

Jürgen Bocklage-Ryannel Cyril Lorquet

Johan Thelin

Table of Contents

[Welcome! 5](#_TOC_250035)

[Acknowledgements 7](#_TOC_250034)

[Authors 10](#_TOC_250033)

[Qt and Qt Quick 14](#_TOC_250032)

[Qt Building Blocks 16](#_TOC_250031)

[Qt 6 Introduction 20](#_TOC_250030)

[Quick Start 29](#_TOC_250029)

[Installing Qt 6 SDK 30](#_TOC_250028)

[Hello World 32](#_TOC_250027)

[Application Types 36](#_TOC_250026)

[Summary 50](#_TOC_250025)

[Qt Creator IDE 51](#_TOC_250024)

[The User Interface 52](#_TOC_250023)

[Registering your Qt Kit 54](#_TOC_250022)

[Managing Projects 55](#_TOC_250021)

[Using the Editor 57](#_TOC_250020)

[Locator 58](#_TOC_250019)

[Debugging 60](#_TOC_250018)

[Shortcuts 61](#_TOC_250017)

[Quick Starter 63](#_TOC_250016)

[QML Syntax 64](#_TOC_250015)

[Core Elements 74](#_TOC_250014)

[Components 84](#_TOC_250013)

[Simple Transformations 89](#_TOC_250012)

[Positioning Elements 94](#_TOC_250011)

[Layout Items 102](#_TOC_250010)

[Input Elements 106](#_TOC_250009)

[Advanced Techniques 115](#_TOC_250008)

[Fluid Elements 116](#_TOC_250007)

[Animations 117](#_TOC_250006)

[States and Transitions 144](#_TOC_250005)

[Advanced Techniques 151](#_TOC_250004)

[UI Controls 152](#_TOC_250003)

[Introduction to Controls 153](#_TOC_250002)

[An Image Viewer 162](#_TOC_250001)

[Common Patterns 184](#_TOC_250000)

The Imagine Style 207

Summary 214

Model-View-Delegate 215

Concept 216

Basic Models 218

Dynamic Views 226

Delegate 245

Advanced Techniques 262

Summary 284

Canvas Element 286

Convenience API 290

Gradients 292

Shadows 294

Images 296

Transformation 298

Composition Modes 300

Pixel Buffers 303

Canvas Paint 306

Porting from HTML5 Canvas 309

Shapes 318

A Basic Shape 319

Building Paths 323

Filling Shapes 328

Animating Shapes 336

Summary 339

Effects in QML 340

Particle Concept 342

Simple Simulation 344

Particle Parameters 348

Directed Particles 351

Affecting Particles 358

Particle Groups 365

Particle Painters 377

Graphics Shaders 379

Shader Elements 381

Fragment Shaders 385

Wave Effect 392

Vertex Shader 395

Curtain Effect 409

Summary 415

Multimedia 416

Playing Media 417

Sound Effects 427

Video Streams 432

Capturing Images 434

Summary 441

Qt Quick 3D 442

The Basics 443

Working with Assets 454

Materials and Light 465

Animations 473

Mixing 2D and 3D Contents 480

Summary 483

Networking 485

Serving UI via HTTP 486

Templates 494

HTTP Requests 496

Local files 502

REST API 506

Authentication using OAuth 516

Web Sockets 536

Summary 544

Storage 545

Settings 546

Local Storage - SQL 549

Dynamic QML 555

Loading Components Dynamically 556

Creating and Destroying Objects 565

Tracking Dynamic Objects 572

Summary 577

JavaScript 578

Browser/HTML vs Qt Quick/QML 581

JS Language 583

JS Objects 586

Creating a JS Console 589

Qt and C++ 596

A Boilerplate Application 599

The QObject 607

Build Systems 611

Common Qt Classes 618

Models in C++ 628

Extending QML with C++ 645

Understanding the QML Run-time 646

Plugin Content 652

Creating the plugin 655

FileIO Implementation 658

Using FileIO 661

Summary 672

Qt for Python 674

Introduction 675

Installing 677

Building an Application 679

Limitations 699

Summary 700

Qt for MCUs 701

Setup 702

Hello World - for MCUs 705

Integrating with C++ 713

Working with Models 721

Summary 728

# Welcome!

Welcome to The Qt 6 Book - A book about QML. This text will guide you through QML, Qt's language for creating dynamic user interfaces.

I believe that the ability to build declarative（声明式编程A 1）, reactive（反应式A 2）, hardware accelerated（硬件加速） user interfaces executing at native performance（原生性能） across *all* major platforms (and some not so major) is a game（规则） changer. When starting with Qt, it was almost as if I had my secret weapon to building software quickly. QML takes that to the next level.

How is this book different from the Qt documentation? I hear you ask. The intention is to build a complement（补充（体系））. This book is meant as a book that you can read from front to back where each chapter builds on what you've previously learned. But it can also be used as a way for the experienced reader to get oriented in（了解） a new topic. Each chapter focuses on a specific topic and introduces the concepts from Qt and QML. However, the Qt documentation will always provide the full picture and is a great reference to look up the details about all elements, properties, enumerations, and more.

I wish you a pleasant read!

*Johan Thelin*

## Structure

The book can be said to be split into three parts. The split is not clear cut enough to motivate（支持） a strict division of chapters, but more of a guideline that we've tried to follow when writing it.

The first few chapters, let's say until somewhere around chapter 5 - 7 can be considered an introduction. If you want to learn QML, you should make sure to read these chapters.

The following chapters, 6-14, can be seen as fairly（相当地） separate（独立） chapters introducing independent topics, even though the models from chapter 7 are used in many more places. Feel free to dive into these in the order that you like and learn about the topics that you are curious（好奇） about.

The remainder（剩余部分） of the book focuses on more advanced topics such as details of JavaScript, mixing C++ and QML, and the Qt for Python bindings and QML. These are important topics and I really want you to read them. To build a full application with QML you need to understand these topics, but their main focus is not on QML.

## Never Ending Work in Progress

The Qt 6 Book is a never ending work in progress. We welcome contributors and are planning to open up（开放） our infrastructure（基础设施） to let you contribute both by reporting issues and by contributing fixes and new content. The end goal is to present you with a printed book when the material has reached a maturity（成熟） level that we are happy with, but we want to share this with you already now and to learn from your feedback what to improve, and what additional content to add.

# Acknowledgements（鸣谢）

This book would not have been possible to create without the kind sponsorship（赞助）from **The Qt Company**. It is a privilege（荣幸） to be able to work on a project such as this, and their help has been invaluable. I would like to give a special mention to (alphabetically（字母顺序）):

 empenzes  Fabian K

 Luca Di sera  magoldst-qt

 Maurice Kalinowski  Mitch Curtis

 nezticle

 Tino Pyssysalo  Ulf Hermann

 Vladimir Minenko

## Contributors

This book was made possible by the great community contributors. I would like to thank them all (alphabetically):

alexshen arky

DavidAdamsPerimeter delvianv

guoci nittwitt

 oleksis

 LorenDB  paulmasri  QtSCH

 ruudschouten  task-jp

 topecongiro  VideoCarp

 wangchunlin5013

## History

This book is based on [The QML Book [](https://qmlbook.github.io/) (https://qmlbook.github.io/) , originally written for Qt 5. I would like to thank all contributors to that book (alphabetically):](https://qmlbook.github.io/)

aamirglb alexeRadu

andreabedini amura11 bakku cibersheep dbelyaev danielbaak DocWicking empyrical Ge0

gillesfernandez gitter-badger gsantner

hckr iitaka1142

 jiakuan  justinfx

 maggu2810

 marco-piccolino  mariopal

 mark-summerfield  mhubig

 micdoug

 Mihaylov93

 moritzsternemann  RossRogers

 Swordfish90  sycy600

 trolley  29jm

I would also like to give a special mention to Pelagicore, The Qt Company and Felgo for help the development of The QML Book by sponsoring our work and being generally awesome when it comes to feedback and support.

# Authors

The Qt 6 Book has been written by a team of authors. They are:

## Johan Thelin



Johan works as a system architect（系统架构师） building automotive solutions. He has a background from over twenty years of device creation based on Linux, Qt

and more. He has written for various papers and blogs, presented at numerous conferences, and provides advice on how to build software, and software organizations. As an avid believer in free and open source [solutions, he founded and organizes the foss-north conference [](https://foss-north.se/) (https://foss-north.se) .](https://foss-north.se/)

[You can find out more about Johan at LinkedIn [](https://www.linkedin.com/in/johanthelin)](https://www.linkedin.com/in/johanthelin)

[(https://www.linkedin.com/in/johanthelin) , his blog [](http://www.thelins.se/johan/blog/)](http://www.thelins.se/johan/blog/)

[(http://www.thelins.se/johan/blog/) , and](http://www.thelins.se/johan/blog/) [his homepage [](http://e8johan.se/) (http://e8johan.se)](http://e8johan.se/)

.

## Jürgen Bocklage-Ryannel



Jürgen is the CEO of ApiGear, which is a collaborative machine interface design tool that enables teams to collaborative design software interfaces with automated monitoring and simulation solutions.

He was the co-founder of Pelagicore AG and was responsible there as Chief User Interface Architect for the early versions of the Daimler MBUX.

He focuses currently on an API driven workflow to design and create the interfaces between the user experience and the underlying services for

different platforms.

[You can find out more about Jürgen at LinkedIn [](https://www.linkedin.com/in/jryannel/) (https://www.linkedin.com/in/jryannel/) .](https://www.linkedin.com/in/jryannel/)

## Cyril Lorquet



[Co-Founder and CEO of the belgian company Eunoia Studio [](https://www.eunoia.be/)](https://www.eunoia.be/)

[(https://www.eunoia.be) , Cyril helps organizations turn their](https://www.eunoia.be/) know-how into software products. Since 2009, he has been working on software products in various contexts (construction, healthcare, hydrology, marketing, ...) -

several of them involving Qt. Software engineer at heart, he has a passion for design processes, software development and change management.

[You can find out more about Cyril at LinkedIn [](https://www.linkedin.com/in/cyrillorquet) (https://www.linkedin.com/in/cyrillorquet) .](https://www.linkedin.com/in/cyrillorquet)

# Qt and Qt Quick

This book provides you with a walk through of the different aspects（方面） of application development using the new Qt 6. It focuses on the Qt Quick technology, but also provides necessary information about writing C++ back-ends（后端） and extensions（扩展） for Qt Quick.

This chapter shall（将） provide a high-level overview of Qt 6. It shows the different application models available for developers, as well as（以及） a showcase（展示） application, as a sneak（前瞻） preview of things to come. Additionally, the chapter aims to provide a wide overview of the Qt content（内容） and how to get in touch with the makers of Qt Company.

## Qt 6 Focus

Qt 5 was released many years ago and introduced a new declarative way of writing stunning（惊人的） user interfaces B1. Since then a lot has changed in the world around us.

Qt 6 will be a continuation of what has been done with Qt 5 and should not be disruptive（引起混乱的） to the majority of users. What makes Qt valuable to its users?

* Its cross-platform nature（性质）
* Its scalability（可扩展性）
* World class（级别） APIs and documentation
* Maintainability, stability and compatibility（兼容性）
* A large developer ecosystem（生态）

Qt 6 evolves the Qt product to new markets while keeping close to the users values.

The desktop market is the root of the Qt offering. It is where most users get the first contact with Qt and it forms the base for the Qt tools and its success.

It is expected（预测） that Qt 6 will grow most in the embedded（嵌入式的） and connected（联网） devices market from high-end（高端的） near desktop performing devices to low-end devices like microcontrollers. Touch screens will come to an exponential（指数的） increasing number to these devices. Many of these devices will have relatively （相当地）simple functionality but require a polished（精良的） and smooth user interface.

At the other end of the spectrum（光谱，范围）（在另一端） there is a demand（需求） for more complex and 2D/3D integrated（集成） user interfaces. The 3D content with 2D elements based interfaces will be common, as will be the use of augmented（增强） and virtual reality.

The growth of connected devices and the higher demand for smooth user interfaces require a simpler workflow to create applications and devices. Integrating UXA3 designers into the development workflow is one of the goals of the Qt 6 series.

## Qt 6 brings us:

* Next generation QML
* Next generation graphics
* Unified（统一的） and consistent（一致的） tooling
* Enhanced Qts C++ APIs
* Component（组件） Marketplace

# Qt Building Blocks（块）

Qt 6 consists of a large number of modules. In general（一般而言）, a module is a library for the developer to use. Some modules are mandatory（强制性的） for a Qt- enabled platform and form the set called *Qt Essentials（必需的） Modules*. Other modules are optional, and form the *Qt Add-On Modules*. The majority of developers may not need to use the latter（后者）, but it’s good to know about them as they provide invaluable（无价的） solutions to common challenges.

## Qt Modules

The Qt Essentials modules are mandatory for any Qt-enabled platform. They offer the foundation to develop modern Qt 6 Applications using Qt [Quick 2. The full list of modules is available in the Qt documentation module list [](https://doc.qt.io/qt-6/qtmodules.html#qt-essentials) (https://doc.qt.io/qt-6/qtmodules.html#qt-essentials) .](https://doc.qt.io/qt-6/qtmodules.html#qt-essentials)

## Core-Essential Modules

The minimal set of Qt 6 modules to start QML programming.

**Qt Core** - Core non-graphical classes used by other modules.

**Qt D-BUS A4** - Classes for inter-process（进程间） communication over the D-Bus protocol on linux.

**Qt GUI** - Base classes for graphical user interface (GUI) components. Includes OpenGL.

**Qt Network** - Classes to make network programming easier and more portable（便携式-易移植）.

**Qt QML** - Classes for QML and **JavaScript languages.**

**Qt Quick** - A declarative（声明式的） framework for building highly dynamic applications with custom user interfaces.

**Qt Quick Controls** - Provides lightweight QML types for creating performant user interfaces for desktop, embedded, and mobile devices. These types employ a simple styling architecture and are very efficient. **Qt Quick Layouts** - Layouts are items that are used to arrange（安排） Qt Quick 2 based items in the user interface.

**Qt Quick Test** - A unit（单元） test framework for QML applications, where the test cases（用例） are written as JavaScript functions.

