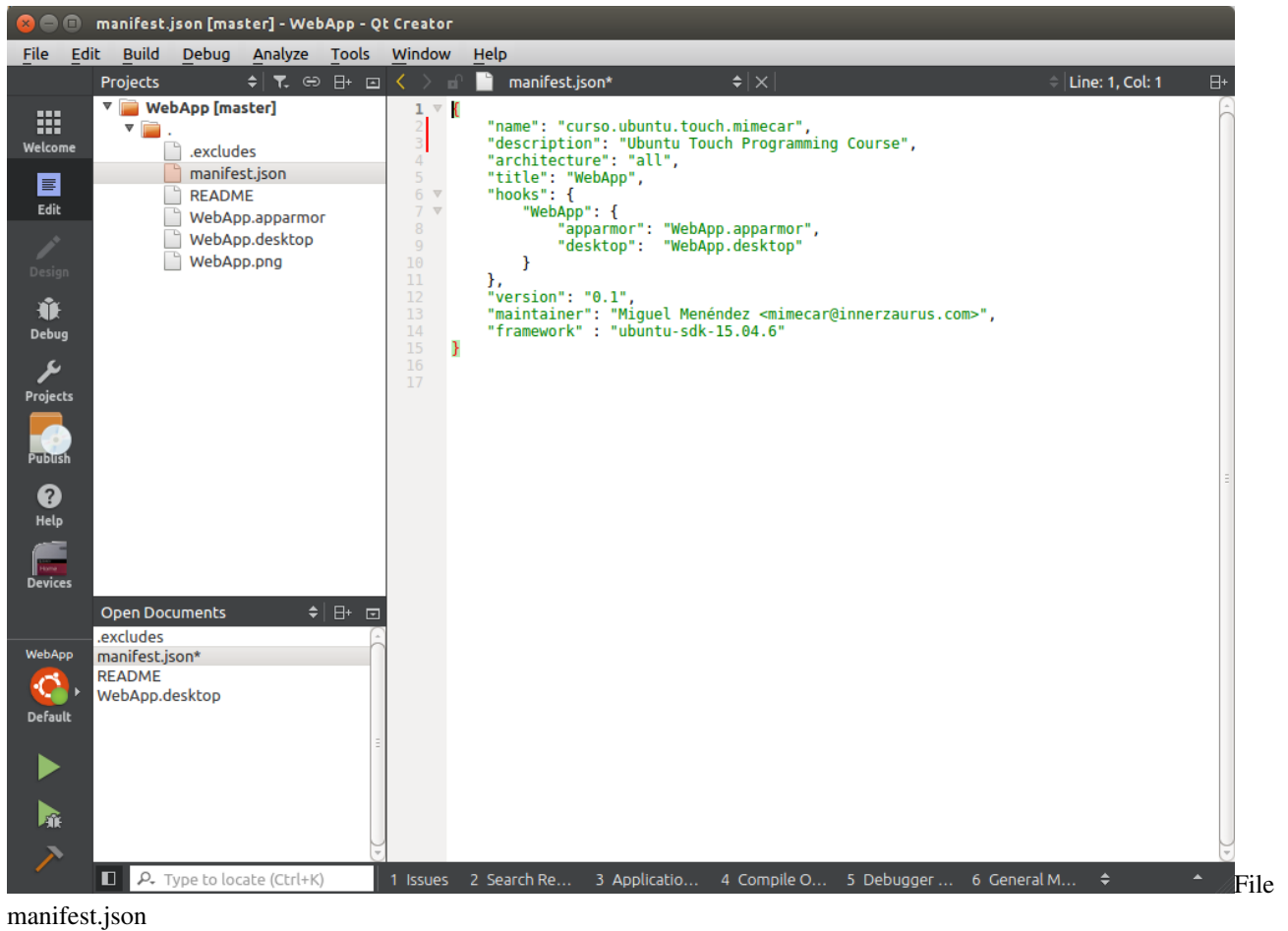
The project of a Web App is very simple and consists of several files.

- `.excludes`: extensions and folders that are out of IDE parse.



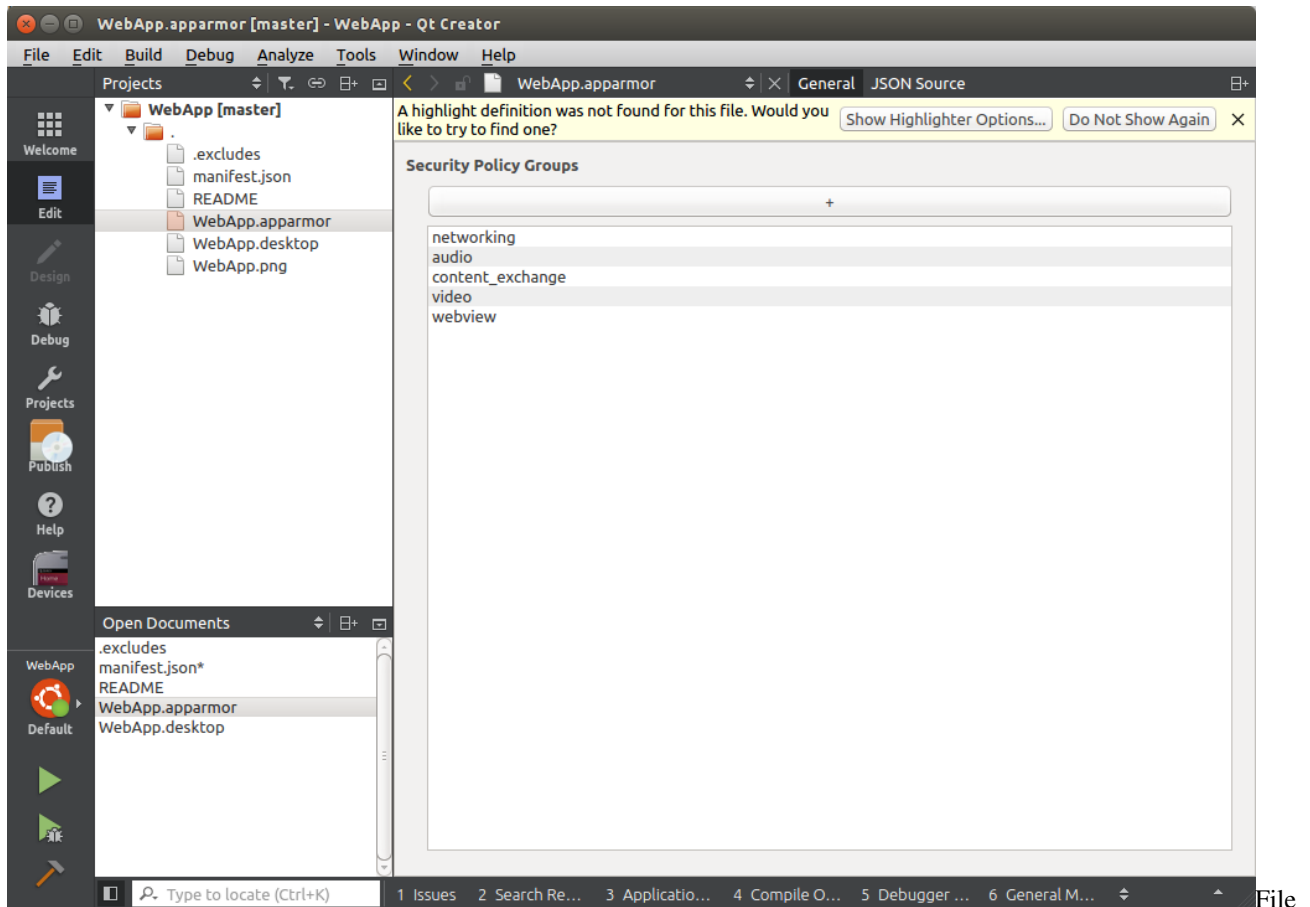
`.excludes`

- `manifest.json`: contains the application information that will be displayed to the user. This file is processed automatically when you upload the application to the Ubuntu Store. All applications have an associated name that is added to all packages. In the example is the text “mimecar”.



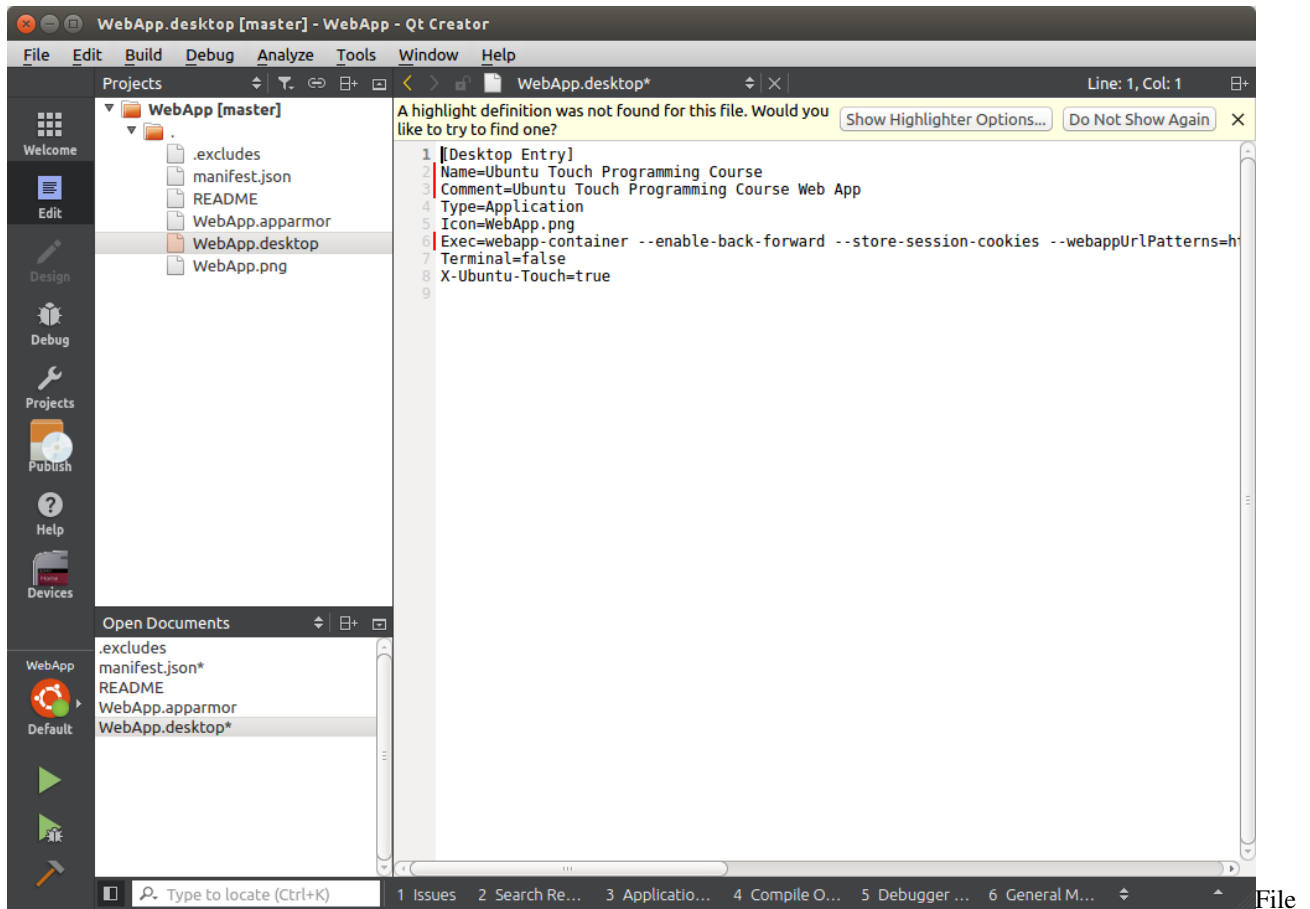
manifest.json

- WebApp.apparmor: application permissions.



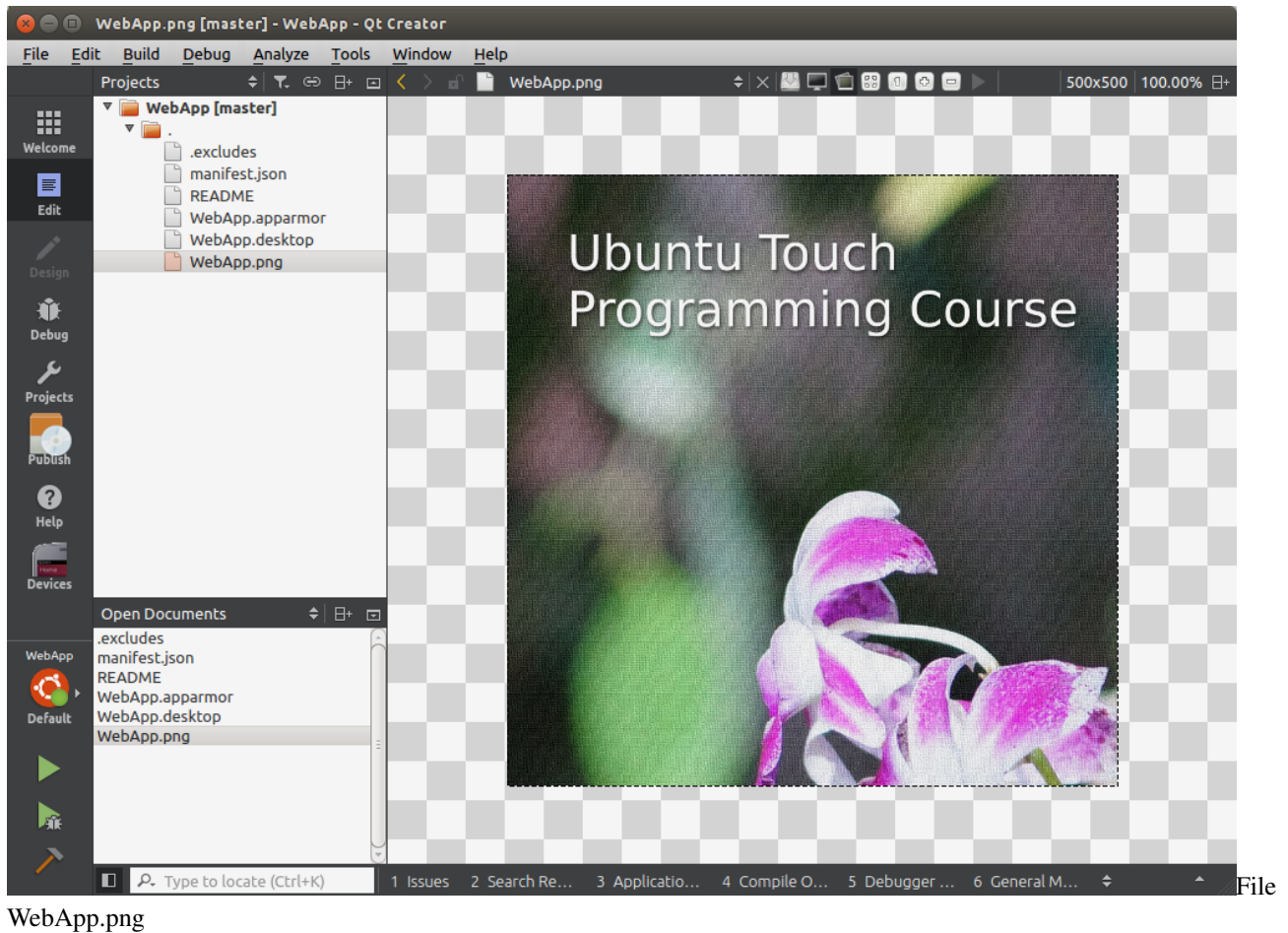
WebApp.apparmor

- *WebApp.desktop*: information for the App Scope to launch the Web App.



WebApp.desktop

- *WebApp.png*: application icon.



3.4 Modifications

There are three modifications to the template provided by the Ubuntu Touch SDK. The first one is to change the URL that is opened by the WebApp.desktop file. You have to change the URL that is displayed in the Exec line. URLs that are within the same domain will appear in the Web App. If you click on a link that goes to another domain, it will be opened in the web browser.

```
Exec=webapp-container --enable-back-forward --store-session-cookies
--webappUrlPatterns=https?://m.WebApp.com/* http://m.WebApp.com %u
```

Must be replaced by:

```
Exec=webapp-container --enable-back-forward --store-session-cookies
--webappUrlPatterns=https?://mimecar.gitbooks.io/ubuntu-touch-programming-course/
content/* https://mimecar.gitbooks.io/ubuntu-touch-programming-course/content/
%u
```

The next step is to replace the icon file (WebApp.png) with a PNG image. This image will be the app icon displayed in the scope. The resolution must be 256x256 or higher.

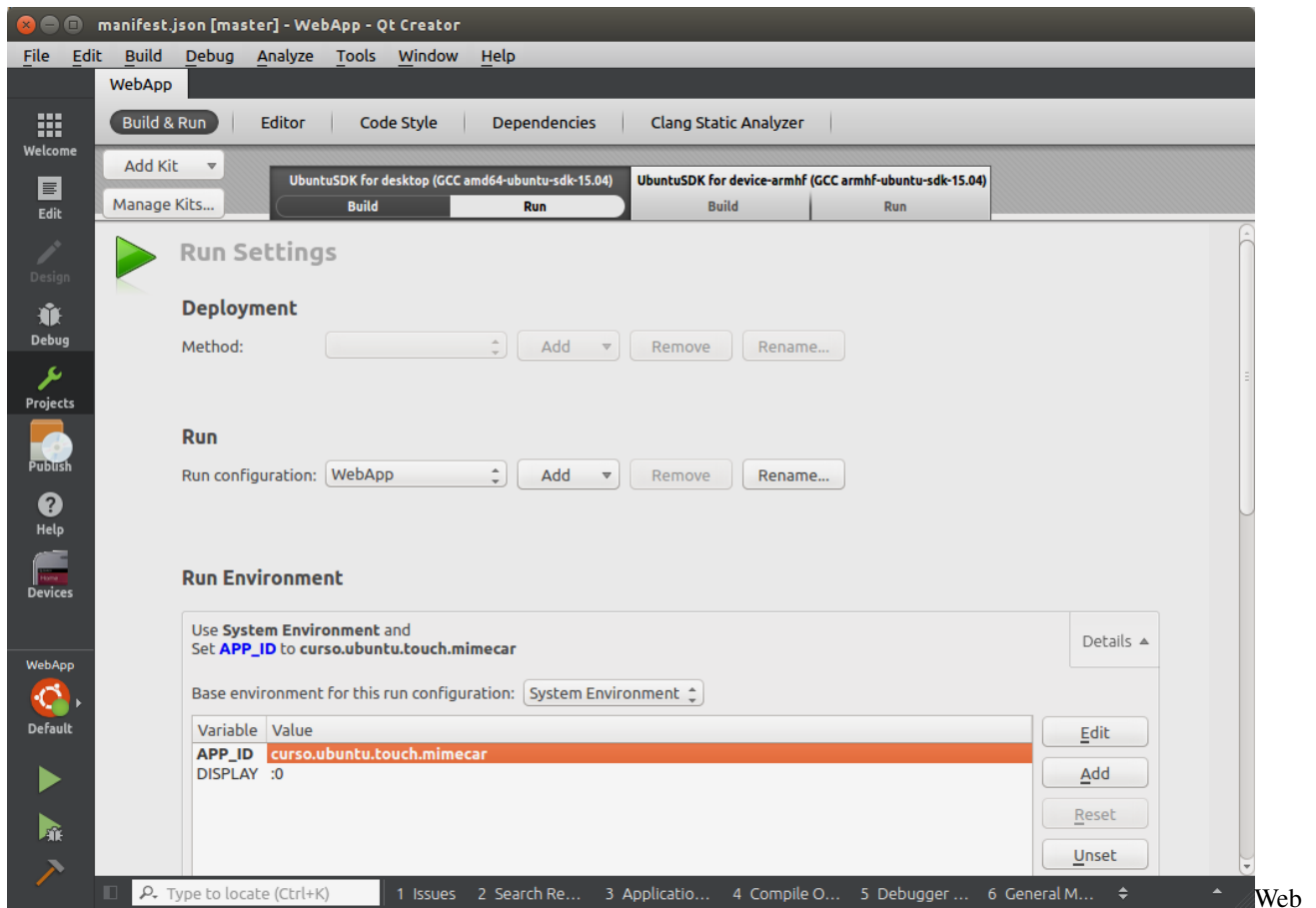
Finally, open the manifest.json file and complete the application information.

3.5 Web App Testing

Press the Play button to launch the application on the desktop. The Desktop kit is selected by default. Note that the SDK template has a bug that does not allow launching the application.

To solve this follow these steps:

- Project button (left sidebar).
- Under Kit, select 'Run'.
- Click on the combo that appears next to 'Run Environment'.
- Click on the Add button and put the data shown in the screen capture.



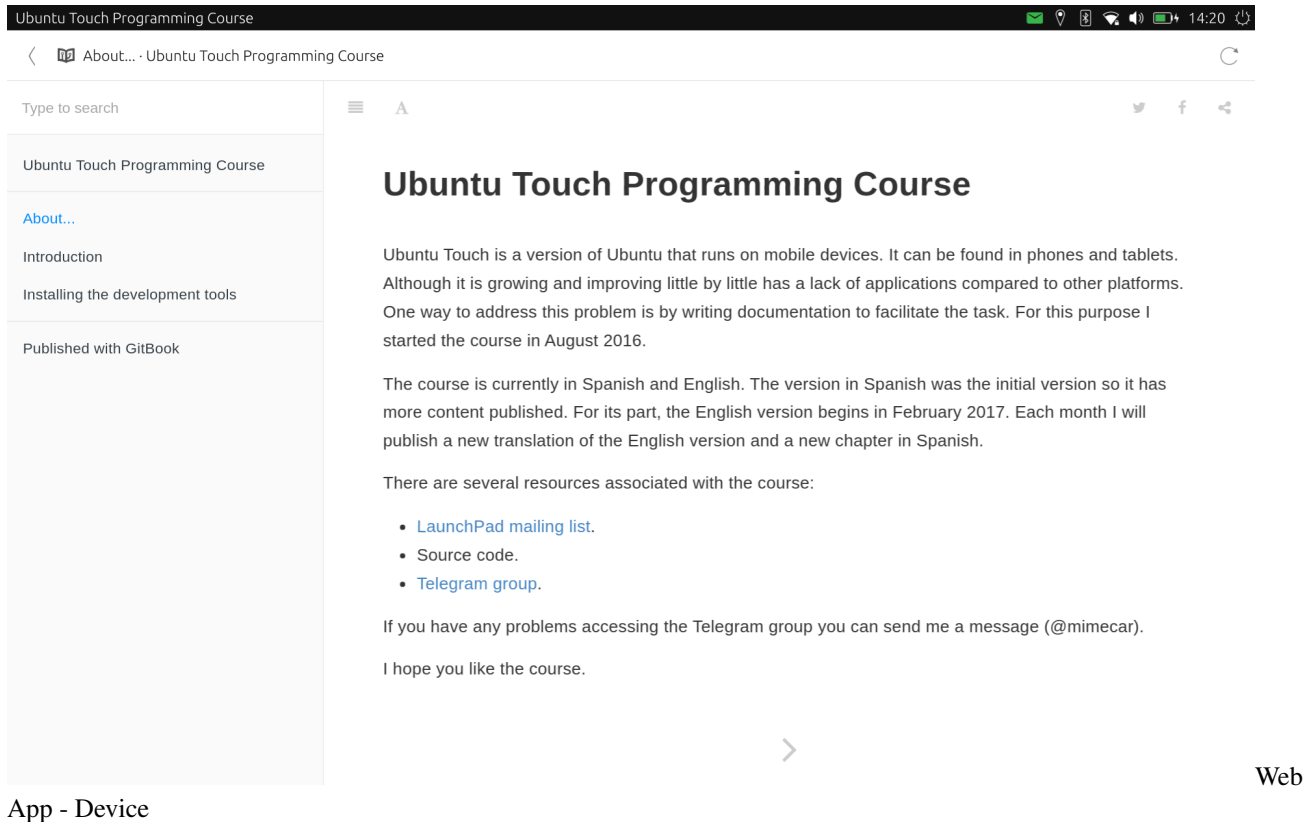
App - Desktop

The value of APP_ID must match the 'name' field in the file 'manifest.json'



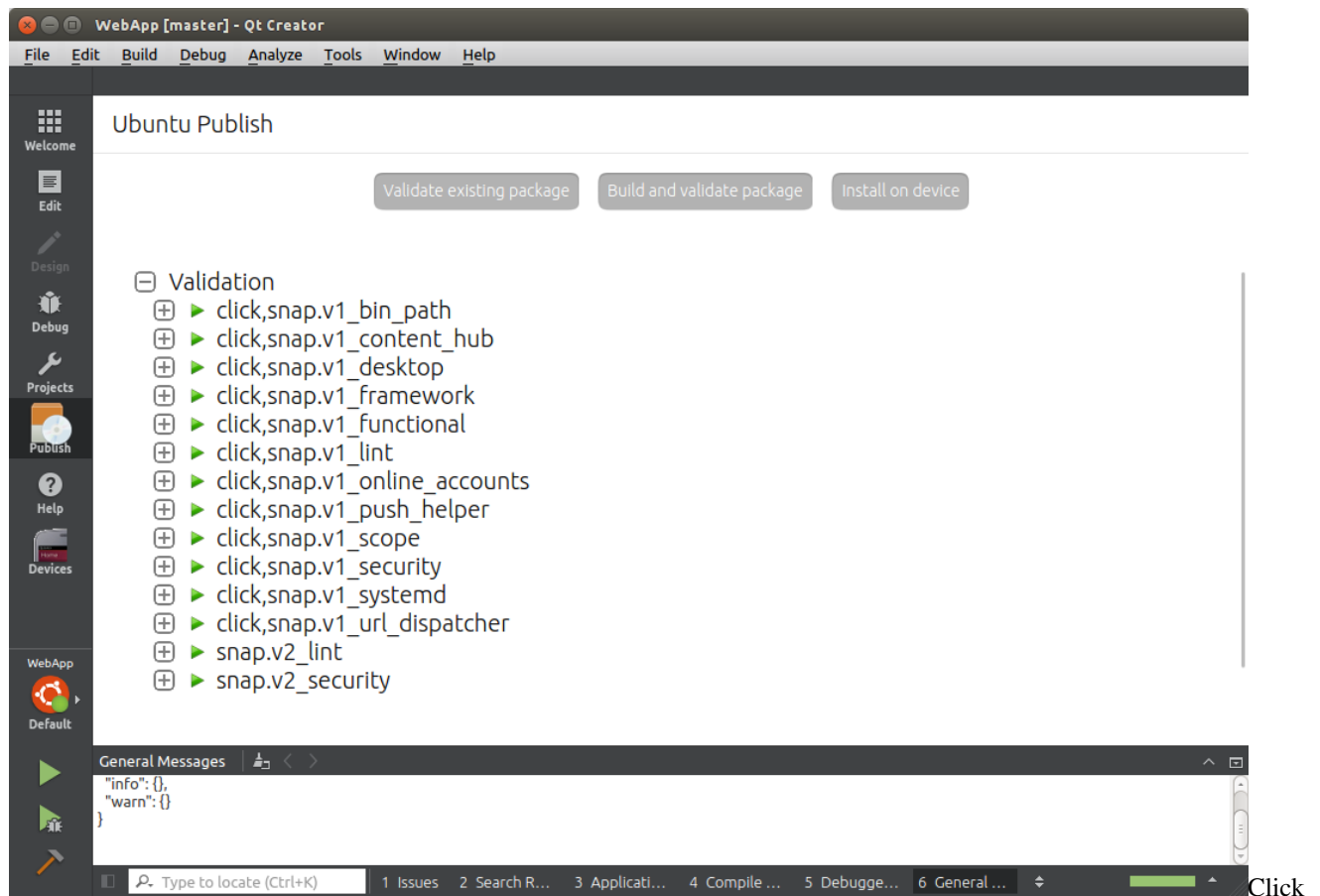
App - Desktop

If you select Ubuntu Target Kit, you can see the result on the device. It is important that the circle next to the Ubuntu Kit is green. Otherwise there will be no connection to the device.



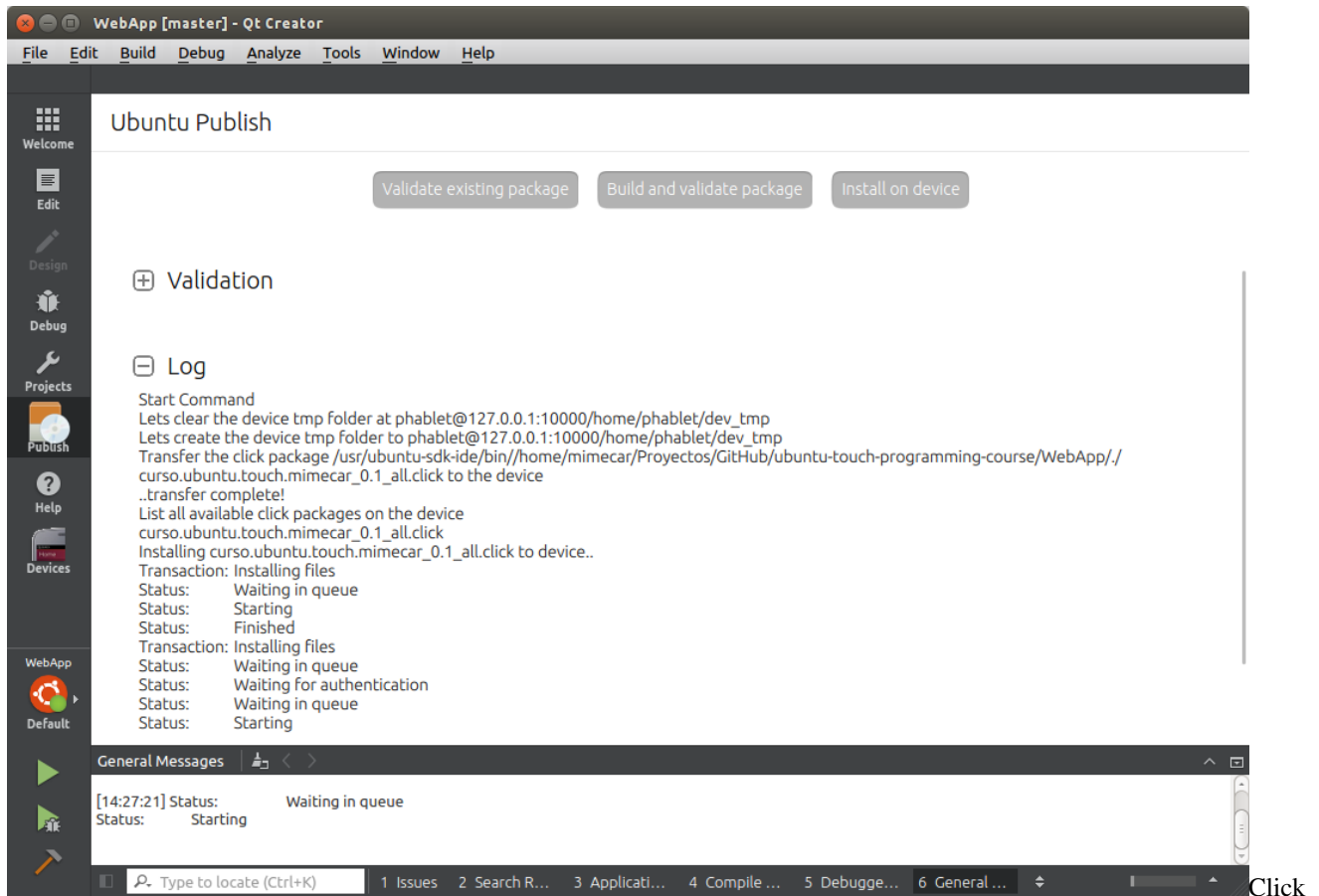
3.6 Creating the click package

The Web App works on the device but only while it is connected to the computer. To remain permanently on the device, it is necessary to install it on the system using a click package. Click on the Build and validate click package button.



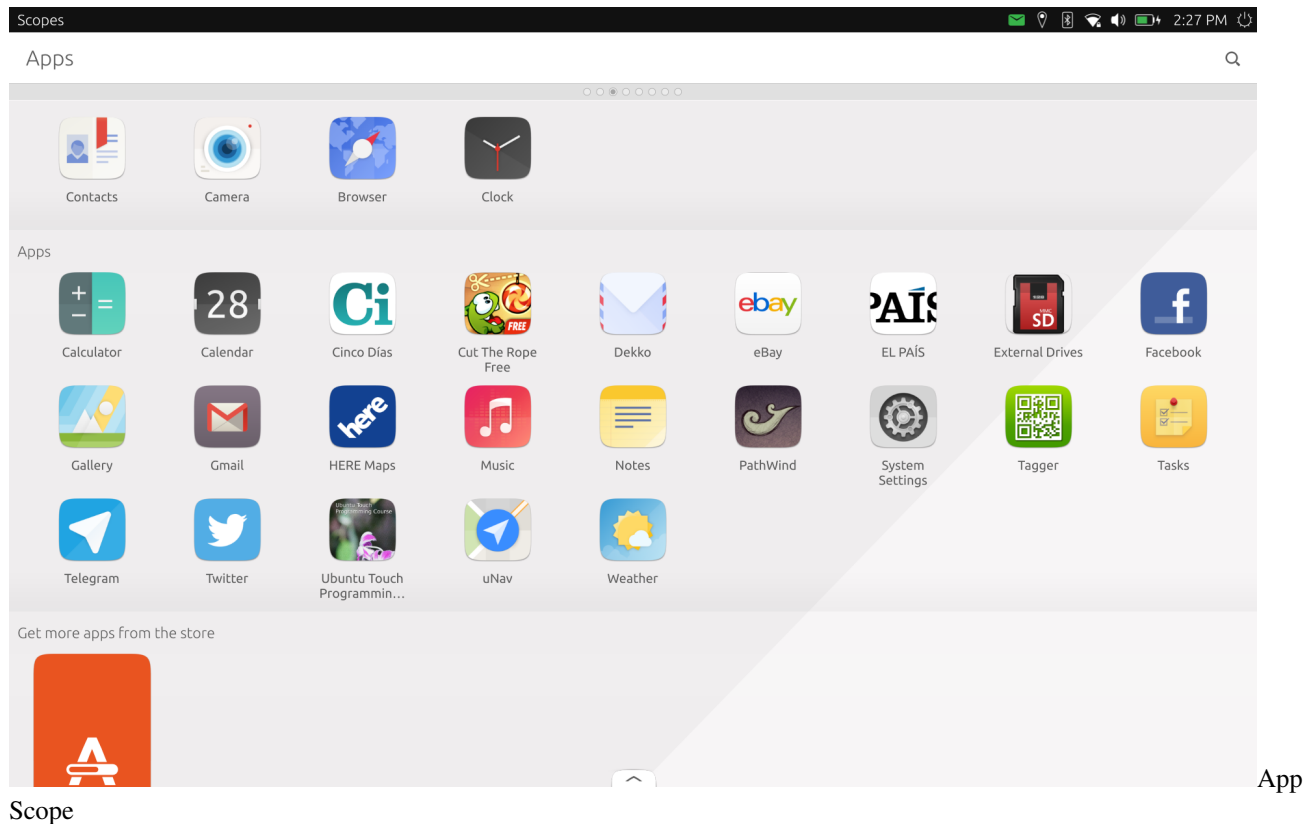
Build

If there is no problems, click the Install on device button. It is possible that it gives a connection problem. If so, go to Devices, select the device and in the section Control click on the button Open SSH connection to the device.



Install

The result is the application installed on the device.