**Qt Test** - Classes for unit testing Qt applications and libraries.

**Qt Widgets** - Classes to extend Qt GUI with C++ widgets.

QtNetwork

QtQuickControls

QtCore

QtTest

QtGui

QtQml

QtQuick

QtQuickTest

QtQuickLayout

## Qt Add-On Modules

Besides the essential modules, Qt offers additional modules that target specific purposes. Many add-on modules are either feature-complete（功能完整） and exist for backwards（向后） compatibility（兼容）, or are only applicable（可应用于） to certain platforms. Here is a list of some of the available add-on modules, but make sure you [familiarize yourself with them all in the Qt documentation add-ons list [](https://doc.qt.io/qt-6/qtmodules.html#qt-add-ons) (https://doc.qt.io/qt-6/qtmodules.html#qt-add-ons) and in the list below.](https://doc.qt.io/qt-6/qtmodules.html#qt-add-ons)

 **Network**: Qt Bluetooth / Qt Network Authorization

 **UI Components**: Qt Quick 3D / Qt Quick Timeline / Qt Charts / Qt Data Visualization / Qt Lottie Animation / Qt Virtual Keyboard

 **Graphics**: Qt 3D / Qt Image Formats / Qt OpenGL / Qt Shader Tools / Qt SVG / Qt Wayland Compositor

 **Helper**: Qt 5 Core Compatibility APIs / Qt Concurrent / Qt Help / Qt Print Support / Qt Quick Widgets / Qt SCXML / Qt SQL / Qt State Machine / Qt UI Tools / Qt XML

#### TIP

As these modules are not part of the release, the state of each module may differ depending on how many contributors are active and how well it’s tested.

## Supported Platforms

Qt supports a variety of platforms including all major desktop and embedded platforms. Through the **Qt Platform Abstraction**, it’s now easier than ever to port（移植） Qt to your own platform if required.

Testing Qt 6 on a platform is time-consuming（费时的）. A subset（子集） of platforms was selected by the Qt Project to build the reference platforms set. These platforms are thoroughly（彻底的） tested through the system testing to ensure the best quality. However, keep in mind that no code is error-free.

# Qt Project（Qt项目本身）

From the [Qt Wiki [](http://wiki.qt.io/) (http://wiki.qt.io/) :](http://wiki.qt.io/)

“The Qt Wiki is a meritocratic（精英统治的） consensus（一致意见）-based community（精英共识社区） interested in Qt. Anyone who

shares that interest can join the community, participate（参加）

in its decision-making processes, and contribute to Qt’s development.”

The Qt Wiki is a place where Qt users and contributors share their insights（洞察力）. It forms the base for other users to contribute. The biggest contributor is The Qt Company, which holds also the commercial（商业） rights to Qt.

Qt has an open-source aspect（方面） and a commercial aspect for companies. The commercial aspect is for companies which can not or will not comply（遵守） with the open-source licenses. Without the commercial aspect（方面）, these companies would not be able to use Qt and it would not allow The Qt Company to contribute so much code to the Qt Project.

There are many companies worldwide, which make the living out of（以…为生） consultancy（咨询） and product development using Qt on the various platforms. There are many open-source projects and open-source developers, which rely on Qt as their major development library. It feels good to be part of this vibrant（充满活力的） community and to work with this awesome tools and libraries. Does it make you a better person? Maybe:-)

**Contribute here:** [**http://wiki.qt.io/**](http://wiki.qt.io/)

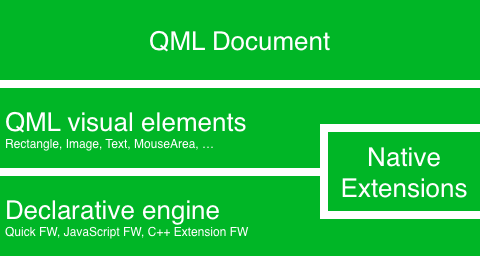
# Qt 6 Introduction

## Qt Quick

Qt Quick is the umbrella term for the user interface technology used in Qt 6. It was introduced in Qt 4 and now expanded for Qt 6. Qt Quick itself is a collection of several technologies:

QML - Markup language for user interfaces JavaScript - The dynamic scripting language

Qt C++ - The highly portable enhanced c++ library



Similar to HTML, QML is a markup language. It is composed of tags, called types in Qt Quick, that are enclosed in curly brackets: Item {} . It was designed from the ground up for the creation of user interfaces, speed and easier reading for developers. The user interface can be enhanced further using JavaScript code. Qt Quick is easily extendable with your own native functionality using Qt C++. In short, the declarative UI is called the front-end

and the native parts are called the back-end. This allows you to separate the computing intensive and native operation of your application from the user interface part.

In a typical project, the front-end is developed in QML/JavaScript. The

back-end code, which interfaces with the system and does the heavy lifting, is developed using Qt C++. This allows a natural split between the more design-oriented developers and the functional developers. Typically, the back-end is tested using Qt Test, the Qt unit testing framework, and exported for the front-end developers to use.

## Digesting a User Interface

Let’s create a simple user interface using Qt Quick, which showcases some aspects of the QML language. In the end, we will have a paper windmill with rotating blades.



We start with an empty document called main.qml . All our QML files will have the suffix .qml . As a markup language (like HTML), a QML document needs to have one and only one root type. In our case, this is the

Image type with a width and height based on the background image geometry:

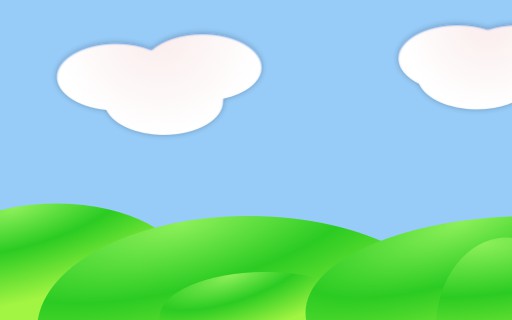
import QtQuick Image {

id: root

source: "images/background.png"

}

As QML doesn’t restrict the choice of type for the root type, we use an Image type with the source property set to our background image as the root.



#### TIP

Each type has properties. For example, an image has the properties width and height , each holding a count of pixels. It also has other properties, such as source . Since the size of the image type is automatically derived from the image size, we don’t need to set the width and height properties ourselves.

The most standard types are located in the QtQuick module, which is made available by the import statement at the start of the .qml file.

The id is a special and optional property that contains an identifier that can be used to reference its associated type elsewhere in the document. Important: An id property cannot be changed after it has been set, and it cannot be set during runtime. Using root as the id for the root-type is a convention used in this book to make referencing the top-most type predictable in larger QML documents.

The foreground elements, representing the pole and the pinwheel in the user interface, are included as separate images.





We want to place the pole horizontally in the center of the background, but offset vertically towards the bottom. And we want to place the pinwheel in the middle of the background.

Although this beginners example only uses image types, as we progress you will create more sophisticated user interfaces that are composed of many different types.

Image {

id: root

...

Image {

id: pole

anchors.horizontalCenter: parent.horizontalCenter anchors.bottom: parent.bottom

source: "images/pole.png"

}

Image {

id: wheel anchors.centerIn: parent

source: "images/pinwheel.png"

}

...

}

To place the pinwheel in the middle, we use a complex property called

anchor . Anchoring allows you to specify geometric relations between

parent and sibling objects. For example, place me in the center of another type ( anchors.centerIn: parent ). There are left, right, top, bottom, centerIn, fill, verticalCenter and horizontalCenter relations on both ends.

Naturally, when two or more anchors are used together, they should complement each other: it wouldn’t make sense, for instance, to anchor a type’s left side to the top of another type.

For the pinwheel, the anchoring only requires one simple anchor.

#### TIP

Sometimes you will want to make small adjustments, for example, to nudge a type slightly off-center. This can be done with anchors.horizontalCenterOffset or with anchors.verticalCenterOffset . Similar adjustment properties are also available for all the other anchors. Refer to the documentation for a full list of anchors properties.

#### TIP

Placing an image as a child type of our root type (the Image ) illustrates an important concept of a declarative language. You describe the visual appearance of the user interface in the order of layers and grouping, where the topmost layer (our background image) is drawn first and the child layers are drawn on top of it in the local coordinate system of the containing type.

To make the showcase a bit more interesting, let’s make the scene interactive. The idea is to rotate the wheel when the user presses the mouse somewhere in the scene.

We use the MouseArea type and make it cover the entire area of our root type.

Image {

id: root

...

MouseArea {

anchors.fill: parent onClicked: wheel.rotation += 90

}

...

}

The mouse area emits signals when the user clicks inside the area it covers. You can connect to this signal by overriding the onClicked function. When a signal is connected, it means that the function (or functions) it corresponds to are called whenever the signal is emitted. In this case, we say that when there’s a mouse click in the mouse area, the type whose id is wheel (i.e., the pinwheel image) should rotate by +90 degrees.

#### TIP

This technique works for every signal, with the naming convention being on + SignalName in title case. Also, all properties emit a signal when their value changes. For these signals, the naming convention is:

js

`on${property}Changed`

For example, if a width property is changed, you can observe it with

onWidthChanged: print(width) .

The wheel will now rotate whenever the user clicks, but the rotation takes place in one jump, rather than a fluid movement over time. We can achieve smooth movement using animation. An animation defines how a property

change occurs over a period of time. To enable this, we use the Animation type’s property called Behavior . The Behavior specifies an animation for a defined property for every change applied to that property. In other words, whenever the property changes, the animation is run. This is only one of many ways of doing animation in QML.

Image {

id: root Image {

id: wheel

Behavior on rotation { NumberAnimation {

duration: 250

}

}

}

}

Now, whenever the wheel’s rotation property changes, it will be animated using a NumberAnimation with a duration of 250 ms. So each 90-degree turn will take 250 ms, producing a nice smooth turn.



#### TIP

You will not actually see the wheel blurred. This is just to indicate the rotation. (A blurred wheel is in the assets folder, in case you’d like to experiment with it.)

Now the wheel looks much better and behaves nicely, as well as providing a very brief insight into the basics of how Qt Quick programming works.

# Quick Start

This chapter will introduce you to developing with Qt 6. We will show you how to install the Qt SDK and how you can create as well as run a simple *hello world* application using the Qt Creator IDE.

# Installing Qt 6 SDK

The Qt SDK includes the tools you need to build desktop or embedded [applications. You can grab the latest version from the Qt Company [](https://qt.io/) (https://qt.io) ’s homepage. There is an offline and online installer. The author personally prefers the online installer package as it allows you to install and update multiple Qt releases. This is the recommended way to start. The SDK itself has a maintenance tool, which allows you to update the SDK to the latest version.](https://qt.io/)

The Qt SDK is easy to install and comes with its own IDE for rapid development called *Qt Creator*. The IDE is a highly productive environment for Qt coding and recommended to all readers. Many developers use Qt from the command line, however, and you are free to use the code editor of your choice.

When installing the SDK, you should select the default option and ensure that at least Qt 6.2 is enabled. Then you’re ready to go.

## Update Qt

The Qt SDK comes with an own maintenance tool located under the

${install\_dir} . It allows to add and/or update Qt SDK components.

## Build from Source

[To build Qt from source you can follow the guide from the Qt Wiki [](https://wiki.qt.io/Building_Qt_6_from_Git) (https://wiki.qt.io/Building\_Qt\_6\_from\_Git) .](https://wiki.qt.io/Building_Qt_6_from_Git)

# Hello World

To test your installation, we will create a small *hello world* application. Please, open Qt Creator and create a Qt Quick UI Project ( File ‣ New File or Project ‣ Other Project ‣ Qt Quick UI Prototype ) and name the project HelloWorld .

#### TIP

The Qt Creator IDE allows you to create various types of applications. If not otherwise stated, we always use a Qt Quick UI prototype project. For a production application you would often prefer a CMake based project, but for fast prototyping this type is better suited.

#### TIP

A typical Qt Quick application is made out of a runtime called the QmlEngine which loads the initial QML code. The developer can register C++ types with the runtime to interface with the native code. These C++ types can also be bundled into a plugin and then dynamically loaded using an import statement. The qml tool is a pre-made runtime which is used directly. For the beginning, we will not cover the native side of development and focus only on the QML aspects of Qt 6. This is why we start from a prototype project.

Qt Creator creates several files for you. The HelloWorld.qmlproject file is the project file, where the relevant project configuration is stored. This file

is managed by Qt Creator, so don’t edit it yourself.

Another file, HelloWorld.qml , is our application code. Open it and try to understand what the application does before you read on.

// HelloWorld.qml

import QtQuick import QtQuick.Window

Window {

width: 640

height: 480 visible: true

title: qsTr("Hello World")

}

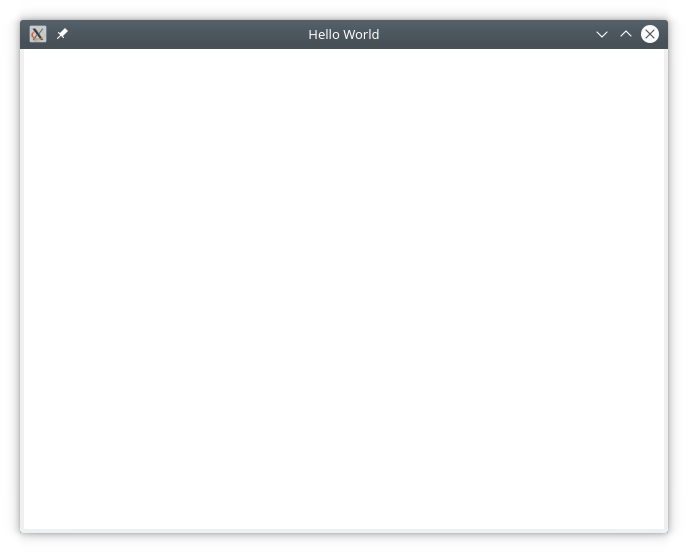
The HelloWorld.qml program is written in the QML language. We’ll discuss the QML language more in-depth in the next chapter. QML describes the user interface as a tree of hierarchical elements. In this case, a window of 640 x 480 pixels, with a window title “Hello World”.

To run the application on your own, press the



Run tool on the left side, or select Build > Run from the menu.

In the background, Qt Creator runs qml and passes your QML document as the first argument. The qml application parses the document, and launches the user interface. You should see something like this:



Qt 6 works! That means we’re ready to continue.

#### TIP

If you are a system integrator, you’ll want to have Qt SDK installed to get the latest stable Qt release, as well as a Qt version compiled from source for your specific device target.

#### TIP

Build from Scratch

If you’d like to build Qt 6 from the command line, you’ll first need to grab a copy of the code repository and build it. Visit Qt’s wiki for an up-to-date explanation of how to build Qt from git.

After a successful compilation (and 2 cups of coffee), Qt 6 will be available in the qtbase folder. Any beverage will suffice, however, we suggest coffee for best results.