3.7 Conclusions

A Web App is very simple to program. The Ubuntu Touch SDK does most of the work. The Web App disappears when the device is disconnected. To install it, you must create a Click package. If you want to share the application with other users, you have to publish the Click package in the Ubuntu store.

3.8 References

[Web App tutorial](#)

3.9 Source Code

- [Source code on GitHub.](#)

3.10 People who have collaborated

- Larrea Mikel: revision of the chapter in Spanish.
- Cesar Herrera: revision of the English translation.
- Joan CiberSheep: revision of the English translation.

4.1 QML language

QML is a language that is based on JavaScript and is used to create the user interface of an application. It allows you to use both traditional components (buttons, lists, etc.) and graphic elements to which logic is added. An example of the first case is the Ubuntu Touch user interface. For the second case there are several examples on the [QT Web](#). QML handles the visualization but not the logic that implements the application.

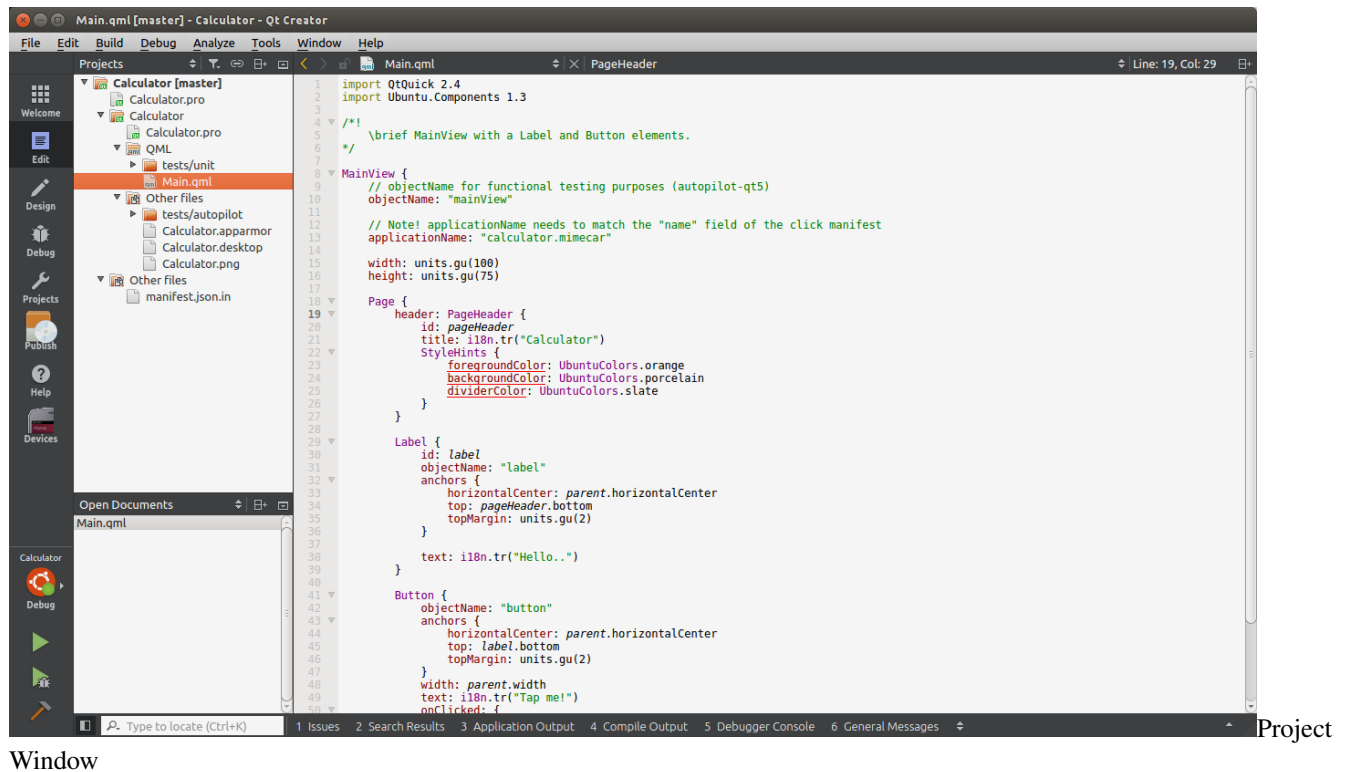
This logic can be written in several languages, depending on the needs we have. If the application is a game and needs computing power, the language chosen would be C/C++. On the other hand, if the performance of the app is not critical, you can use JavaScript. At the moment the applications of the course will use JavaScript and later will be written in C/C++.

The configuration of the project must be the following:

- Project type: QML App with Simple UI (qmake).
- Kits: select all installed kits.

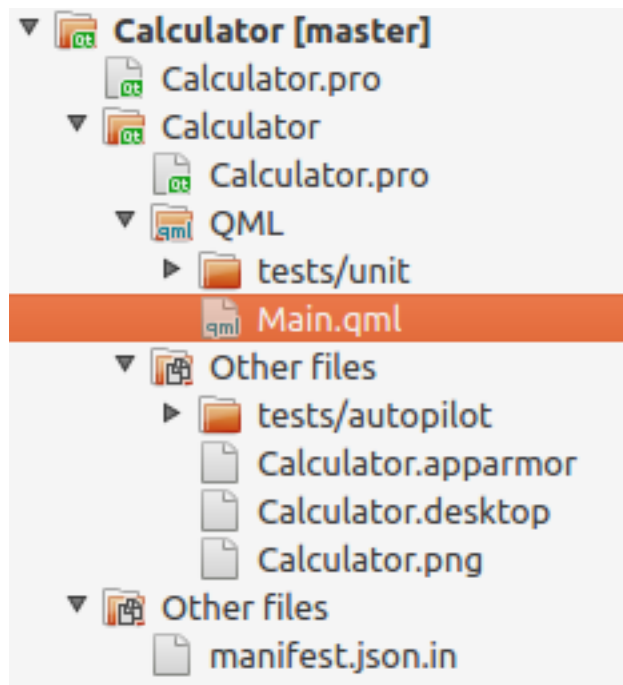
4.2 Project structure

The project will be created with the chosen template and will open by default, the user interface (QML) file.



Window

The structure of the project varies slightly comparing it to the one that had the Web App.



Project Structure

Files that can be modified are displayed in bold type. The remaining files are for internal use of the IDE and should not be modified.

- Calculator.pro: main project file.
- Calculator

- Calculator.pro: information for the compiler.
- QML
 - * Tests / unit: The files in this folder are used to make validations of the application.
 - * **Main.qml**: file containing the user interface of a screen.
- Other files
 - * Tests / autopilot: The files in this folder are used to run the application.
 - * **Calculator.apparmor**: application permissions
 - * **Calculator.desktop**: application information for the application launcher.
 - * **Calculator.png**: application icon.
- Other files
 - * **Manifest.json.in**: information for the Ubuntu store with application data.

The IDE has a tab that with the text ‘Design’. This tab analyzes the generated QML code and converts it into the user interface. The tool does not work well and gives errors even if the QML code is valid.

You can see this on the project screenshot on the words that are marked in red on lines 23, 24 and 25. For this reason, user interface tests will be performed by running the application on the computer.

4.3 File structure

All QML files have a common structure that you will see below. QML allows you to use a number of components that are already defined. This information is passed to the project with the import statement. It is possible to use different versions of the components although it would be normal to choose the latest available version.

```
import QtQuick 2.4
import Ubuntu.Components 1.3
```

4.4 Comments

Comments are guides that the programmer writes about specific parts of the code. I recommend writing comments on code parts that are complex. It is best to write them when the code is still fresh. If you wait a while you may not remember all the details.

Comments can be write in two ways:

- If the comment has several lines you can use

```
/*
 * Description
 */
```

- Whereas if you have only one line

```
// Description
```

- In the sample code it looks like this:

```
/*!  
  \brief MainView with a Label and Button elements.  
*/
```

4.5 MainView

MainView is the root element of the user interface. It automatically adapts to the rotation of the device.

```
MainView {
```

This block of code doesn't need to be modified. It is used internally.

```
// objectName for functional testing purposes (autopilot-qt5)  
objectName: "mainView"  
  
// Note! applicationName needs to match the "name" field of the click manifest  
applicationName: "calculadora.innerzaurus"
```

4.6 Grid Unit

The next two statements are responsible for defining the initial size of the screen. An important detail is that the dimensions are not defined in pixels but in some units called gridUnits (gu). The reason to invent new units and not using pixels that already exists is that gridUnits is universal to the different screen sizes and translates. An example will make this clear.

Suppose that we have a device with a resolution of 600 x 800 pixels. If a rectangle is defined with the dimensions 300x800 pixels, will occupy the middle of the screen. Now if you use a device that has a screen resolution of 1080x1920 pixels, the rectangle will not reach the middle of the screen. To avoid this problem, define the dimensions in gu (gridUnit). If the dimensions are defined in gu, internally, the system will calculate those dimensions with the characteristics of the screen. The final result will be that the rectangle always has the same size regardless of the screen resolution.

```
width: units.gu(100)  
height: units.gu(75)
```

4.7 Page Element

The Page element defines a view. It is recommended that it be included within a MainView element (our specific case) or an AdaptivePageLayout element. It contains a header with an identifier, title and style that applies to the elements of the header. The style is defined within the StyleHints element.

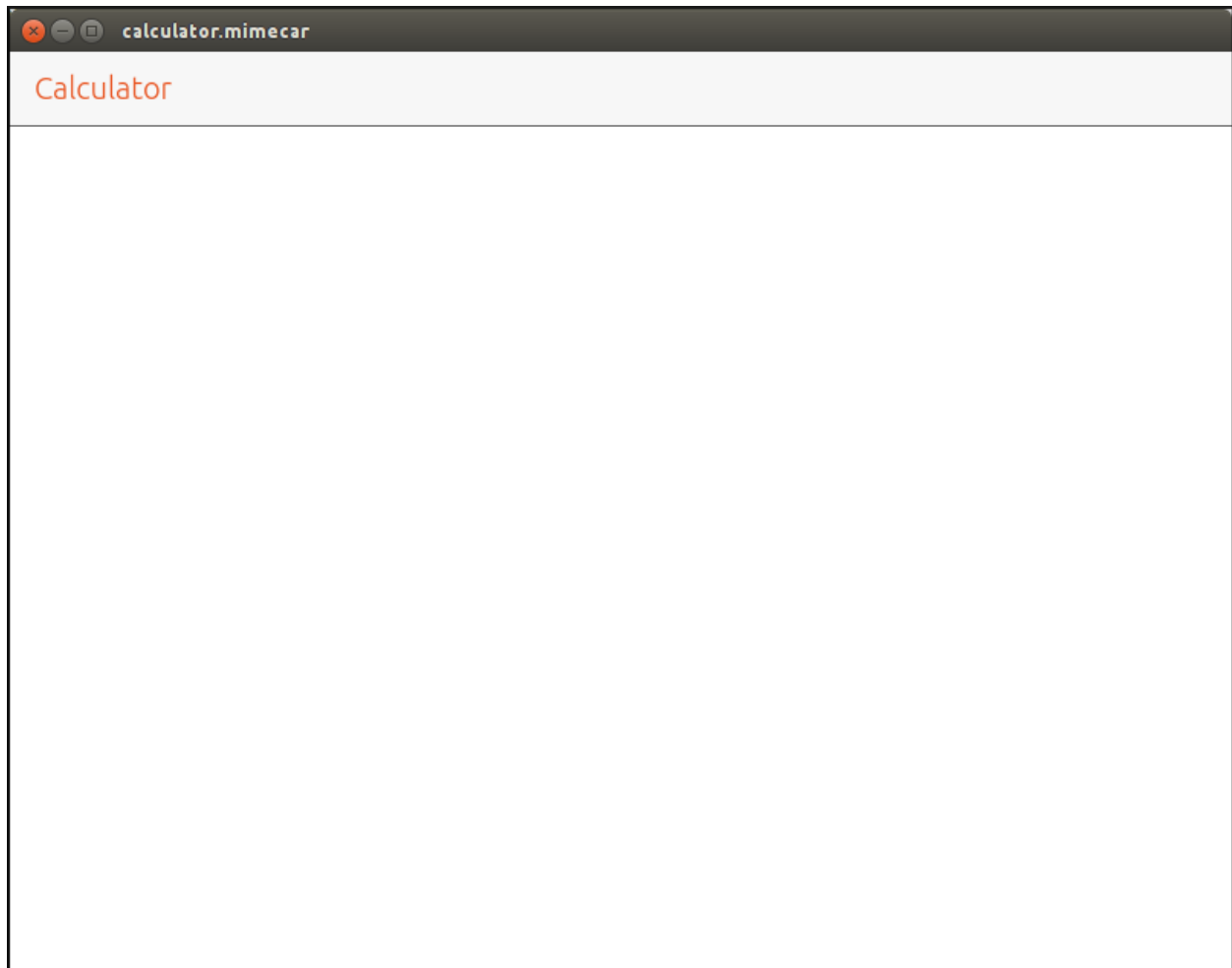
```
Page {  
    header: PageHeader {  
        id: pageHeader  
  
        title: i18n.tr("Calculator")  
  
        StyleHints {  
            foregroundColor: UbuntuColors.orange
```

(continues on next page)

(continued from previous page)

```
        backgroundColor: UbuntuColors.porcelain
        dividerColor: UbuntuColors.slate
    }
}
```

To see the user interface that is created it is necessary to click on the Play button. Although you can also try on a physical device I recommend that you work with the desktop. The test process is faster and at first you will use it a lot.



In the default QML file there are several components. To make it easier to learn it will be created from scratch with a more detailed explanation. You have to delete the lines from 28 to 53.

Below you will see an initial design of the calculator with labels. It will help to introduce the way components works and will get a minimum base to continue with QML.

4.8 Label

A label is a component that displays text. Its minimum structure is:

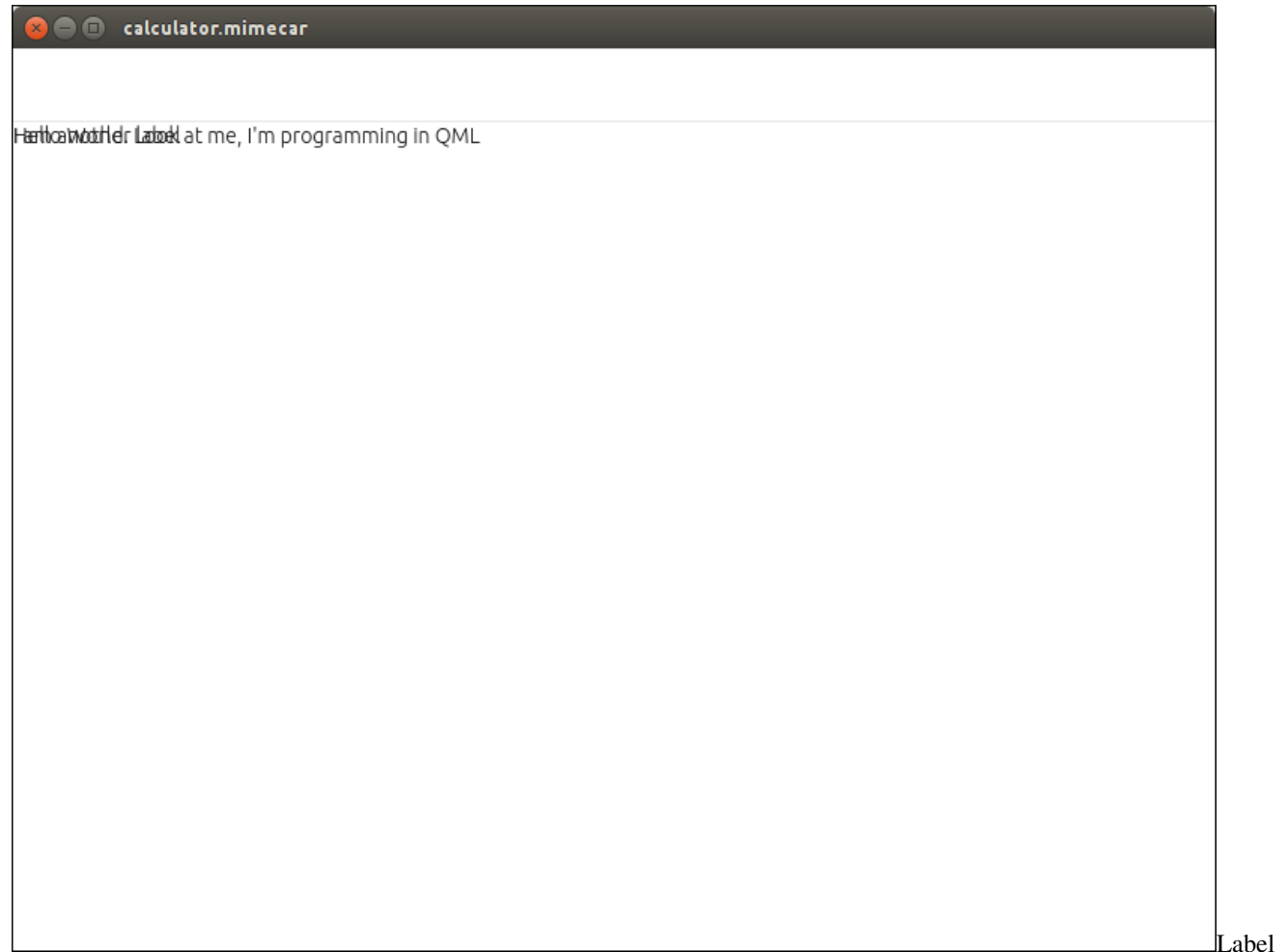
```
Label {
    text: "Hello World"
}
```

Add the following code from line 28. Then run the application on the desktop.

```
Label {
    text: "Hello World. Look at me, I'm programming in QML"
}

Label {
    text: "I am another label"
}
```

Now comment the lines in the header and run the application.



Overlay

The labels are shown, but the two are overlaid. In one hand there is the definition of the component and on the other hand, its organization. If the header is displayed, the tags do not show and if the header is removed the tags overlap.

4.9 Component Organization

Components can be organized in rows, in columns or in a grid—that combines rows and columns. To distribute labels in a column they have to be inside a Column element.

```

Column {

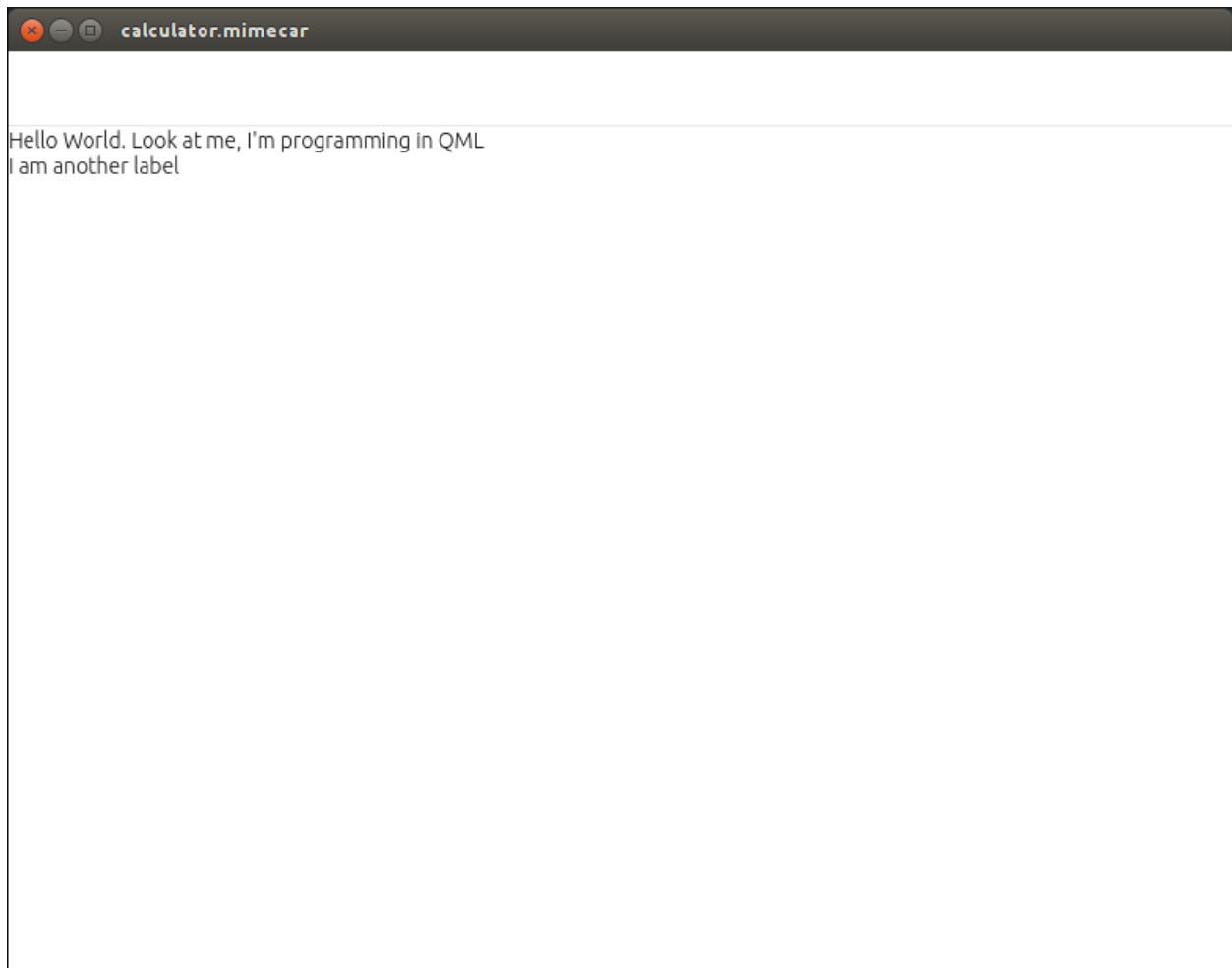
    Label {
        text: "Hello World. Look at me, I'm programming in QML"
    }

    Label {
        text: "I am another label"
    }

}

```

With this modifications we can see the improvement.



Column

of Labels

When you restore the header code, the labels disappear again. It's not that qml doesn't like us. The labels are actually hidden by the header, so it is necessary to define a relation between the header and the column

```

Column {
    anchors.top: pageHeader.bottom

    Label {
        text: "Hello World. Look at me, I'm programming in QML"
    }
}

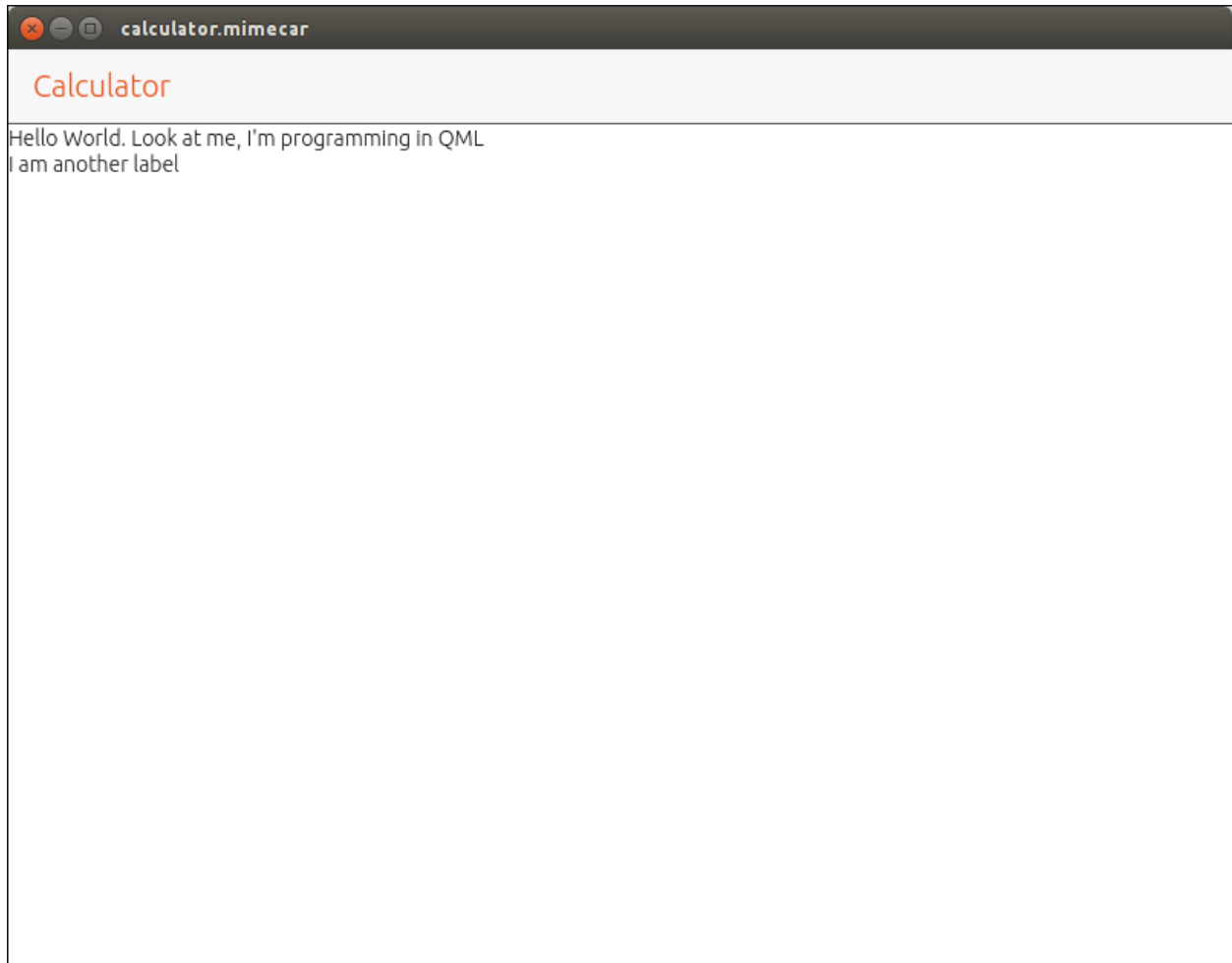
```

(continues on next page)

(continued from previous page)

```
}  
  
Label {  
    text: "I am another label"  
}  
  
}
```

The result is the expected now.



Chapter

End

4.10 Exercises

In the previous chapters you finished the exercises too quickly. In this chapter this will not be the case. The exercises are:

- **Exercise 01.** Create three columns one after the other and show the numeric keypad numbers (1, 4, 7, 2, 5, 8, 3, 6, 9; 0).
- **Exercise 02.** Distribute keypad numbers as a table by combining columns and rows. The element to create the rows is Row and is used similarly to the element Column. You may use the documentation included in the SDK

but you may not search the solution on the Internet. To view the SDK documentation, press the F1 key with a QML component selected.

4.11 Source Code

- [Source code on GitHub](#).

4.12 People who have collaborated

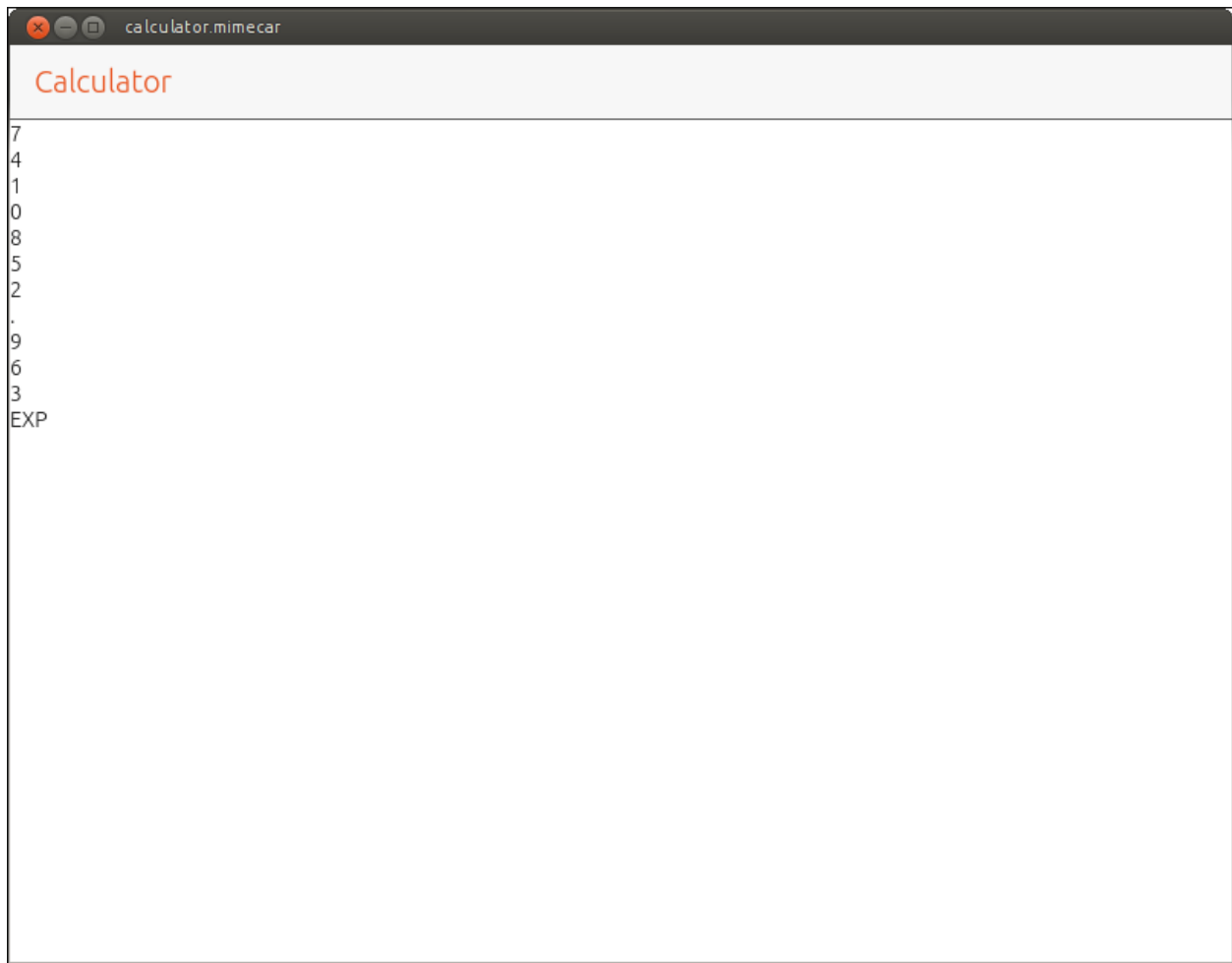
- Larrea Mikel: revision of the chapter in Spanish.
- Cesar Herrera: revision of the English translation.
- Joan CiberSheep: revision of the English translation.

5.1 Organizing components

In exercise 1 you had to distribute the calculator numbers in columns. To make the design of the calculator clearer, let's add the distribution of the numbers in a table. I have added a couple of new texts to simplify the later development.

Column 1	Column 2	Column 3	—	—	—	7	8	9	4	5	6	1	2	3	0	.	EXP
----------	----------	----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----

This table has three columns. If the numbers are grouped into columns (as requested in the exercise) the result would be the one shown in the next image.



Exercise

1

Source code:

```
// First column
Column {

    anchors.top: pageHeader.bottom
    id: column1

    Label {
        text: "7"
    }

    Label {
        text: "4"
    }

    Label {
        text: "1"
    }

    Label {
        text: "0"
    }

}
```

(continues on next page)

(continued from previous page)

```
}

// Second column
Column {

    anchors.top: column1.bottom

    id: column2

    Label {
        text: "8"
    }

    Label {
        text: "5"
    }

    Label {
        text: "2"
    }

    Label {
        text: "."
    }
}

// Third column
Column {

    anchors.top: column2.bottom
    id: column3

    Label {
        text: "9"
    }

    Label {
        text: "6"
    }

    Label {
        text: "3"
    }

    Label {
        text: "EXP"
    }
}
```

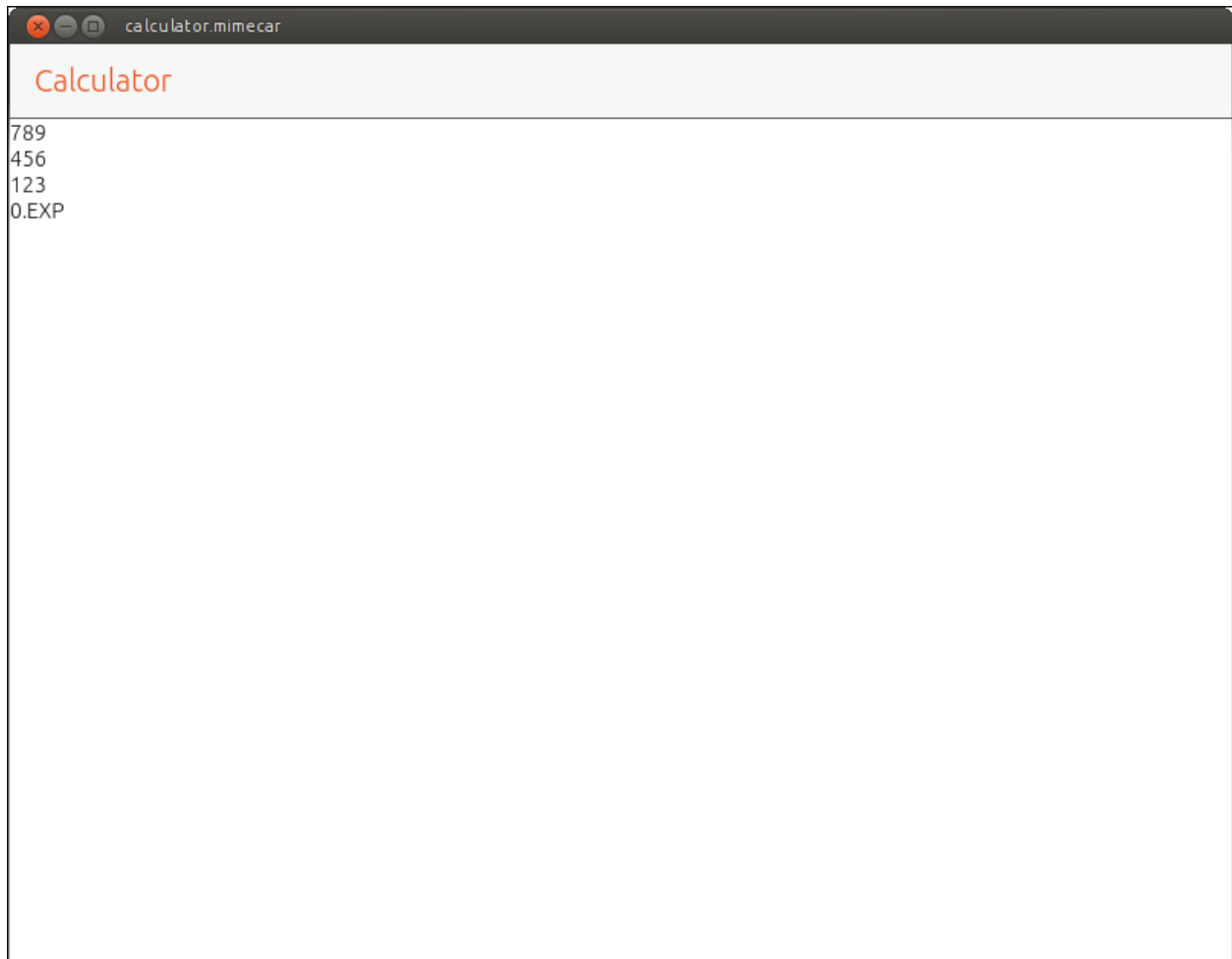
Unless you are programming a design calculator, the screen capture is different from the initial table. The columns are correct but they are distributed one underneath the other, not in parallel as they should. To distribute the numbers in a row you have to use the Row layout.

5.2 The Row Layout

The Row layout works just like a Column element with a difference: the components are distributed in a row. The structure for this layout is as follows:

```
Row {  
    // Component  
}
```

If the table is defined using only rows, all numbers would be distributed in a single row. To solve this problem we need to separate the table into rows and columns. The rows will contain the numbers horizontally, in four rows in total. The next step is to distribute the four rows in a column.



Exercise

2

The source code is similar to the previous case. You just have to change the way in which the components are organized.

```
// First column  
Column {  
  
    anchors.top: pageHeader.bottom  
    id: column1
```

(continues on next page)

(continued from previous page)

```
// First row
Row {

    Label {
        text: "7"
    }

    Label {
        text: "8"
    }

    Label {
        text: "9"
    }
}

// Second row
Row {

    Label {
        text: "4"
    }

    Label {
        text: "5"
    }

    Label {
        text: "6"
    }
}

// Third row
Row {

    Label {
        text: "1"
    }

    Label {
        text: "2"
    }

    Label {
        text: "3"
    }
}

// Fourth row
Row {

    Label {
        text: "0"
    }

    Label {
```

(continues on next page)

(continued from previous page)

```
        text: "."
    }

    Label {
        text: "EXP"
    }
}
```

In this way the components are distributed as wanted. By separating the definition of rows and columns it is possible to have in one row 3 components and in another row, only one. If all rows in the table have the same number of elements you can use the Grid layout, which simplifies the QML code and gives us more flexibility.

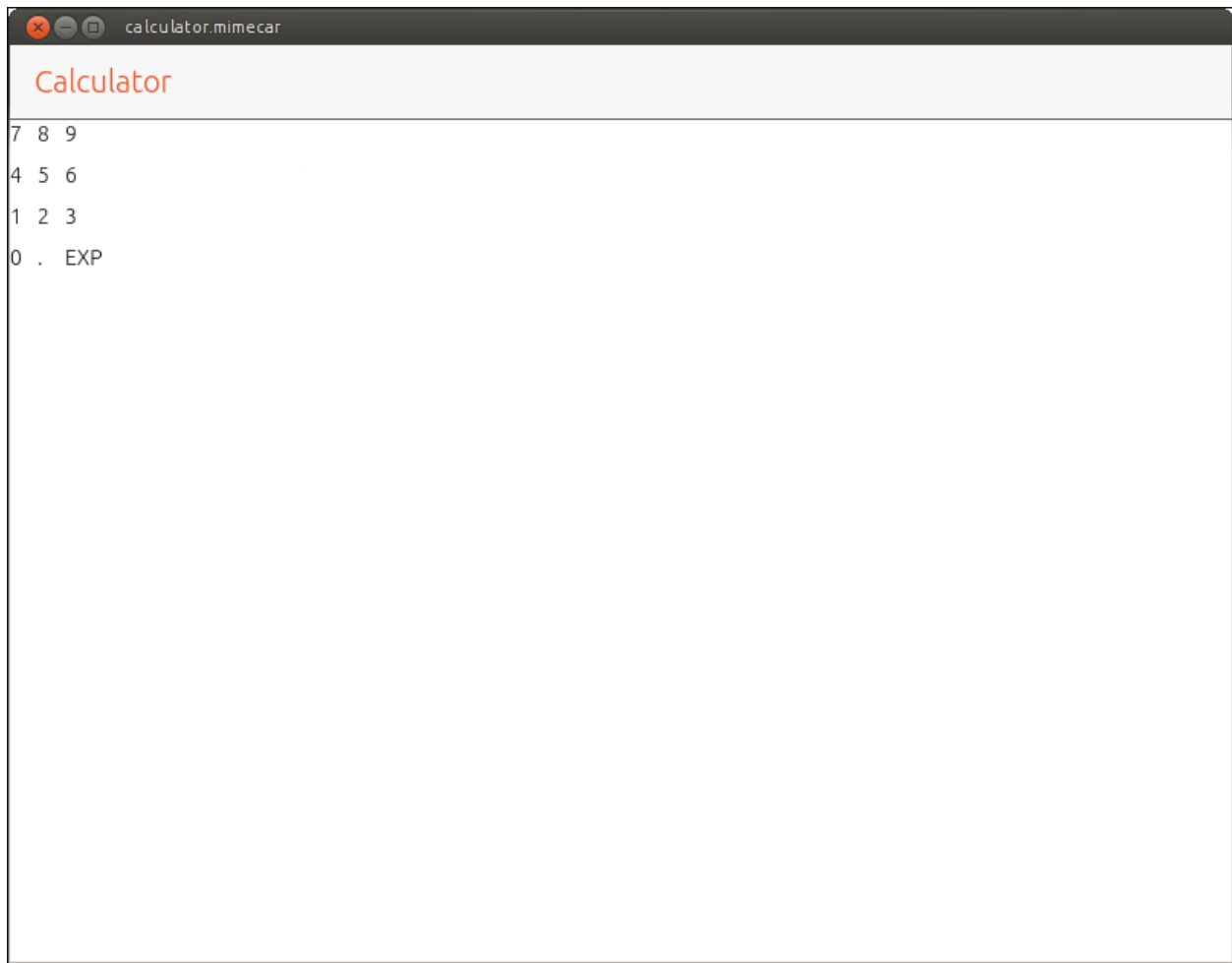
5.3 Grid Layout

This layout can be used if all rows and columns have the same number of components. The structure that follows is as shown:

```
Grid {
    columns: 3

    // Components
}
```

The most important parameter is the number of columns. The layout distributes the components automatically using the number of columns as a limit. If you enter an extra component, it automatically adds a row and adds that component in the first column.



Exercise

3

When defining a layout you can add a series of parameters that modify its behavior. In this case, only the number of columns is defined. As you can see in the code, it's not defined how the components are distributed.

```
Grid {  
  
    anchors.top: pageHeader.bottom  
    spacing: 10  
    columns: 3  
  
    // Row 1  
    Label {  
        text: "7"  
    }  
  
    Label {  
        text: "8"  
    }  
  
    Label {  
        text: "9"  
    }  
  
    // Row 2
```

(continues on next page)

(continued from previous page)

```
Label {
    text: "4"
}

Label {
    text: "5"
}

Label {
    text: "6"
}

// Row 3
Label {
    text: "1"
}

Label {
    text: "2"
}

Label {
    text: "3"
}

// Row 4
Label {
    text: "0"
}

Label {
    text: "."
}

Label {
    text: "EXP"
}

}
```

5.4 The Button

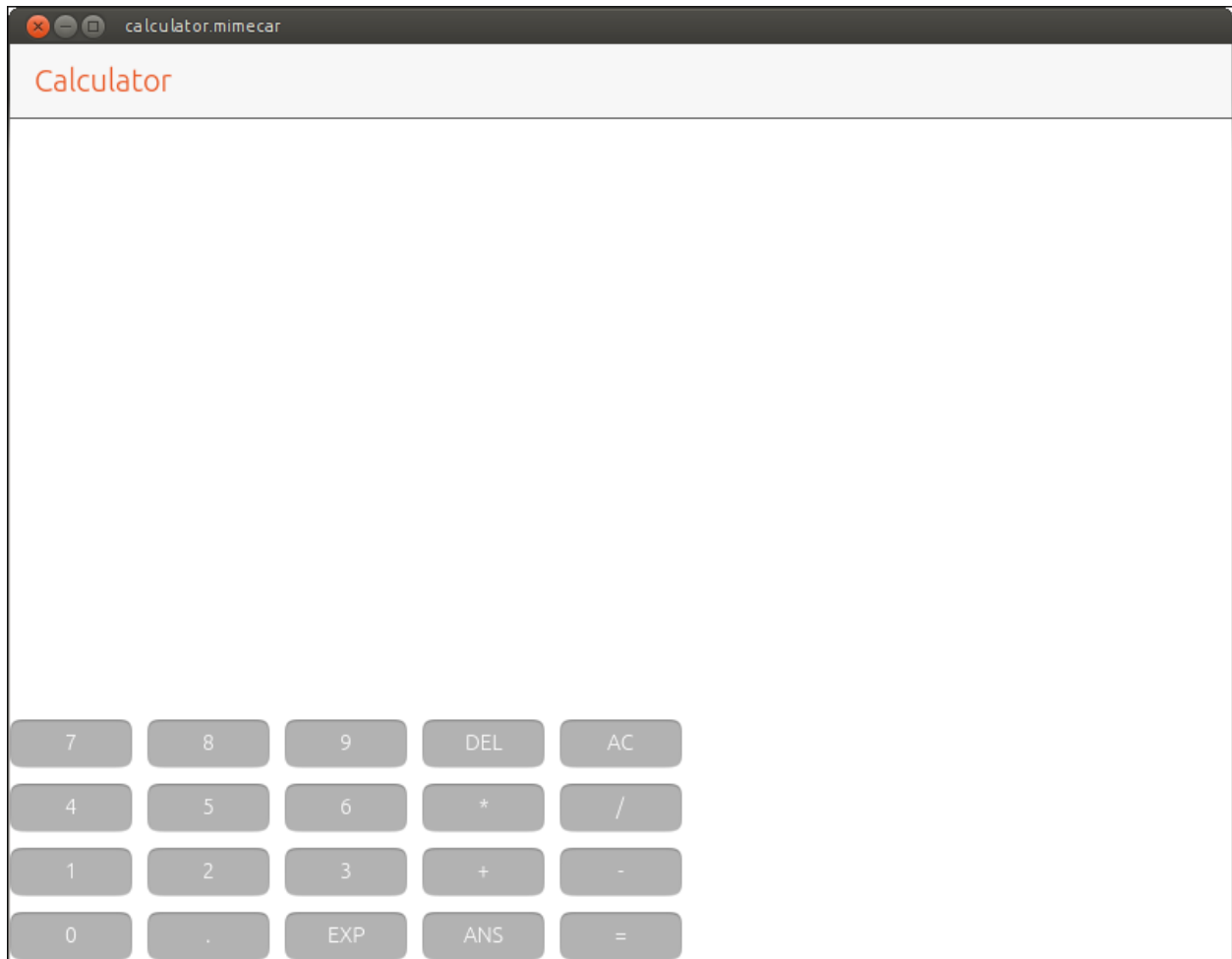
The label is a component that allows to display information to the user. The calculator that is being developed must allow the user to enter numbers and the operations to be performed with. To do this you have to use another component: the button.

A button has an associated text that tells the user the function it performs. When the user clicks or taps it, an action is generated that allows to execute an associated code. The button structure is a bit more complex than the label structure. We will learn it gradually.

```
Button {
    text: "Text"
}
```

Based on the code that shows the labels in a grid, we need to make several changes:

- Replace the labels with buttons. To do this, it is enough to change the Label text by the object Button.
- Modify the anchor of the grid so that it is aligned at the bottom of the screen.
- Add two new columns to the table with the buttons shown in the image.



Exercise

4

Source code:

```
Grid {

    anchors.bottom: parent.bottom
    spacing: 10
    columns: 5

    // First row
    Button {
        text: "7"
    }

    Button {
        text: "8"
    }

    Button {
```

(continues on next page)

(continued from previous page)

```
        text: "9"
    }

    Button {
        text: "DEL"
    }

    Button {
        text: "AC"
    }

    // Second row
    Button {
        text: "4"
    }

    Button {
        text: "5"
    }

    Button {
        text: "6"
    }

    Button {
        text: "*"
    }

    Button {
        text: "/"
    }

    // Third row
    Button {
        text: "1"
    }

    Button {
        text: "2"
    }

    Button {
        text: "3"
    }

    Button {
        text: "+"
    }

    Button {
        text: "-"
    }

    // Fourth row
    Button {
        text: "0"
    }
}
```

(continues on next page)

(continued from previous page)

```
Button {  
    text: "."  
}  
  
Button {  
    text: "EXP"  
}  
  
Button {  
    text: "ANS"  
}  
  
Button {  
    text: "="  
}  
}
```

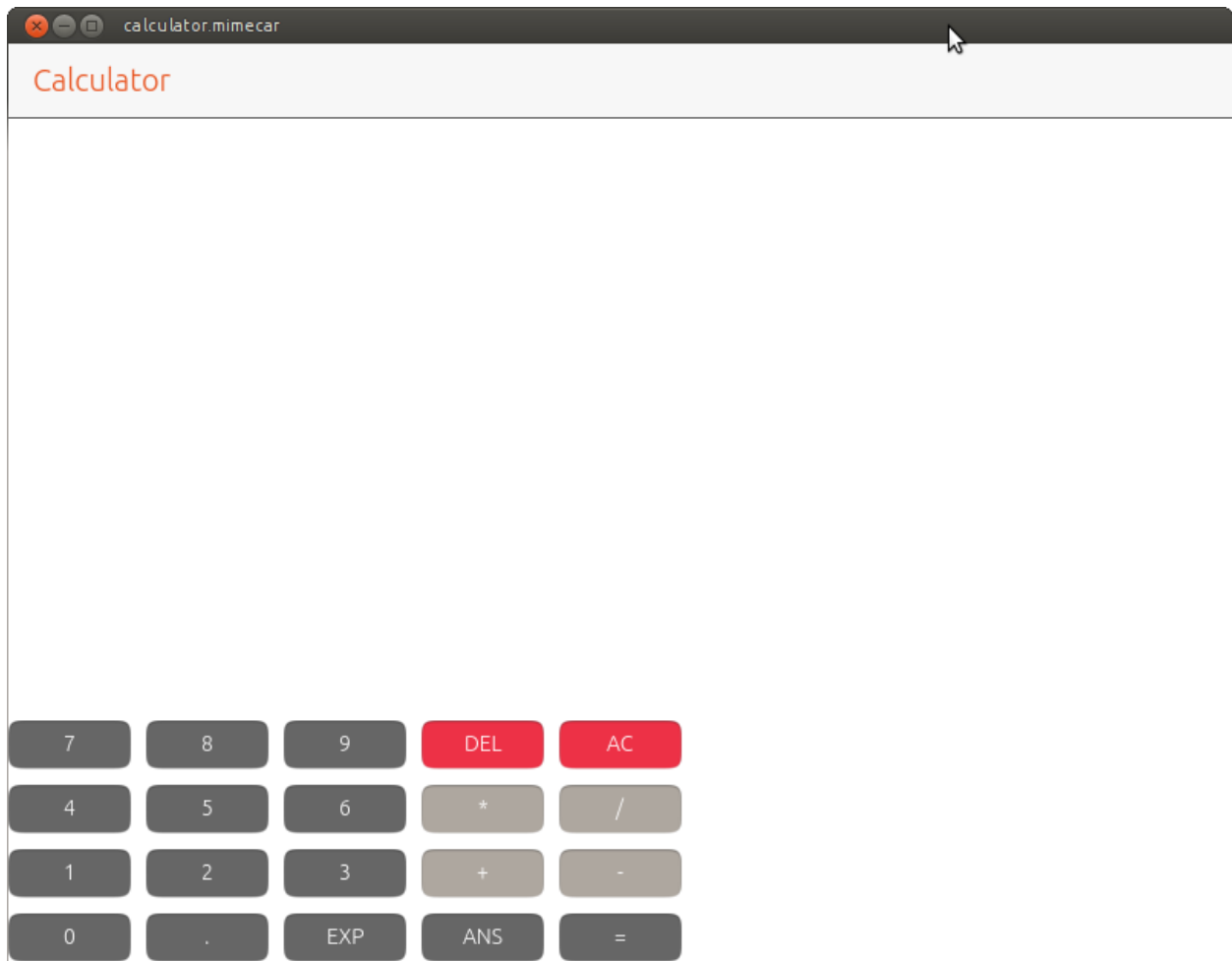
At this point, the application is starting to look like a calculator. There are two changes that would greatly improve its appearance though. The first is the colour of the buttons. The calculators have dark buttons and depending on the function, that colour varies. The second change is the size of the buttons. Do not forget that the calculator must work on a real device and normally the user fingers take up more space than the mouse pointer.

5.5 Colour Settings

Ubuntu Touch has a series of colours defined in its palette. If they are used in the design of the user interface, the application will look integrated into the system. To define the background colour of a button, the `color` property is used.

```
Button {  
    text: "8"  
    color: UbuntuColors.graphite  
}
```

You have to change the background colour of all buttons. The numbers should be dark gray. The operations a light gray and the buttons DEL and AC, reddish color. To complete the exercise you have to change the background color. It doesn't matter if it doesn't match the one on the screenshot.



Exercise

5

Source code:

```
Grid {  
  
    anchors.bottom: parent.bottom  
    spacing: 10  
    columns: 5  
  
    // First row  
    Button {  
        text: "7"  
        color: UbuntuColors.graphite  
    }  
  
    Button {  
        text: "8"  
        color: UbuntuColors.graphite  
    }  
  
    Button {  
        text: "9"  
        color: UbuntuColors.graphite  
    }  
}
```

(continues on next page)

(continued from previous page)

```
Button {
    text: "DEL"
    color: UbuntuColors.red
}

Button {
    text: "AC"
    color: UbuntuColors.red
}

// Second row
Button {
    text: "4"
    color: UbuntuColors.graphite
}

Button {
    text: "5"
    color: UbuntuColors.graphite
}

Button {
    text: "6"
    color: UbuntuColors.graphite
}

Button {
    text: "*"
    color: UbuntuColors.warmGrey
}

Button {
    text: "/"
    color: UbuntuColors.warmGrey
}

// Third row
Button {
    text: "1"
    color: UbuntuColors.graphite
}

Button {
    text: "2"
    color: UbuntuColors.graphite
}

Button {
    text: "3"
    color: UbuntuColors.graphite
}

Button {
    text: "+"
    color: UbuntuColors.warmGrey
}
```

(continues on next page)

(continued from previous page)

```
Button {
    text: "-"
    color: UbuntuColors.warmGrey
}

// Fourth row
Button {
    text: "0"
    color: UbuntuColors.graphite
}

Button {
    text: "."
    color: UbuntuColors.graphite
}

Button {
    text: "EXP"
    color: UbuntuColors.graphite
}

Button {
    text: "ANS"
    color: UbuntuColors.graphite
}

Button {
    text: "="
    color: UbuntuColors.graphite
}
}
```

5.6 Changing the Button Size

The user interface of the calculator can be used on a computer. In a real device the buttons are too small. An important detail is that the interface has to adapt to the characteristics of each device. It is not the same to have a vertical mobile screen than having it horizontal.

For now, we will use a vertical screen in mind to design the app. The buttons have two properties that allow you to define their vertical and horizontal sizes. The text of the button doesn't automatically adapt to the size of the button so it will have to be adapted too. The button would have the following structure:

```
Button {

    // Font size
    text: "7"
    font.pointSize: 17

    // Background color
    color: UbuntuColors.graphite

    // Button dimensions
    width: buttonWidth
```

(continues on next page)

(continued from previous page)

```

    height: buttonHeight
}

```

Repeat the same structure on all buttons. To make it easier to adjust the size of the buttons, we will define two variables. The variables are defined before the view (Page) and have the following structure:

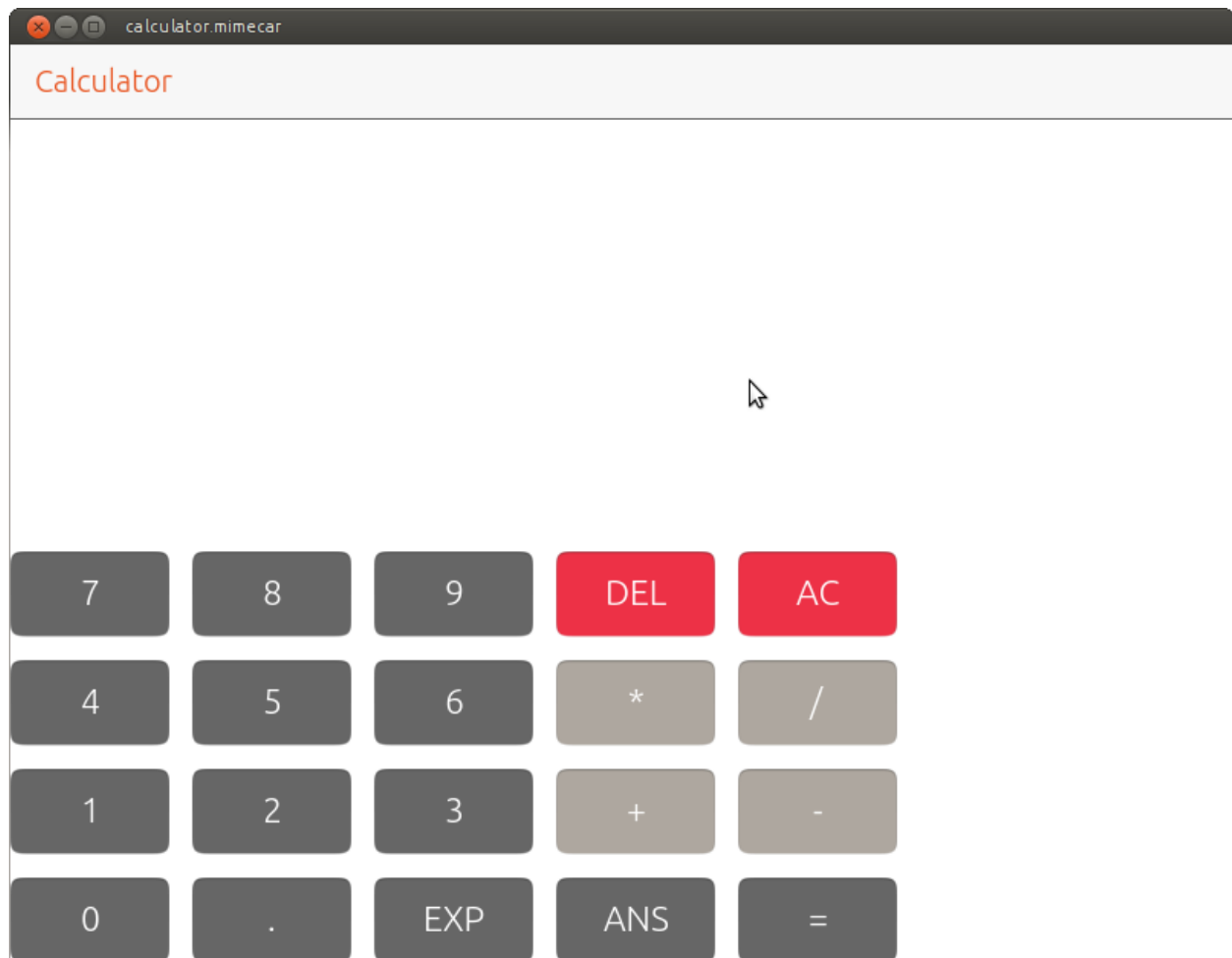
```

property real buttonWidth: units.gu(13)
property real buttonHeight: units.gu(7)

```

They are defined with the word ‘property’ to indicate that are a variable. ‘Real’ defines a number with decimals. Remember that the dimensions must be expressed in the form of GridUnits.

You need to adjust the size of the text and the dimensions of all buttons on the calculator. The result must be similar to the following:



Exercise

6

Source code:

```

Grid {
    anchors.bottom: parent.bottom
    anchors.left: parent.left
}

```

(continues on next page)

(continued from previous page)

```
anchors.right: parent.right

spacing: 15
columns: 5

// First row
Button {
    text: "7"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "8"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "9"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "DEL"

    font.pointSize: 17
    color: UbuntuColors.red

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "AC"

    font.pointSize: 17
    color: UbuntuColors.red

    width: buttonWidth
    height: buttonHeight
}

// Second row
```

(continues on next page)

(continued from previous page)

```
Button {
    text: "4"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "5"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "6"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "*"

    font.pointSize: 17
    color: UbuntuColors.warmGrey

    width: buttonWidth
    height: buttonHeight
}

Button {
    text: "/"

    font.pointSize: 17
    color: UbuntuColors.warmGrey

    width: buttonWidth
    height: buttonHeight
}

// Third row
Button {
    text: "1"

    font.pointSize: 17
    color: UbuntuColors.graphite
```

(continues on next page)

(continued from previous page)

```
        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "2"

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "3"

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "+"

        font.pointSize: 17
        color: UbuntuColors.warmGrey

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "-"

        font.pointSize: 17
        color: UbuntuColors.warmGrey

        width: buttonWidth
        height: buttonHeight
    }

    // Fourth row
    Button {
        text: "0"

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "."
```

(continues on next page)

(continued from previous page)

```
        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "EXP"

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "ANS"

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }

    Button {
        text: "="

        font.pointSize: 17
        color: UbuntuColors.graphite

        width: buttonWidth
        height: buttonHeight
    }
}
```

This chapter ends with this example. The next step is to add a label at the top of the screen to show the operations and their result.

5.7 People who have collaborated

- Larrea Mikel: revision of the chapter in Spanish.
- Cesar Herrera: revision of the English translation.
- Joan CiberSheep: revision of the English translation.

6.1 Formatting labels and Events

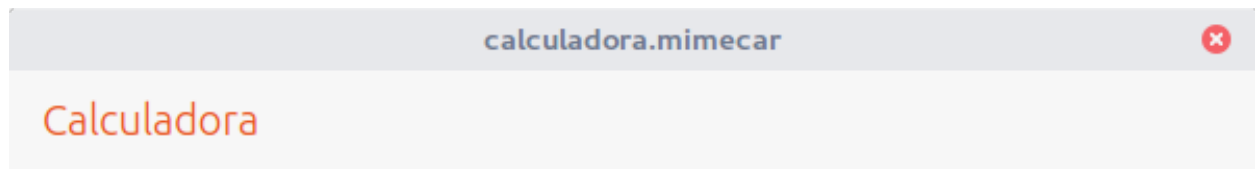
Starting with the previous code, the calculator with buttons, we will add in this chapter more information regarding labels. A label is a component which shows information to the user. It can be used both for showing fixed and information that can change. The first case, fixed information, was already presented at the beginning of this course, now we are going to focus on the second one. When the user pushes the buttons of the calculator it has to get feedback about what has been done. The buttons change aspect in an automatic way but they have no information about the operations performed. This problem will be fixed using a label which shows the values of the operation.

Besides showing the text in the label, it is necessary to save the numbers and the operation somewhere. A couple of auxiliary variables will be used for that purpose. The flow of the app would be:

- User introduces the first number
- Mathematical operation is selected
- User introduces the second number
- The ANS button is pushed.

The easiest way would be concatenating text on the button and the label to show the operation to the user. The problem with this approach is it would make necessary to extract that information of the label later to do the operations by code. Unfortunately, this solution has more cons than pros. A different approach comes following the opposite way: the buttons update some variables and, then, update the labels. In this way, the label would show only the internal status (model) and it would be easier to handle. The use of methods (functions) will be also introduced to implement this feature.

Let's start taking as initial point the calculator app from previous chapter as shown in the next figure.



Code

of the calculator used as starting point

6.2 Label to show operations

In a first iteration the label will be fixed in the upper part of the grid of buttons. The gap at the top will be used later to show a history of the operations with another component to work with text. The first step will be, therefore, to create the component with a text field by default. The point is, how do we start? Based on the comments related to the previous exercise of the grid done by the participants, there are still some doubts about how the anchor works. From this moment, anchors will be used quite frequently, so it is important to remember its functioning to continue this chapter.

The components in QML need to be seen as boxes. Each box has four anchors in the up, down, left, right positions. Playing with those anchors is possible to distribute the components on the screen in an arbitrary way. There are two ways for the distribution of the components:

- By defining relationships between them using anchors.
- By using a layout.

In the calculator the two previous ways have been already applied during this course. The buttons are distributed using a layout grid. The grid has the same behaviour than a component in QML, which makes it, in terms of design, a box with four anchors. The components are placed inside a Page which is at the same time, a box with anchors.

To move the grid of buttons, the anchors of the page and the anchors of the grid need to be related. Depending on this relationship, the organization will change in the screen. The easiest way is to link the position of the grid bottom to the same position in the page:

```
anchors.bottom: page.bottom
```

One of the common doubts arriving at this point of the course is: where is “page” defined? The component in the code is Page and QML is case sensitive. When a component is created, several attributes must be defined. Among those attributes, there is an identifier which allows to access the component from other parts of the code. This id is not mandatory to show the component, but it is mandatory when you want to use the component later. The Page definition would be:

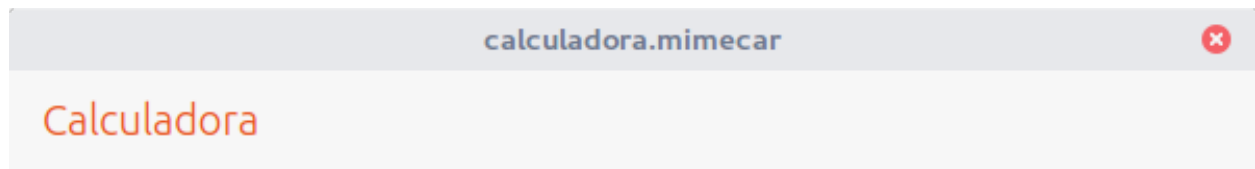
```
Page {
    id:page
    ...
}
```

Now, any component could access the information in Page. The identifier (“id”) can be both used in the components and in the layouts. If an id is added to the grid, it will be possible to anchor the label to the component. In this new way, if the grid grows when adding new components, the label will adapt to the new position without additional changes.

Let’s create the label and assign an ID:

```
Label {
    anchors.bottom: grid.top
    text: "Operation and result"
}

Grid {
    id: grid
    ...
}
```



Operación y resultado

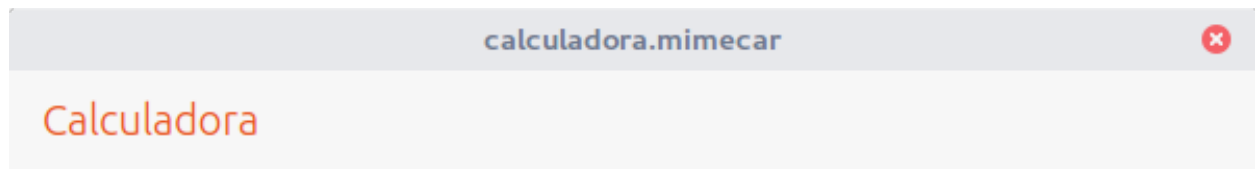


Label

anchored to the grid

The label is placed in the upper part of the grid but the text remains a little bit weird in the left part of the screen. Following the comparison with the box, the label is a box which starts in the first “O” and it ends in the last “o”. To align it to the right, it is necessary to add a new anchor, which must be the same for label and grid

```
anchors.right: grid.right
```



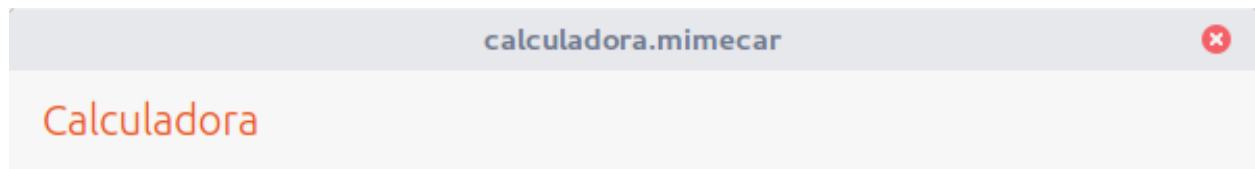
Label

placed

An important detail, which cannot be appreciated in the figure due to the background, is that neither grid nor label are aligned with the right part of the screen. There is still an empty part with no use. It is possible to assign the label to the anchor of the page instead of using the corresponding one of the grid, but in that case buttons and labels would be misplaced. A solution to this problem will be shown later in this course.

Before continuing with the app it is necessary to fix a little bit the appearance of the label. The text must be bigger and the font will be changed for a smaller monospace type of font, that means, all characters equally spaced. Both modifications will be applied to the Label, which has been created previously.

The same logic applied in the buttons, using the attribute `font.pointsize`, will be follow to change of the fontsize. The final result should be close to:



Label

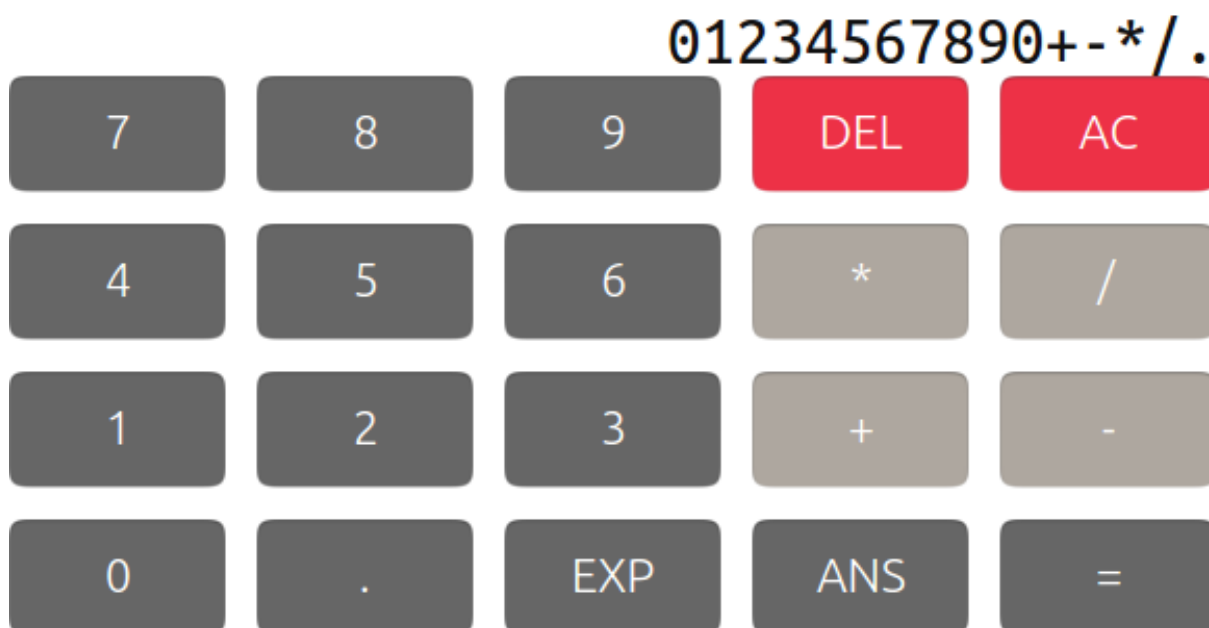
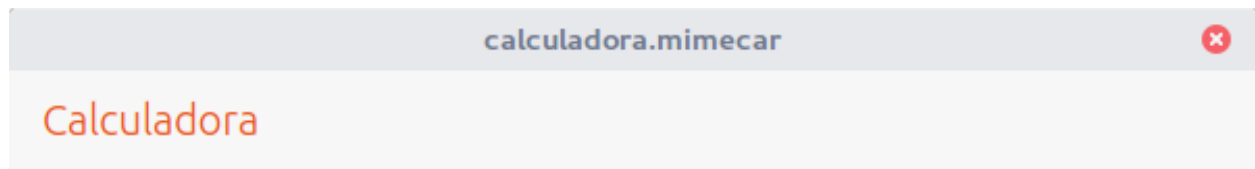
with size modified

In this example the fontsize is 25 px. You can find an example of a similar code in the buttons and it will be great for you to struggle a little bit with this part of the app. The doubts of the previous part appeared just in the parts of the code which were not specifically in the explanation. It would be very easy for me to solve your doubts, but it is important for you to look for your own solution when facing problems.