If you want to test your compilation, you can now run the example with the default runtime that comes with Qt 6:

$ qtbase/bin/qml

# Application Types

This section is a run through of different application types one can write with Qt 6. It’s not limited to the selection presented here, but it will give you a better idea of what you can achieve with Qt 6 in general.

## Console Application

A console application does not provide a graphical user interface, and is usually called as part of a system service or from the command line. Qt 6 comes with a series of ready-made components which help you create cross-platform console applications very efficiently. For example, the networking file APIs, string handling, and an efficient command line parser. As Qt is a high-level API on top of C++, you get programming speed paired with execution speed. Don’t think of Qt as being *just* a UI toolkit - it has so much more to offer!

### String Handling

This first example demonstrates how you could add 2 constant strings. Admittedly, this is not a very useful application, but it gives you an idea of what a native C++ application without an event loop may look like.

// module or class includes #include <QtCore>

// text stream is text-codec aware

QTextStream cout(stdout, QIODevice::WriteOnly);

int main(int argc, char\*\* argv)

{

// avoid compiler warnings Q\_UNUSED(argc) Q\_UNUSED(argv)

QString s1("Paris"); QString s2("London");

// string concatenation

QString s = s1 + " " + s2 + "!"; cout << s << Qt::endl;

}

### Container Classes

This example adds a list, and list iteration, to the application. Qt comes with a large collection of container classes that are easy to use, and has the same API paradigms as other Qt classes.

QString s1("Hello"); QString s2("Qt"); QList<QString> list;

// stream into containers list << s1 << s2;

// Java and STL like iterators QListIterator<QString> iter(list); while(iter.hasNext()) {

cout << iter.next(); if(iter.hasNext()) { cout << " ";

}

}

cout << "!" << Qt::endl;

Here is a more advanced list function, that allows you to join a list of strings into one string. This is very handy when you need to proceed line based

text input. The inverse (string to string-list) is also possible using the

QString::split() function.

QString s1("Hello"); QString s2("Qt");

// convenient container classes QStringList list;

list << s1 << s2;

// join strings

QString s = list.join(" ") + "!"; cout << s << Qt::endl;

### File IO

In the next snippet, we read a CSV file from the local directory and loop over the rows to extract the cells from each row. Doing this, we get the table data from the CSV file in ca. 20 lines of code. File reading gives us a byte stream, to be able to convert this into valid Unicode text, we need to use the text stream and pass in the file as a lower-level stream. For writing CSV files, you would just need to open the file in write mode, and pipe the lines into the text stream.

QList<QStringList> data;

// file operations

QFile file("sample.csv"); if(file.open(QIODevice::ReadOnly)) {

QTextStream stream(&file);

// loop forever macro forever {

QString line = stream.readLine();

// test for null string 'String()' if(line.isNull()) {

break;

}

// test for empty string 'QString("")'

if(line.isEmpty()) { continue;

}

QStringList row;

// for each loop to iterate over containers foreach(const QString& cell, line.split(",")) {

row.append(cell.trimmed());

}

data.append(row);

}

}

// No cleanup necessary.

This concludes the section about console based applications with Qt.

## C++ Widget Application

Console based applications are very handy, but sometimes you need to have a graphical user interface (GUI). In addition, GUI-based applications will likely need a back-end to read/write files, communicate over the network, or keep data in a container.

In this first snippet for widget-based applications, we do as little as needed to create a window and show it. In Qt, a widget without a parent is a

window. We use a scoped pointer to ensure that the widget is deleted when the pointer goes out of scope. The application object encapsulates the Qt runtime, and we start the event loop with the exec() call. From there on, the application reacts only to events triggered by user input (such as mouse or keyboard), or other event providers, such as networking or file IO. The application only exits when the event loop is exited. This is done by calling quit() on the application or by closing the window.

When you run the code, you will see a window with the size of 240 x 120 pixels. That’s all.

include <QtGui>

int main(int argc, char\*\* argv)

{

QApplication app(argc, argv); QScopedPointer<QWidget> widget(new CustomWidget()); widget->resize(240, 120);

widget->show(); return app.exec();

}

### Custom Widgets

When you work on user interfaces, you may need to create custom-made widgets. Typically, a widget is a window area filled with painting calls.

Additionally, the widget has internal knowledge of how to handle keyboard and mouse input, as well as how to react to external triggers. To do this in Qt, we need to derive from QWidget and overwrite several functions for painting and event handling.

#pragma once include <QtWidgets>

class CustomWidget : public QWidget

{

Q\_OBJECT

public:

explicit CustomWidget(QWidget \*parent = 0); void paintEvent(QPaintEvent \*event);

void mousePressEvent(QMouseEvent \*event); void mouseMoveEvent(QMouseEvent \*event);

private:

QPoint m\_lastPos;

};

In the implementation, we draw a small border on our widget and a small rectangle on the last mouse position. This is very typical for a low-level custom widget. Mouse and keyboard events change the internal state of the widget and trigger a painting update. We won’t go into too much detail about this code, but it is good to know that you have the possibility. Qt comes with a large set of ready-made desktop widgets, so it’s likely that you don’t have to do this.

include "customwidget.h" CustomWidget::CustomWidget(QWidget \*parent) :

QWidget(parent)

{

}

void CustomWidget::paintEvent(QPaintEvent \*)

{

QPainter painter(this);

QRect r1 = rect().adjusted(10,10,-10,-10); painter.setPen(QColor("#33B5E5")); painter.drawRect(r1);

QRect r2(QPoint(0,0),QSize(40,40)); if(m\_lastPos.isNull()) {

r2.moveCenter(r1.center());

} else {

r2.moveCenter(m\_lastPos);

}

painter.fillRect(r2, QColor("#FFBB33"));

}

void CustomWidget::mousePressEvent(QMouseEvent \*event)

{

m\_lastPos = event->pos(); update();

}

void CustomWidget::mouseMoveEvent(QMouseEvent \*event)

{

m\_lastPos = event->pos(); update();

}

### Desktop Widgets

The Qt developers have done all of this for you already and provide a set of desktop widgets, with a native look on different operating systems. Your job, then, is to arrange these different widgets in a widget container into larger panels. A widget in Qt can also be a container for other widgets. This is accomplished through the parent-child relationship. This means we need to make our ready-made widgets, such as buttons, checkboxes, radio buttons, lists, and grids, children of other widgets. One way to accomplish this is displayed below.

Here is the header file for a so-called widget container.

class CustomWidget : public QWidget

{

Q\_OBJECT

public:

explicit CustomWidget(QWidget \*parent = 0); private slots:

void itemClicked(QListWidgetItem\* item); void updateItem();

private:

QListWidget \*m\_widget; QLineEdit \*m\_edit; QPushButton \*m\_button;

};

In the implementation, we use layouts to better arrange our widgets. Layout managers re-layout the widgets according to some size policies when the

container widget is re-sized. In this example, we have a list, a line edit, and a button, which are arranged vertically and allow the user to edit a list of cities. We use Qt’s signal and slots to connect sender and receiver objects.

CustomWidget::CustomWidget(QWidget \*parent) : QWidget(parent)

{

QVBoxLayout \*layout = new QVBoxLayout(this); m\_widget = new QListWidget(this);

layout->addWidget(m\_widget);

m\_edit = new QLineEdit(this); layout->addWidget(m\_edit);

m\_button = new QPushButton("Quit", this); layout->addWidget(m\_button); setLayout(layout);

QStringList cities;

cities << "Paris" << "London" << "Munich"; foreach(const QString& city, cities) {

m\_widget->addItem(city);

}

connect(m\_widget, SIGNAL(itemClicked(QListWidgetItem\*)), th connect(m\_edit, SIGNAL(editingFinished()), this, SLOT(updat connect(m\_button, SIGNAL(clicked()), qApp, SLOT(quit()));

}

void CustomWidget::itemClicked(QListWidgetItem \*item)

{

Q\_ASSERT(item);

m\_edit->setText(item->text());

}

void CustomWidget::updateItem()

{

QListWidgetItem\* item = m\_widget->currentItem(); if(item) {

item->setText(m\_edit->text());

}

}

### Drawing Shapes

Some problems are better visualized. If the problem at hand looks remotely like geometrical objects, Qt graphics view is a good candidate. A graphics view arranges simple geometrical shapes in a scene. The user can interact with these shapes, or they are positioned using an algorithm. To populate a graphics view, you need a graphics view and a graphics scene. The scene is attached to the view and is populated with graphics items.

Here is a short example. First the header file with the declaration of the view and scene.

class CustomWidgetV2 : public QWidget

{

Q\_OBJECT

public:

explicit CustomWidgetV2(QWidget \*parent = 0); private:

QGraphicsView \*m\_view; QGraphicsScene \*m\_scene;

};

In the implementation, the scene gets attached to the view first. The view is a widget and gets arranged in our container widget. In the end, we add a

small rectangle to the scene, which is then rendered on the view.

include "customwidgetv2.h"

CustomWidget::CustomWidget(QWidget \*parent) : QWidget(parent)

{

m\_view = new QGraphicsView(this); m\_scene = new QGraphicsScene(this); m\_view->setScene(m\_scene);

QVBoxLayout \*layout = new QVBoxLayout(this); layout->setMargin(0);

layout->addWidget(m\_view); setLayout(layout);

QGraphicsItem\* rect1 = m\_scene->addRect(0,0, 40, 40, Qt::No rect1->setFlags(QGraphicsItem::ItemIsFocusable|QGraphicsIte

}

## Adapting Data

Up to now, we have mostly covered basic data types and how to use widgets and graphics views. In your applications, you will often need a larger amount of structured data, which may also need to be stored persistently. Finally, the data also needs to be displayed. For this, Qt uses models. A simple model is the string list model, which gets filled with strings and then attached to a list view.

m\_view = new QListView(this); m\_model = new QStringListModel(this); view->setModel(m\_model);

QList<QString> cities;

cities << "Munich" << "Paris" << "London"; m\_model->setStringList(cities);

Another popular way to store and retrieve data is SQL. Qt comes with SQLite embedded, and also has support for other database engines (e.g. MySQL and PostgreSQL). First, you need to create your database using a schema, like this:

CREATE TABLE city (name TEXT, country TEXT); INSERT INTO city VALUES ("Munich", "Germany"); INSERT INTO city VALUES ("Paris", "France");

INSERT INTO city VALUES ("London", "United Kingdom");

To use SQL, we need to add the SQL module to our .pro file

QT += sql

And then we can open our database using C++. First, we need to retrieve a new database object for the specified database engine. With this database object, we open the database. For SQLite, it’s enough to specify the path to the database file. Qt provides some high-level database models, one of which is the table model. The table model uses a table identifier and an

optional where clause to select the data. The resulting model can be attached to a list view as with the other model before.

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE"); db.setDatabaseName("cities.db");

if(!db.open()) {

qFatal("unable to open database");

}

m\_model = QSqlTableModel(this); m\_model->setTable("city");

m\_model->setHeaderData(0, Qt::Horizontal, "City"); m\_model->setHeaderData(1, Qt::Horizontal, "Country");

view->setModel(m\_model); m\_model->select();

For a higher level model operations, Qt provides a sorting file proxy model that allows you sort, filter, and transform models.

QSortFilterProxyModel\* proxy = new QSortFilterProxyModel(this); proxy->setSourceModel(m\_model);

view->setModel(proxy);

view->setSortingEnabled(true);

Filtering is done based on the column that is to be filters, and a string as filter argument.

proxy->setFilterKeyColumn(0);

proxy->setFilterCaseSensitivity(Qt::CaseInsensitive); proxy->setFilterFixedString(QString)

The filter proxy model is much more powerful than demonstrated here. For now, it is enough to remember it exists.

!!! note

This has been an overview of the different kind of classic appl

*Coming next: Qt Quick to the rescue.*

## Qt Quick Application

There is an inherent conflict in modern software development. The user interface is moving much faster than our back-end services. In a traditional technology, you develop the so-called front-end at the same pace as the

back-end. This results in conflicts when customers want to change the user interface during a project, or develop the idea of a user interface during the project. Agile projects, require agile methods.

Qt Quick provides a declarative environment where your user interface (the front-end) is declared like HTML and your back-end is in native C++ code. This allows you to get the best of both worlds.

This is a simple Qt Quick UI below

import QtQuick Rectangle {

width: 240; height: 240 Rectangle {

width: 40; height: 40 anchors.centerIn: parent color: '#FFBB33'

}

}

The declaration language is called QML and it needs a runtime to execute it. Qt provides a standard runtime called qml . You can also write a custom runtime. For this, we need a quick view and set the main QML document as a source from C++. Then you can show the user interface.

#include <QtGui> #include <QtQml>

int main(int argc, char \*argv[])

{

QGuiApplication app(argc, argv); QQmlApplicationEngine engine("main.qml"); return app.exec();

}

Let’s come back to our earlier examples. In one example, we used a C++ city model. It would be great if we could use this model inside our declarative QML code.

To enable this, we first code our front-end to see how we would want to use a city model. In this case, the front-end expects an object named cityModel which we can use inside a list view.

import QtQuick Rectangle {

width: 240; height: 120 ListView {

width: 180; height: 120 anchors.centerIn: parent model: cityModel

delegate: Text { text: model.city }

}

}

To enable the cityModel , we can mostly re-use our previous model, and add a context property to our root context. The root context is the other root-element in the main document.

m\_model = QSqlTableModel(this);

... // some magic code QHash<int, QByteArray> roles; roles[Qt::UserRole+1] = "city";

roles[Qt::UserRole+2] = "country"; m\_model->setRoleNames(roles);

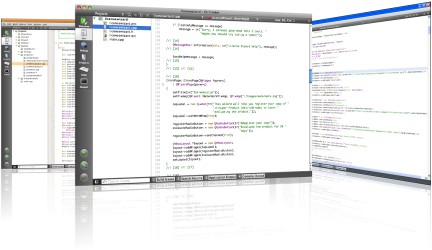
engine.rootContext()->setContextProperty("cityModel", m\_model);

# Summary

We have seen how to install the Qt SDK and how to create our first application. Then we walked you through the different application types to give you an overview of Qt, showing off some features Qt offers for application development. I hope you got a good impression that Qt is a very rich user interface toolkit and offers everything an application developer can hope for and more. Still, Qt does not lock you into specific libraries, as you can always use other libraries, or even extend Qt yourself. It is also rich when it comes to supporting different application models: console, classic desktop user interface, and touch user interface.