The attribute `font.family` can be used to change font family. The following code has to be added inside of the label:

```
font.family: "UbuntuMono"
```

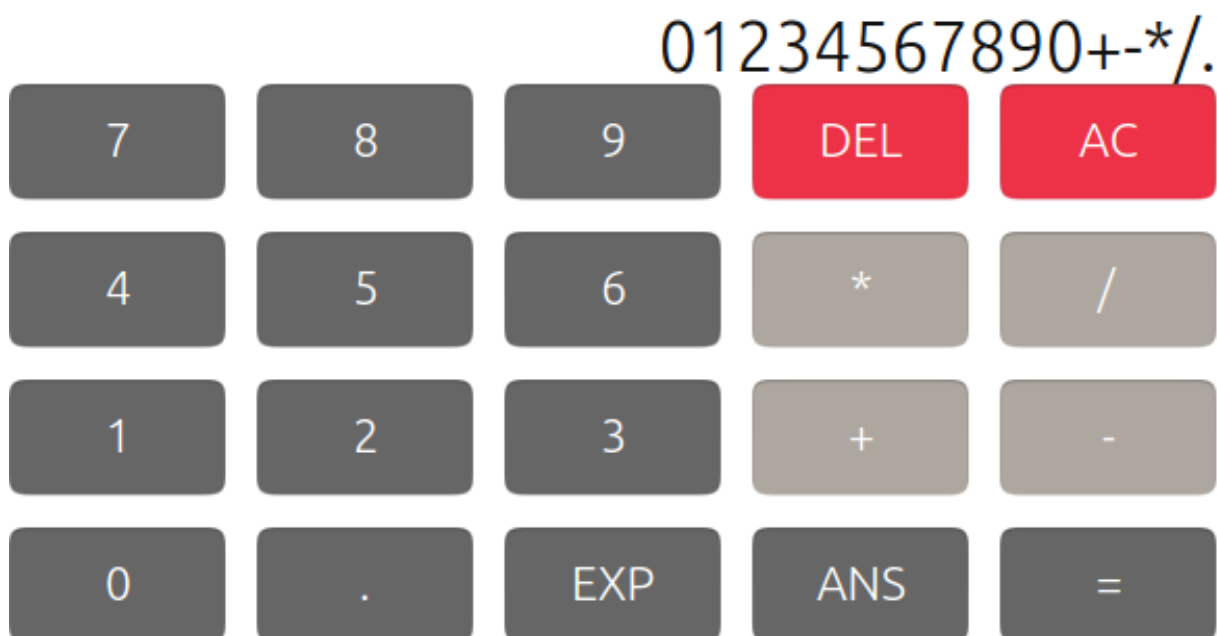
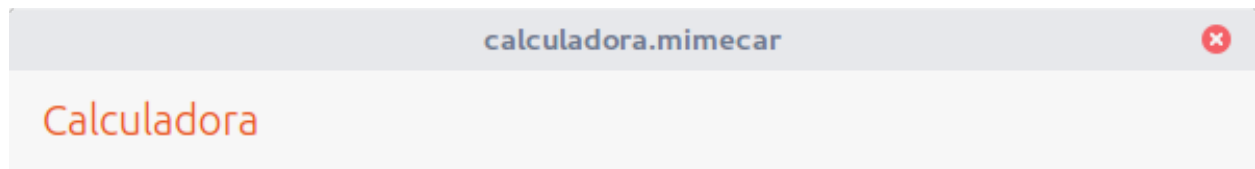
Just with text characters is more complicated to distinguish the effect of the monospace font. That is the reason why I have replaced the initial text by numbers and operators.



Monospace

font

The same code but with the font by default. There is no difference in the numbers when changing font type but you can see some major changes in the operator symbols. Notice specially the change in position in the zero respect the “AC” button in both figures.



Normal

font

6.3 Adding logic to the buttons

The objective is making the buttons pressed by the user appear as text of the label. If the user presses number 7, the number 7 will appear as text. When the user presses another button, the text present in the label will be concatenated with the text of the new button.

When components are used in QML, several signals are generated automatically. For example, button pressed or button released. Those events can be processed in the code and do other actions when detected. As an example of this behaviour, the signal `onClicked` will be processed. The code of button “7” will be set as follows:

```
Button {
    text: "7"

    font.pointSize: 17
    color: UbuntuColors.graphite

    width: buttonWidth
    height: buttonHeight

    onClicked: {
        result.text = result.text + "7"
    }
}
```

To access the label, it is necessary to create first an identifier (id) with the name “result”. When pressing the button, an **onClick signal** is generated. As that signal is the one being captured, the code in curly brackets is being executed. Basically, it makes the attribute text to concatenate the value it had with the new value of the number.

As the final exercise of this chapter, I will ask you to add the logic to all the buttons of the calculator in the same way. The buttons of the operations can show in the label the same text as the one on the button. At the end, the app needs to show in the label the buttons pressed.

This part is a little bit shorter than the previous ones. Besides learning how the QML components work, I am introducing logic to the app. This logic allows the app to perform certain actions when users work with it. In the course, I start from scratch considering the knowledge of the user in programming QML and general programming. That is the reason why I am introducing concepts step by step to make it easy to follow.

If you have any question, please contact me using the contact details associated with the course and I will be pleased to answer all your questions.

6.4 People who have collaborated

- Santiago Mohr: English translation.

QML App - Database access

This chapter will show how to access to a *SQLite* database from a *QML* + JavaScript application. *SQLite* is the database used by *Ubuntu Touch (UBports)* (and other mobile devices). For more informations about *SQLite* see its [website](#) and the links proposed at the end of the chapter.

The prerequisites necessary to understand this tutorial are a knowledge of *QML*, JavaScript, Ubuntu Touch SDK with his IDE and a base SQL language knowledge. If you don't have that prerequisites you are invited to read the dedicated sections of the *QML* course and the suggested tutorial at the end of the chapter.

Premise: for the database access will be used *QtQuick.LocalStorage* object (provided by QT library), and not *UIDb-QT* (the *QML* module created by Canonical). This choice is due to the fact that *UIDb-QT* currently seems incomplete and not flexible like the JavaScript solution.

7.1 Introduction

In this chapter we'll be showed a custom *QML* application to provide examples of the common operations performed on a database: Create, Research, Update, Delete (ie: a C.R.U.D. application). That sample application will have a basic user interface (our focus is on the database access part) and don't have an important real use; was chosen and created only for his didactic relevancy.

Note: This chapter don't cover the steps necessary to package, test and deploy the application (will be the focus of next chapters).

7.2 The sample application: functional presentation

The application that we are going to describe is a "weather temperature recorder". Using it, the user can insert the daily temperature value of his favourite city and manage the saved ones: update, delete or search them. At first start-up, the application must provide default configuration values for the target city and the temperature unit (ie: Celsius or Fahrenheit degree). The user can accept the proposed values or insert custom ones (the saved configuration can be updated later if the user want do it).

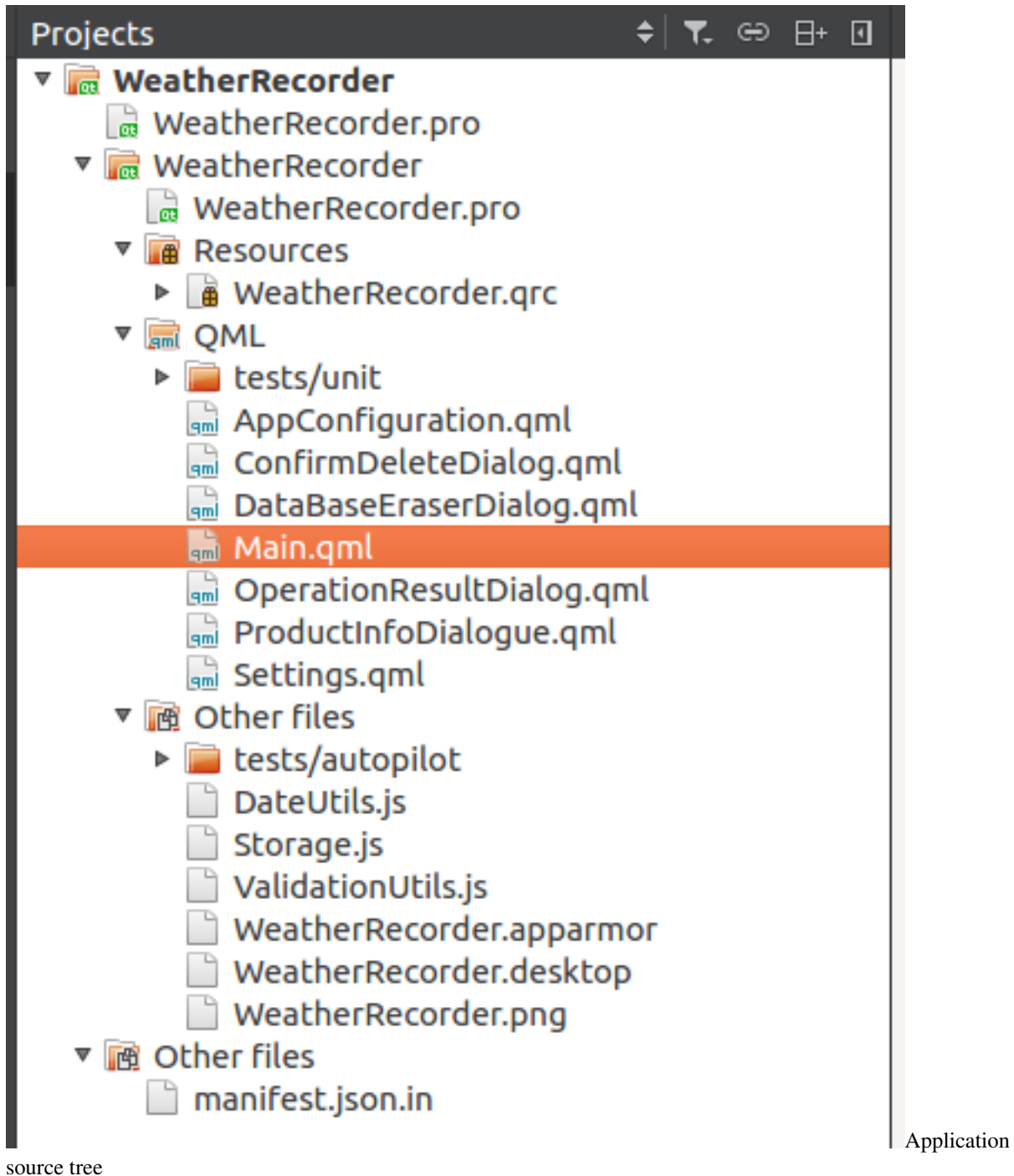
Each day the user can insert *only* one temperature value. Saved values can be searched by date. If a value is found, is possible modify or delete it. Is also possible execute a batch deletion of all the saved temperature values. The application must provide a validation system to prevent invalid insert, like empty or not numeric values.

Let's start:

1. Set-up the Ubuntu IDE and SDK as described at the beginning of this course creating at least a “Desktop” kit.
2. Download and import our sample project (named “WeatherRecorder”) in the Ubuntu IDE (File → Open File or Project... → choose the folder containing the file ‘WeatherRecorder.pro’)

During the import, Ubuntu SDK ask to choose the “Target kits” for the project (ie: the platform where the application will be executed). We want run our project form the IDE, so that is necessary flag an existing “Desktop Kit” or create a new one. You can find more details about “Kit” at the beginning of this course.

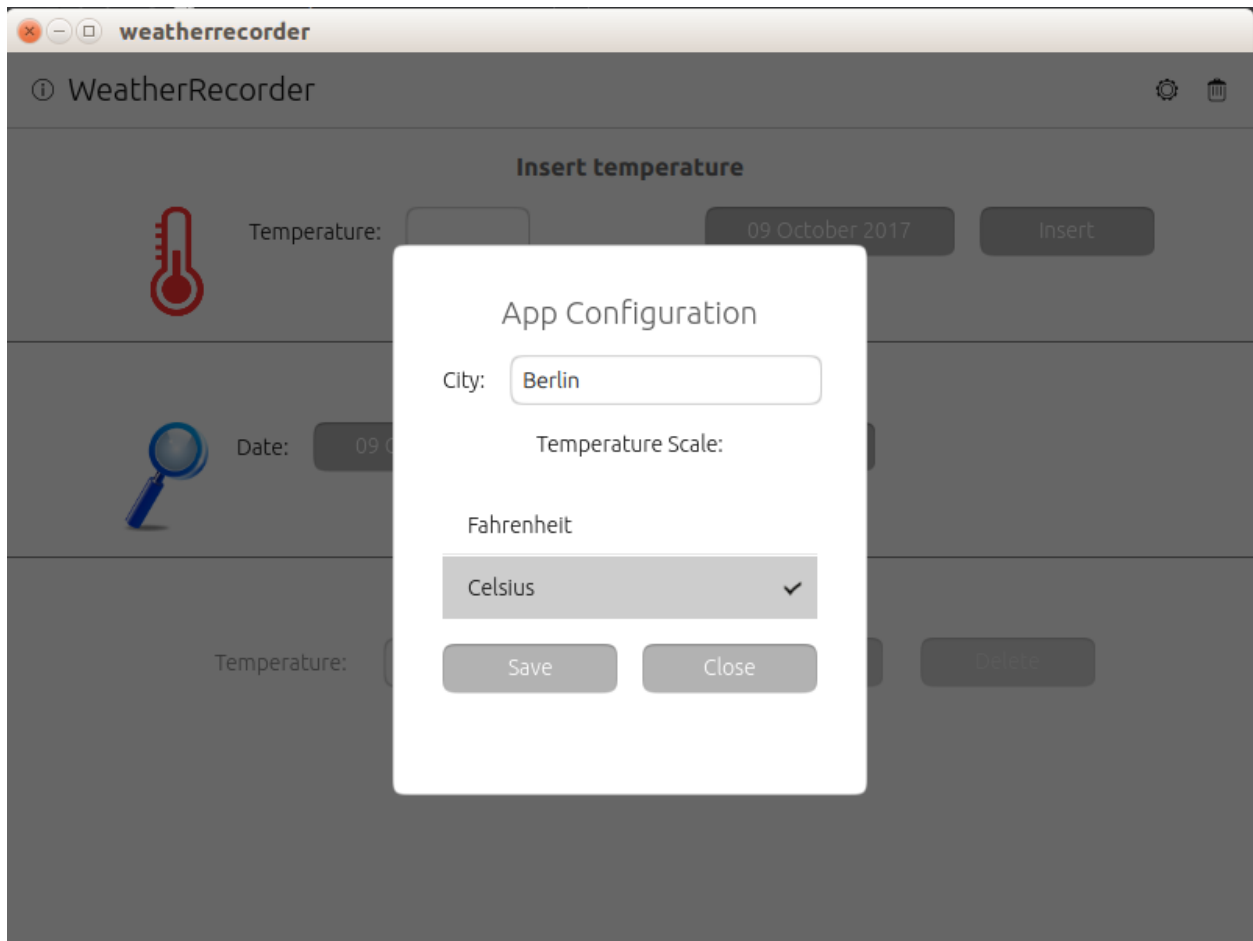
At the end of import process, expand the nodes in “Projects” view. The structure must be like the following one:



Feel free to open and explore any files and study the project structure before continue. Before concentrate our attention on the database access, we want provide some background informations that could be useful for the future.

The entry point of a QML application is the *Main.qml* file, it contains the code that create and start the full application. Running for the first time our application (press the green “Play” icon of Ubuntu SDK) is shown a configuration dialogue filled with default values. If you like them, press “Save” button, otherwise modify them before save.

The following image display the configuration page shown at first start-up:



Application

configuration wizard

Let's see more details about what happens when the application starts. If you open the *Main.qml* file you can find this code (you can use CTRL+ F to perform a search):

```
Component.onCompleted: {
    //some code... omitted for shortness
}
```

All the code placed inside that block will be executed when the event *onCompleted* is raised. In this case each time the application starts, because the above code is placed inside the *QML Page* component. But, the *onComplete* event is raised for any other component, so that can be used to initialize it instead of the application.

For example:

```
Label{
    id:myLabel
    width: units.gu(10)
    Component.onCompleted: {
        myLabel.text = /* get his value from the database */
    }
}
```

When the *onComplete* event of the *Label* component is raised (ie: when is drawn), is executed the code that initialize the *Label* (e.g. loading the text to display from a database or other source). In general, the *onComplete* event is used to perform some initialization tasks at application or single component level.

In our specific case, at *onComplete* event of the application, are performed two operations:

1. Database and tables creation (we'll talk about that in the next section).
2. Inserted default values (ie: the ones shown in the configuration dialogue shown at first start-up).

Now, someone could say: "... we don't want perform that operations at each application start !" Right, to prevent this, the solution adopted is the use of *Settings* component. It provides persistent application settings (ie. user preferences or other customizations to be persisted on a configuration file).

In our *Main.qml* we have:

```
Settings {
    id:settings
    /* flag to show or not the App configuration pop-up */
    property bool isFirstUse : true;
}
```

It declare a boolean property named *isFirstUse* initialized to *true*. After the first initialization operations, inside the *onComplete* management block, is set to *false*. With this solution, next time the configuration dialogue will be not shown and the database initialization will be skipped.

To distinguish the first start form the other ones, we use an *if* check, so that the decision logic is the following:

```
Component.onCompleted: {
    if(settings.isFirstUse)
    {
        //create database, insert default data
    }else{
        //if necessary do something else
    }
}
```

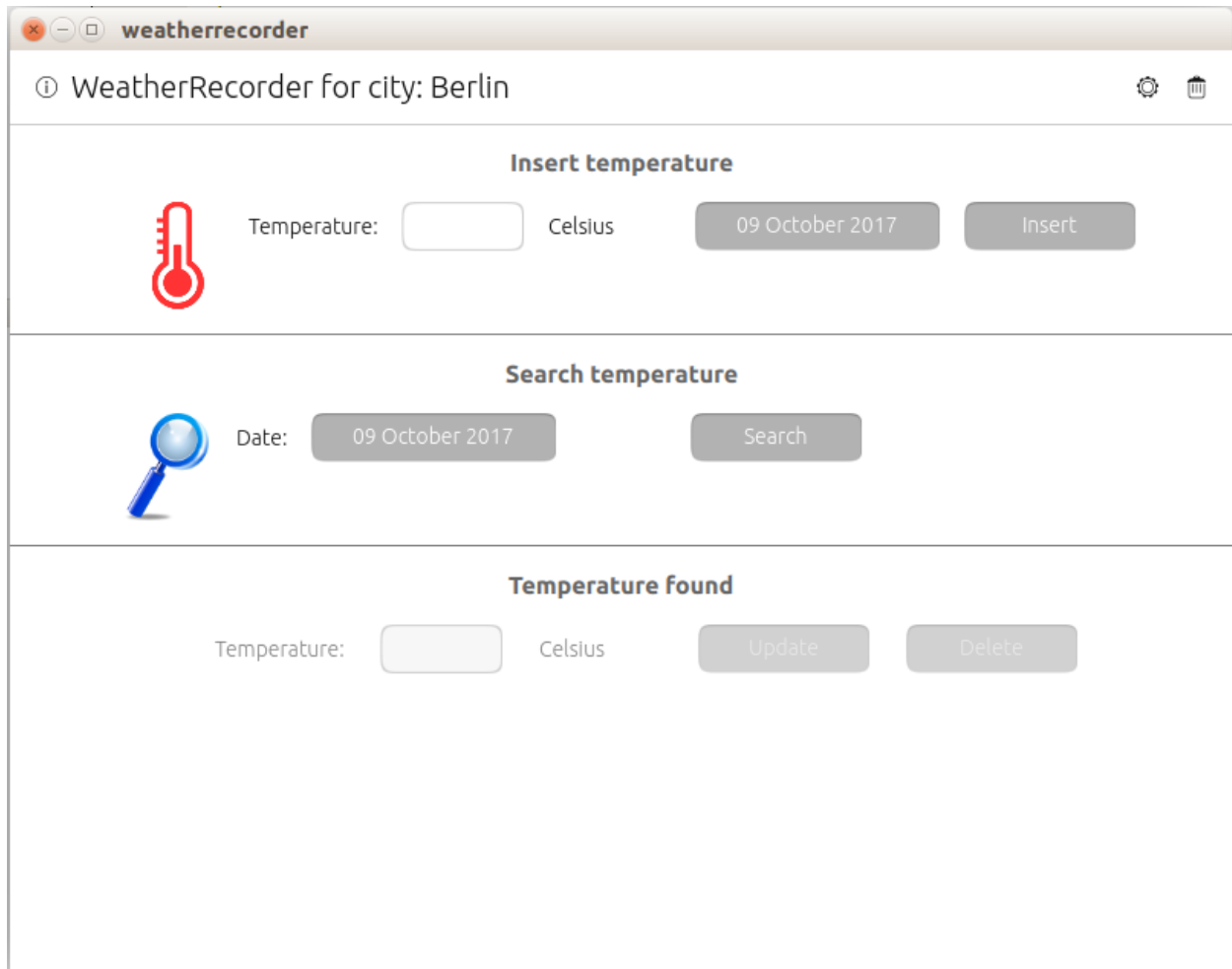
But, where are saved the Setting values (ie: our "isFirstuse" flag)? They are saved in a .conf file placed in that folder:

```
~userHome/.config/<applicationName>/<applicationName>.conf
```

In our case is *weatherrecorder* (is the value assigned at the parameter "applicationName" in *Main.qml*). Feel free to open that file, is a simple text. Removing it, or setting to *true* our flag value, at the next application start will be executed the initialization operations (try it if you want).

Also, note that in the project sources there is a file named *Settings.qml*. It must contains all the flags/variable to be stored in the *weatherrecorder.conf* file. The *Settings* component can be used to implements any other logic, not only for the use case described here.

After the configuration saving the home page of the application is the following:



Application

home page

The user interface and his use is quite simple, so that we omit his description. Before continue we suggest to try it to get more confidence with the application behaviour. Just some notes about the header bar of the application that could be useful for future applications. We can see two “action” bars: one in the upper right corner and another in the upper left one.

The left one is named *leadingActionBar*, the one on the right is named *trailingActionBar*. Look in the `Main.qml` source code and his code comments to know how are implemented (perform a search in that file with a CTRL+F).

In the next sections, we’ll look at the chapter focus: the database access from a QML + JavaScript application.

7.3 Introduction at the Database access

As said before, we don’t use *UIDb-QT* API but `QtQuick.LocalStorage` object to perform the database access. The database type used by *Ubuntu Touch (Ubports)* is *SQLite* a file based database that support SQL as query language. To use it from a *QML* application are not necessary installations or configurations: everything is provided by *QML* and *Ubuntu SDK*.

Our database access logic is contained in the file named `Storage.js` (there is no naming convention for that file, you can use another one or use DAO pattern if you prefer). The following examples are taken from that file.

The `QtQuick.LocalStorage` module must be imported in any *QML* file that need it, using this statement:

```
import QtQuick.LocalStorage 2.0
```

Is also necessary import the JavaScript file containing the database access functions (*Storage.js* in our case):

```
import "Storage.js" as Storage
```

This is the import syntax to use any JavaScript from QML file, not only for our sample. To invoke the functions contained in *Storage.js* is used the declared alias *Storage*. Is suggested use a name similar at the JavaScript file for the alias. Remember that alias names **MUST** have the first letter in upper Case.

7.4 The Database interaction

Follow the description of the database operations performed by our application (see *Storage.js* file for the full code).

7.4.1 Database and table(s) creation

In any *QML* application that need a database, the first operation to be performed is the database and table(s) creation. This operations (like any others) require a connection at the physical database. The connection is created with the *QtQuick.LocalStorage* object and usually is a best practice create an utility JavaScript function that return a reference to it, like this:

```
function getDatabase() {
    return LocalStorage.openDatabaseSync("weatherRecorder_db", "1.0",
    ↪ "StorageDatabase", 1000000);
}
```

For more details about the API *openDatabaseSync* parameters see: <http://doc.qt.io/archives/qt-5.5/qtquick-localstorage-qmlmodule.html> (we omit it for shortness). Before perform any database operation is necessary obtain a connection with it. For this purpose, the above function can be re-used in any other JavaScript function that need the database (See *Storage.js* file).

When a connection is obtained, is possible execute a database *SQL* query. Note: Database connections are automatically closed during JavaScript garbage collection (a cleaning process executed by JavaScript engine to free memory).

Let's see the database table creation code (we omit the full code to have a compact view. See the source file for details):

```
/* create the necessary tables */
function createTables() {
    var db = getDatabase();
    db.transaction(
        function(tx) {
            tx.executeSql(' < SQL Create statement 1 > ');
            tx.executeSql(' < SQL Create statement 2 > ');
        });
}
```

Note: as first thing, is called the *getDatabase()* function the return a database connection. Like the above code, each time we need to perform a database query (ie any C.R.U.D. operation) is necessary open a transaction that is passed at the callback function containing the *SQL* statements to run.

If the callback function throws exceptions (ie: an error), the transaction is rolled back (a 'transaction' is a set of operations that must be executed with success, if one of them fails is executed an undo to restore the status before the execution).

For our example application, in the *createTables()* function are created two tables: one to store temperature values and one for the configuration parameters. No foreign keys are used.

For *configuration* table the creation SQL code is:

```
CREATE TABLE IF NOT EXISTS configuration(id INTEGER PRIMARY KEY AUTOINCREMENT,
      param_name TEXT, param_value TEXT)
```

For *temperature* table is:

```
CREATE TABLE IF NOT EXISTS temperature(id INTEGER PRIMARY KEY AUTOINCREMENT, date_
↪TEXT, temperature_value REAL)
```

(The above code is the one placed inside the < SQL Create statement > placeholder).

Obviously, the declared data-type of the table columns are the ones supported by *SQLite* database (the type “date” is not supported, so that is declared as TEXT). Note the use of a field named “id” and marked as Primary key (PK) in the two tables.

That field is flagged as *AUTOINCREMENT* so that each time a new record is inserted in the table his value is automatically incremented by *SQLite*. The use of an “autoincrement” field as PK is not mandatory if there are field(s) that can identify only one record in the table (we choose to use it to provide a further useful feature of *SQLite*).

After the execution of *createTables()* function the *SQLite* database file is created and two tables are inserted.

We have said that *SQLite* is a file based database; but, where are placed our database files? The location depends on the application name. The general full path is:

```
/<home-folder>/.local/share/<applicationName>/Databases/
```

where is the values provided in the *applicationName* parameter in *Main.qml*:

```
applicationName: "weatherrecorder"
```

So that, our database is placed at:

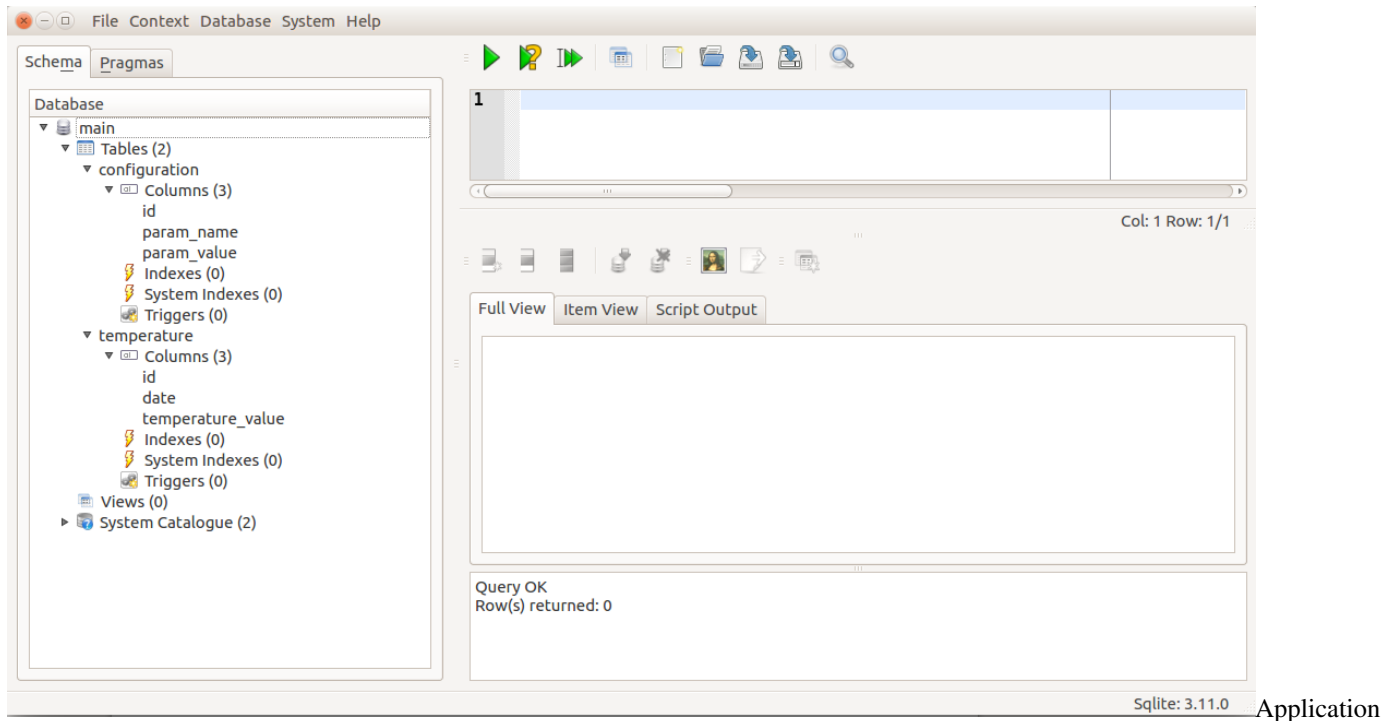
```
/<home-folder>/.local/share/weatherrecorder/Databases/
```

On *Ubuntu Touch* (UBports) systems is */home/phablet/* on Desktop systems replace ‘phablet’ with the name of the logged in user (type the command *whoami* in a shell to know his name if you are not sure).

That folder contains two files: a *.sqlite* and a *.ini* one. The database file is the one with *.sqlite* extension (the name is an unique id created by *QTQuick.LocalStorage*). The *.ini* file is a file descriptor associated at the *.sqlite* one. Don’t worry about their strange names, is not necessary remember or manage them.

To see our two tables and perform *SQL* query, open the *.sqlite* file with a graphic interface like *Sqlliteman* available in the “Ubuntu software Center”.

If you open the database file with *Sqlliteman* you got a view like this one:



source tree

On the left side we can see the tables found in the database ("configuration" and "temperature") with their columns. In the upper right text area we can execute some *SQL* statements. For example, write this one and press the *Play* icon in the menu bar:

```
select * from configuration;
```

The result is the content of the *configuration* table (ie: our configuration parameters provided at the first application start-up). If you right click on table name is possible perform some administration tasks on the table (add or edit column, delete the table and so on).

Obviously any modification made with *Sqliteman* affect the application that uses the database (e.g.: if a table is dropped our application will crash because is necessary modify the application).

In *QML* + *JavaScript* development, *Sqliteman* (or any other similar tools) is used only for checks purpose, like running *SQL* extraction statements or data insertion/editing. For example to insert sample data or correct wrong ones. We'll talk about *SQL* statements in the next sections.

7.4.2 Research operation

With a research operation we look inside the database table(s) for a specific record(s) matching some search criteria. That operation is done with a *SQL select* query that return a set of zero or N table rows matching the criteria (for us *record* and *table row* are synonym). The returned *set* is usually identified with the *result-set* name.

An example of research, can be found in the function named *getTemperatureValueByDate* in *Storage.js* file. That function is invoked when the user perform a search operation in the user interface passing as input the date for which want to know the temperature value.

Here the JavaScript function that perform the search:

```
function getTemperatureValueByDate (date) {
```

(continues on next page)

(continued from previous page)

```

var db = getDatabase();
var targetDate = new Date (date);

/* return a formatted date like: 2017-04-30 (yyyy-mm-dd) */
var fullTargetDate = DateUtils.formatDateToString(targetDate);
var rs = "";

db.transaction(function(tx) {
    rs = tx.executeSql("SELECT temperature_value FROM temperature t where date(t.
↪date) = date('"+fullTargetDate+"')");
    } );

/* check if value is missing or not */
if (rs.rows.length > 0) {
    return rs.rows.item(0).temperature_value;
} else {
    return "N/A";
}
}

```

Note at the beginning the *getDatabase()* function (described in the previous section) and how we can invoke functions contained in another JavaScript file (look at the statement *DateUtils.formatDateToString(targetDate);*). The technique adopted to execute a *SQL* query is the same used for the table creation: create a transaction and inside the callback function insert *SQL* statements.

The above example show how to manage the results returned by the *SQL* query. Is used a JavaScript variable named *rs* that act as an handle to the result-set informations. The access at the records/rows must be performed with the built-in field named *rows* (a keyword name). It is a pointer at the set of returned records used to access at a specific one or iterate over all the records).

For example:

```
rs.rows.item(0).temperature_value;
```

get access only at the *first* record of the returned result-set (note: the count is zero-based) and from it take the value of table column *temperature_value*. We can do that because we have executed a *select* query; using other *SQL* statements (see following sections) the returned information will not be a set of table records but a simple number.

To iterate over the full result-set records, for example to print the *temperature_value*, this JavaScript code must be used:

```

for(var i = 0; i < rs.rows.length; i++) {
    console.log("Temperature found:" + rs.rows.item(i).temperature_value);
}

```

where *console* is a built-in object used to print messages on the console (is similar at *System.out.println("a message")* in Java).

7.4.3 Update operation

The update operation is performed with a *SQL update* query. An example of update is the *updateTemperature* JavaScript function used to update an already stored temperature value.

```

function updateTemperature (date,tempValue) {
    var db = getDatabase();

```

(continues on next page)

(continued from previous page)

```

    var fullDate = new Date (date);

    /* return a formatted date like: 2017-09-30 (yyyy-mm-dd) */
    var dateFormatted = DateUtils.formatDateToString(fullDate);
    var res = "";
    db.transaction(function(tx) {

        var rs = tx.executeSql('UPDATE temperature SET temperature_value=?WHERE date=?
↪; ', [tempValue, dateFormatted]);

        if (rs.rowsAffected > 0) {
            res = "OK";
        } else {
            res = "KO";
        }
    }
);
return res;
}

```

In the above example is shown another use of the query result-set. The returned value is a number NOT a set of record(s), due to the facts that we have executed an *SQL* update query that return only the number of table row(s) updated. That value is contained in the field named *rowsAffected* (is a fix name you can't change it). The above JavaScript function return a string message whose value depends on *rowsAffected* value. Is also possible return the *rowsAffected* value without any decision logic; the choice depends on the logic to implement.

7.4.4 Insert operation

To insert a new record/row in a table is used a *SQL insert* statement. In our use case we perform that operation when a new temperature value is inserted:

```

/* Insert a new temperature value in the give date */
function insertTemperature(date,tempValue) {
    var db = getDatabase();
    var fullDate = new Date (date);
    var res = "";
    var dateFormatted = DateUtils.formatDateToString(fullDate);

    db.transaction(function(tx) {
        var rs = tx.executeSql('INSERT INTO temperature (date, temperature_
↪value) VALUES (?,?); ', [dateFormatted, tempValue]);

        if (rs.rowsAffected > 0) {
            res = "OK";
        } else {
            res = "Error";
        }
    });

    return res;
}

```

The logic and the returned value are similar at the update operation, except for the *SQL* statement executed.

7.4.5 Delete operation

The last operation that can be performed is the *delete* one. For example, this operation is executed when the user want delete all the saved temperature values, pressing the *trash* icon in the menu bar. This is the JavaScript function used:

```
/* Remove ALL the saved Temperature values NOT the tables. Return the number of rows
deleted */
function deleteAllTemperatureValues() {
    var db = getDatabase();
    var rs;
    db.transaction(function(tx) {
        rs = tx.executeSql('DELETE FROM temperature;');
    });

    return rs.rowsAffected;
}
```

In this case we want to mark that *rowsAffected* is the number of deleted table rows and that value is returned by the JavaScript function, without any decision logic. For example, the caller of the function can assign it to a *QML* Label component to be displayed or use it for a different purpose.

7.5 Conclusions

We have shown how to create from scratch a *SQLite* database and how to insert and retrieve data using *QML* + *JavaScript*. Also was demonstrated how can be used the *onComplete* event and how to save user preferences or custom values with the *Settings* object. The provided sample code and the solutions proposed can be reused as base for a custom application or improve the sample one.

We invite to modify the source code to clarify some possible doubts (this is the best solution to understand and improve your knowledge). Also, take a look at the comments placed inside the source code.

In the next chapter we'll talk about charts. We will see how to create a chart displaying our saved temperature values using *QCharts.js* library.

7.6 References

Here some link to deepen some arguments introduced in this chapter:

- For *SQL* language: https://www.w3schools.com/sql/sql_intro.asp only the syntax for the base statements: insert, update, delete, select, create.
- For *SQLite*: <http://www.sqlitetutorial.net/>
- For *QML* API: <https://docs.ubuntu.com/phone/en/apps/qml/index> the official reference for Ubuntu Touch documentation.

7.7 People who have collaborated

- Fulvio Russo: author
- Miguel Menéndez: revision of the chapter in English. Translation into Spanish.

8.1 Introduction

The purpose of this chapter is show how to draw charts from QML + Javascript applications for Ubuntu Touch (UB-Ports).

8.2 Requirement

To follow this chapter is required an Ubuntu SDK IDE configured with a “Desktop Kit”, a knowledge of QML and Javascript. For more informations about environment configuration you can look at the beginning of this course.

Is also necessary a base knowledge of SQL language and how to access at a SQLite database from QML. If the last requirements are missing you can get them from the dedicated chapter of this course.

8.3 Premise

This chapter uses as sample application the one created in the previous chapter about QML and database access. Just for quick summary, that sample application was titled “WeatherRecorder” and was designed to save and manage the daily temperature values for a favourite city.

Now, in this chapter, we want improve that application adding an analytic feature: display the monthly temperature values with a chart.

8.4 Charts with QML

The presented solution uses the [open source Javascript library Chart.js](#) as drawing library which uses the new HTML5 elements. To use it form QML is necessary a “binding” library placed in the middle between Charts.js and the QML

code. That binding is provided by another open source library named `qchart.js` that define a custom QML Component representing a chart that wrap `Chart.js` object.

Thanks to Jwintz, the author of `qchart.js`, for his great job! But don't worry, we don't need to manage any low level details about that.

8.5 Chart support configuration

If you import our sample application into Ubuntu SDK IDE, the following configuration steps can be skipped. They must be executed if you are creating a new QML application from scratch and you need chart support.

To draw charts from our QML application is necessary include two files (you can find all the necessary files from <https://github.com/jwintz/qchart.js>):

8.5.1 Qchart.js

Is the Javascript drawing library containing the core functions to draw the chart, calculates the axis scales and so on. It is a version of `Chart.js` library already bundled with the “`qchart.js`” one (with maybe with some little patches). To use it is necessary this import statement:

```
import "QChart.js" as Charts
```

Note: the above import syntax is the same necessary to use any other Javascript file from QML.

8.5.2 QChart.qml

Define the new QML component (named “`Qchart`”) to be used in our QML code to draw a chart. The use is made with a block like this:

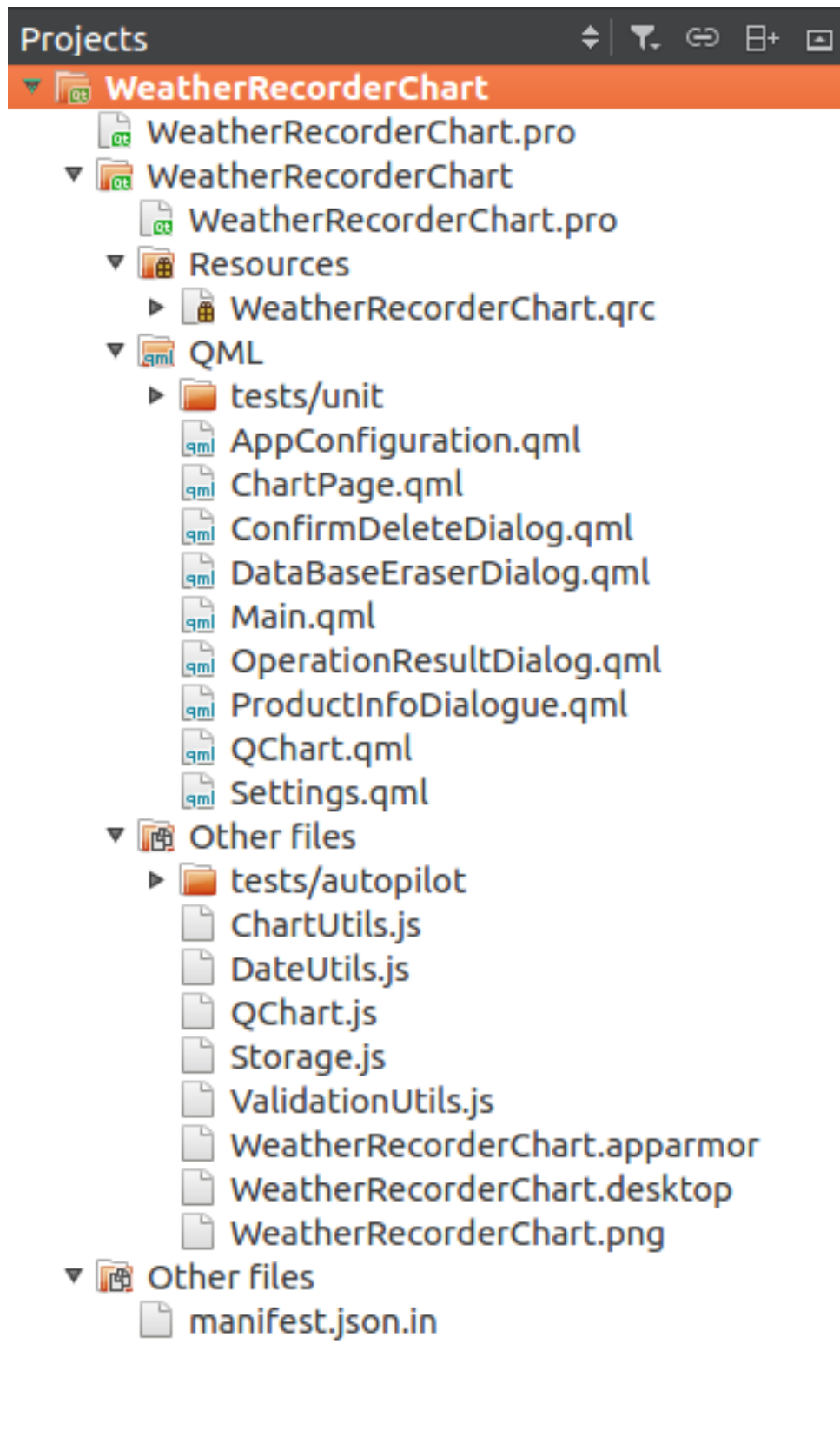
```
QChart {  
    //chart configuration options  
}
```

As last step, is necessary include that files in the `.qrc` file (a special project file that list all the files that composing the project). Now the QML application is ready to draw charts.

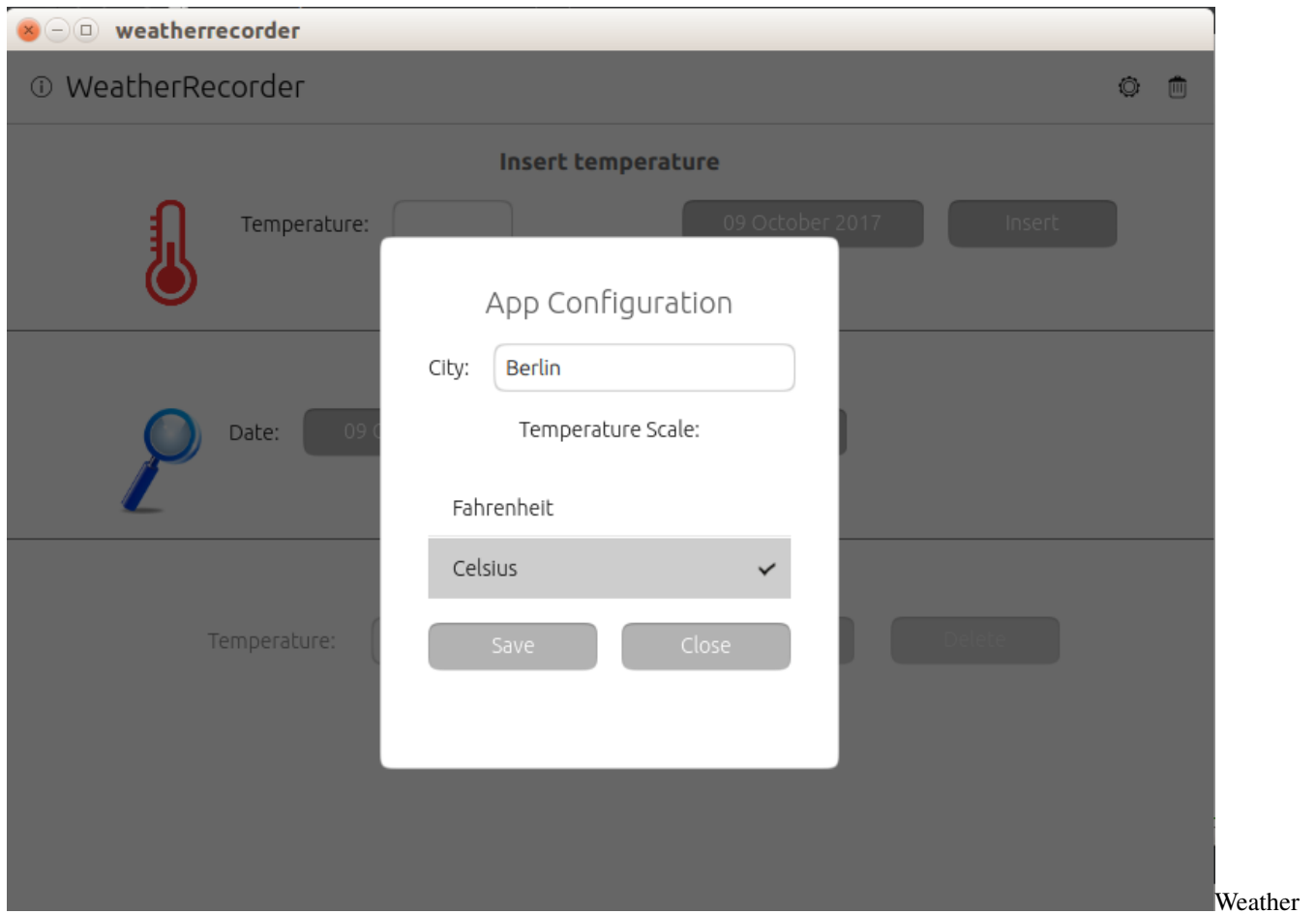
8.6 Introduction at the sample application

As noted above, we will use the application presented in the chapter dedicated at QML and database access and we will improve it. To keep separated the source code of the two applications, and prevent confusion, for this chapter there is a separated application named “`WeatherRecorderChart`” created using the previous one “`WeatherRecorder`” as base.

If you import the “`WeatherRecorderChart`” project into Ubuntu SDK IDE in the project source tree you should get a view like this:

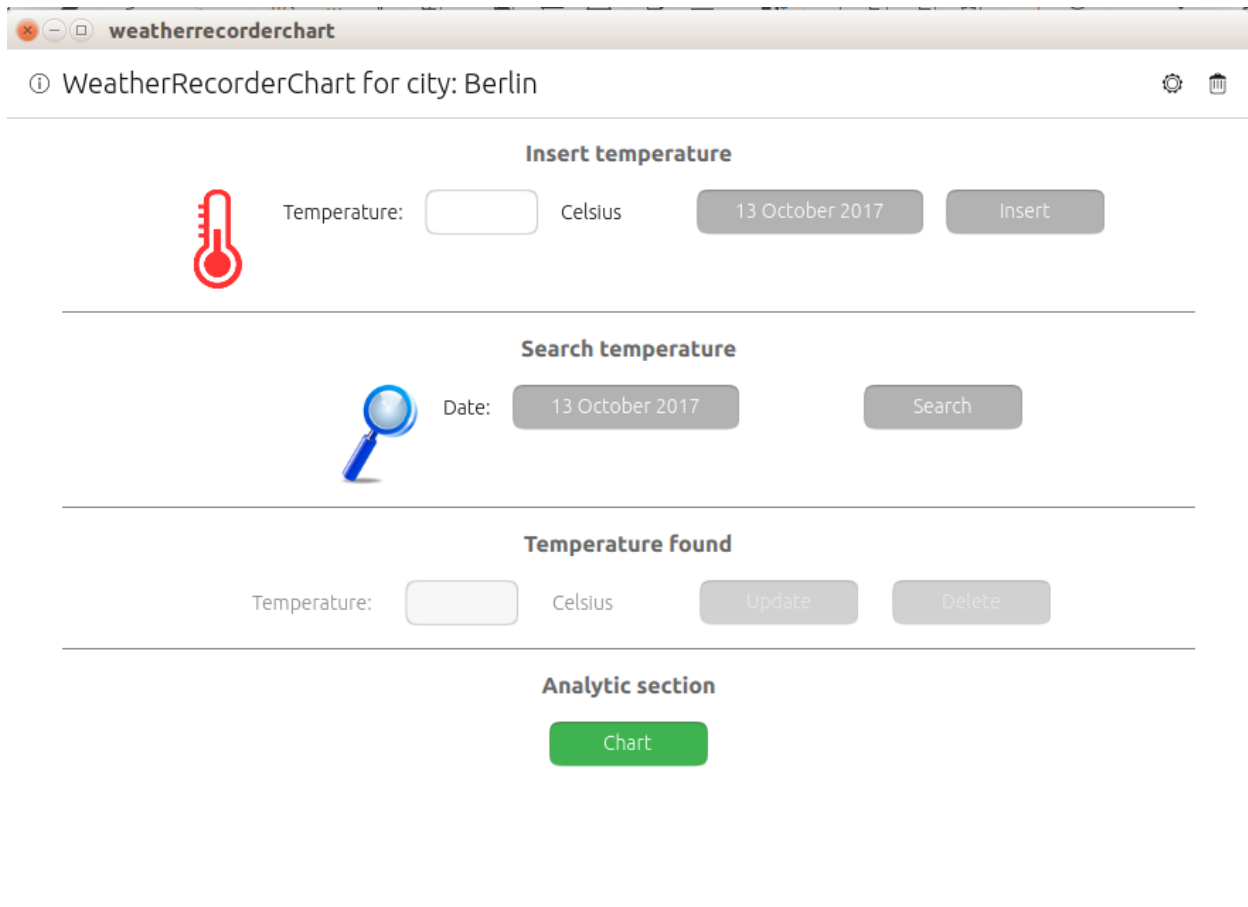


If you run the application for the first time (pressing the green “Play” icon) you get that application page:



Recorder Configuration

When you have accepted (or updated) the proposed configuration you get the new Application home page:



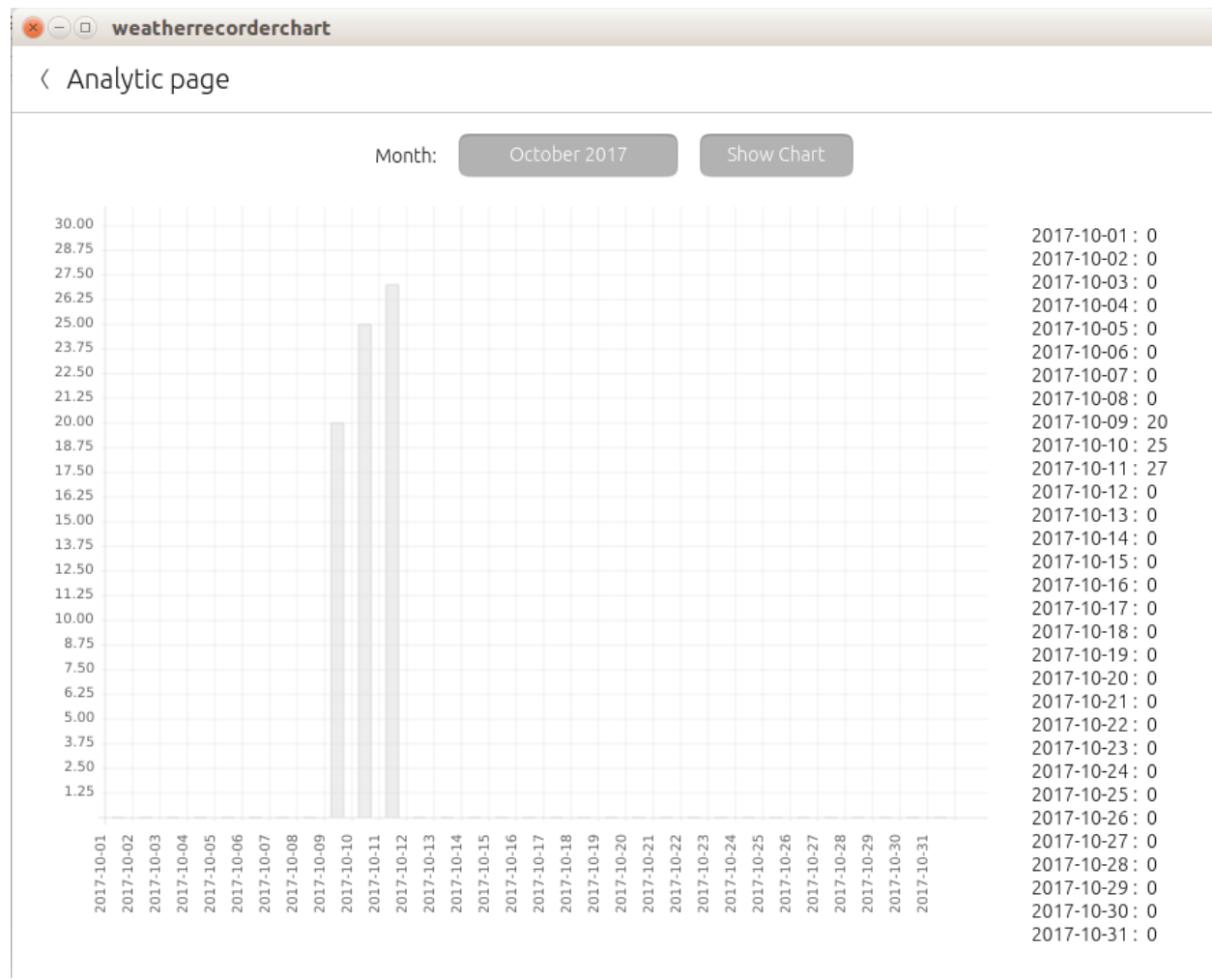
Weather

Recorder new design

Note: next application start-up the configuration windows will not shown (we have described what happen at application start-up in the previous chapter).

Important: the new application (like the old one) uses a SQLite database to save the temperature values. But, the location of the database files is different, due to the fact that “applicationName” field in Main.qml have a different value (please, see chapter about database access for info about database location).

In comparison at the other sample application “WeatherRecorder” we note a new section at the end of the page named “Analytic section”. If you press the green button “Chart” you get a new page that allow you to choose a target month and then display his temperature values with a bar chart like this:



Page

Analytic

On the right there is a simple chart legend that show the values used to draw the chart. If no values are found, the chart will be empty (this is the case when all the temperature values are missing for the chosen month). Insert some values and retry to redraw the chart.

8.7 New application features in details

The previous section have already shown the new chart page added at our application. Here, we want describe in more details the changes added.

In the user interface there is new section, whose source code can be found in `Main.qml` file. This section is composed of two new QML Row components with id “chartSectionHeader” and “chartRow” (try to search them with CTRL + F) appended at the ones already existing.

To have a new separated page for the chart, and his legend, we have introduced a new QML Component: `PageStack`. Very quickly, it is a component used to manage the navigation between pages, that works like a stack: the page on the top is the one showed. To add/remove a page from the stack there are the methods “push” and “pop”.

As first thing, when the full application starts, is necessary push on the `PageStack` the first page to show, this is done using the “onComplete” event of the `PageStack` component raised when it is created (see the previous chapter dedicated a database access with QML for more information about the use of “onComplete” event).

That initialization is made with this statements:

```
// Listing 01
PageStack {
    id: pageStack

    /* set the firts page of the application */
    Component.onCompleted: {
        pageStack.push(mainPage);
    }

    // a set of Page components
}
```

With the above code is pushed on the top of the stack the QML page with id “mainPage”. That identifiers is the one of the home page shown on application start. Try to search that identifiers in the code.

Now, we can look at the entry point of our chart page: the “Chart” button, his source code (form Main.qml) is the following:

```
// Listing 02
Button {
    id:showCahrtButton
    text: i18n.tr("Chart")
    width: units.gu(14)
    color: UbuntuColors.green
    onClicked: {
        pageStack.push(chartPage)
    }
}
```

When the user press the button (i.e.: the “onClick” event is raised) a new page is pushed on the top of our PageStack component and will be shown replacing the previous one. Which is the page shown? Is the one with “chartPage” identifiers. Try a “CTRL + click” on that identifiers passed at the “push” function (or try or search with CTRL + F).

Doing this you will be redirect to the following code:

```
// Listing 03
Component {
    id: chartPage
    ChartPage{}
}
```

That statement, define and import a custom component whose source code is contained in the ChartPage.qml file (try a CTRL+Click on ChartPage{} to open that file).

(Note: listing 03 shows how a QML Component, defined in an external file, can be imported into another QML file). In our case the imported QML Component is the Page whit the id “chartPage”.

In ChartPage.qml we find the page implementation:

```
// Listing 04
Page {
    id: chartPage

    //... other code omitted for shortness
}
```

8.8 The Chart page in details

Now, that the navigation flow is explained, we can focus on the page that create the chart, whose source code is contained in `ChartPage.qml` file.

Note: we omit explanations of the header part of that page because contains components already presented in the previous chapters of the course (`DatePicker`, `Buttons`. etc.).

Our description starts when the user press the “Show Chart” button. Here the code executed when the button “onClick” event is raised:

```
// Listing 05
onClicked: {
    /* extract the year, month, day from the variable 'targetDate' */
    var dateParts = chartPage.targetDate.split("-");
    /* build a JS Date object using string tokens (month is 0-based) */
    var date = new Date(dateParts[0], dateParts[1] - 1, dateParts[2]);
    /* calculates first and last day of the month */
    var firstDayMonth = new Date( date.getFullYear(),date.getMonth(), 1);
    var lastDayMonth = new Date( date.getFullYear(), date.getMonth() + 1, 0);
    /* set the data-set at the chart and make visible the chart and legend */
    temperatureChart.chartData = ChartUtils.getChartData(firstDayMonth,lastDayMonth);

    /* make visible the chart and the legend */
    temperatureChartContainer.visible = true;
}
```

When the user choose a month with the `DatePicker`, that value is assigned at the Javascript variable named “chartPage” as a formatted value `yyyy-mm-dd` (see: the Component with id “`popoverTargetMonthPicker`”).

After a splitting of that value (see “listing 05”, is created a Javascript `Date` object to use his native methods to calculate the first and last days of the chosen month (this calculation must be done at runtime because we don’t know the user choice).

The calculated first and last days are passed as input at the Javascript function “`getChartData`” that create the XY data-set for the chart:

```
// Listing 06
temperatureChart.chartData= ChartUtils.getChartData(firstDayMonth,lastDayMonth);
```

That returned data-set is assigned at the field named “chartData” of the QML Component with id “`temperatureChart`”, which is our chart declared as follow:

```
// Listing 07
/* The monthly temperature chart */
QChart{
    id: temperatureChart;
    width: parent.width
    height: parent.height;
    chartAnimated: false;

    /* for all the options see: QChart.js */
    chartOptions: {"barStrokeWidth": 0};

    /* chartData: set when the user press 'Show Chart' button*/
    chartType: Charts.ChartType.BAR;
}
```

Listing 07 show the declaration of the chart QML Component defined by the Qchart.js library described at the beginning of the chapter. Note how we specify that we want a Bar chart.

8.9 Chart creation in details

What happens at QML components level was described in the previous sections. Now, we want to study the Javascript code that builds the chart data-set.

All the functions used to create our temperature chart, are contained in ChartUtils.js file (imported in our QML file). Note: ChartUtils.js file was created for this application, is NOT owned by any library.

Let's look at that file.

The entry point is the function: `function getChartData(fromDate, toDate)` invoked when the "Show Chart" button is pressed and the first/last days of the month are calculated.

That function performs the following steps:

8.9.1 Step 1

Calculate the amount of days contained in the chosen month (i.e. 28 – 31) using the function "getDifferenceInDays" contained in DateUtils.js file:

```
// Listing 08

/* the amount of days for the target month (e.g.: 30) */
var monthDays = DateUtils.getDifferenceInDays(fromDate,toDate);
```

8.9.2 Step 2

Prepare a key-value associative Javascript Array whose size is the amount of days (this will be the base for our chart data-set). The key part of the Array is a date in the target month, the associated value is the temperature. All the entry values are initialized to 0 (zero). This is necessary to manage the cases when a temperature value is missing (e.g. the user forgot to insert it). The missing temperature values are the ones shown with "N/A" in the user interface.

Note: setting them to "zero" is only a choice, was also possible to set a different default value. This is the code that performs the operations described in 'step 2':

```
// Listing 09
function prepareEmptyDataset(fromDate, monthDays){

    /* Init an empty associative array */
    var xyDataSet = {};
    for(var i=0;i < monthDays+1; i++) {
        /* initialize to zero the temperature value for the date */
        xyDataSet[DateUtils.addDaysAndFormat(fromDate, i)] = 0;
    }
    return xyDataSet;
}
```

8.9.3 Step 3

Extract from the SQLite database the temperature values for the chosen months, then update the key-value Javascript Array prepared in the previous step. (for details about SQLite database access from QML we invite you to read the dedicated chapter previously published).

This activity is done in the following Javascript function: `function updateXYdataset(fromDate, toDate, xyDataSet)` (we omit the full code for shortness)

When the update of the Array is finished, our chart data-set will contains a not zero temperature value only for the days updated by the user. For the others ones remain the default value set during the initialization. We have now the XY data-set for our chart.

8.9.4 Step 4

The last step is extract the single X and Y values to have two distinct sets. This is done with to dedicated Javascript functions:

- `function getXaxisValue(xyDataSet)`: extract the keys of the Javascript array, they are our X values
- `function getYaxisValue(xyDataSet)`: extract the values form the array, they are our Y values.

This split is necessary because the Javascript Object accepted by the chart library require to separated sets (see step 5).

8.9.5 Step 5

Create this Javascript object to be assigned at our QChart QML component (declared in listing 7)

```
// Listing 10
var ChartBarData = {
  labels: < our X axis values extracted at step 4 above>,
  datasets: [{
    fillColor: "rgba(220,220,220,0.5)",
    strokeColor: "rgba(220,220,220,1)",
    data: < our Y axis values extracted at step 4 above>
  }]
}
```

The assignment to QChart component is done with the following statement `temperatureChart.chartData = ChartUtils.getChartData(firstDayMonth, lastDayMonth)`; where “temperatureChart” is the identifier of our QChart Component (listing 7) and “chartData” is his field to be initialized with an object like the one in listing 10.

Note: In other cases, when the chart data-set is already known or doesn’t depends on runtime choices (i.e. the target month in our case), the setting of the “chartData” field (of QChart component) can be done when the QChart component is declared (i.e. in listing 7) instead of after a processing.

8.10 The chart legend

On the right of the chart is drawn a simple legend filled with the X-Y values used to draw the chart. It is not mandatory, we insert it to completeness and to give at the reader new ideas about possible future customizations.

That legend is created in the **ChartPage.qml** using a ListModel QML Component that act as a container for the values to display. That model is filled whit the following function (see ChartUtils.js):

```
// Listing 11

/* fill the Listmodel used to create the Chart legend */
function getChartLegendData(xyDataSet) {
    customRangeChartListModel.clear();
    for(var key in xyDataSet) {
        customRangeChartListModel.append({"date": key, "temp":xyDataSet[key]});
    }
}
```

It perform an iteration on the full XY data-set and extract the values to place (as Javascript Object key-value) in the ListModel. We invite to read the official documentation for more information about ListModel. We choose to not give more details because that arguments are out of the Chapter purpose.

To show the values contained in the ListModel is used an [UbuntuListView Component](#). The implementation and the configuration of our UbuntuListView can be found in ChartPage.qml file. Here a code snapshot with some omission for shortness:

```
// Listing 12
/* ListView that display the values in the legend */
UbuntuListView {
    id:chartLegendListView
    anchors.fill: parent

    /* disable the dragging of the model list elements */
    boundsBehavior: Flickable.StopAtBounds
    model: /* id of the ListModel containing the data */

    delegate:
        /* 'delegate' is the component that define the layout
        used to display the value from the ListModel */
        Component {
            id: customReportChartLegend
            Rectangle {
                id: wrapper
                height: legendEntry.height
                border.color: UbuntuColors.graphite
                border.width:units.gu(0.5)

                Label {
                    id: legendEntry
                    /* 'key' and 'temp' are the key used to
                    store date and temperature values in the
                    ListModel */
                    text: date+" : "+temp
                    fontSize: Label.XSmall
                }
            }
        }
}
```

Only a couple of tips about how UbuntuListView works: it receive in input (with the field named “model”) the ListModel containing the values to display and a QML component that define the layout used to show them in the page (field “delegate”). The “delegate” component access at the values in the ListModel using the key values used to store them (i.e. “date” and “temp” looking at listing 11).

8.11 Conclusions

We have shown how to draw charts from an QML + Javascript application using values from a SQLite database. To reach that we have made some choices, like to use a Bar Chart or use “Zero” as default value in case of missing value. Maybe, our choices and some details can be disputable but our focus was provide at the reader a knowledge about drawing charts with QML and suggest new ideas for future applications.

As exercise, the reader can try the use of different chart type (e.g. a line bar) or add another chart about pressure values for the city...

We invite the reader to look at the application source code (and his comments) for more details omitted for shortness in this chapter.

8.12 References

Here some useful links about the arguments explained in the chapter:

- [Ubuntu touch QML API](#)
- [The repository of Qchart.js library](#)

8.13 People who have collaborated

- Fulvio Russo: author
- Miguel Menéndez: revision of the chapter in English. Translation into Spanish.

Compilation of documentation

The Qt programming course has evolved over time. Initially it started as a course focused on Ubuntu Touch and little by little it has been opened to desktop applications. To make it more affordable, among other reasons, I will modify some things of the course.

The changes affect both the tool that uses the course, which will become Sphinx and the internal organization of the course. In the current chapters it is assumed that the user uses the programming environment of the Ubuntu SDK. This SDK is no longer available and there are more modern alternatives that solve many of the current problems. Initially I will take the desktop as a base environment, and I will make the appropriate comments for the parts that are exclusive to Ubuntu Touch.

Although it may seem so, the changes are not so radical. The code and logic that is followed in the different chapters is practically the same. Only some elements related to Qt will change.

Although I will be working on the course throughout the month, I will upload the documentation on the last Friday of each month. At that time I will compile the documentation and upload it to the InnerZaurus website.

9.1 Installation

The instructions apply to distributions derived from Ubuntu. The packages that have to be installed are:

- build-essential
- Sphinx and Python 3 files

Commands:

```
sudo apt-get install build-essential
sudo apt-get install python3-sphinx sphinxsearch python3-pip
```

Extensions for Sphinx:

```
pip3 install sphinx-intl
pip3 install --upgrade recommonmark
pip3 install sphinx_rtd_theme
```

If you want run de compilation (English):

```
make html
```

To compile a translation (Spanish):

```
make gettext  
sphinx-intl update -p build/gettext -l es  
make -e SPHINXOPTS="-D language='es'" html
```

10.1 Introduction

In the previous chapter we have seen an introduction to the programming course with Ubuntu Touch. The next step is to prepare the development environment. When programming in any language a SDK (Software Development Kit) is used. The SDK consists of a set of tools that process the source code and generate the executable for the platform that is being used. Ubuntu Touch is not an exception and also has its own SDK.

Applications in GNU / Linux are in repositories. The repository contains a set of applications that can be easily installed. On some occasions there are applications that are not in the official repositories. For these cases, it is possible to use personal repositories, also known as PPA repositories. The Ubuntu SDK is in a PPA repository that will have to be added to the system in order to be installed. By this limitation, it can only be programmed on distributions that can work with PPA repositories, that is to say, all the distributions that use Ubuntu as a base. If the distribution uses RPM packages, as in the case of OpenSuse or Fedora, alternative measures will have to be taken.

There are two other ways to do this: create a Live USB with Ubuntu or use a virtual machine that has Ubuntu installed. The first one is simpler and works reasonably well if the USB memory is fast. The second way is more comfortable since it does not depend on a USB stick, but requires a more powerful computer to run the virtualized operating system. I will not go into how to create a Live USB or virtual machine. Anyone in one of these two cases can ask in the mailing list and I will guide him with the most important steps.

My development environment is:

- Ubuntu 16.04 LTS.
- Aquaris E4.5 with OTA-14.
- Aquaris E5 HD with OTA-14.
- Aquaris M10 FHD with OTA-14.

It is possible that in other distributions there are some intermediate steps that need to be done to configure the SDK. If that is the case you can indicate it.

10.2 Installing the Ubuntu Touch SDK

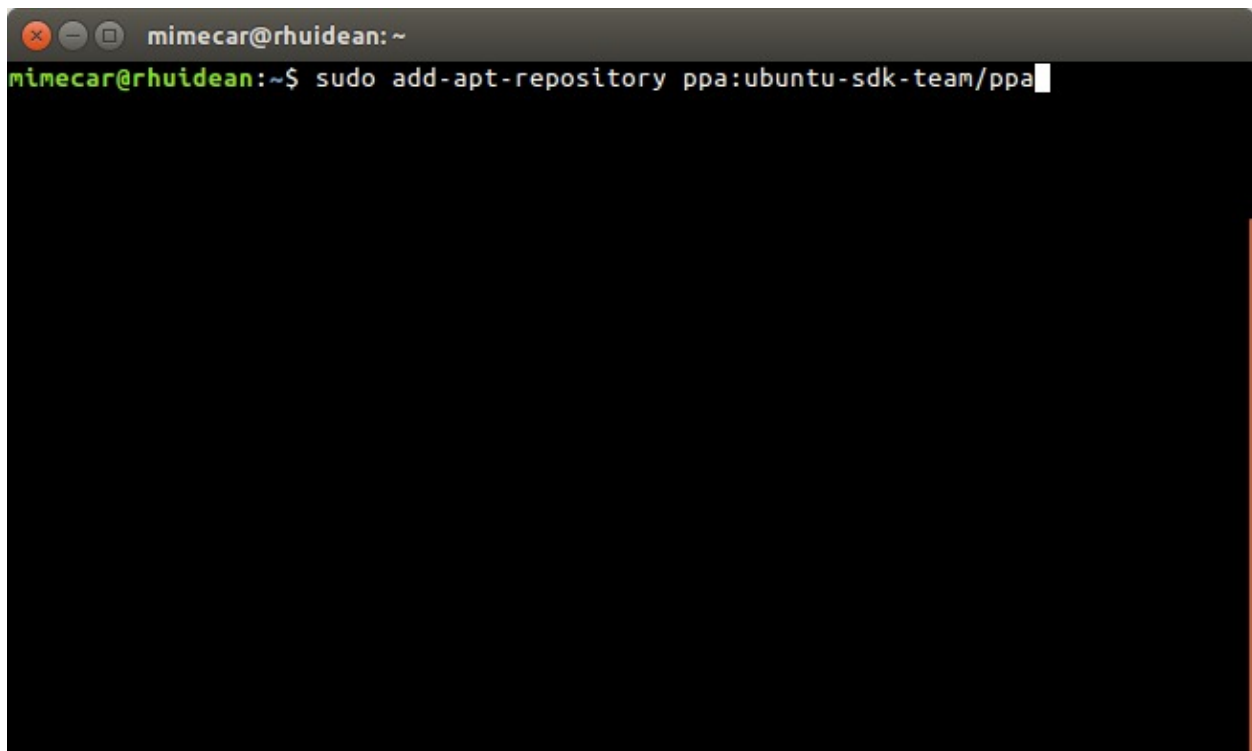
Installing the SDK is very simple and should not take more than a couple of minutes. It is advisable to have the system updated before you start. For the case of Ubuntu, this can be done with the commands:

```
sudo apt-get update && sudo apt-get upgrade && sudo apt-get dist-upgrade
```

You use `sudo` because package installation requires administrator permissions. The first command updates the list of repositories. If there is no error, the second command is executed, which is responsible for updating the applications. Finally the third command updates some operating system packages that are not updated by default.