# Qt Creator IDE

Qt Creator is the default integrated development environment for Qt. It’s written from Qt developers for Qt developers. The IDE is available on all major desktop platforms, e.g. Windows/Mac/Linux. We have already seen customers using Qt Creator on an embedded device. Qt Creator has a lean efficient user interface and it really shines in making the developer productive. Qt Creator can be used to run your Qt Quick user interface but also to compile c++ code and this for your host system or for another device using a cross-compiler.



**WARNING**

Update screenshots!

# The User Interface

When starting Qt Creator you are greeted by the *Welcome* screen. There you will find the most important hints on how to continue inside Qt Creator and your recently used projects. You will also see the sessions list, which might be empty for you. A session is a collection of projects and configurations stored for fast access. This comes really handy when you have several customers with larger projects.

On the left side, you will see the mode-selector. The mode selectors support typical steps from a developer workflow.

 **Welcome mode**: For your orientation.

 **Edit mode**: Focus on the code

 **Design mode**: Focus on the UI design

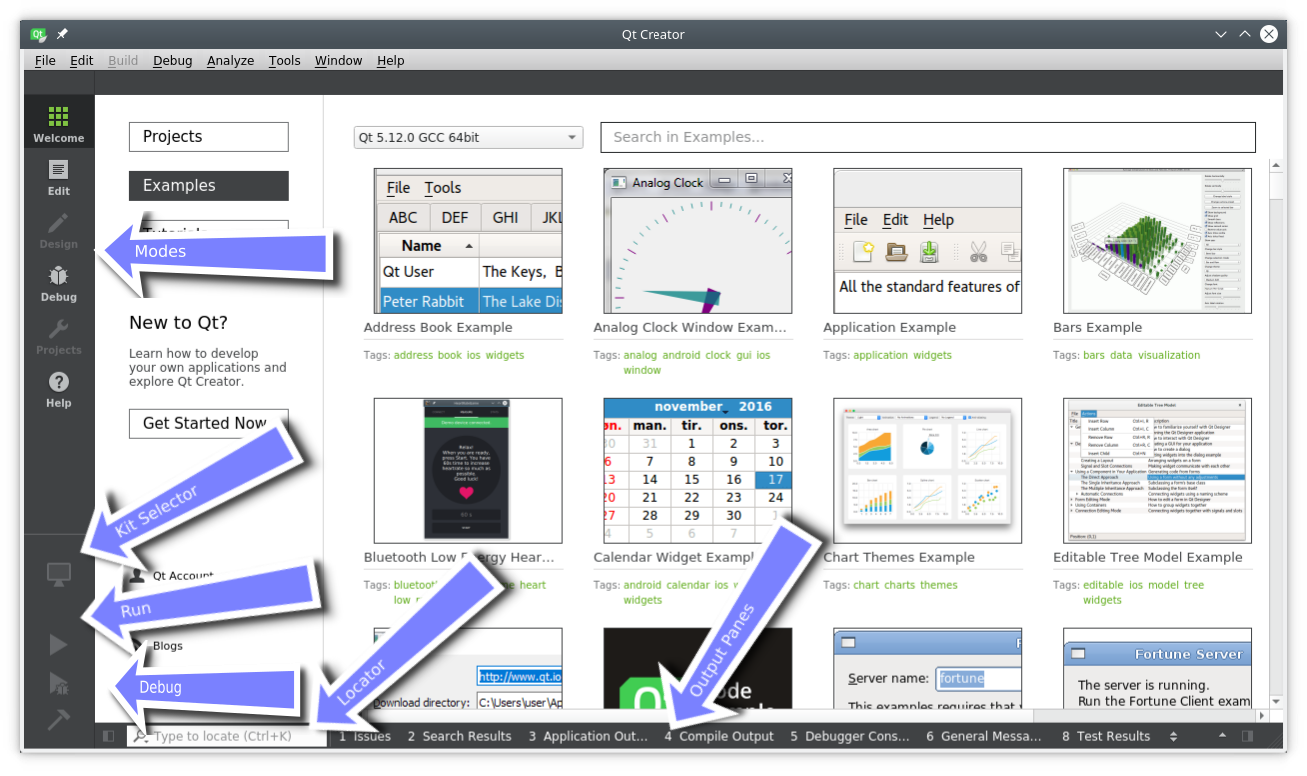
 **Debug mode**: Retrieve information about a running application

 **Projects mode**: Modify your projects run and build configuration

 **Analyze mode**: For detecting memory leaks and profiling

 **Help mode**: Easy access to the Qt documentation

Below the mode-selectors, you will find the actual project-configuration selector and the run/debug



Most of the time you will be in the edit mode with the code-editor in the

central panel. From time to time, you will visit the Projects mode when you need to configure your project. And then you press Run . Qt Creator is smart enough to ensure your project is fully built before running it.

In the bottom are the output panes for issues, application messages, compile messages, and other messages.

# Registering your Qt Kit

The Qt Kit is probably the most difficult aspect when it comes to working with Qt Creator initially. A Qt Kit is a set of a Qt version, compiler and device and some other settings. It is used to uniquely identify the combination of tools for your project build. A typical kit for the desktop would contain a C++ compiler and a Qt version (e.g. Qt 6.xx.yy) and a device (“Desktop”). After you have created a project you need to assign a kit to a project before Qt Creator can build the project. Before you are able to create a kit first you need to have a compiler installed and have a Qt version registered. A Qt version is registered by specifying the path to the qmake executable. Qt Creator then queries qmake for information required to identify the Qt version. This is also true for Qt 6 where CMake is the preferred build tool.

Adding a kit and registering a Qt version is done in the Settings ‣ Kits

entry. There you can also see which compilers are registered.

#### TIP

Please first check if your Qt Creator has already the correct Qt version registered and then ensure a Kit for your combination of compiler and Qt and device is specified. **You can not build a project without a kit.**

# Managing Projects

Qt Creator manages your source code in projects. You can create a new project by using File ‣ New File or Project . When you create a project you have many choices of application templates. Qt Creator is capable of creating desktop, embedded, mobile applications and even python projects using Qt for Python. There are templates for applications which uses Widgets or Qt Quick or even bare-bone projects just using a console. For a beginner, it is difficult to choose, so we pick three project types for you.

**Other Project / QtQuick UI Prototype**: Great for playing around with QML as there is no C++ build step involved. Mostly suitable for prototyping only.

**Applications (Qt Quick) / Qt Quick Application (Empty)**: Creates a bare C++ project with cmake support and a QML main document to render an empty window. This is the typical default starting point for all native QML application.

**Libraries / Qt Quick 2.0 Extension Plug-in**: Use this wizard to create a stub for a plug-in for your Qt Quick UI. A plug-in is used to extend Qt Quick with native elements. This is ideally to create a re-usable Qt Quick library.

**Applications (Qt) / Qt Widgets Application**: Creates a starting point for a desktop application using Qt Widgets. This would be your starting point if you plan to create a traditional C++ widgets based application.

**Applications (Qt) / Qt Console Application**: Creates a starting point for a desktop application without any user interface. This would be your starting point if you plan to create a traditional C++ command line tool using Qt C++.

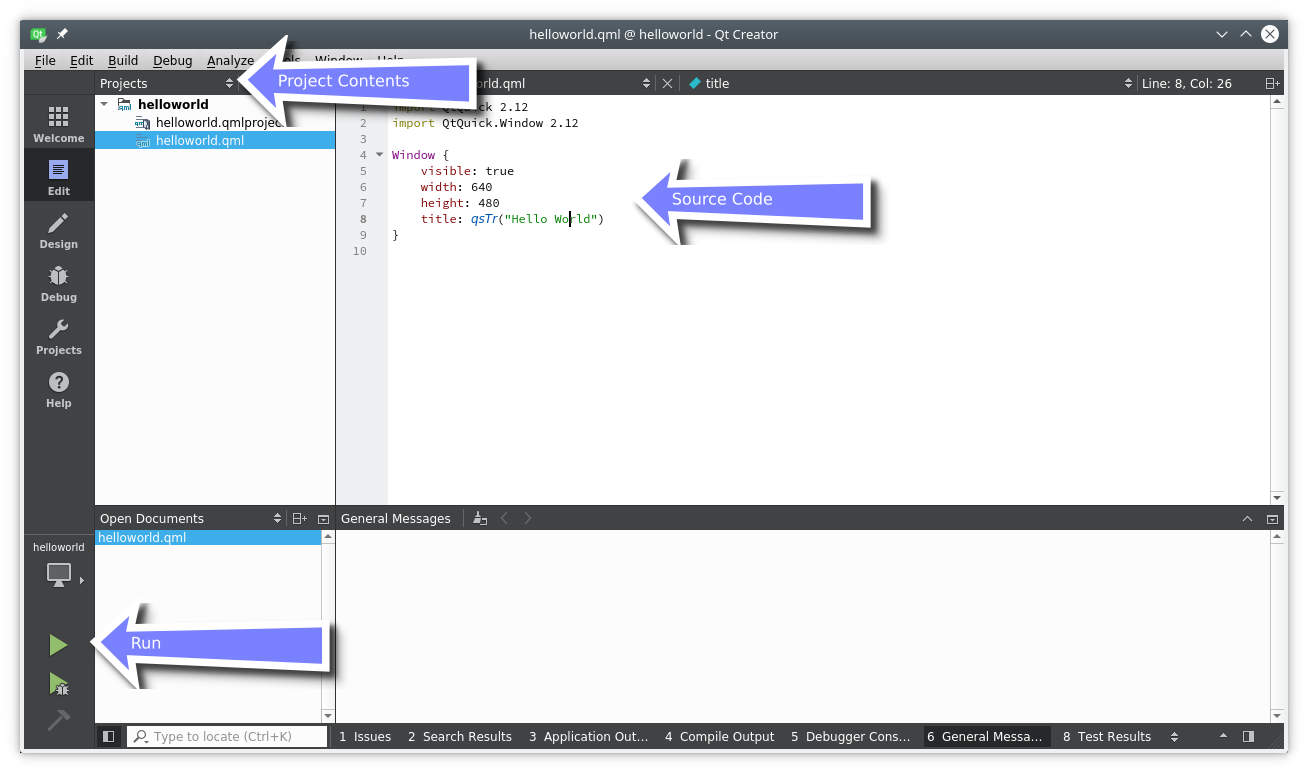
#### TIP

During the first parts of the book, we will mainly use the **QtQuick UI Prototype** type or the **Qt Quick Application**, depending on whether we also use some C++ code with Qt Quick. Later to describe some c++ aspects we will use the **Qt Console Application** type. For extending Qt Quick with our own native plug- ins we will use the *Qt Quick 2.0 Extension Plug-in* wizard type.

# Using the Editor

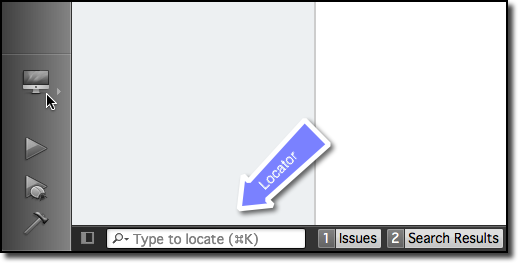
When you open a project or you just created a new project Qt Creator will switch to the edit mode. You should see on the left of your project files and in the center area the code editor. Selecting files on the left will open them in the editor.

The editor provides syntax highlighting, code-completion, and quick-fixes. Also, it supports several commands for code refactoring. When working with the editor you will have the feeling that everything reacts immediately. This is thanks to the developers of Qt Creator which made the tool feel really snappy.



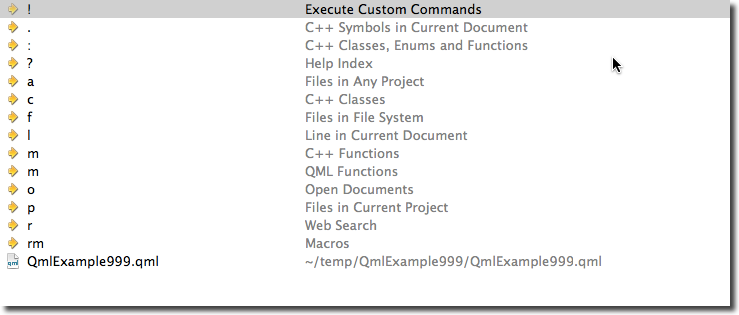
# Locator

The locator is a central component inside Qt Creator. It allows developers to navigate fast to specific locations inside the source code or inside the help. To open the locator press Ctrl+K .



A pop-up is coming from the bottom left and shows a list of options. If you just search a file inside your project just hit the first letter from the file name. The locator also accepts wild-cards, so \\*main.qml will also work.

Otherwise, you can also prefix your search to search for the specific content type.



Please try it out. For example to open the help for the QML element Rectangle open the locator and type ? rectangle . While you type the locator will update the suggestions until you found the reference you are looking for.

# Debugging

Qt Creator is an easy to use and well designed IDE to code your Qt C++ and QML projects. It has world class CMake support and is pre-configured for Qt C++ development. Due to its excellent C++ support it can also be used for any other vanilla C++ projects.

#### TIP

Hmm, I just realized I have not used debugging a lot. I hope this is a good sign. Need to ask someone to help me out here. In the [meantime have a look at the Qt Creator documentation [](http://http/doc.qt.io/qtcreator/index.html) (http://http://doc.qt.io/qtcreator/index.html) .](http://http//doc.qt.io/qtcreator/index.html)

# Shortcuts

Shortcuts are the difference between a nice-to-use editor and a

professional editor. As a professional you spend hundreds of hours in front of your application. Each shortcut which makes your work-flow faster counts. Luckily the developers of Qt Creator think the same and have added literally hundreds of shortcuts to the application.

To get started we have collection some basic shortcuts (in Windows notation):

 Ctrl+B - Build project

 Ctrl+R - Run Project

 Ctrl+Tab - Switch between open documents

 Ctrl+K - Open Locator

 Esc - Go back (hit several times and you are back in the editor)

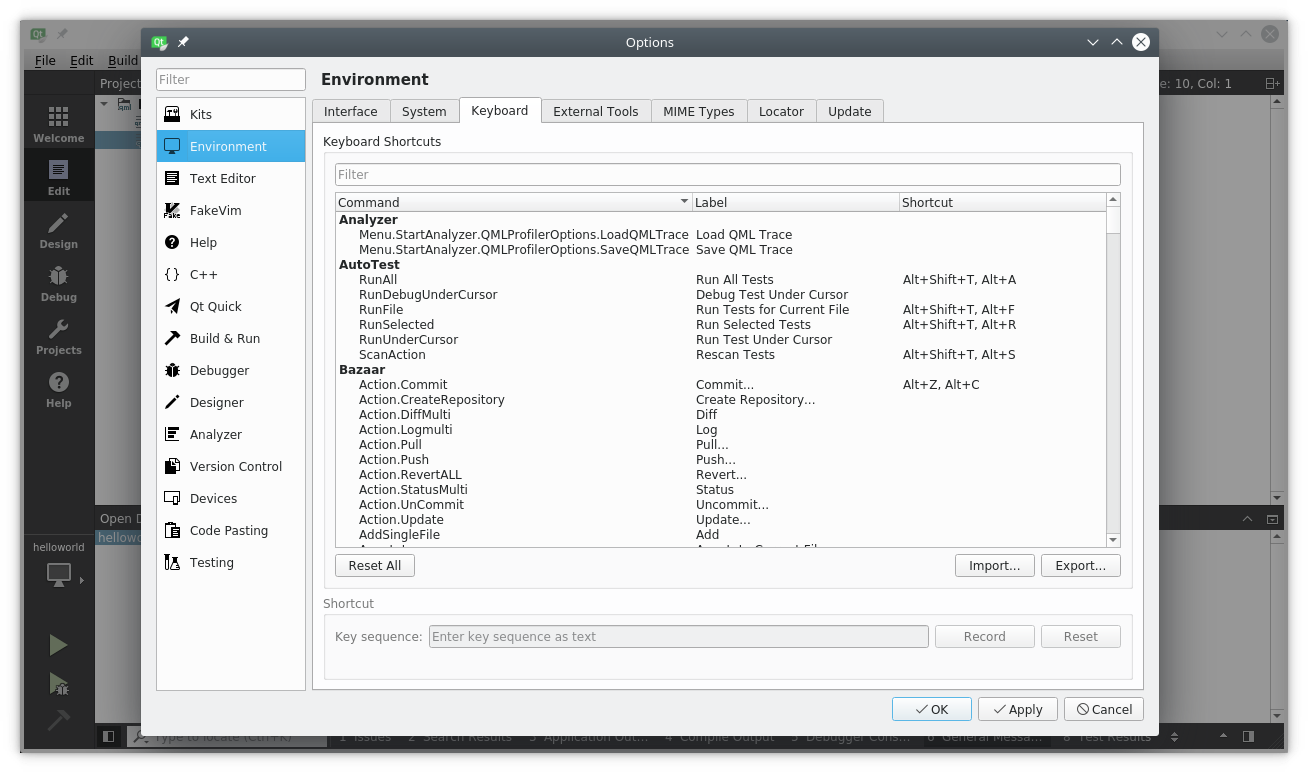
 F2 - Follow Symbol under cursor

 F4 - Switch between header and source (only useful for c++ code)

[List of Qt Creator shortcuts [](http://doc.qt.io/qtcreator/creator-keyboard-shortcuts.html) (http://doc.qt.io/qtcreator/creator-keyboard- shortcuts.html) from the documentation.](http://doc.qt.io/qtcreator/creator-keyboard-shortcuts.html)

## Configure Shortcuts

You can configure the shortcuts from inside creator using the settings dialog.



# Quick Starter

This chapter provides an overview of QML, the declarative user interface language used in Qt 6. We will discuss the QML syntax, which is a tree of elements, followed by an overview of the most important basic elements. Later we will briefly look at how to create our own elements, called components and how to transform elements using property manipulations. Towards the end, we will look at how to arrange elements together in a layout and finally have a look at elements where the user can provide input.

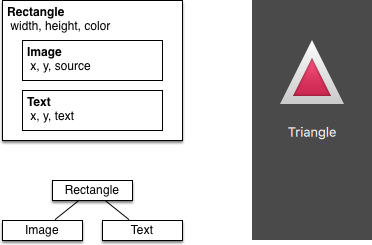
# QML Syntax

QML is a declarative language used to describe how objects relate to each other. QtQuick is a framework built on QML for building the user interface of your application. It breaks down the user interface into smaller elements, which can be combined into components. QtQuick describes the look and the behavior of these user interface elements. This user interface description can be enriched with JavaScript code to provide simple but also more complex logic. In this perspective, it follows the HTML-JavaScript pattern but QML and QtQuick are designed from the ground up to describe user interfaces, not text-documents.

In its simplest form, QtQuick lets you create a hierarchy of elements. Child elements inherit the coordinate system from the parent. An x,y coordinate is always relative to the parent.

#### TIP

QtQuick builds on QML. The QML language only knows of elements, properties, signals and bindings. QtQuick is a framework built on QML. Using default properties, the hierarchy of QtQuick elements can be constructed in an elegant way.



Let’s start with a simple example of a QML file to explain the different syntax.

// RectangleExample.qml import QtQuick

// The root element is the Rectangle Rectangle {

// name this element root id: root

// properties: <name>: <value> width: 120; height: 240

// color property color: "#4A4A4A"

// Declare a nested element (child of root) Image {

id: triangle

// reference the parent

x: (parent.width - width)/2; y: 40 source: 'assets/triangle\_red.png'

}

// Another child of root Text {

// un-named element

// reference element by id

y: triangle.y + triangle.height + 20

// reference root element width: root.width

color: 'white'

horizontalAlignment: Text.AlignHCenter text: 'Triangle'

}

}

 The import statement imports a module. An optional version in the form of <major>.<minor> can be added.

 Comments can be made using // for single line comments or /\* \*/

for multi-line comments. Just like in C/C++ and JavaScript

 Every QML file needs to have exactly one root element, like HTML  An element is declared by its type followed by { }

 Elements can have properties, they are in the form name: value

 Arbitrary elements inside a QML document can be accessed by using their id (an unquoted identifier)

 Elements can be nested, meaning a parent element can have child elements. The parent element can be accessed using the parent keyword

With the import statement you import a QML module by name. In Qt5 you had to specify a major and minor version (e.g. 2.15 ), this is now

optional in Qt6. For the book content we drop this optional version number

as normally you automatically want to choose the newest version available from your selected Qt Kit.

#### TIP

Often you want to access a particular element by id or a parent element using the parent keyword. So it’s good practice to name your root element “root” using id: root . Then you don’t have to think about how the root element is named in your QML document.

#### TIP

You can run the example using the Qt Quick runtime from the command line from your OS like this:

$ $QTDIR/bin/qml RectangleExample.qml

Where you need to replace the $QTDIR to the path to your Qt installation. The qml executable initializes the Qt Quick runtime and interprets the provided QML file.

In Qt Creator, you can open the corresponding project file and run the document RectangleExample.qml .

## Properties

Elements are declared by using their element name but are defined by using their properties or by creating custom properties. A property is a simple key-value pair, e.g. width: 100 , text: 'Greetings' , color: '#FF0000' . A property has a well-defined type and can have an initial value.

Text {

// (1) identifier id: thisLabel

// (2) set x- and y-position x: 24; y: 16

// (3) bind height to 2 \* width height: 2 \* width

// (4) custom property property int times: 24

// (5) property alias

property alias anotherTimes: thisLabel.times

// (6) set text appended by value text: "Greetings " + times

// (7) font is a grouped property font.family: "Ubuntu" font.pixelSize: 24

// (8) KeyNavigation is an attached property KeyNavigation.tab: otherLabel

// (9) signal handler for property changes onHeightChanged: console.log('height:', height)

// focus is need to receive key events focus: true

// change color based on focus value color: focus ? "red" : "black"

}

Let’s go through the different features of properties:

1. id is a very special property-like value, it is used to reference elements inside a QML file (called “document” in QML). The id is not a string type but rather an identifier and part of the QML syntax. An id needs to be unique inside a document and it can’t be reset to a different value, nor may it be queried. (It behaves much like a reference in the

C++ world.)

1. A property can be set to a value, depending on its type. If no value is given for a property, an initial value will be chosen. You need to consult the documentation of the particular element for more information about the initial value of a property.
2. A property can depend on one or many other properties. This is called *binding*. A bound property is updated when its dependent properties change. It works like a contract, in this case, the height should always be two times the width .
3. Adding new properties to an element is done using the property qualifier followed by the type, the name and the optional initial value ( property <type> <name> : <value> ). If no initial value is given, a default initial value is chosen.

#### TIP

You can also declare one property to be the default property using default keyword. If another element is created inside the element and not explicitly bound to a property, it is bound to the default property. For instance, This is used when you add child

elements. The child elements are added automatically to the default property children of type list if they are visible elements.

1. Another important way of declaring properties is using the alias keyword ( property alias <name>: <reference> ). The alias keyword allows us to forward a property of an object or an object itself

from within the type to an outer scope. We will use this technique later when defining components to export the inner properties or element ids to the root level. A property alias does not need a type, it uses the type of the referenced property or object.

1. The text property depends on the custom property times of type int. The int based value is automatically converted to a string type. The expression itself is another example of binding and results in the text being updated every time the times property changes.
2. Some properties are grouped properties. This feature is used when a property is more structured and related properties should be grouped together. Another way of writing grouped properties is font { family: "Ubuntu"; pixelSize: 24 } .
3. Some properties belong to the element class itself. This is done for global settings elements which appear only once in the application (e.g. keyboard input). The writing is <Element>.<property>: <value> .
4. For every property, you can provide a signal handler. This handler is called after the property changes. For example, here we want to be notified whenever the height changes and use the built-in console to log a message to the system.

**WARNING**

An element id should only be used to reference elements inside your document (e.g. the current file). QML provides a mechanism called "dynamic scoping", where documents loaded later on overwrite the element IDs from documents loaded earlier. This makes it possible to reference element IDs from previously loaded documents if they have not yet been overwritten. It’s like creating global variables. Unfortunately, this frequently leads to really bad code in practice, where the program depends on the order of execution. Unfortunately, this can’t be turned off. Please only use this with care; or, even better, don’t use this mechanism at all. It’s better to export the element you want to provide to the outside world using properties on the root element of your document.

## Scripting

QML and JavaScript (also known as ECMAScript) are best friends. In the *JavaScript* chapter we will go into more detail on this symbiosis. Currently, we just want to make you aware of this relationship.

Text {

id: label x: 24; y: 24

// custom counter property for space presses property int spacePresses: 0

text: "Space pressed: " + spacePresses + " times"

// (1) handler for text changes. Need to use function to ca onTextChanged: function(text) {

console.log("text changed to:", text)

}

// need focus to receive key events focus: true

// (2) handler with some JS Keys.onSpacePressed: {

increment()

}

// clear the text on escape Keys.onEscapePressed: {

label.text = ''

}

// (3) a JS function function increment() {

spacePresses = spacePresses + 1

}

}

1. The text changed handler onTextChanged prints the current text every time the text changed due to the space bar being pressed. As we use a parameter injected by the signal, we need to use the function syntax here. It's also possible to use an arrow function ( (text) => {} ), but we feel function(text) {} is more readable.
2. When the text element receives the space key (because the user pressed the space bar on the keyboard) we call a JavaScript function increment() .
3. Definition of a JavaScript function in the form of function <name> (<parameters>) { ... } , which increments our counter spacePresses . Every time spacePresses is incremented, bound properties will also be updated.

## Binding

The difference between the QML : (binding) and the JavaScript = (assignment) is that the binding is a contract and keeps true over the lifetime of the binding, whereas the JavaScript assignment ( = ) is a one time value assignment.

The lifetime of a binding ends when a new binding is set on the property or even when a JavaScript value is assigned to the property. For example, a key handler setting the text property to an empty string would destroy our increment display:

Keys.onEscapePressed: { label.text = ''

}

After pressing escape, pressing the space bar will not update the display anymore, as the previous binding of the text property (*text: “Space pressed: ” + spacePresses + ” times”*) was destroyed.

When you have conflicting strategies to change a property as in this case (text updated by a change to a property increment via a binding and text cleared by a JavaScript assignment) then you can’t use bindings! You need to use assignment on both property change paths as the binding will be destroyed by the assignment (broken contract!).

# Core Elements

Elements can be grouped into visual and non-visual elements. A visual element (like the Rectangle ) has a geometry and normally presents an area on the screen. A non-visual element (like a Timer ) provides general functionality, normally used to manipulate the visual elements.

Currently, we will focus on the fundamental visual elements, such as

Item , Rectangle , Text , Image and MouseArea . However, by using the Qt Quick Controls 2 module, it is possible to create user interfaces built from standard platform components such as buttons, labels and sliders.

## Item Element

Item is the base element for all visual elements as such all other visual elements inherits from Item . It doesn’t paint anything by itself but defines all properties which are common across all visual elements:

**Geometry** - x and y to define the top-left position, width and height for the expansion of the element, and z for the stacking order to lift elements up or down from their natural ordering.

**Layout handling** - anchors (left, right, top, bottom, vertical and horizontal center) to position elements relative to other elements with optional margins .

**Key handling** - attached Key and KeyNavigation properties to control key handling and the focus property to enable key handling in the first place.

**Transformation** - scale and rotate transformation and the generic

transform property list for *x,y,z* transformation, as well as a

transformOrigin point.

 **Visual** - opacity to control transparency, visible to show/hide elements, clip to restrain paint operations to the element boundary, and smooth to enhance the rendering quality.

 **State definition** - states list property with the supported list of states, the current state property, and the transitions list property to animate state changes.

To better understand the different properties we will try to introduce them throughout this chapter in the context of the element presented. Please remember these fundamental properties are available on every visual element and work the same across these elements.

#### TIP

The Item element is often used as a container for other elements, similar to the *div* element in HTML.

## Rectangle Element

Rectangle extends Item and adds a fill color to it. Additionally it supports borders defined by border.color and border.width . To create rounded rectangles you can use the radius property.

Rectangle {

id: rect1 x: 12; y: 12

width: 76; height: 96 color: "lightsteelblue"

}

Rectangle {

id: rect2

x: 112; y: 12

width: 76; height: 96 border.color: "lightsteelblue" border.width: 4

radius: 8

}



#### TIP

Valid color values are colors from the SVG color names (see [http://www.w3.org/TR/css3-color/#svg-color [](http://www.w3.org/TR/css3-color/#svg-color) (http://www.w3.org/TR/css3-color/#svg-color) ). You can provide colors in QML in different ways, but the most common way is an RGB string (‘#FF4444’) or as a color name (e.g. ‘white’).](http://www.w3.org/TR/css3-color/#svg-color)

A random color can be created using some JavaScript:

color: Qt.rgba( Math.random(), Math.random(), Math.rando

Besides a fill color and a border, the rectangle also supports custom gradients:

Rectangle {

id: rect1 x: 12; y: 12

width: 176; height: 96 gradient: Gradient {

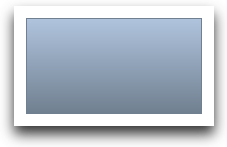
GradientStop { position: 0.0; color: "lightsteelblue" }

GradientStop { position: 1.0; color: "slategray" }

}

border.color: "slategray"

}



A gradient is defined by a series of gradient stops. Each stop has a position and a color. The position marks the position on the y-axis (0 = top, 1 = bottom). The color of the GradientStop marks the color at that position.