Add the Ubuntu Touch SDK repository with the command:

```
sudo add-apt-repository ppa:ubuntu-sdk-team/ppa
```



Add

Ubuntu SDK PPA

The repository information will be displayed. You can continue by pressing the ENTER key.

```

mimecar@rhuidean:~
Ubuntu 14.04 LTS had Qt 5.x, Ubuntu UI Toolkit and SDK updates during its main development cycle. This PPA offers post-release SDK updates. The emulator included (or a mobile device attached to the computer) makes it possible to test apps in the very latest Ubuntu phone/tablet environment.

Ubuntu 15.04 and 15.10 have reached end of its support, please upgrade to 16.04 LTS as soon as possible.

== Earlier Ubuntu versions ==

Old Ubuntu 12.04 LTS Qt 5.0 packages have been moved to https://launchpad.net/~canonical-qt5-edgers/+archive/ubuntu/ubuntu1204-qt5/

GENERAL NOTES
-----

1. Qt4 and Qt5 developer tools are co-installable thanks to the qtchooser tool. See 'man qtchooser' for more information.

2. The packaging is mostly done at Debian git (http://anonscm.debian.org/gitweb/pkg-kde/qt)).
   Más información: https://launchpad.net/~ubuntu-sdk-team/+archive/ubuntu/ppa
Pulse [Intro] para continuar o ctrl-c para cancelar

```

PPA

Info

All repositories have a signature that assures us that the installed packages come from the repository. This signature is added to the system and will be used when repository packages are installed.

```

mimecar@rhuidean:~
Old Ubuntu 12.04 LTS Qt 5.0 packages have been moved to https://launchpad.net/~canonical-qt5-edgers/+archive/ubuntu/ubuntu1204-qt5/

GENERAL NOTES
-----

1. Qt4 and Qt5 developer tools are co-installable thanks to the qtchooser tool. See 'man qtchooser' for more information.

2. The packaging is mostly done at Debian git (http://anonscm.debian.org/gitweb/pkg-kde/qt)).
   Más información: https://launchpad.net/~ubuntu-sdk-team/+archive/ubuntu/ppa
Pulse [Intro] para continuar o ctrl-c para cancelar

gpg: anillo «/tmp/tmpugjb6_1m/secring.gpg» creado
gpg: anillo «/tmp/tmpugjb6_1m/pubring.gpg» creado
gpg: solicitando clave C7122F9B de hkp servidor keyserver.ubuntu.com
gpg: /tmp/tmpugjb6_1m/trustdb.gpg: se ha creado base de datos de confianza
gpg: clave C7122F9B: clave pública "Launchpad PPA for Ubuntu SDK team" importada
gpg: Cantidad total procesada: 1
gpg: importadas: 1 (RSA: 1)
OK
mimecar@rhuidean:~$

```

PPA

Key

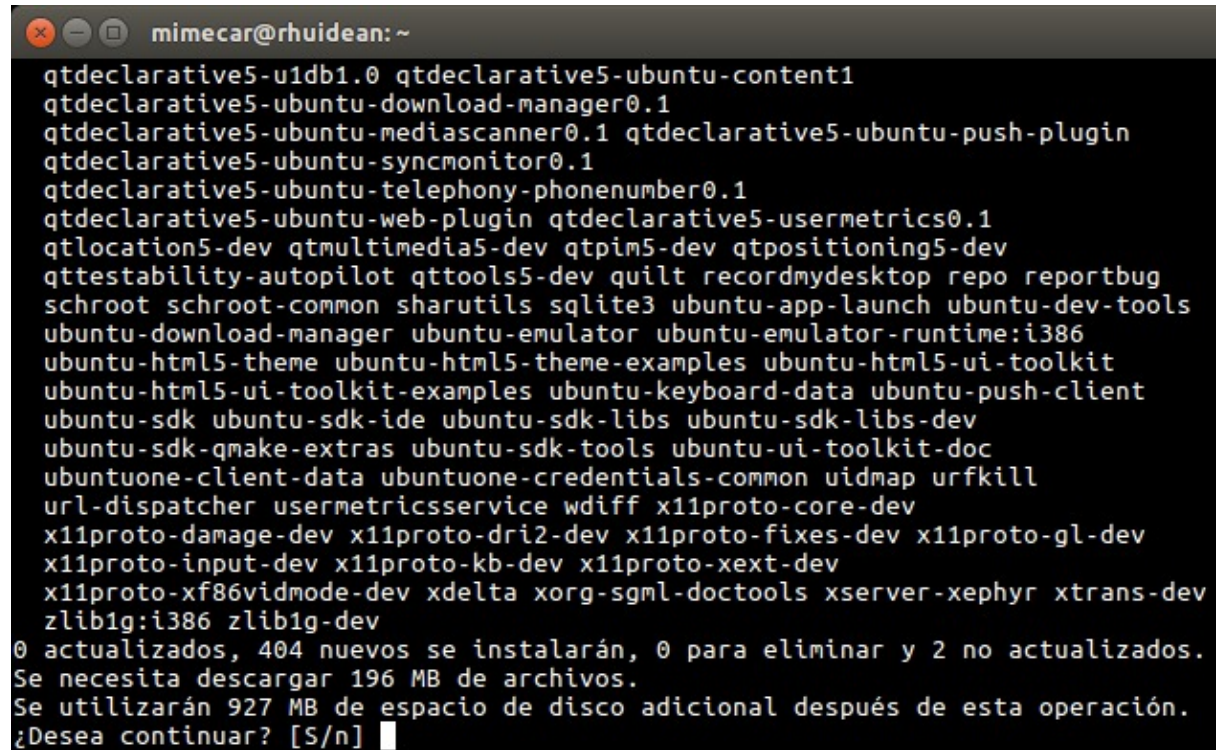
After adding a repository it is necessary to update the information of the packages that it contains. You can do this

with the command:

```
sudo apt-get update
```

Everything is ready. The last step of this section is to install the Ubuntu Touch SDK.

```
sudo apt-get install ubuntu-sdk
```

A terminal window titled 'mimecar@rhuidean: ~' showing the output of the command 'sudo apt-get install ubuntu-sdk'. The output lists numerous packages to be installed, including qtdeclarative5-ubuntub1.0, qtdeclarative5-ubuntu-content1, qtdeclarative5-ubuntu-download-manager0.1, qtdeclarative5-ubuntu-mediascanner0.1, qtdeclarative5-ubuntu-push-plugin, qtdeclarative5-ubuntu-synmonitor0.1, qtdeclarative5-ubuntu-telephony-phonenum0.1, qtdeclarative5-ubuntu-web-plugin, qtdeclarative5-usermetrics0.1, qtlocation5-dev, qtmultimedia5-dev, qtpim5-dev, qtpositioning5-dev, qttestability-autopilot, qttools5-dev, quilt, recordmydesktop, repo, reportbug, schroot, schroot-common, sharutils, sqlite3, ubuntu-app-launch, ubuntu-dev-tools, ubuntu-download-manager, ubuntu-emulator, ubuntu-emulator-runtime:i386, ubuntu-html5-theme, ubuntu-html5-theme-examples, ubuntu-html5-ui-toolkit, ubuntu-html5-ui-toolkit-examples, ubuntu-keyboard-data, ubuntu-push-client, ubuntu-sdk, ubuntu-sdk-ide, ubuntu-sdk-libs, ubuntu-sdk-libs-dev, ubuntu-sdk-qmake-extras, ubuntu-sdk-tools, ubuntu-ui-toolkit-doc, ubuntuone-client-data, ubuntuone-credentials-common, uidmap, urfkill, url-dispatcher, usermetricsservice, wdiff, x11proto-core-dev, x11proto-damage-dev, x11proto-dri2-dev, x11proto-fixes-dev, x11proto-gl-dev, x11proto-input-dev, x11proto-kb-dev, x11proto-xext-dev, x11proto-xf86vidmode-dev, xdelta, xorg-sgml-doctools, xserver-xephyr, xtrans-dev, zlib1g:i386, and zlib1g-dev. It also shows the number of packages to be updated, new, removed, and not updated, the size of the files to be downloaded, and the additional disk space required. The prompt '¿Desea continuar? [S/n]' is visible at the bottom.

```
mimecar@rhuidean: ~
qtdeclarative5-ub1db1.0 qtdeclarative5-ubuntu-content1
qtdeclarative5-ubuntu-download-manager0.1
qtdeclarative5-ubuntu-mediascanner0.1 qtdeclarative5-ubuntu-push-plugin
qtdeclarative5-ubuntu-synmonitor0.1
qtdeclarative5-ubuntu-telephony-phonenum0.1
qtdeclarative5-ubuntu-web-plugin qtdeclarative5-usermetrics0.1
qtlocation5-dev qtmultimedia5-dev qtpim5-dev qtpositioning5-dev
qttestability-autopilot qttools5-dev quilt recordmydesktop repo reportbug
schroot schroot-common sharutils sqlite3 ubuntu-app-launch ubuntu-dev-tools
ubuntu-download-manager ubuntu-emulator ubuntu-emulator-runtime:i386
ubuntu-html5-theme ubuntu-html5-theme-examples ubuntu-html5-ui-toolkit
ubuntu-html5-ui-toolkit-examples ubuntu-keyboard-data ubuntu-push-client
ubuntu-sdk ubuntu-sdk-ide ubuntu-sdk-libs ubuntu-sdk-libs-dev
ubuntu-sdk-qmake-extras ubuntu-sdk-tools ubuntu-ui-toolkit-doc
ubuntuone-client-data ubuntuone-credentials-common uidmap urfkill
url-dispatcher usermetricsservice wdiff x11proto-core-dev
x11proto-damage-dev x11proto-dri2-dev x11proto-fixes-dev x11proto-gl-dev
x11proto-input-dev x11proto-kb-dev x11proto-xext-dev
x11proto-xf86vidmode-dev xdelta xorg-sgml-doctools xserver-xephyr xtrans-dev
zlib1g:i386 zlib1g-dev
0 actualizados, 404 nuevos se instalarán, 0 para eliminar y 2 no actualizados.
Se necesita descargar 196 MB de archivos.
Se utilizarán 927 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]
```

Ubuntu

SDK Packages

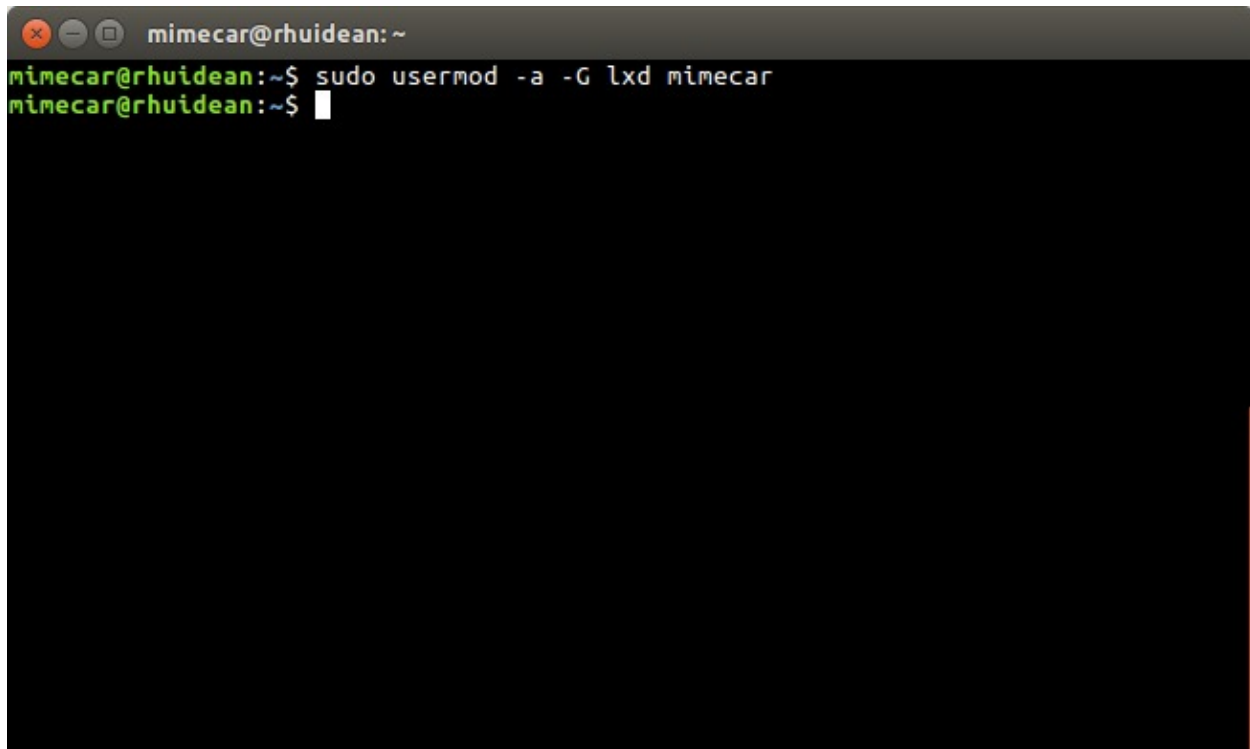
Although only one package (ubuntu-sdk) is put in the command, all the dependencies of that package are automatically installed, so it works without problems. The installation may take a while depending on the Internet connection. It's a good time to let the computer work and have a good coffee.

10.3 Setting up the Environment

The Ubuntu Touch SDK includes tools to generate applications and an editor to make programming easier. The tools take the source code of the application and process it. If the target is a computer there is no problem. Instead, if the target is a device that uses Ubuntu Touch, we are in a different scenario. These devices use the ARM architecture, which is different from the one used by the PC. To generate the executable you have to use a cross-compiler that runs in a container (LXD in the latest versions of the SDK).

To work with LXD you have to add the user that is used on the computer to the group lxd.

```
sudo usermod -a -G lxd user
```



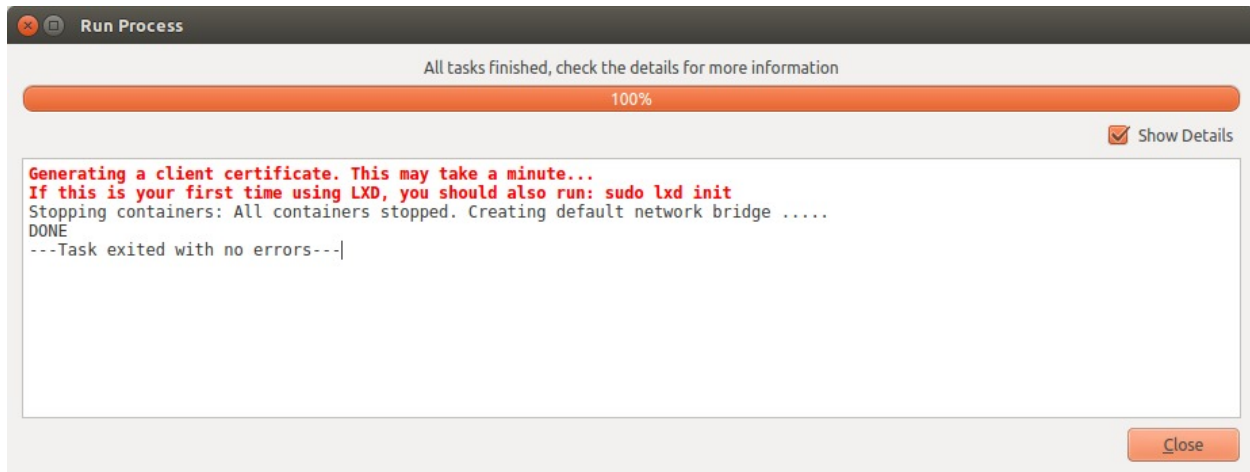
```
mimecar@rhuidean:~$ sudo usermod -a -G lxd mimecar
mimecar@rhuidean:~$
```

Usermod

After writing the command you have to close the user's session and login again. You can start working with this small modification. To start, we need to run the ubuntu-sdk-ide application, it can be launched from the desktop start menu or by pressing ALT + F2 and typing its name. The first time it's launched, a screen appears asking in which lxd has to be configured. Just press 'Yes' to generate the default configuration.



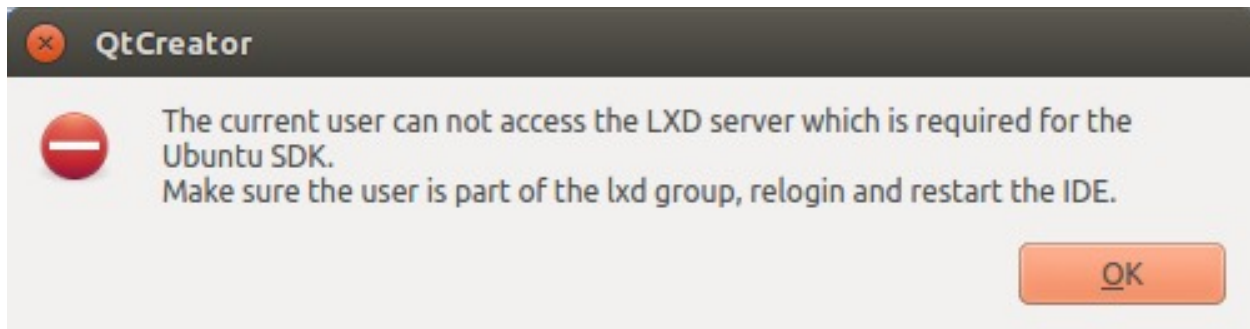
Container Initialization



Container

Generation

If the user is not in lxd group, the following error shown in the screen will be output. In that case, it is sufficient to logout and login again. When launching the IDE of Ubuntu Touch should give you no errors.

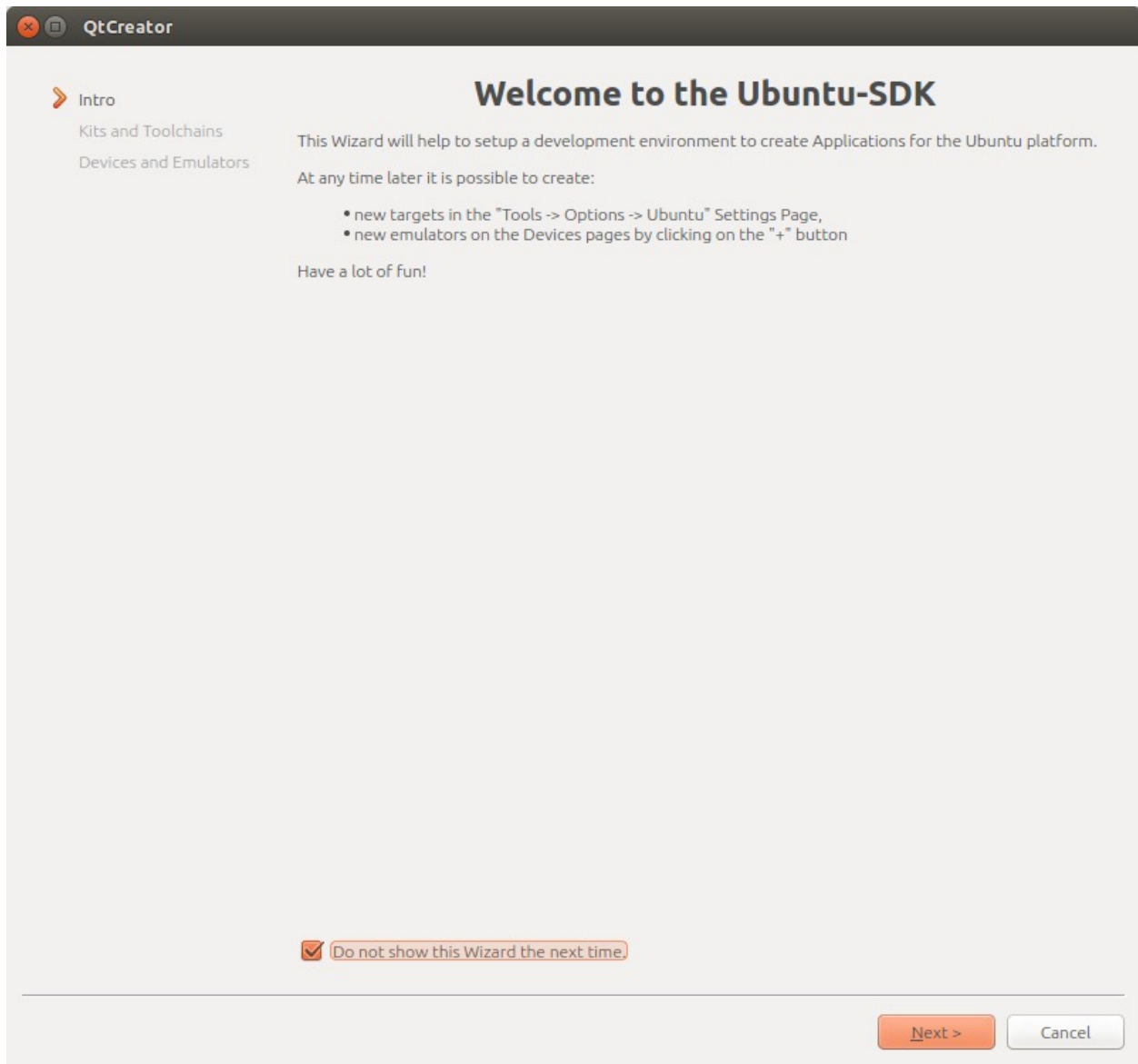


LXD

Error

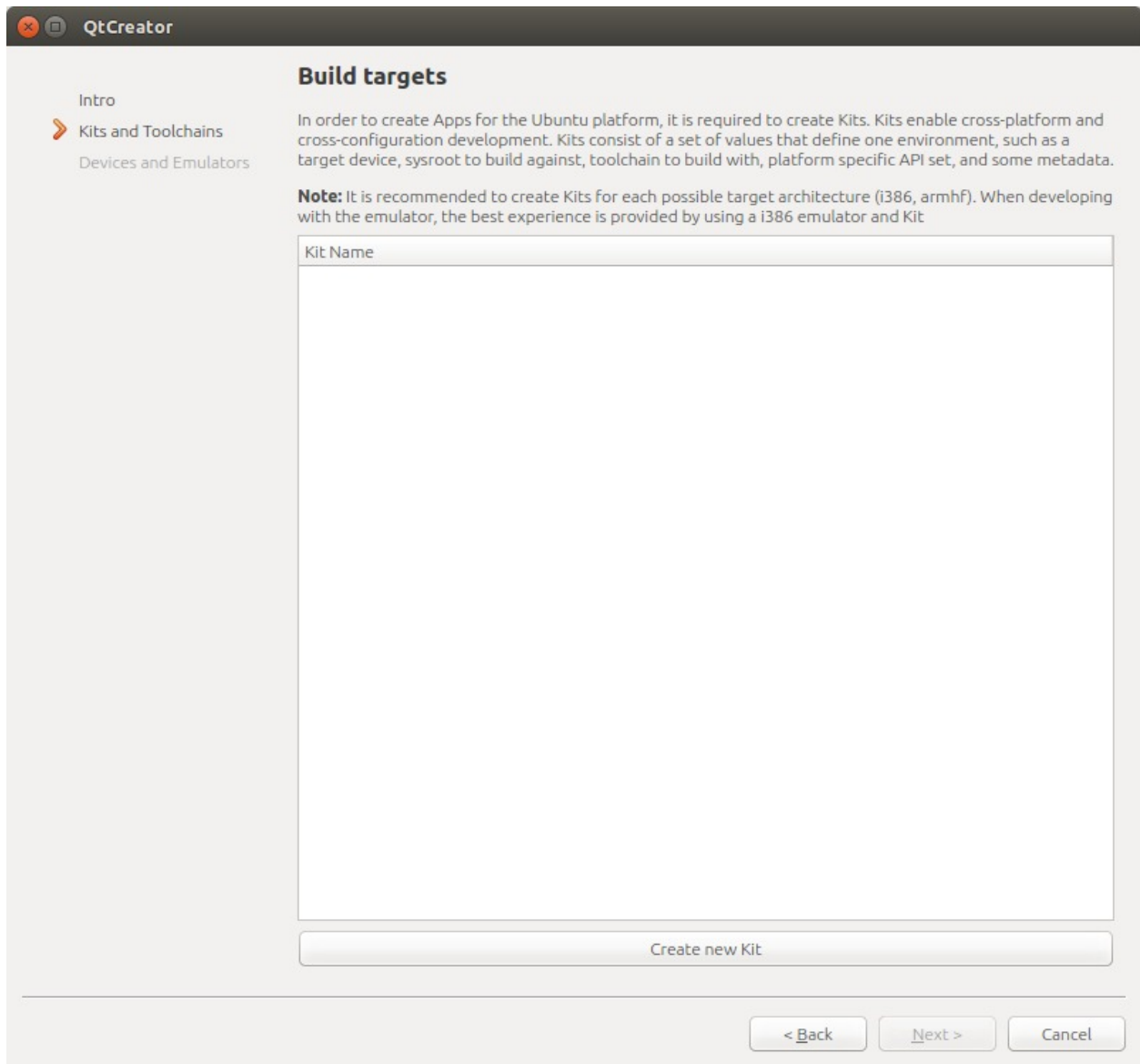
10.4 Configuration Wizard

The first screen of the wizard is an introduction to Qt Creator. To continue, click the 'Next' button.



Intro

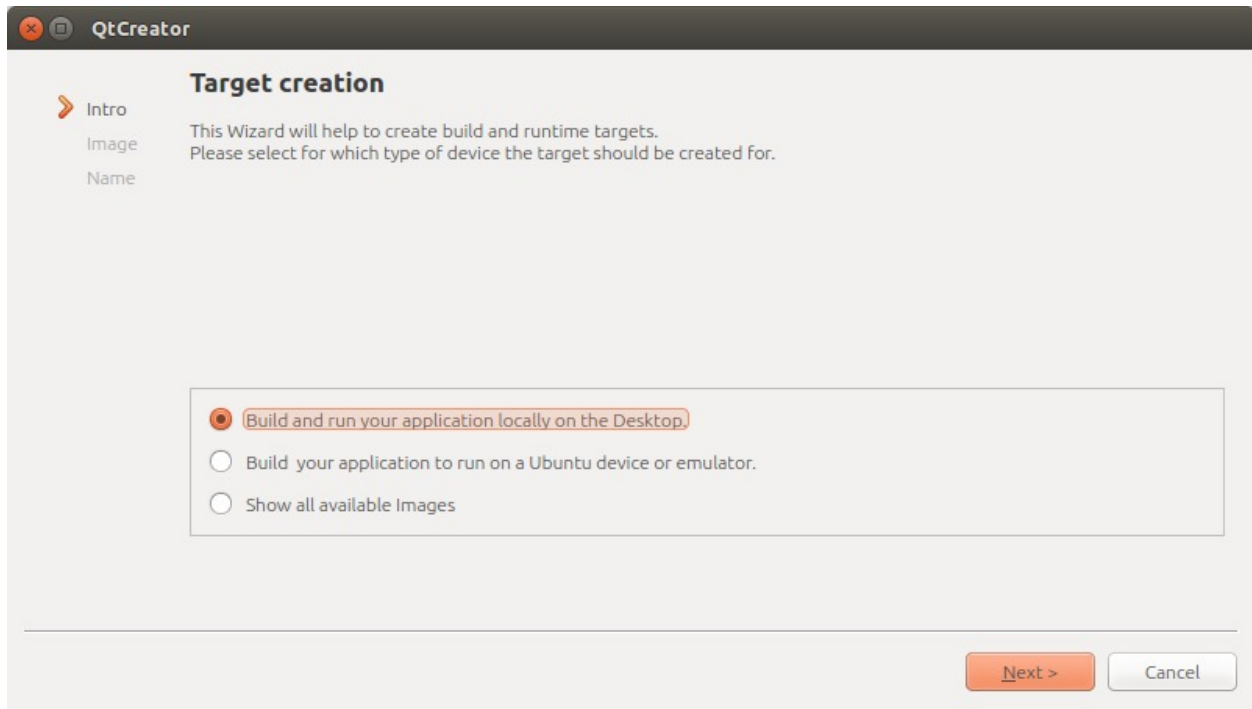
Although the course is focused on mobile devices such as phones or tablets, you will also apply the knowledge learn to create desktop applications. Each of these devices has associated a kit, a set of tools, which are based on the code that is programmed and generates the necessary files for each device.



Kits

Creation

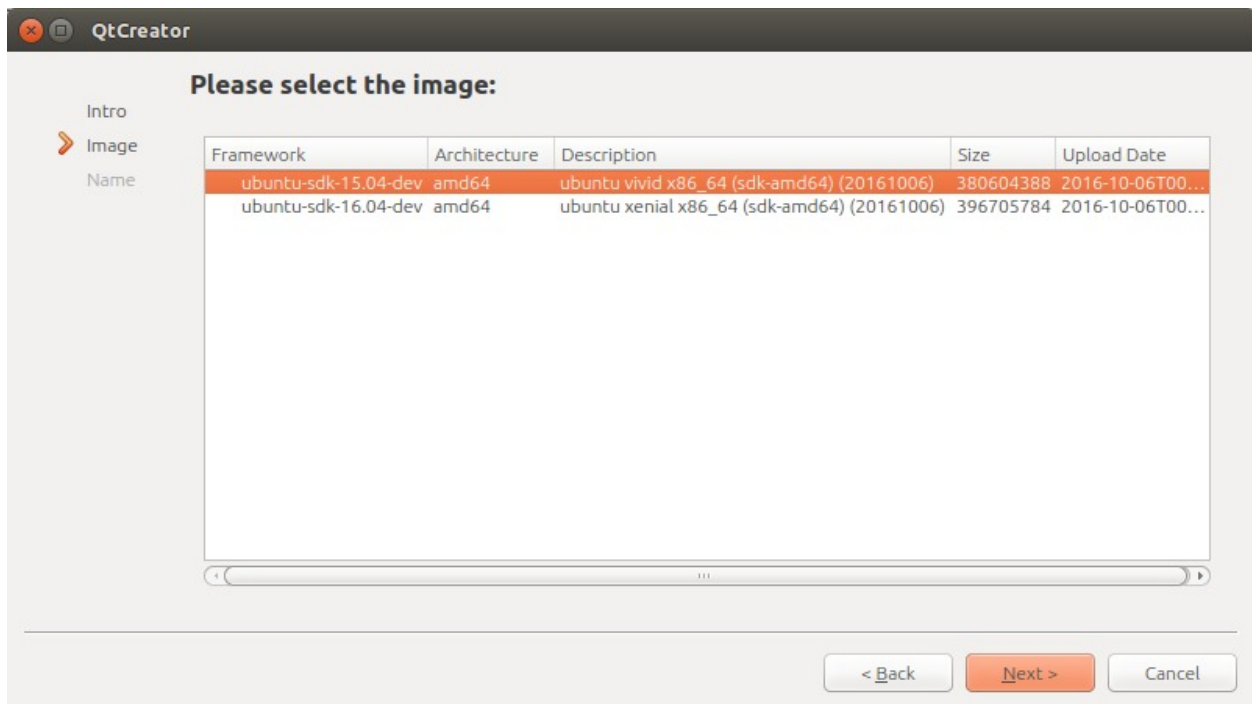
Click the 'Create new Kit button'. The first option must be selected.



Kit

Desktop

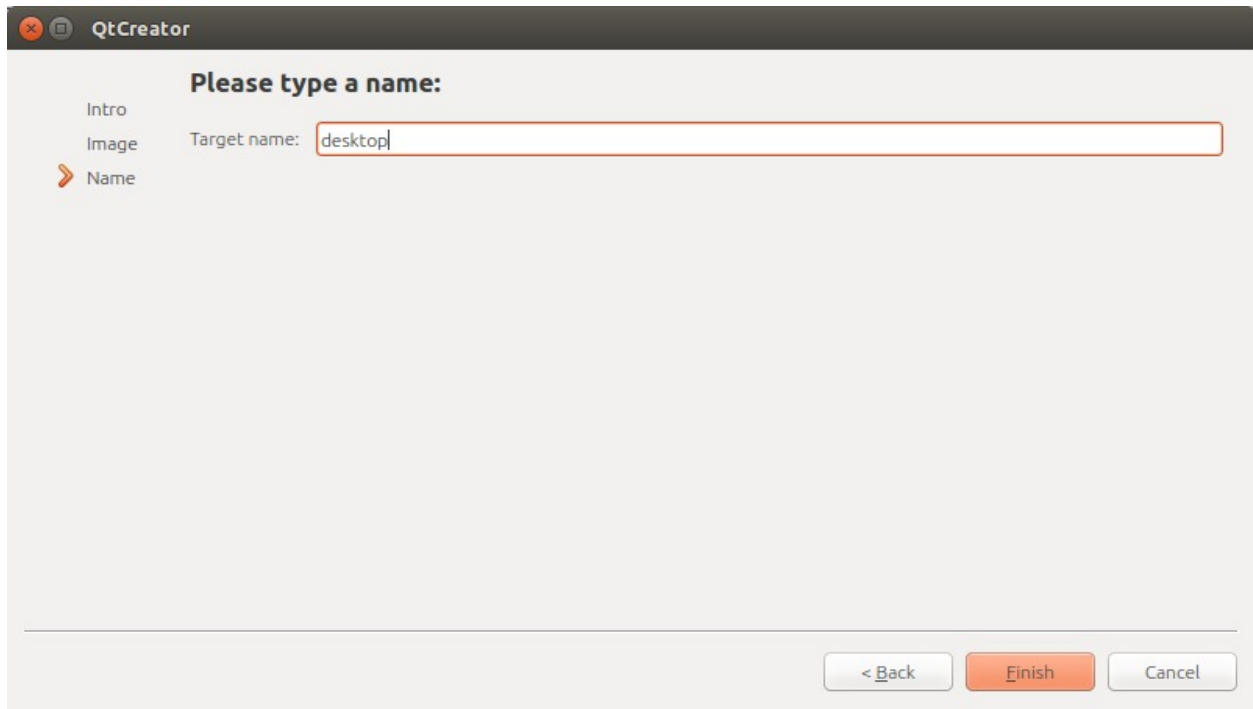
In the list there are several kits to download. Ubuntu Touch is currently based on Vivid so you have to select that option.