#### TIP

A rectangle with no *width/height* set will not be visible. This happens often when you have several rectangles width (height) depending on each other and something went wrong in your composition logic. So watch out!

#### TIP

It is not possible to create an angled gradient. For this, it’s better to use predefined images. One possibility would be to just rotate the rectangle with the gradient, but be aware the geometry of a rotated rectangle will not change and thus will lead to confusion as the geometry of the element is not the same as the visible area. From the author's perspective, it’s really better to use designed gradient images in that case.

## Text Element

To display text, you can use the Text element. Its most notable property is the text property of type string . The element calculates its initial width and height based on the given text and the font used. The font can be influenced using the font property group (e.g. font.family , font.pixelSize , …). To change the color of the text just use the color property.

Text {

text: "The quick brown fox" color: "#303030" font.family: "Ubuntu" font.pixelSize: 28

}



Text can be aligned to each side and the center using the horizontalAlignment and verticalAlignment properties. To further enhance the text rendering you can use the style and styleColor property, which allows you to render the text in outline, raised and sunken mode.

For longer text, you often want to define a *break* position like *A very … long text*, this can be achieved using the elide property. The elide property allows you to set the elide position to the left, right or middle of your text.

In case you don’t want the ‘…’ of the elide mode to appear but still want to see the full text you can also wrap the text using the wrapMode property (works only when the width is explicitly set):

Text {

width: 40; height: 120 text: 'A very long text'

// '...' shall appear in the middle elide: Text.ElideMiddle

// red sunken text styling style: Text.Sunken styleColor: '#FF4444'

// align text to the top verticalAlignment: Text.AlignTop

// only sensible when no elide mode

// wrapMode: Text.WordWrap

}

A Text element only displays the given text, and the remaining space it occupies is transparent. This means it does not render any background decoration, and so it's up to you to provide a sensible background if desired.

#### TIP

Be aware that the initial width of a Text item is dependant on the font and text string that were set. A Text element with no width set and no text will not be visible, as the initial width will be 0.

#### TIP

Often when you want to layout Text elements you need to differentiate between aligning the text inside the Text element boundary box and aligning the element boundary box itself. In the former, you want to use the horizontalAlignment and verticalAlignment properties, and in the latter case, you want to manipulate the element geometry or use anchors.

## Image Element

An Image element is able to display images in various formats (e.g. PNG , JPG , GIF , BMP , WEBP ). *For the full list of supported image* [*formats, please consult the Qt documentation [](https://doc.qt.io/qt-6/qimagereader.html#supportedImageFormats)* *(https://doc.qt.io/qt- 6/qimagereader.html#supportedImageFormats)* . Besides the source](https://doc.qt.io/qt-6/qimagereader.html#supportedImageFormats)

property to provide the image URL, it contains a fillMode which controls the resizing behavior.

Image {

x: 12; y: 12

// width: 72

// height: 72

source: "assets/triangle\_red.png"

}

Image {

x: 12+64+12; y: 12

// width: 72 height: 72/2

source: "assets/triangle\_red.png" fillMode: Image.PreserveAspectCrop clip: true

}



#### TIP

A URL can be a local path with forward slashes ( “./images/home.png” ) or a web-link (e.g. “[http://example.org/home.png [](http://example.org/home.png) (http://example.org/home.png) ”).](http://example.org/home.png)

#### TIP

Image elements using PreserveAspectCrop should also enable clipping to avoid image data being rendered outside the Image boundaries. By default clipping is disabled ( clip: false ). You need to enable clipping ( clip: true ) to constrain the painting to the elements bounding rectangle. This can be used on any visual [element, but should be used sparingly [](https://doc.qt.io/qt-6/qtquick-performance.html#clipping) (https://doc.qt.io/qt- 6/qtquick-performance.html#clipping) .](https://doc.qt.io/qt-6/qtquick-performance.html#clipping)

#### TIP

Using C++ you are able to create your own image provider using QQuickImageProvider . This allows you to create images on the fly and make use of threaded image loading.

## MouseArea Element

To interact with these elements you will often use a MouseArea . It’s a rectangular invisible item in which you can capture mouse events. The

mouse area is often used together with a visible item to execute commands when the user interacts with the visual part.

Rectangle {

id: rect1 x: 12; y: 12

width: 76; height: 96 color: "lightsteelblue" MouseArea {

id: area

width: parent.width height: parent.height

onClicked: rect2.visible = !rect2.visible

}

}

Rectangle {

id: rect2

x: 112; y: 12

width: 76; height: 96 border.color: "lightsteelblue" border.width: 4

radius: 8

}



#### TIP

This is an important aspect of Qt Quick: the input handling is separated from the visual presentation. This allows you to show the user an interface element where the actual interaction area can be larger.

#### TIP

[For more complex interaction, see Qt Quick Input Handlers [](https://doc.qt.io/qt-6/qtquickhandlers-index.html) (https://doc.qt.io/qt-6/qtquickhandlers-index.html) . They are intended to be used instead of elements such as MouseArea and Flickable and offer greater control and flexibility. The idea is to handle one interaction aspect in each handler instance instead of centralizing the handling of all events from a given source in a single element, which was the case before.](https://doc.qt.io/qt-6/qtquickhandlers-index.html)

# Components

A component is a reusable element. QML provides different ways to create components. Currently, we will look only at the simplest form - a file-based component. A file-based component is created by placing a QML element in a file and giving the file an element name (e.g. Button.qml ). You can use the component like every other element from the Qt Quick module. In our case, you would use this in your code as Button { ... } .

For example, let’s create a rectangle containing a text component and a mouse area. This resembles a simple button and doesn’t need to be more complicated for our purposes.

Rectangle { // our inlined button ui id: button

x: 12; y: 12

width: 116; height: 26 color: "lightsteelblue" border.color: "slategrey" Text {

anchors.centerIn: parent text: "Start"

}

MouseArea {

anchors.fill: parent onClicked: {

status.text = "Button clicked!"

}

}

}

Text { // text changes when button was clicked

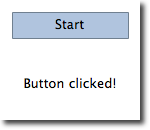
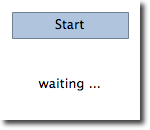
id: status x: 12; y: 76

width: 116; height: 26 text: "waiting ..."

horizontalAlignment: Text.AlignHCenter

}

The UI will look similar to this. In the first image, the UI is in its initial state, and in the second image the button has been clicked.



Now our task is to extract the button UI into a reusable component. For this, we should think about a possible API for our button. You can do this by imagining how someone else should use your button. Here’s what I came up with:

// minimal API for a button Button {

text: "Click Me"

onClicked: { /\* do something \*/ }

}

I would like to set the text using a text property and to implement my own click handler. Also, I would expect the button to have a sensible initial

size, which I can overwrite (e.g. with width: 240 for example).

To achieve this we create a Button.qml file and copy our button UI inside. Additionally, we need to export the properties a user might want to change at the root level.

// Button.qml import QtQuick Rectangle {

id: root

// export button properties property alias text: label.text signal clicked

width: 116; height: 26 color: "lightsteelblue" border.color: "slategrey"

Text {

id: label anchors.centerIn: parent text: "Start"

}

MouseArea {

anchors.fill: parent onClicked: {

root.clicked()

}

}

}

We have exported the text property and the clicked signal at the root level. Typically we name our root element root to make referencing it easier. We use the alias feature of QML, which is a way to export properties inside nested QML elements to the root level and make this available for the

outside world. It is important to know that only the root level properties can be accessed from outside this file by other components.

To use our new Button element we can simply declare it in our file. So the earlier example will become a little bit simplified.

Button { // our Button component id: button

x: 12; y: 12

text: "Start" onClicked: {

status.text = "Button clicked!"

}

}

Text { // text changes when button was clicked id: status

x: 12; y: 76

width: 116; height: 26 text: "waiting ..."

horizontalAlignment: Text.AlignHCenter

}

Now you can use as many buttons as you like in your UI by just using Button { ... } . A real button could be more complex, e.g. providing feedback when clicked or showing a nicer decoration.

#### TIP

If you want to, you could even go a step further and use an Item as a root element. This prevents users from changing the color of the button we designed, and provides us with more control over the exported API. The target should be to export a minimal API. Practically, this means we would need to replace the root Rectangle with an Item and make the rectangle a nested element in the root item.

Item {

id: root

width: 116; height: 26

property alias text: label.text signal clicked

Rectangle {

anchors.fill parent color: "lightsteelblue" border.color: "slategrey"

}

...

}

With this technique, it is easy to create a whole series of reusable components.

## Simple Transformations

A transformation manipulates the geometry of an object. QML Items can, in general, be translated, rotated and scaled. There is a simple form of these operations and a more advanced way.

Let’s start with the simple transformations. Here is our scene as our starting point.

A simple translation is done via changing the x,y position. A rotation is done using the rotation property. The value is provided in degrees (0 .. 360). A scaling is done using the scale property and a value <1 means the element is scaled down and >1 means the element is scaled up.

Rotation and scaling do not change an item's geometry: the x,y and width/height haven’t changed; only the painting instructions are transformed.

Before we show off the example I would like to introduce a little helper: the ClickableImage element. The ClickableImage is just an image with a mouse area. This brings up a useful rule of thumb - if you have copied a chunk of code three times, extract it into a component.

// ClickableImage.qml

// Simple image which can be clicked import QtQuick

Image {

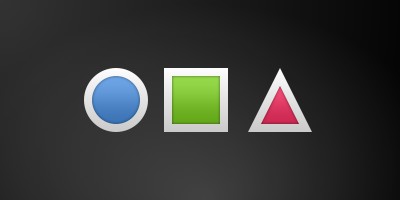
id: root signal clicked

MouseArea {

anchors.fill: parent onClicked: root.clicked()

}

}



We use our clickable image to present three objects (box, circle, triangle). Each object performs a simple transformation when clicked. Clicking the background will reset the scene.

// TransformationExample.qml import QtQuick

Item {

// set width based on given background width: bg.width

height: bg.height

Image { // nice background image id: bg

source: "assets/background.png"

}

MouseArea {

id: backgroundClicker

// needs to be before the images as order matters

// otherwise this mousearea would be before the other e

// and consume the mouse events anchors.fill: parent

onClicked: {

// reset our little scene circle.x = 84

box.rotation = 0

triangle.rotation = 0

triangle.scale = 1.0

}

}

ClickableImage { id: circle x: 84; y: 68

source: "assets/circle\_blue.png" antialiasing: true

onClicked: {

// increase the x-position on click x += 20

}

}

ClickableImage { id: box

x: 164; y: 68

source: "assets/box\_green.png" antialiasing: true

onClicked: {

// increase the rotation on click rotation += 15

}

}

ClickableImage { id: triangle x: 248; y: 68

source: "assets/triangle\_red.png" antialiasing: true

onClicked: {

// several transformations

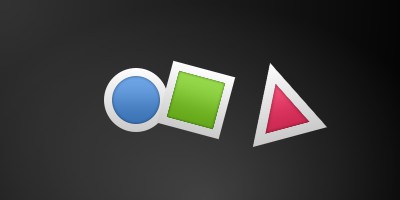
rotation += 15

scale += 0.05

}

}

// ...



The circle increments the x-position on each click and the box will rotate on each click. The triangle will rotate and scale the image up on each click, to demonstrate a combined transformation. For the scaling and rotation operation we set antialiasing: true to enable anti-aliasing, which is switched off (same as the clipping property clip ) for performance reasons. In your own work, when you see some rasterized edges in your graphics, then you should probably switch smoothing on.

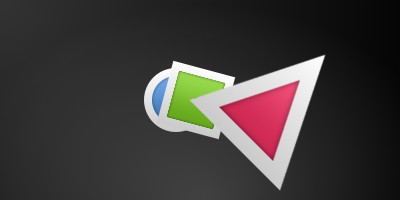
#### TIP

To achieve better visual quality when scaling images, it is recommended to scale down instead of up. Scaling an image up with a larger scaling factor will result in scaling artifacts (blurred image). When scaling an image you should consider using smooth: true to enable the usage of a higher quality filter at the cost of performance.

The background MouseArea covers the whole background and resets the object values.

#### TIP

Elements which appear earlier in the code have a lower stacking order (called z-order). If you click long enough on circle you will see it moves below box . The z-order can also be manipulated by the z property of an Item.



This is because box appears later in the code. The same applies also to mouse areas. A mouse area later in the code will overlap (and thus grab the mouse events) of a mouse area earlier in the code.

Please remember: *the order of elements in the document matters*.

# Positioning Elements

There are a number of QML elements used to position items. These are called positioners, of which the Qt Quick module provides the following: Row , Column , Grid and Flow . They can be seen showing the same contents in the illustration below.

#### TIP

Before we go into details, let me introduce some helper elements: the red, blue, green, lighter and darker squares. Each of these components contains a 48x48 pixel colorized rectangle. As a reference, here is the source code for the RedSquare :

// RedSquare.qml import QtQuick Rectangle {

width: 48

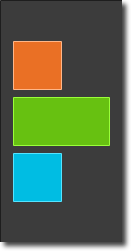
height: 48 color: "#ea7025"

border.color: Qt.lighter(color)

}

Please note the use of Qt.lighter(color) to produce a lighter border color based on the fill color. We will use these helpers in the next examples to make the source code more compact and readable. Please remember, each rectangle is initially 48x48 pixels.

The Column element arranges child items into a column by stacking them on top of each other. The spacing property can be used to distance each of the child elements from each other.



// ColumnExample.qml import QtQuick DarkSquare {

id: root width: 120

height: 240

Column {

id: column anchors.centerIn: parent spacing: 8

RedSquare { } GreenSquare { width: 96 } BlueSquare { }

}

}

The Row element places its child items next to each other, either from the left to the right, or from the right to the left, depending on the

layoutDirection property. Again, spacing is used to separate child items.



// RowExample.qml import QtQuick BrightSquare {

id: root

width: 400; height: 120

Row {

id: row anchors.centerIn: parent spacing: 20

BlueSquare { } GreenSquare { } RedSquare { }

}

}

The Grid element arranges its children in a grid. By setting the rows and columns properties, the number of rows or columns can be constrained. By not setting either of them, the other is calculated from the number of child items. For instance, setting rows to 3 and adding 6 child items will result in 2 columns. The properties flow and layoutDirection are used to control the order in which the items are

added to the grid, while spacing controls the amount of space separating the child items.



// GridExample.qml import QtQuick BrightSquare {

id: root width: 160

height: 160

Grid {

id: grid rows: 2

columns: 2 anchors.centerIn: parent spacing: 8

RedSquare { } RedSquare { } RedSquare { } RedSquare { }

}

}

The final positioner is Flow . It adds its child items in a flow. The direction of the flow is controlled using flow and layoutDirection . It can run sideways or from the top to the bottom. It can also run from left to right or in the opposite direction. As the items are added in the flow, they are wrapped to form new rows or columns as needed. In order for a flow to work, it must

have a width or a height. This can be set either directly, or though anchor layouts.



// FlowExample.qml import QtQuick BrightSquare {

id: root width: 160

height: 160

Flow {

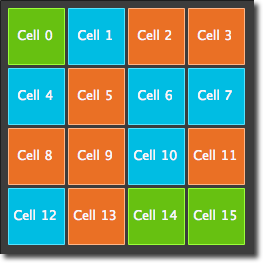
anchors.fill: parent anchors.margins: 20

spacing: 20 RedSquare { } BlueSquare { } GreenSquare { }

}

}

An element often used with positioners is the Repeater . It works like a for-loop and iterates over a model. In the simplest case a model is just a value providing the number of loops.



// RepeaterExample.qml import QtQuick DarkSquare {

id: root width: 252

height: 252

property variant colorArray: ["#00bde3", "#67c111", "#ea702

Grid{

anchors.fill: parent anchors.margins: 8

spacing: 4 Repeater {

model: 16

delegate: Rectangle {

required property int index

property int colorIndex: Math.floor(Math.random

width: 56; height: 56

color: root.colorArray[colorIndex] border.color: Qt.lighter(color)

Text {

anchors.centerIn: parent

color: "#f0f0f0"

text: "Cell " + parent.index

}

}

}

}

}

In this repeater example, we use some new magic. We define our own colorArray property, which is an array of colors. The repeater creates a series of rectangles (16, as defined by the model). For each loop, it creates the rectangle as defined by the child of the repeater. In the rectangle we chose the color by using JS math functions: Math.floor(Math.random()\*3) . This gives us a random number in the range from 0..2, which we use to select the color from our color array. As noted earlier, JavaScript is a core part of Qt Quick, and as such, the standard libraries are available to us.

A repeater injects the index property into the repeater. It contains the current loop-index. (0,1,..15). We can use this to make our own decisions based on the index, or in our case to visualize the current index with the Text element.

#### TIP

While the index property is dynamically injected into the Rectangle, it is a good practice to declare it as a required property to ease readability and help tooling. This is achieved by the required property int index line.

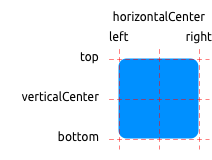
#### TIP

More advanced handling of larger models and kinetic views with dynamic delegates is covered in its own model-view chapter.

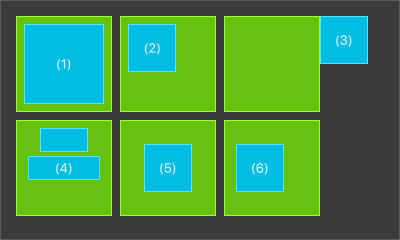
Repeaters are best used when having a small amount of static data to be presented.

# Layout Items

QML provides a flexible way to layout items using anchors. The concept of anchoring is fundamental to Item , and is available to all visual QML elements. Anchors act like a contract and are stronger than competing geometry changes. Anchors are expressions of relativeness; you always need a related element to anchor with.



An element has 6 major anchor lines ( top , bottom , left , right , horizontalCenter , verticalCenter ). Additionally, there is the baseline anchor for text in Text elements. Each anchor line comes with an offset. In the case of the top , bottom , left , and right anchors, they are called margins. For horizontalCenter , verticalCenter and baseline they are called offsets.



1. An element fills a parent element.

GreenSquare {

BlueSquare {

width: 12 anchors.fill: parent anchors.margins: 8

text: '(1)'

}

}

1. An element is left aligned to the parent.

GreenSquare {

BlueSquare {

width: 48

y: 8

anchors.left: parent.left anchors.leftMargin: 8

text: '(2)'

}

}

1. An element's left side is aligned to the parent’s right side.

GreenSquare {

BlueSquare {

width: 48

anchors.left: parent.right text: '(3)'

}

}

1. Center-aligned elements. Blue1 is horizontally centered on the parent. Blue2 is also horizontally centered, but on Blue1 , and its top is aligned to the Blue1 bottom line.

GreenSquare {

BlueSquare {

id: blue1

width: 48; height: 24

y: 8

anchors.horizontalCenter: parent.horizontalCenter

}

BlueSquare {

id: blue2

width: 72; height: 24 anchors.top: blue1.bottom anchors.topMargin: 4

anchors.horizontalCenter: blue1.horizontalCenter text: '(4)'

}

}

1. An element is centered on a parent element

GreenSquare {

BlueSquare {

width: 48 anchors.centerIn: parent

text: '(5)'

}

}

1. An element is centered with a left-offset on a parent element using horizontal and vertical center lines

GreenSquare {

BlueSquare {

width: 48

anchors.horizontalCenter: parent.horizontalCenter anchors.horizontalCenterOffset: -12 anchors.verticalCenter: parent.verticalCenter text: '(6)'

}

}

## Hidden Gems

Our squares have been magically enhanced to enable dragging. Try the example and drag around some squares. You will see that (1) can’t be dragged as it’s anchored on all sides (although you can drag the parent of (1), as it’s not anchored at all). (2) can be vertically dragged, as only the left side is anchored. The same applies to (3). (4) can only be dragged vertically, as both squares are horizontally centered. (5) is centered on the parent, and as such, can’t be dragged. The same applies to (6). Dragging an element means changing its x,y position. As anchoring is stronger than setting the x,y properties, dragging is restricted by the anchored lines. We will see this effect later when we discuss animations.

# Input Elements

We have already used the MouseArea as a mouse input element. Next, we’ll focus on keyboard input. We start off with the text editing elements: TextInput and TextEdit .

## TextInput

TextInput allows the user to enter a line of text. The element supports input constraints such as validator , inputMask , and echoMode .

// textinput.qml import QtQuick Rectangle {

width: 200

height: 80 color: "linen"

TextInput {

id: input1 x: 8; y: 8

width: 96; height: 20 focus: true

text: "Text Input 1"

}

TextInput {

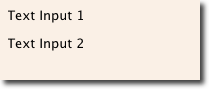
id: input2 x: 8; y: 36

width: 96; height: 20

text: "Text Input 2"

}

}



The user can click inside a TextInput to change the focus. To support switching the focus by keyboard, we can use the KeyNavigation attached property.

// textinput2.qml import QtQuick Rectangle {

width: 200

height: 80 color: "linen"

TextInput {

id: input1 x: 8; y: 8

width: 96; height: 20 focus: true

text: "Text Input 1" KeyNavigation.tab: input2

}

TextInput {

id: input2 x: 8; y: 36

width: 96; height: 20 text: "Text Input 2" KeyNavigation.tab: input1

}

}

The KeyNavigation attached property supports a preset of navigation keys where an element id is bound to switch focus on the given key press.

A text input element comes with no visual presentation beside a blinking cursor and the entered text. For the user to be able to recognize the element as an input element it needs some visual decoration; for example, a simple rectangle. When placing the TextInput inside an element you need make sure you export the major properties you want others to be able to access.

We move this piece of code into our own component called TLineEditV1

for reuse.

// TLineEditV1.qml import QtQuick Rectangle {

width: 96; height: input.height + 8 color: "lightsteelblue" border.color: "gray"

property alias text: input.text property alias input: input

TextInput {

id: input anchors.fill: parent anchors.margins: 4 focus: true

}

}

#### TIP

If you want to export the TextInput completely, you can export the element by using property alias input: input . The first input is the property name, where the 2nd input is the element id.

We then rewrite our KeyNavigation example with the new TLineEditV1

component.

Rectangle {

...

TLineEditV1 {

id: input1

...

}

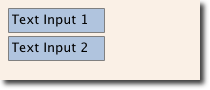
TLineEditV1 {

id: input2

...

}

}



Try the tab key for navigation. You will experience the focus does not change to input2 . The simple use of focus: true is not sufficient. The problem is that when the focus was transferred to the input2 element, the top-level item inside the TlineEditV1 (our Rectangle ) received focus, and did not forward the focus to the TextInput . To prevent this, QML offers the FocusScope .

## FocusScope

A focus scope declares that the last child element with focus: true receives the focus when the focus scope receives the focus. So it forwards the focus to the last focus-requesting child element. We will create a second version of our TLineEdit component called TLineEditV2, using a focus scope as the root element.

// TLineEditV2.qml import QtQuick FocusScope {

width: 96; height: input.height + 8 Rectangle {

anchors.fill: parent color: "lightsteelblue" border.color: "gray"

}

property alias text: input.text property alias input: input

TextInput {

id: input anchors.fill: parent anchors.margins: 4 focus: true

}

}

Our example now looks like this:

Rectangle {

...

TLineEditV2 {

id: input1

...

}

TLineEditV2 {

id: input2

...

}

}

Pressing the tab key now successfully switches the focus between the 2 components and the correct child element inside the component is focused.

## TextEdit

The TextEdit is very similar to TextInput , and supports a multi-line text edit field. It doesn’t have the text constraint properties, as this depends on querying the content size of the text ( contentHeight ,

contentWidth ). We also create our own component called TTextEdit to provide an editing background and use the focus scope for better focus forwarding.

// TTextEdit.qml import QtQuick FocusScope {

width: 96; height: 96 Rectangle {

anchors.fill: parent color: "lightsteelblue" border.color: "gray"

}

property alias text: input.text property alias input: input

TextEdit {

id: input anchors.fill: parent anchors.margins: 4 focus: true

}

}

You can use it like the TLineEdit component

// textedit.qml import QtQuick Rectangle {

width: 136

height: 120 color: "linen"

TTextEdit {

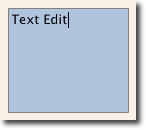
id: input x: 8; y: 8

width: 120; height: 104 focus: true

text: "Text Edit"

}

}



## Keys Element

The attached property Keys allows executing code based on certain key presses. For example, to move and scale a square, we can hook into the up, down, left and right keys to translate the element, and the plus and minus keys to scale the element.

// keys.qml import QtQuick DarkSquare {

width: 400; height: 200

GreenSquare {

id: square x: 8; y: 8

}

focus: true

Keys.onLeftPressed: square.x -= 8 Keys.onRightPressed: square.x += 8 Keys.onUpPressed: square.y -= 8 Keys.onDownPressed: square.y += 8 Keys.onPressed: function (event) {

switch(event.key) { case Qt.Key\_Plus:

square.scale += 0.2 break;

case Qt.Key\_Minus: square.scale -= 0.2

break;

}

}

}



# Advanced Techniques

## Performance of QML

QML and Javascript are interpreted languages. This means that they do not have to be processed by a compiler before being executed. Instead, they are being run inside an execution engine. However, as interpretation is a costly operation, various techniques are used to improve performance.

The QML engine uses just-in-time (JIT) compilation to improve performance. It also caches the intermediate output to avoid having to recompile. This works seamlessly for you as a developer. The only trace of this is that files ending with qmlc and jsc can be found next to the source files.

If you want to avoid the initial start-up penalty induced by the initial parsing you can also pre-compile your QML and Javascript. This requires you to put [your code into a Qt resource file, and is described in detail in the Compiling QML Ahead of Time [](https://doc.qt.io/qt-6/qtquick-deployment.html#ahead-of-time-compilation) (https://doc.qt.io/qt-6/qtquick- deployment.html#ahead-of-time-compilation) chapter in the Qt documentation.](https://doc.qt.io/qt-6/qtquick-deployment.html#ahead-of-time-compilation)

# Fluid Elements

Until now, we have mostly looked at some simple graphical elements and how to arrange and manipulate them.

This chapter is about how to make these changes more interesting by animating them.

Animations are one of the key foundations for modern, slick user interfaces, and can be employed in your user interface via states, transitions and animations. Each state defines a set of property changes and can be combined with animations on state changes. These changes are described as a transition from one state to another state.

Besides animations being used during transitions, they can also be used as standalone elements triggered by some scripted events.

# Animations

Animations are applied to property changes. An animation defines the interpolation curve from one value to another value when a property value changes. These animation curves create smooth transitions from one value to another.

An animation is defined by a series of target properties to be animated, an easing curve for the interpolation curve, and a duration. All animations in Qt Quick are controlled by the same timer and are therefore synchronized.

This improves the performance and visual quality of animations.

##### Animations control how properties change using value interpolation

This is a fundamental concept. QML is based on elements, properties, and scripting. Every element provides dozens of properties, each property is waiting to get animated by you. In the book, you will see this is a spectacular playing field.

You will catch yourself looking at some animations and just admiring their beauty, and your creative genius, too. Please remember then: *animations control property changes and every element has dozens of properties at your disposal*.

##### Unlock the power!



// AnimationExample.qml import QtQuick

Image {

id: root

source: "assets/background.png"

property int padding: 40 property int duration: 4000 property bool running: false

Image {

id: box

x: root.padding;

y: (root.height-height)/2 source: "assets/box\_green.png"

NumberAnimation on x {

to: root.width - box.width - root.padding duration: root.duration

running: root.running

}

RotationAnimation on rotation { to: 360

duration: root.duration running: root.running

}

}

MouseArea {

anchors.fill: parent onClicked: root.running = true

}

}

The example above shows a simple animation applied on the x and rotation properties. Each animation has a duration of 4000 milliseconds (msec). The animation on x moves the x-coordinate from the object gradually over to 240px. The animation on rotation runs from the current angle to 360 degrees. Both animations run in parallel and are started when the MouseArea is clicked.