Kits

Versions for Desktop

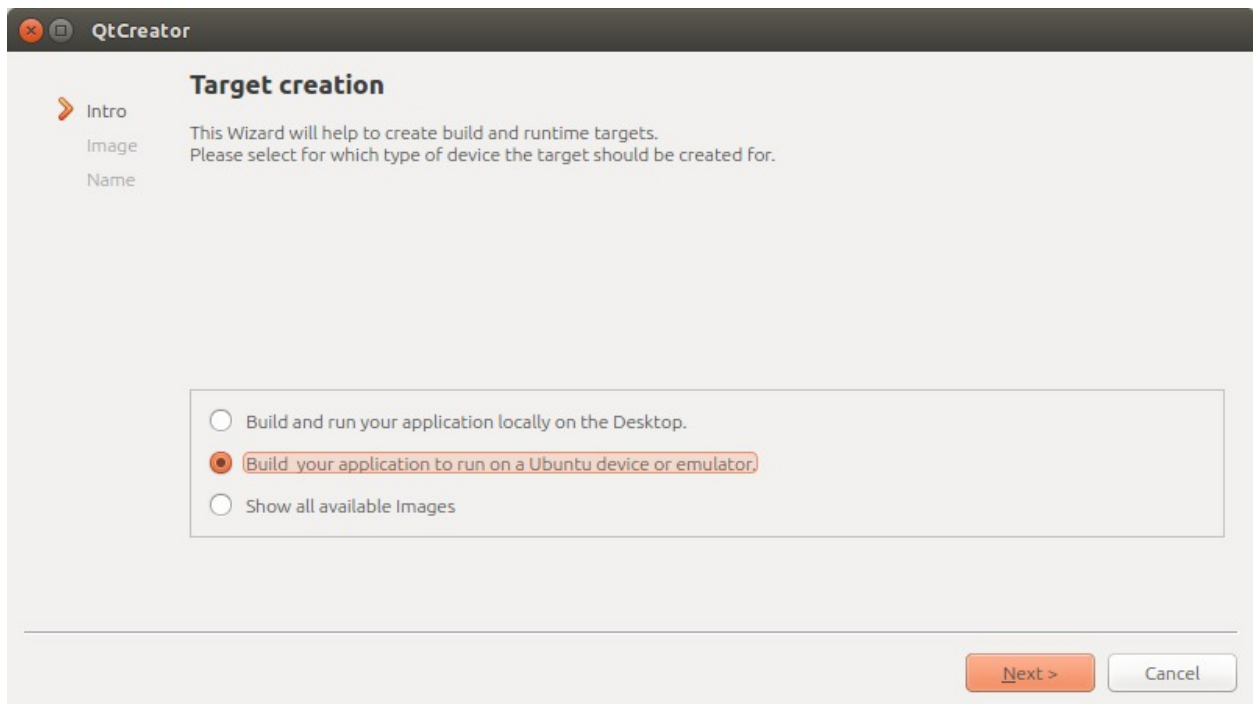
Finally write the name of the kit.



Kit

Name for Desktop

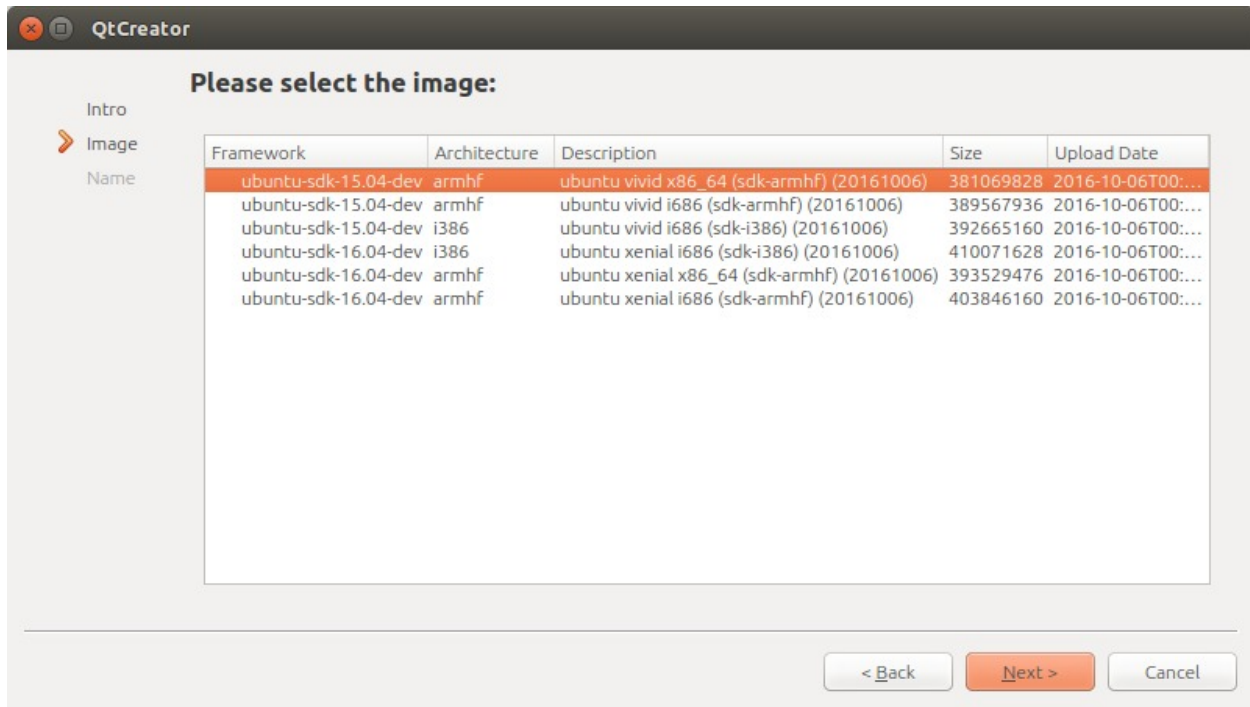
When you complete the last step, you return to the initial screen of the kits. Press the 'Create new Kit' button again and repeat the process by selecting Ubuntu Device.



Kit

Device

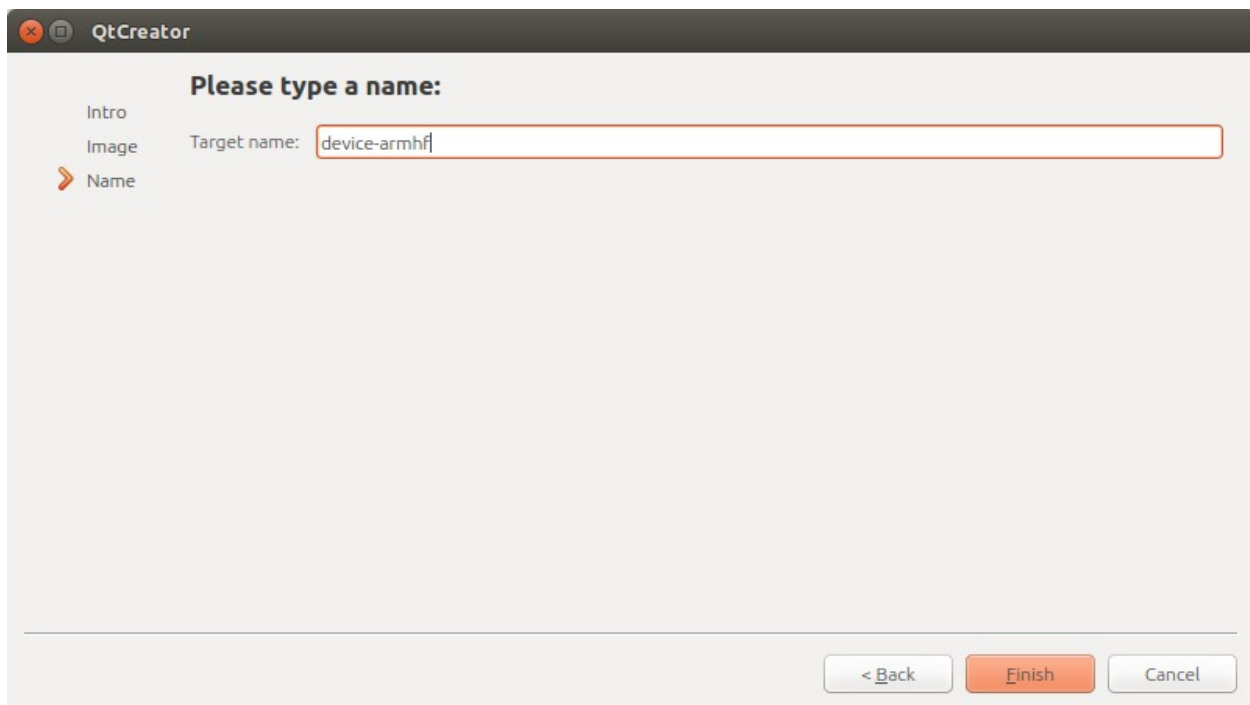
With this kit there are more options to choose from. Phones or tablets that use Ubuntu Touch work with the armhf architecture. In the list you have to select a kit that has that architecture. The version of Ubuntu Touch is Vivid, as in the previous case. Finally you have to choose 686 or x64 depending on whether the computer uses 32 or 64 bits.



Kit

Versions for Device

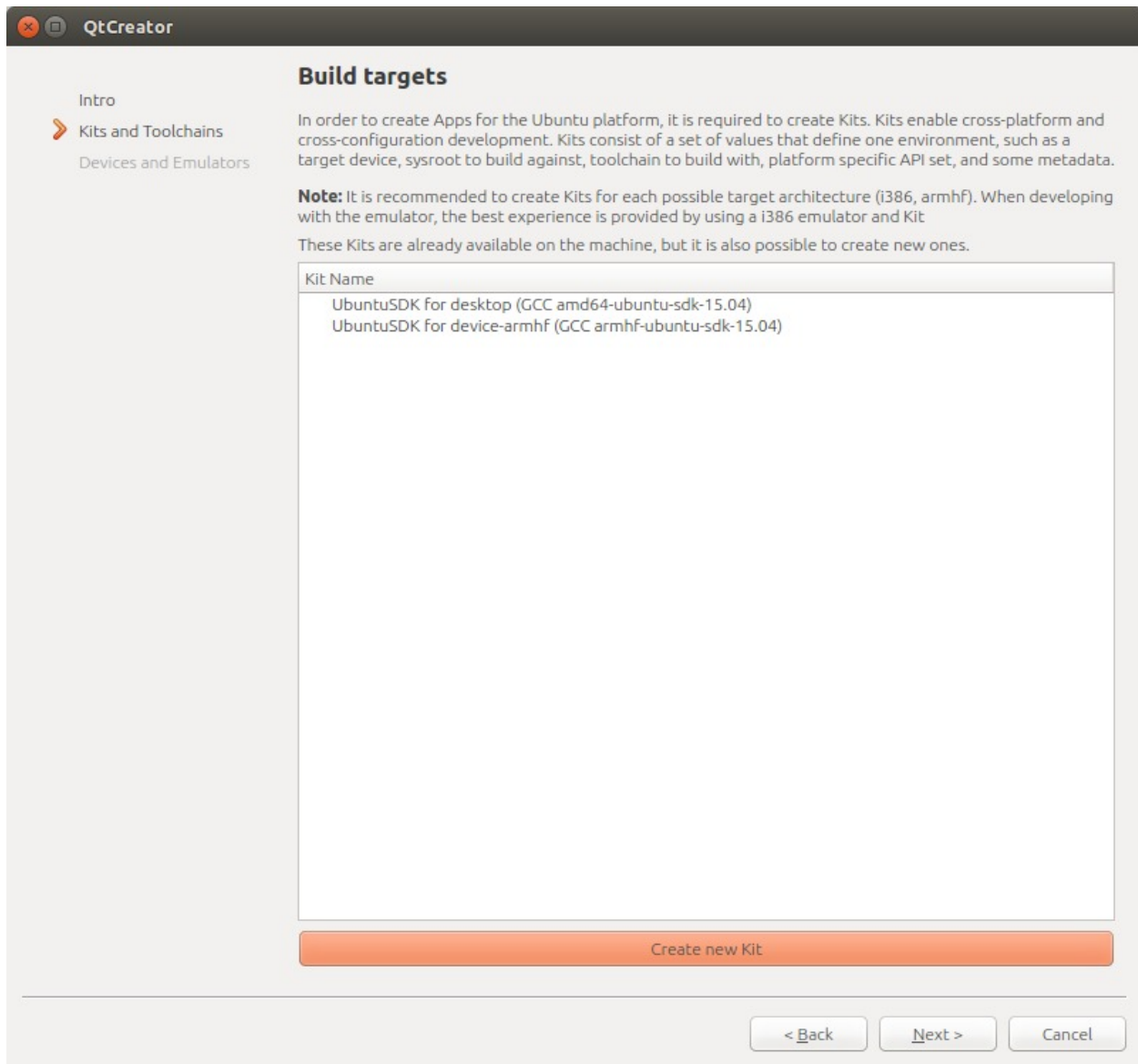
After pressing the 'Next' button, you must write the Kit name.



Kit

Name for Device

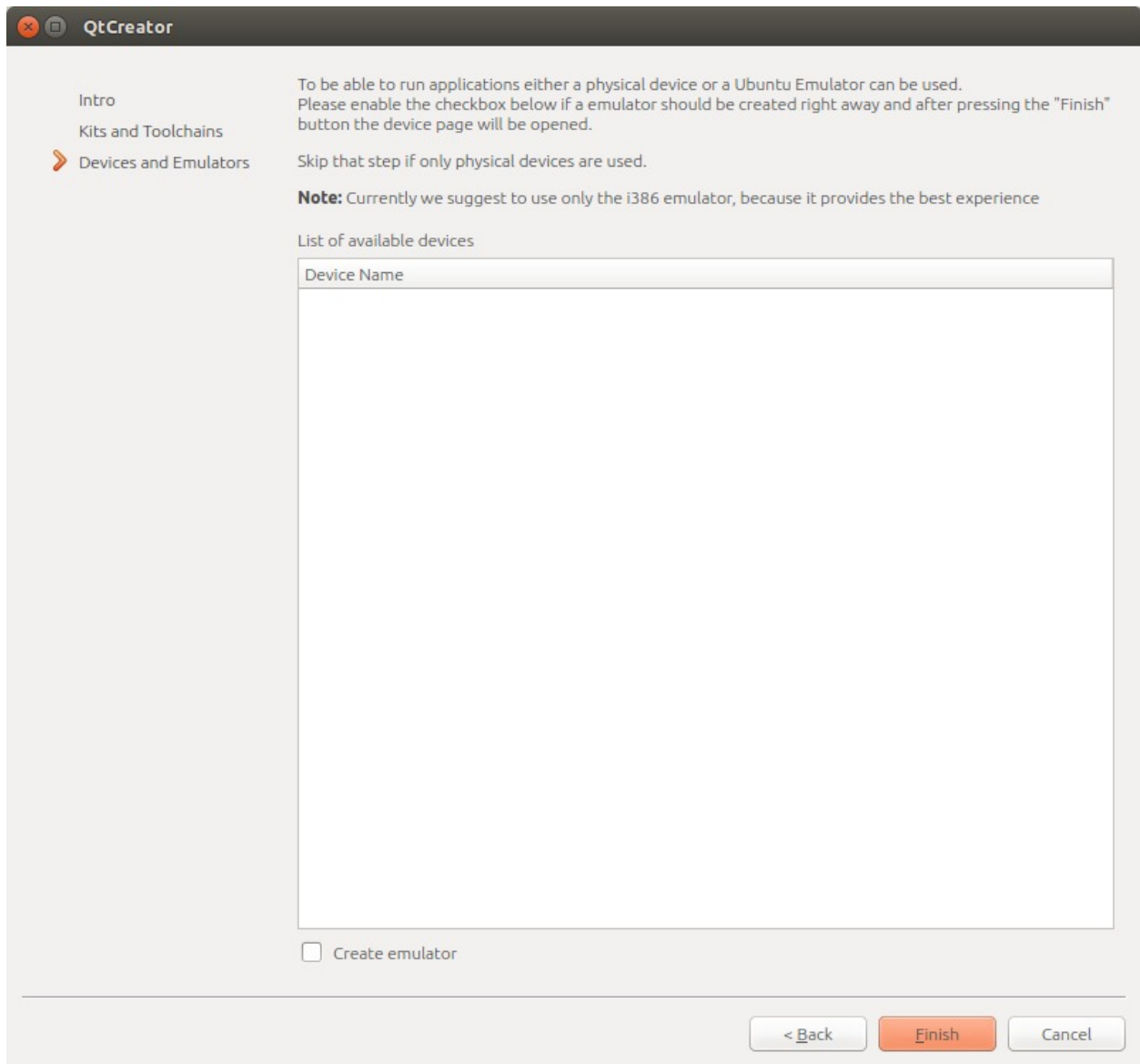
At this time it is not necessary to create additional kits.



Kit

List

In the last step of the wizard you can configure the physical devices and the emulator. There are some problems with the emulator. For now, I recommend not to configure the emulator. Applications can be tested natively on the computer. For this reason you must uncheck the 'Create emulator' box and click on the 'Finish' button.

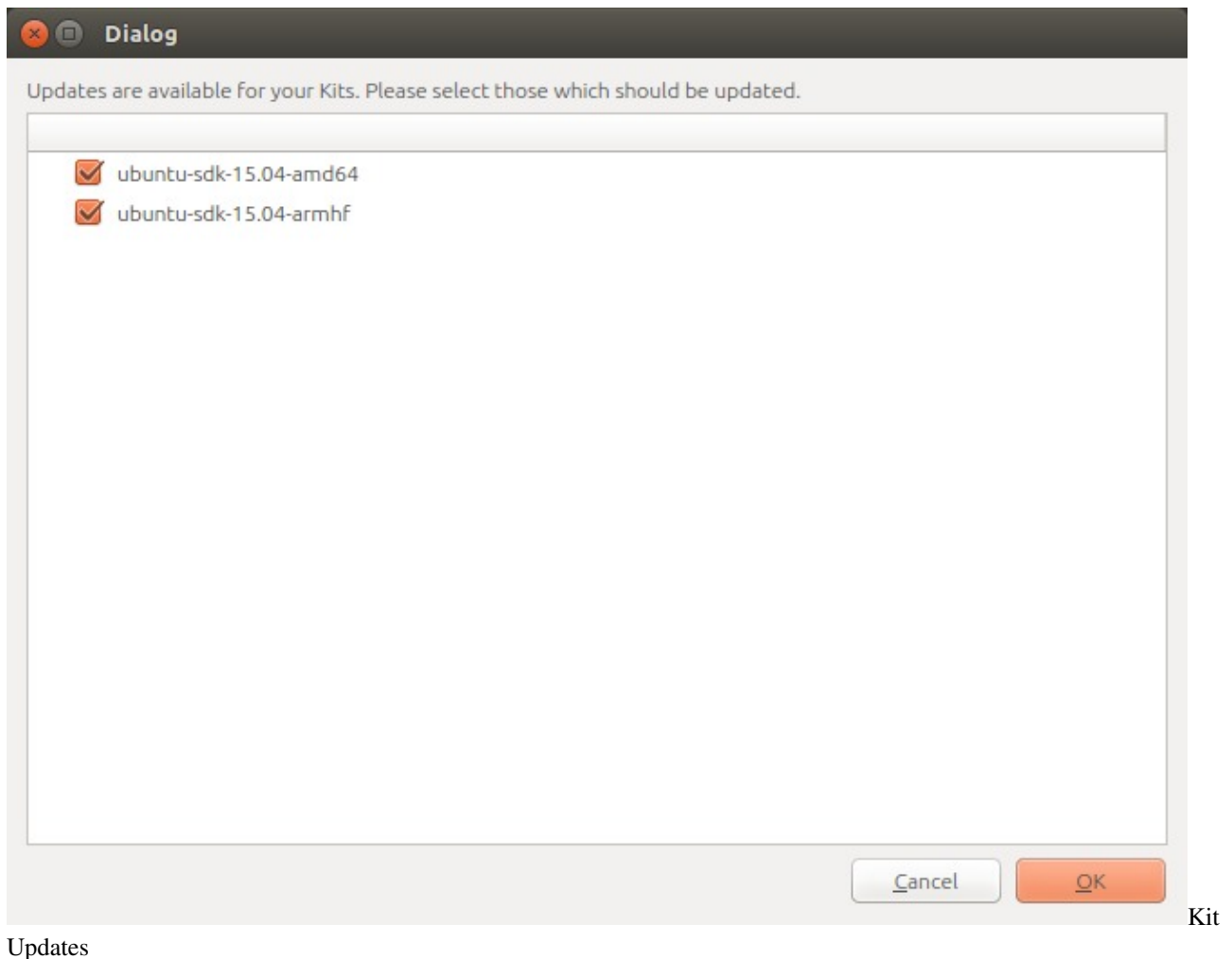


Emulators

List

10.5 Hello World

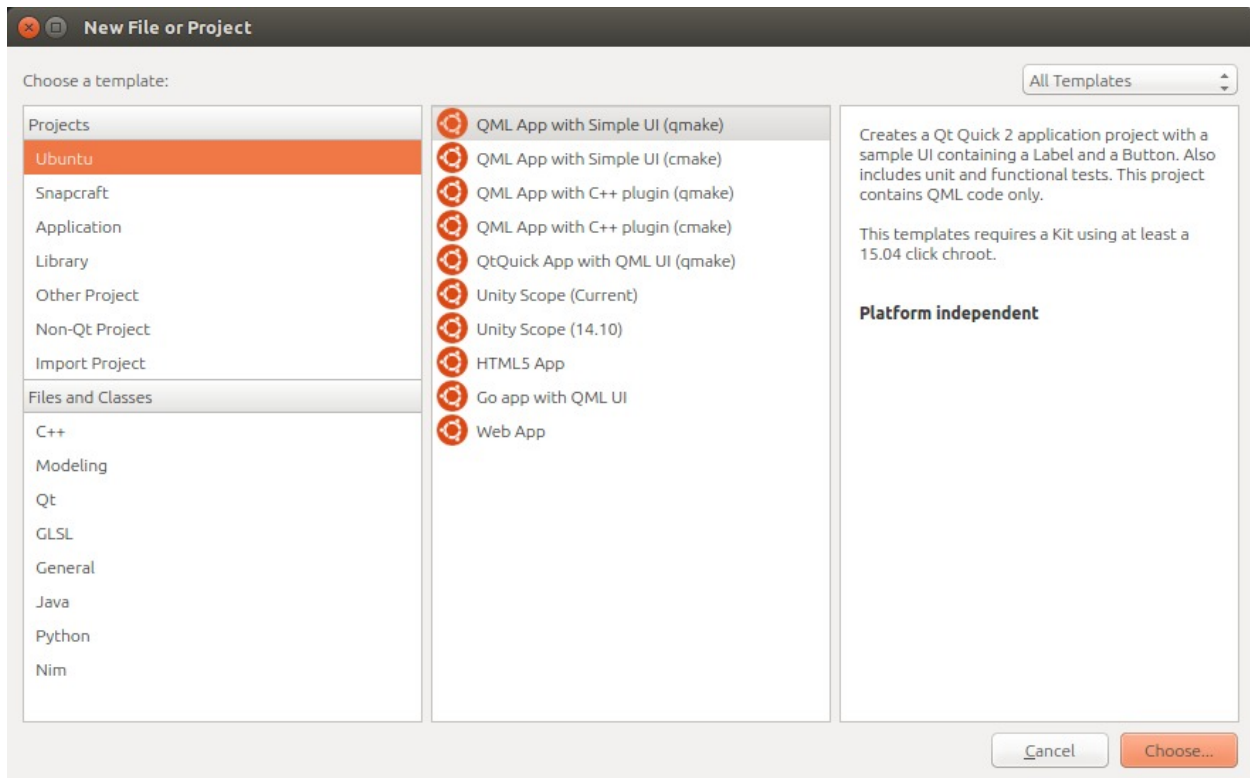
In order to keep the good traditions, the first step is to program a minimal application that allows us to check the correct operation of the SDK. The application will run on the computer natively and on a tablet. As you will see is quite simple to do it. Kits are updated frequently. If the following screen appears, all the kits must be marked and updated.



Updates

10.6 Project Creation

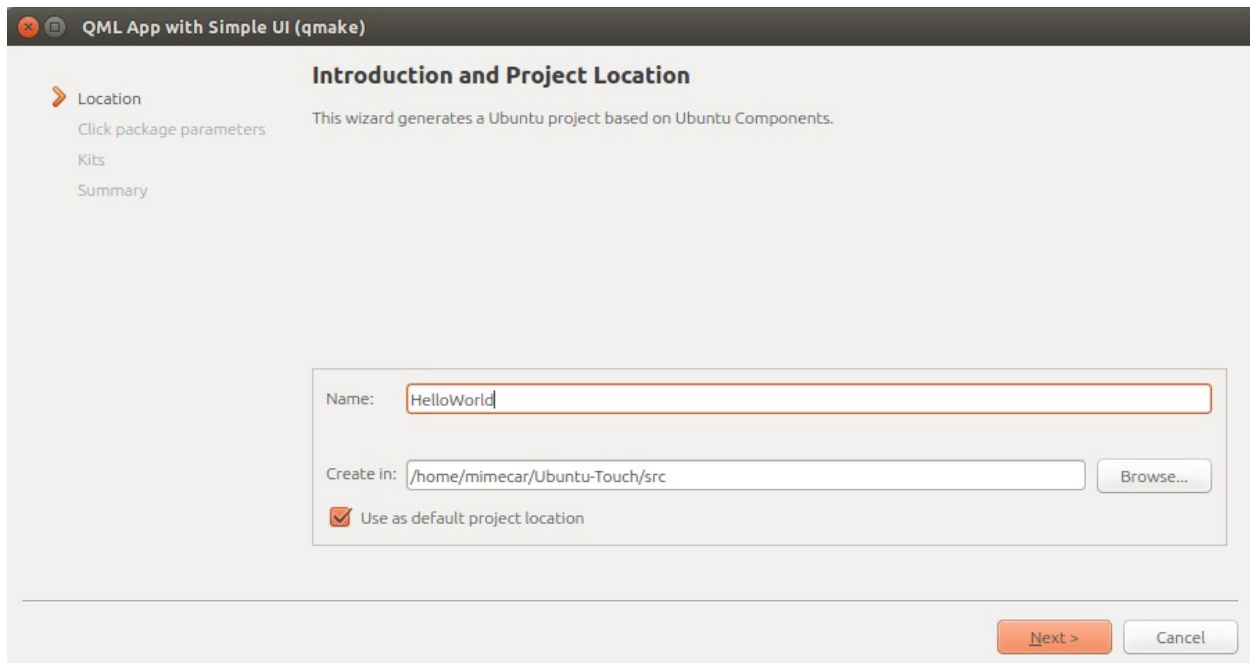
Click on the File menu, New file or project. A window will appear with the types of projects that can be used. The user interface is made with QML which is a scripting language oriented to the creation of graphical interfaces. The logic of the application can be done with several languages. JavaScript will be used for now. Select the first option and then click on the 'Choose' button.



New

Project

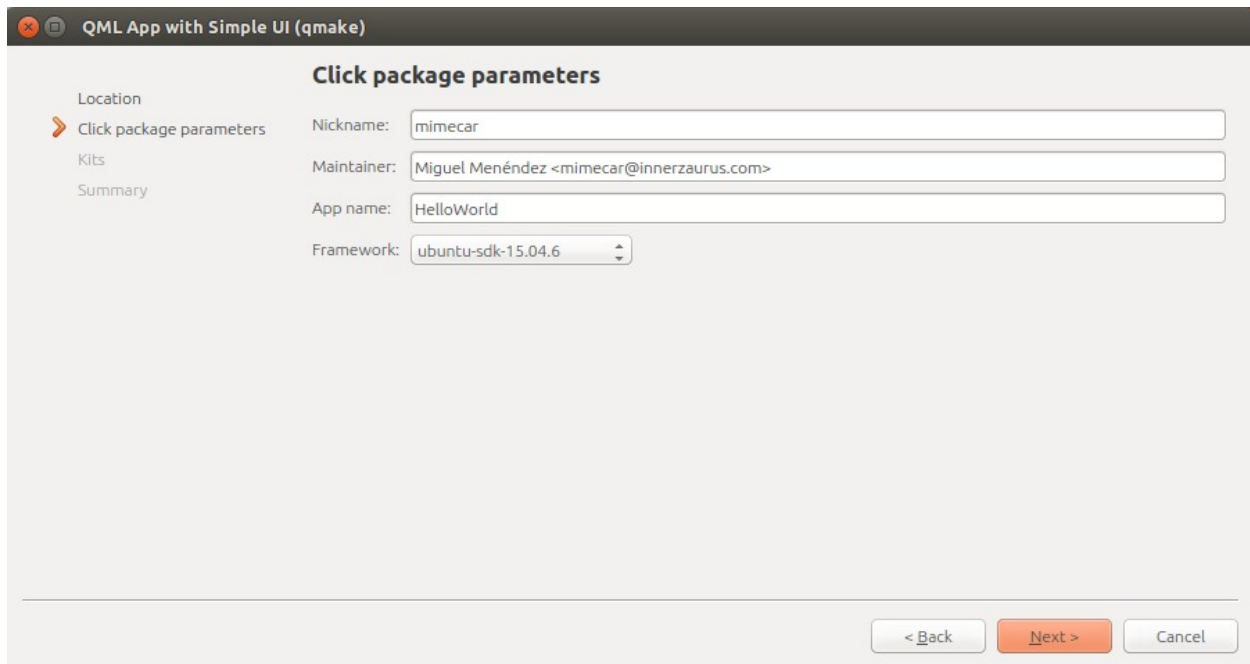
You must choose the folder in which the projects will be saved. The project name cannot contain spaces.



Project

Name

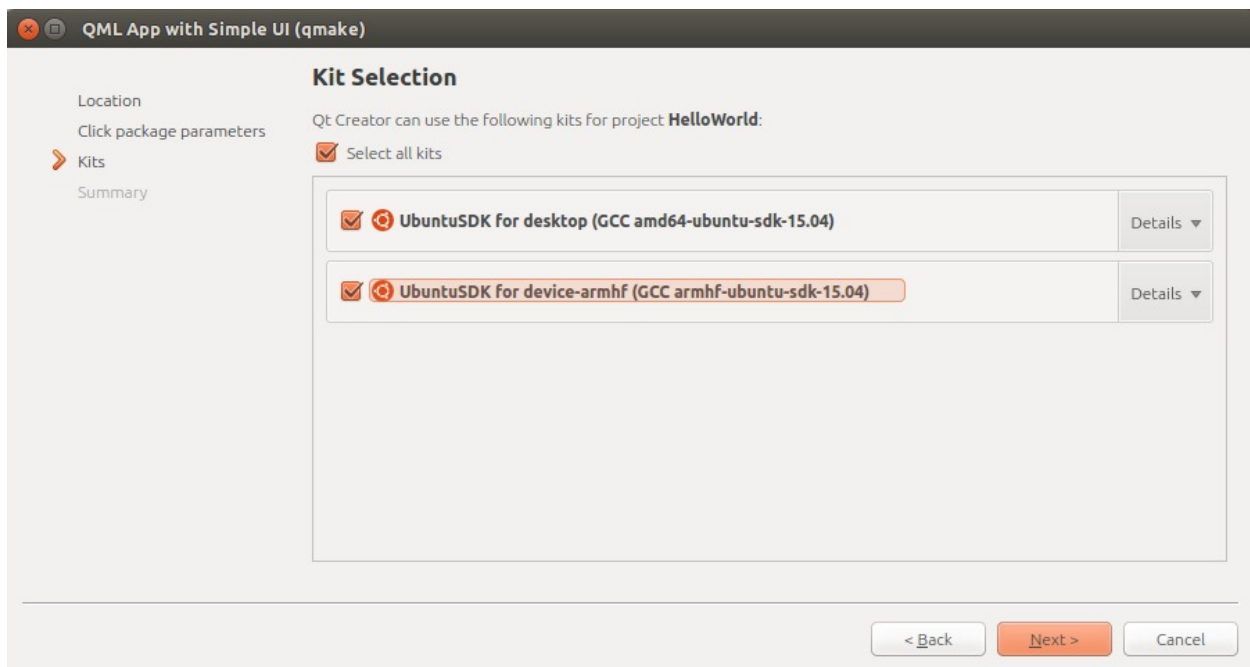
Applications need basic information: user and maintainer. All other parameters should be left as they come by default. It is important to respect the structure in the Maintainer field so that it allows us to continue.



Click

Parameters

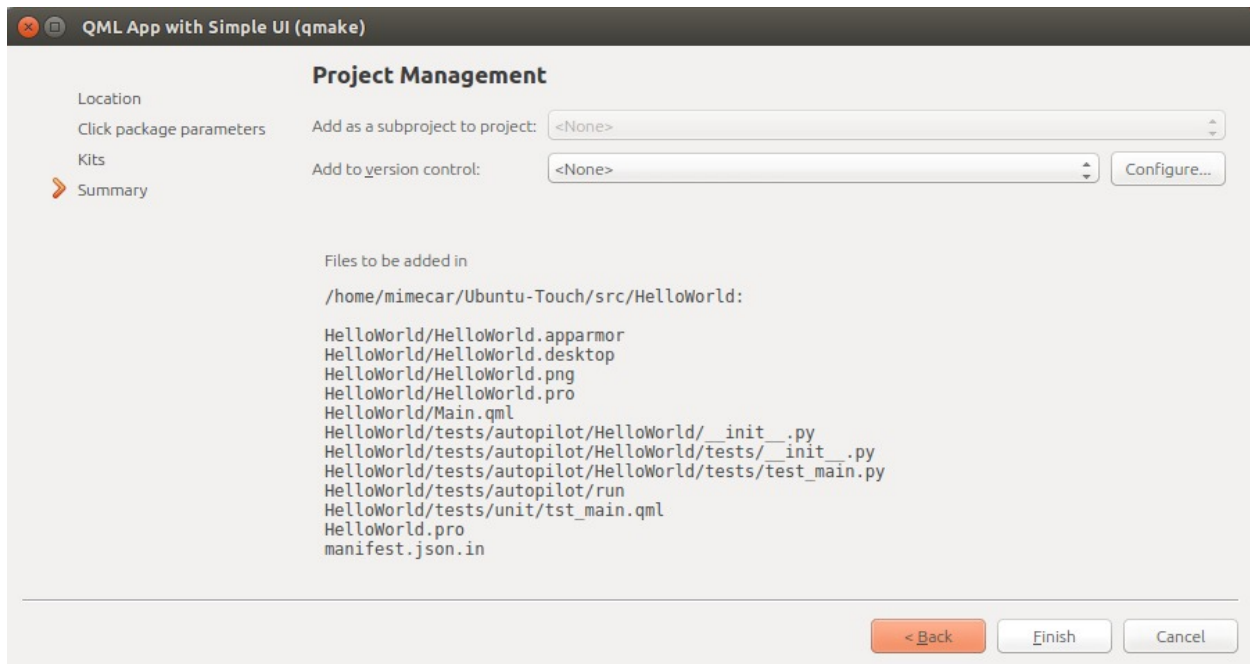
You have to select the kits that you want to use. By default the two will be selected in order to run the application on the computer and on the mobile device.



Project

Kits

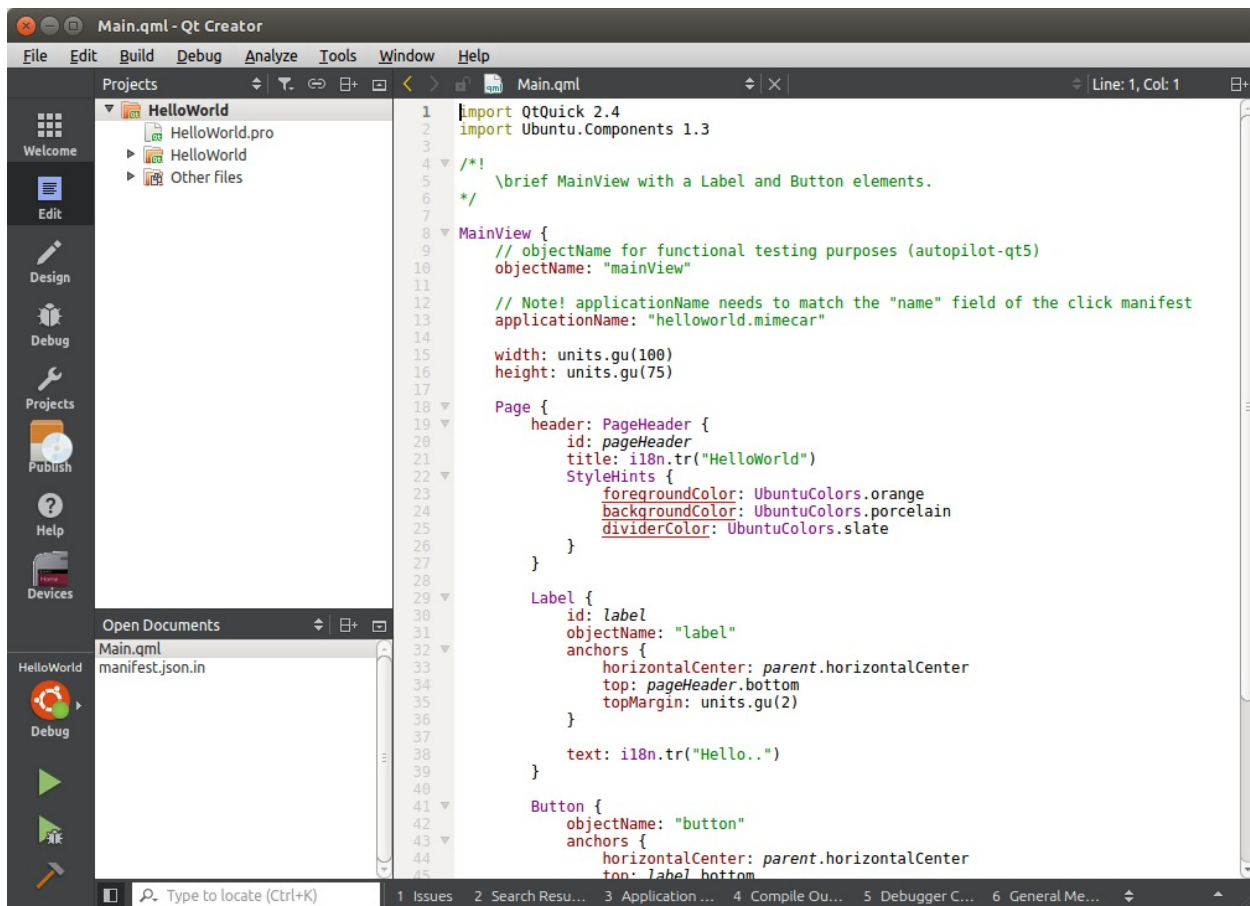
The last screen contains a summary of the wizard steps. Click on the 'Finish' button.



Project

Summary

The project will open automatically.

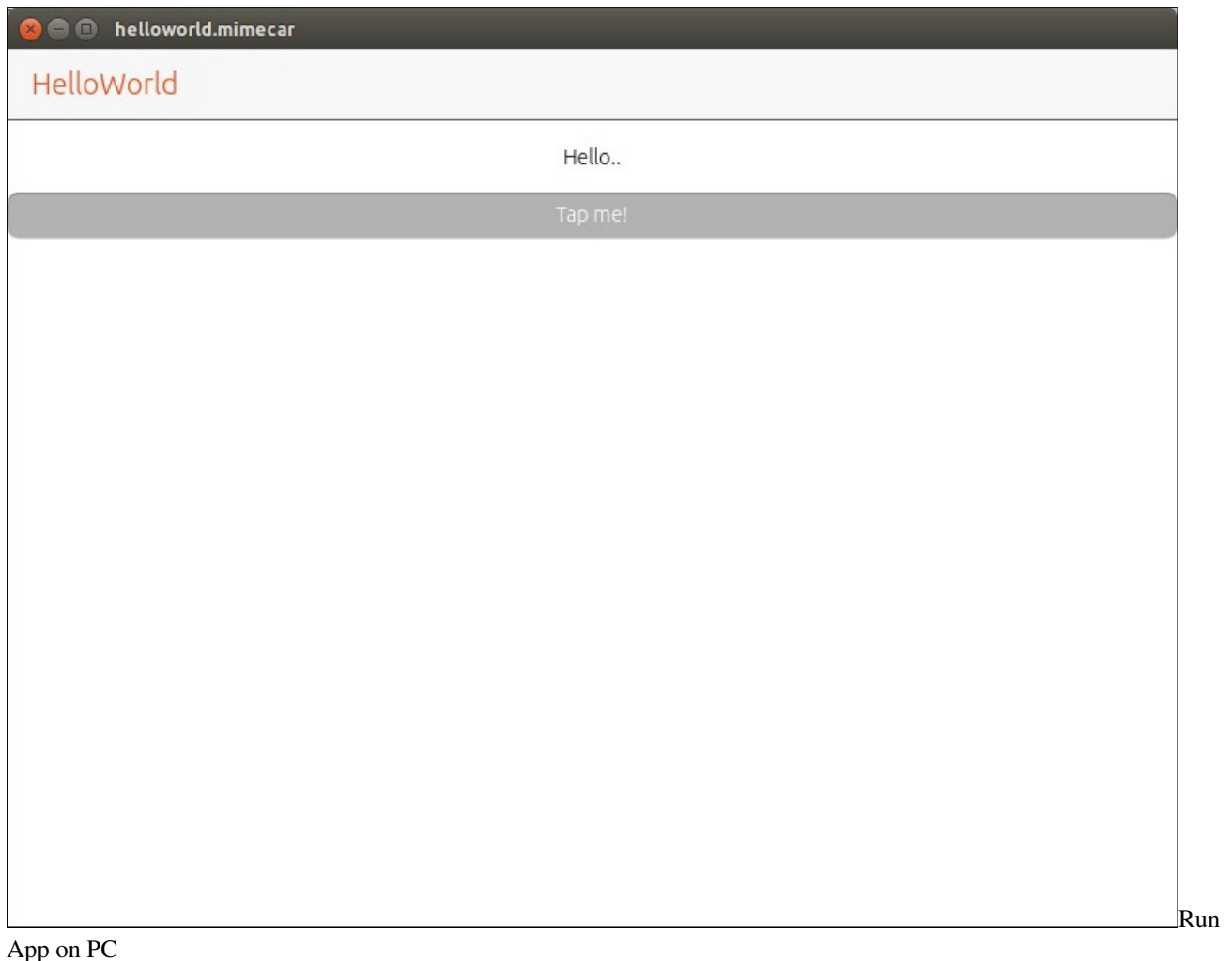


Project

Opened

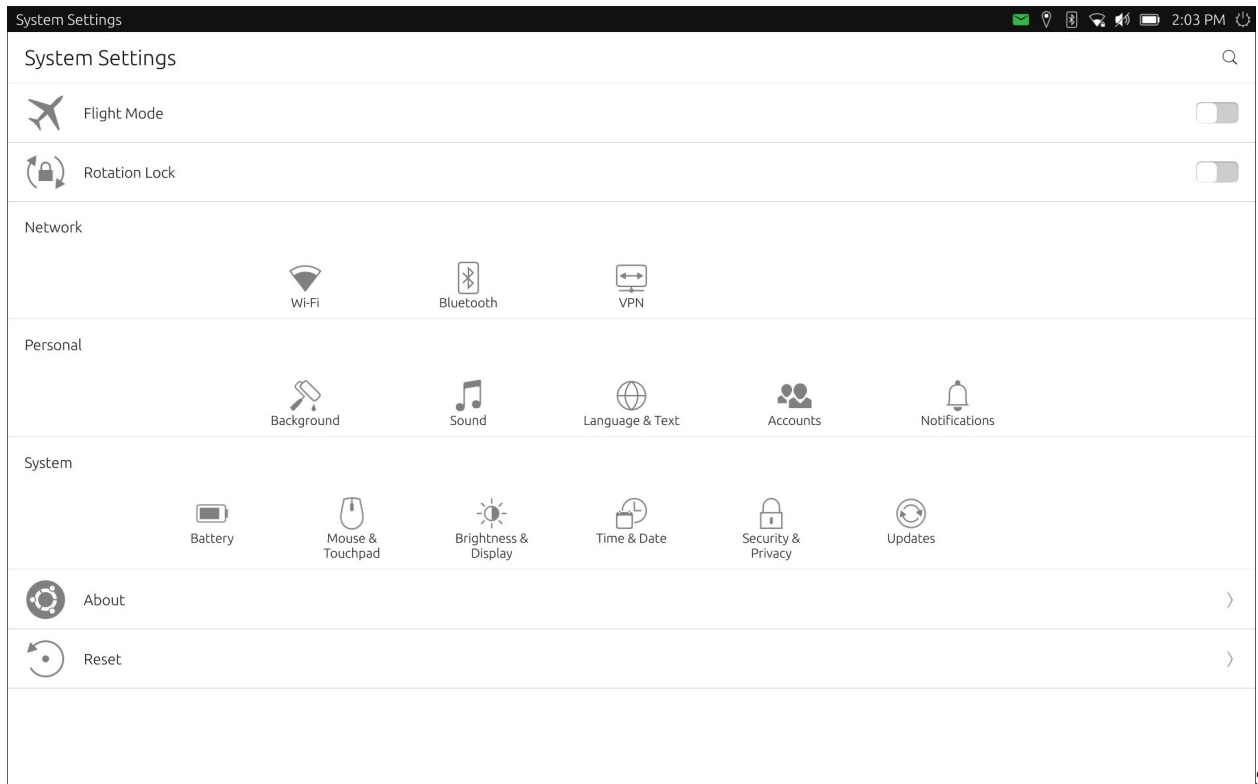
10.7 Running the Application on Your Computer

To run the application you have to press the ‘Play’ button in the lower left part of Qt Creator.



10.8 Running the Application on a Real Device

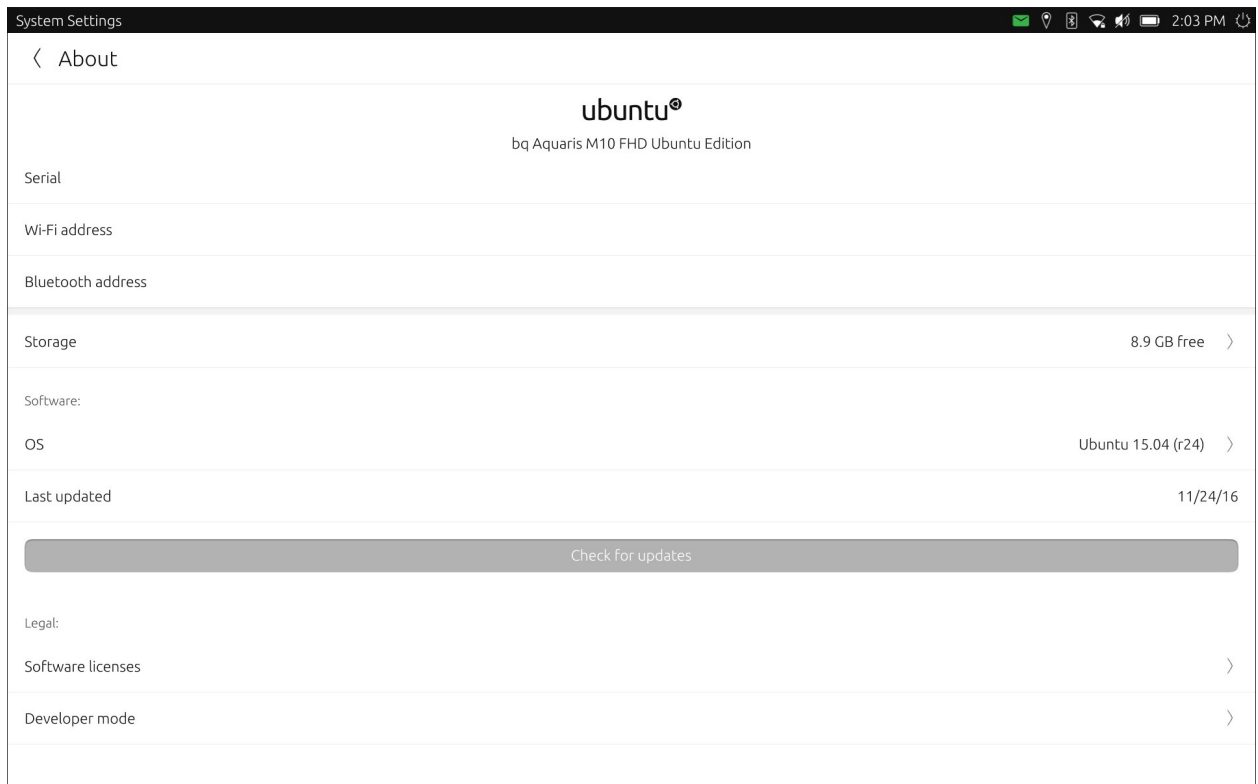
Before running the application, it is necessary to configure the device. For testing I have used an Aquaris M10 FHD tablet with the OTA-14 —although the procedure is the same on the other terminals. Leave ubuntu-sdk-ide open and connect the device via USB to the computer. To activate the development options, access the System Settings.



System

Settings

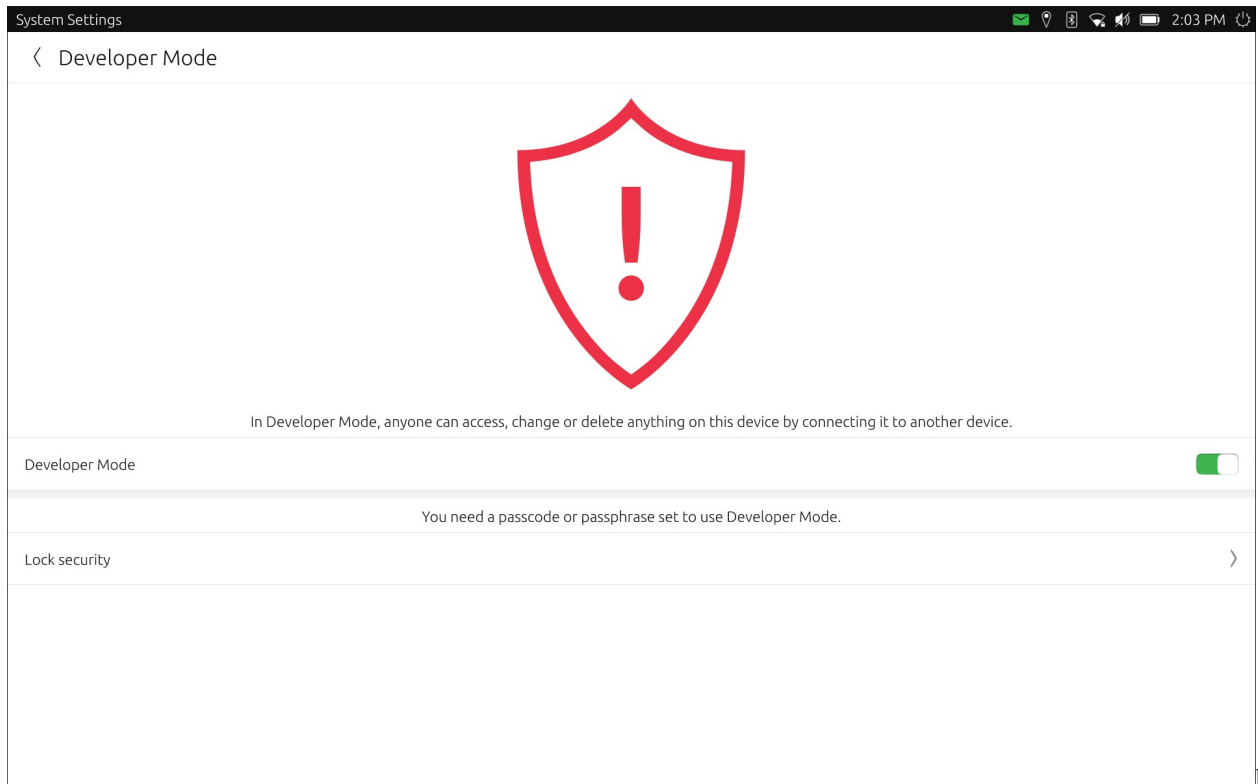
Click on About.



About

Select Developer Mode. This mode is disabled by default because it allows remote control of the device if it is

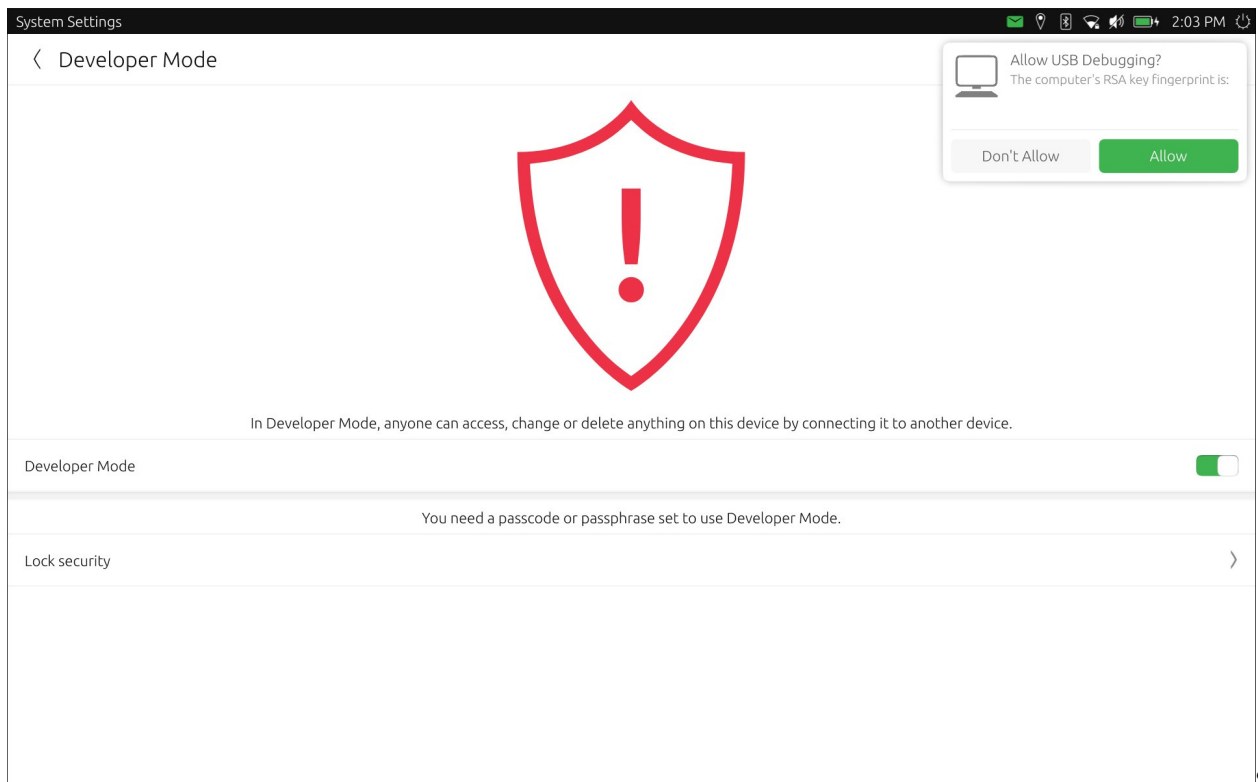
connected by USB to the computer. Check the box.



Developer

Mode

After a few seconds, a notification must appear with the connection request of the computer.

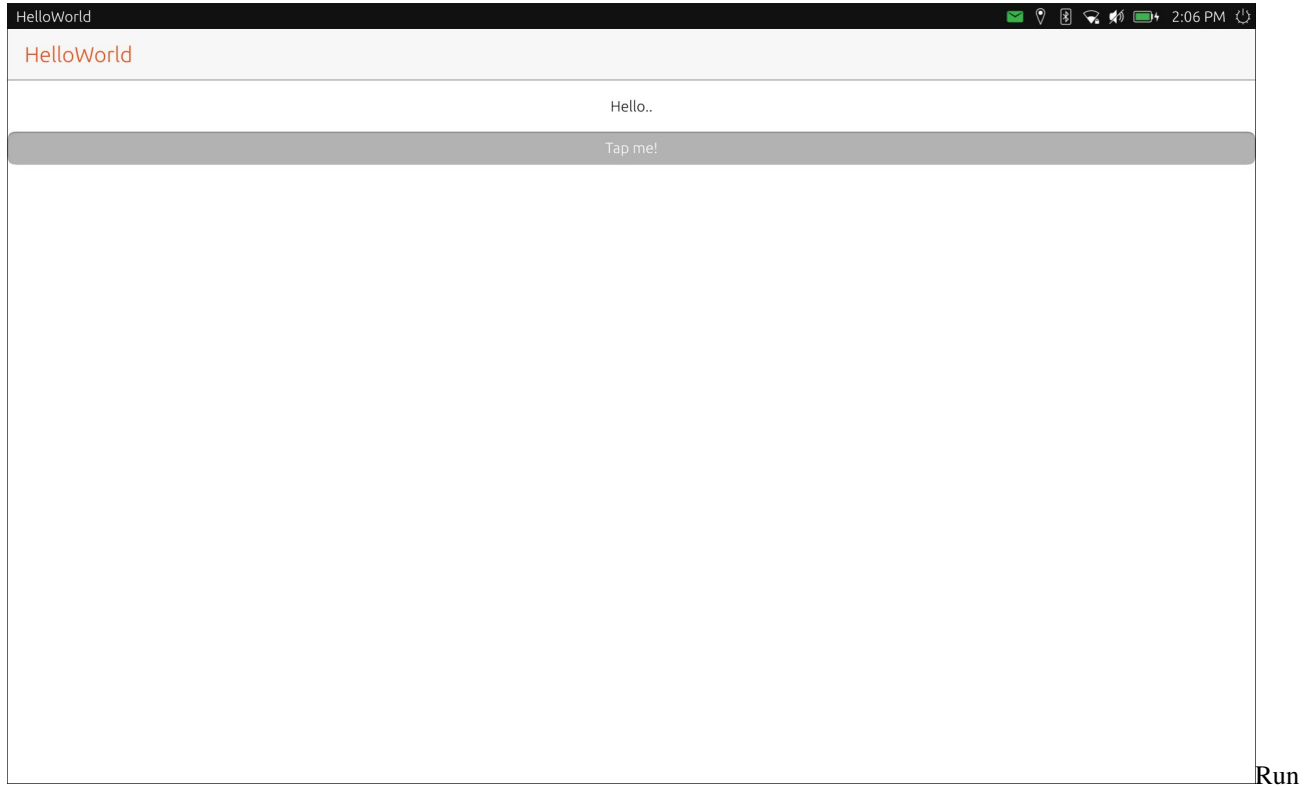


SSH

notification

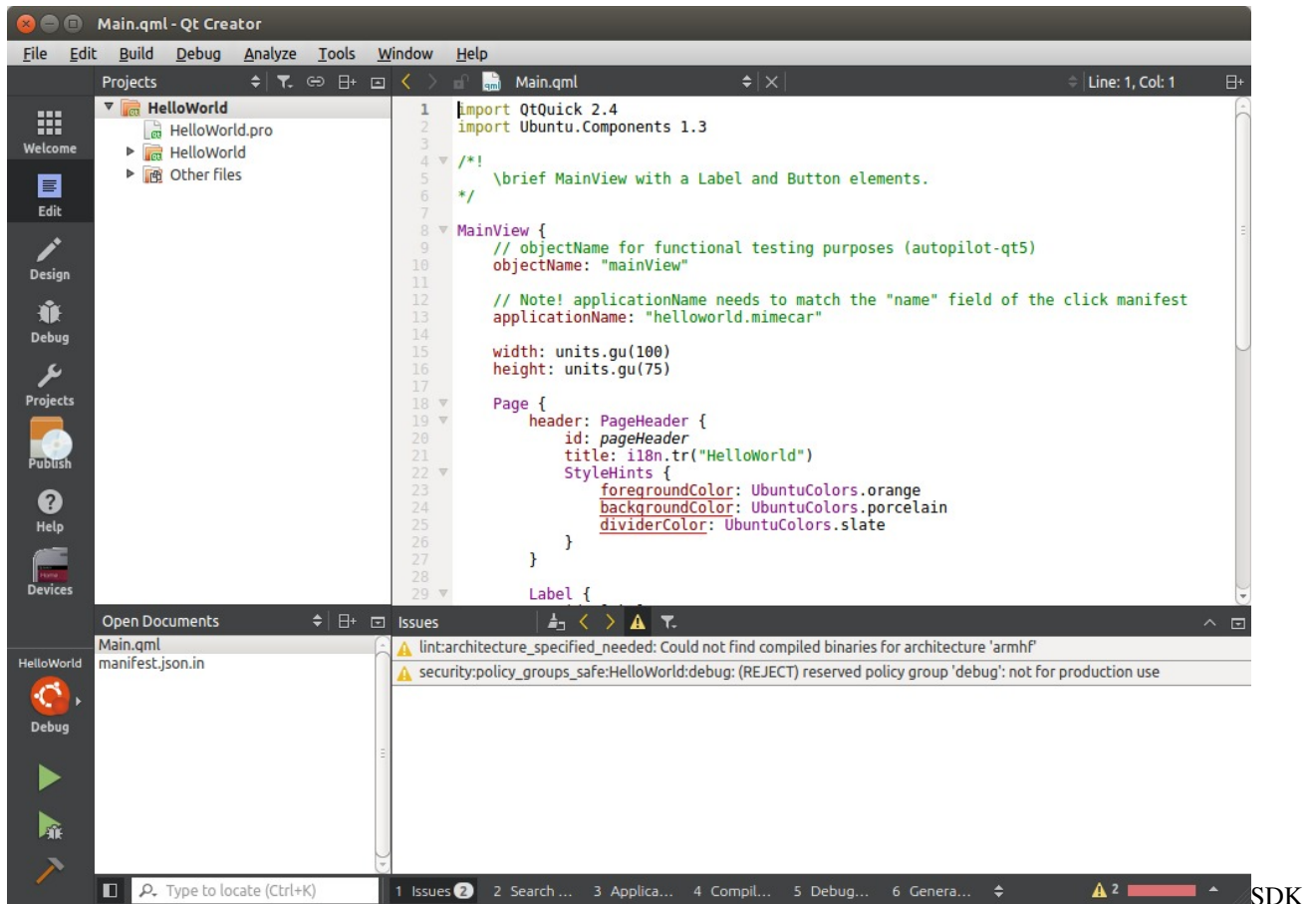
You have now completed all the steps on the side of the tablet. In the IDE, you must select the tablet as the compilation target. To do this, click on the button above the Play and select the device we just configured as the destination.

A few seconds after pressing the Play button, the application will appear on the device.



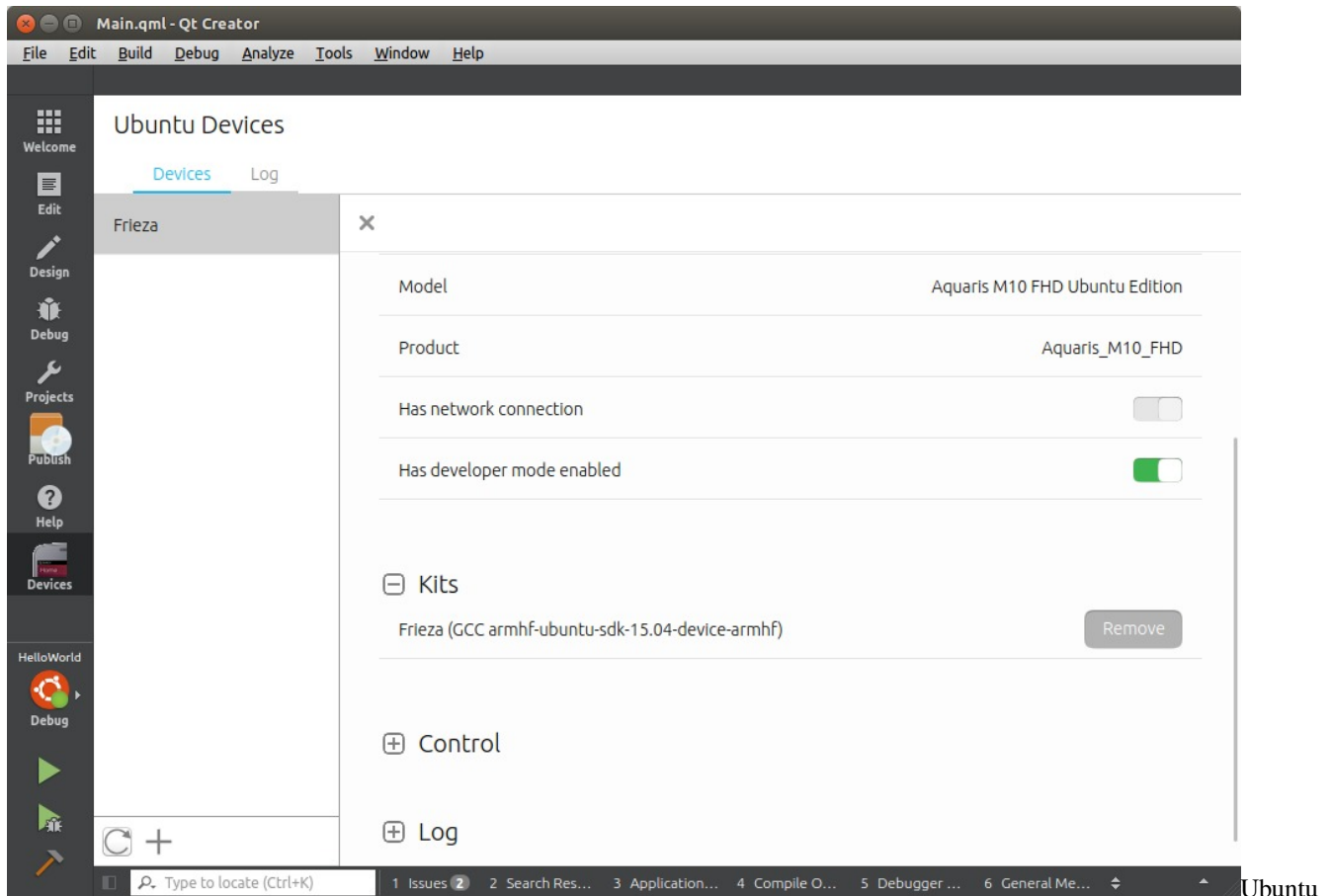
App on Device

The error shown in the following image may appear:



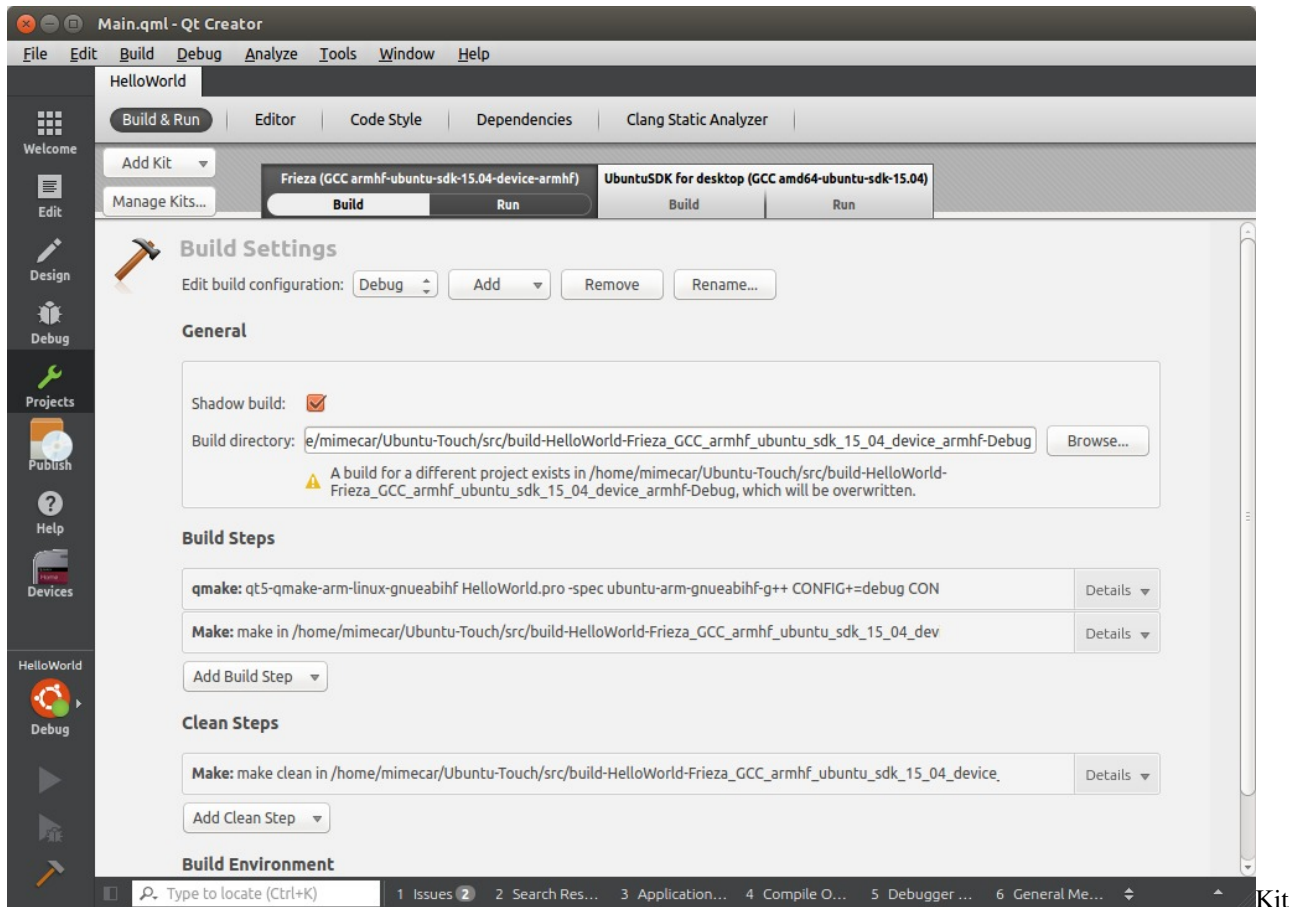
Error

In the sidebar of Qt Creator you have to click on the 'Devices' button. Then click on the device and Kits click on the 'Remove' button. Then click on the 'Create' button.



Devices

Now you have to associate the Kit back to the Project. Click on 'Projects' (on the sidebar), 'Add Kit' and select the Kit that appears in the dropdown.



Selection

Wait a couple of seconds and select the device as the compilation target. It is important that you try the compilation in both cases. In the next chapter I will start with the source code and assume that everything works correctly on both the computer and the test device. If you have any problems you can ask on the mailing list.

10.9 People who have collaborated

- Larrea Mikel: revision of the chapter in Spanish.
- Cesar Herrera: revision of the English translation.
- Joan CiberSheep: revision of the English translation.

CHAPTER 11

Indices and tables

- `genindex`
- `modindex`
- `search`