You can play around with the animation by changing the to and

duration properties, or you could add another animation (for example, on the opacity or even the scale ). **Combining these it could look like the object is disappearing into deep space. Try it out!**

## Animation Elements

There are several types of animation elements, each optimized for a specific use case. Here is a list of the most prominent animations:

PropertyAnimation - Animates changes in property values

NumberAnimation - Animates changes in qreal-type values

ColorAnimation - Animates changes in color values

 RotationAnimation - Animates changes in rotation values

Besides these basic and widely used animation elements, Qt Quick also provides more specialized animations for specific use cases:

PauseAnimation - Provides a pause for an animation

SequentialAnimation - Allows animations to be run sequentially

ParallelAnimation - Allows animations to be run in parallel

AnchorAnimation - Animates changes in anchor values

ParentAnimation - Animates changes in parent values

SmoothedAnimation - Allows a property to smoothly track a value

SpringAnimation - Allows a property to track a value in a spring-like motion

PathAnimation - Animates an item alongside a path

 Vector3dAnimation - Animates changes in QVector3d values

Later we will learn how to create a sequence of animations. While working on more complex animations, there is sometimes a need to change a property or to run a script during an ongoing animation. For this Qt Quick offers the action elements, which can be used everywhere where the other animation elements can be used:

PropertyAction - Specifies immediate property changes during animation

 ScriptAction - Defines scripts to be run during an animation

The major animation types will be discussed in this chapter using small, focused examples.

## Applying Animations

Animation can be applied in several ways:

**Animation on property** - runs automatically after the element is fully loaded

**Behavior on property** - runs automatically when the property value changes

 **Standalone Animation** - runs when the animation is explicitly started using start() or running is set to true (e.g. by a property binding)

*Later we will also see how animations can be used inside state transitions.*

## Clickable Image V2

To demonstrate the usage of animations we reuse our ClickableImage component from an earlier chapter and extended it with a text element.

// ClickableImageV2.qml

// Simple image which can be clicked import QtQuick

Item {

id: root

width: container.childrenRect.width height: container.childrenRect.height property alias text: label.text property alias source: image.source signal clicked

Column {

id: container Image {

id: image

}

Text {

id: label

width: image.width horizontalAlignment: Text.AlignHCenter wrapMode: Text.WordWrap

color: "#ececec"

}

}

MouseArea {

anchors.fill: parent onClicked: root.clicked()

}

}

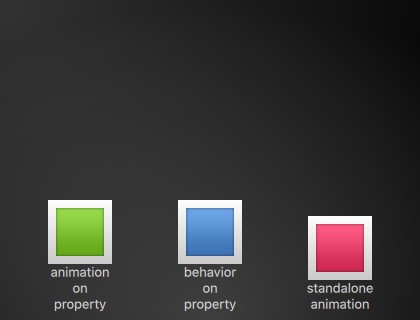
To organize the element below the image we used a Column positioner and calculated the width and height based on the column’s childrenRect property. We exposed text and image source properties, and a clicked signal. We also wanted the text to be as wide as the image, and for it to wrap. We achieve the latter by using the Text element's wrapMode property.

##### Parent/child geometry dependency

Due to the inversion of the geometry-dependency (parent geometry depends on child geometry), we can’t set a width / height on the ClickableImageV2, as this will break our width / height binding.

You should prefer the child’s geometry to depend on the parent’s geometry if the item is more like a container for other items and should adapt to the parent's geometry.

### The objects ascending



The three objects are all at the same y-position ( y=200 ). They all need to travel to y=40 , each of them using a different method with different side- effects and features.

### First object

The first object travels using the Animation on <property> strategy. The animation starts immediately.

ClickableImageV2 { id: greenBox

x: 40; y: root.height-height source: "assets/box\_green.png" text: qsTr("animation on property") NumberAnimation on y {

to: 40; duration: 4000

}

}

When an object is clicked, its y-position is reset to the start position, and this applies to all of the objects. On the first object, the reset does not have

any effect as long as the animation is running.

This can be visually disturbing, as the y-position is set to a new value for a fraction of a second before the animation starts. *Such competing property changes should be avoided*.

### Second object

The second object travels using a Behavior on animation. This behavior tells the property it should animate each change in value. The behavior can be disabled by setting enabled: false on the Behavior element.

ClickableImageV2 { id: blueBox

x: (root.width-width)/2; y: root.height-height source: "assets/box\_blue.png"

text: qsTr("behavior on property") Behavior on y {

NumberAnimation { duration: 4000 }

}

onClicked: y = 40

// random y on each click

// onClicked: y = 40 + Math.random() \* (205-40)

}

The object will start traveling when you click it (its y-position is then set to 40). Another click has no influence, as the position is already set.

You could try to use a random value (e.g. 40 + (Math.random() \\* (205- 40) ) for the y-position. You will see that the object will always animate to the new position and adapt its speed to match the 4 seconds to the destination defined by the duration of the animation.

### Third object

The third object uses a standalone animation. The animation is defined as its own element and can be almost anywhere in the document.

ClickableImageV2 { id: redBox

x: root.width-width-40; y: root.height-height source: "assets/box\_red.png"

onClicked: anim.start()

// onClicked: anim.restart() text: qsTr("standalone animation")

NumberAnimation { id: anim target: redBox properties: "y" to: 40

duration: 4000

}

}

The click will start the animation using the animation's start() function. Each animation has start(), stop(), resume(), and restart() functions. The animation itself contains much more information than the other animation types earlier.

We need to define the target , which is the element to be animated, along with the names of the properties that we want to animate. We also need to define a to value, and, in this case, a from value, which allows a restart of the animation.



A click on the background will reset all objects to their initial position. The first object cannot be restarted except by re-starting the program which triggers the re-loading of the element.

##### Other ways to control Animations

Another way to start/stop an animation is to bind a property to the running property of an animation. This is especially useful when the user-input is in control of properties:

NumberAnimation {

// [...]

// animation runs when mouse is pressed running: area.pressed

}

MouseArea {

id: area

}

## Easing Curves

The value change of a property can be controlled by an animation. Easing attributes allow influencing the interpolation curve of a property change.

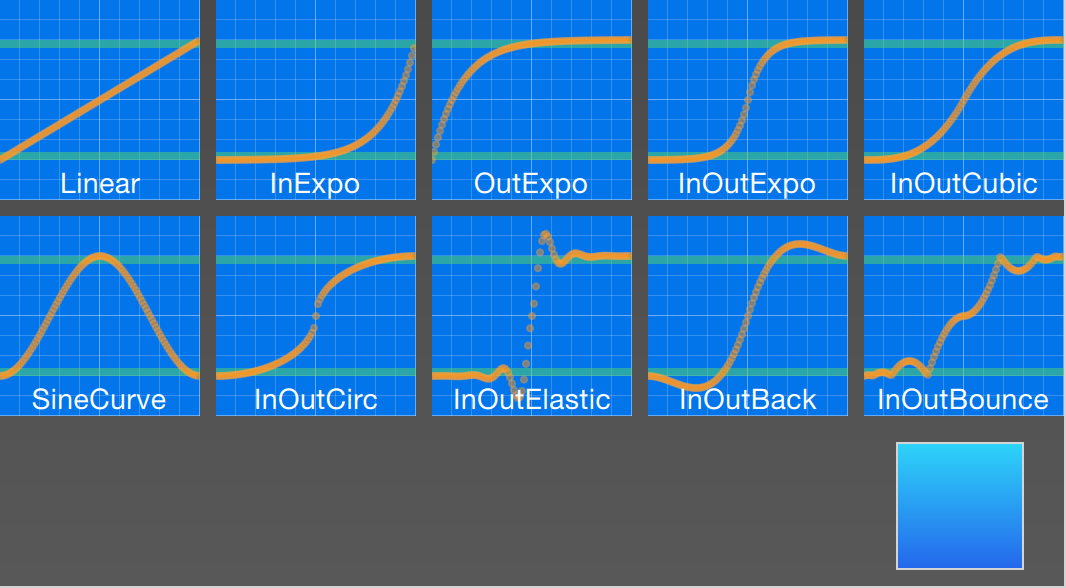
All animations we have defined by now use a linear interpolation because the initial easing type of an animation is Easing.Linear . It’s best visualized with a small plot, where the y-axis is the property to be animated and the x-axis is the time (*duration*). A linear interpolation would draw a straight line from the from value at the start of the animation to the to value at the end of the animation. So the easing type defines the curve of change.

Easing types should be carefully chosen to support a natural fit for a moving object. For example, when a page slides out, the page should initially slide out slowly and then gain momentum to finally slide out at high speed, similar to turning the page of a book.

##### Animations should not be overused.

As with other aspects of UI design, animations should be designed carefully to support the UI flow, not dominate it. The eye is very sensitive to moving objects and animations can easily distract the user.

In the next example, we will try some easing curves. Each easing curve is displayed by a clickable image and, when clicked, will set a new easing type on the square animation and then trigger a restart() to run the animation with the new curve.



The code for this example was made a little bit more complicated. We first create a grid of EasingTypes and a Box which is controlled by the easing types. An easing type just displays the curve which the box shall use for its animation. When the user clicks on an easing curve the box moves in a direction according to the easing curve. The animation itself is a standalone animation with the target set to the box and configured for x- property animation with a duration of 2 seconds.

#### TIP

The internals of the EasingType renders the curve in real time, and the interested reader can look it up in the EasingCurves example.

// EasingCurves.qml import QtQuick

import QtQuick.Layouts

Rectangle {

id: root

width: childrenRect.width height: childrenRect.height

color: '#4a4a4a' gradient: Gradient {

GradientStop { position: 0.0; color: root.color } GradientStop { position: 1.0; color: Qt.lighter(root.co

}

ColumnLayout { Grid {

spacing: 8

columns: 5 EasingType {

easingType: Easing.Linear title: 'Linear' onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InExpo title: "InExpo" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.OutExpo title: "OutExpo" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutExpo title: "InOutExpo" onClicked: {

animation.easing.type = easingType

box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutCubic title: "InOutCubic" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.SineCurve title: "SineCurve" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutCirc title: "InOutCirc" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutElastic title: "InOutElastic" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutBack title: "InOutBack" onClicked: {

animation.easing.type = easingType

box.toggle = !box.toggle

}

}

EasingType {

easingType: Easing.InOutBounce title: "InOutBounce" onClicked: {

animation.easing.type = easingType box.toggle = !box.toggle

}

}

}

Item {

height: 80 Layout.fillWidth: true Box {

id: box

property bool toggle

x: toggle ? 20 : root.width - width - 20 anchors.verticalCenter: parent.verticalCenter gradient: Gradient {

GradientStop { position: 0.0; color: "#2ed5 GradientStop { position: 1.0; color: "#2467

}

Behavior on x { NumberAnimation {

id: animation duration: 500

}

}

}

}

}

}

Please play with the example and observe the change of speed during an animation. Some animations feel more natural for the object and some feel irritating.

Besides the duration and easing.type , you are able to fine-tune animations. For example, the general PropertyAnimation type (from which most animations inherit) additionally supports easing.amplitude , easing.overshoot , and easing.period properties, which allow you to fine-tune the behavior of particular easing curves.

[Not all easing curves support these parameters. Please consult the easing table [](http://doc.qt.io/qt-6/qml-qtquick-propertyanimation.html#easing-prop) (http://doc.qt.io/qt-6/qml-qtquick-propertyanimation.html#easing- prop) from the PropertyAnimation documentation to check if an easing parameter has an influence on an easing curve.](http://doc.qt.io/qt-6/qml-qtquick-propertyanimation.html#easing-prop)

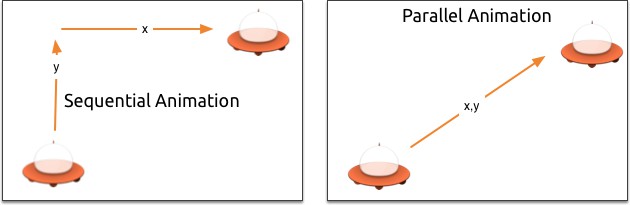
##### Choose the right Animation

Choosing the right animation for the element in the user interface context is crucial for the outcome. Remember the animation shall support the UI flow; not irritate the user.

## Grouped Animations

Often animations will be more complex than just animating one property. You might want to run several animations at the same time or one after another or even execute a script between two animations.

For this, grouped animations can be used. As the name suggests, it’s possible to group animations. Grouping can be done in two ways: parallel or sequential. You can use the SequentialAnimation or the ParallelAnimation element, which act as animation containers for other animation elements. These grouped animations are animations themselves and can be used exactly as such.



### Parallel animations

All direct child animations of a parallel animation run in parallel when started. This allows you to animate different properties at the same time.

// ParallelAnimationExample.qml import QtQuick

BrightSquare { id: root

property int duration: 3000 property Item ufo: ufo

width: 600

height: 400

Image {

anchors.fill: parent

source: "assets/ufo\_background.png"

}

ClickableImageV3 { id: ufo

x: 20; y: root.height-height text: qsTr('ufo')

source: "assets/ufo.png" onClicked: anim.restart()

}

ParallelAnimation { id: anim NumberAnimation {

target: ufo properties: "y" to: 20

duration: root.duration

}

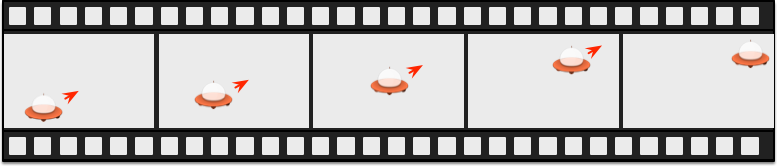
NumberAnimation { target: ufo properties: "x" to: 160

duration: root.duration

}

}

}



### Sequential animations

A sequential animation runs each child animation in the order in which it is declared: top to bottom.

// SequentialAnimationExample.qml import QtQuick

BrightSquare { id: root

property int duration: 3000 property Item ufo: ufo

width: 600

height: 400

Image {

anchors.fill: parent

source: "assets/ufo\_background.png"

}

ClickableImageV3 { id: ufo

x: 20; y: root.height-height text: qsTr('rocket') source: "assets/ufo.png" onClicked: anim.restart()

}

SequentialAnimation { id: anim NumberAnimation {

target: ufo properties: "y" to: 20

// 60% of time to travel up duration: root.duration \* 0.6

}

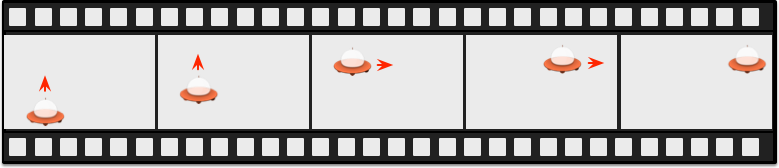
NumberAnimation { target: ufo properties: "x" to: 400

// 40% of time to travel sideways duration: root.duration \* 0.4

}

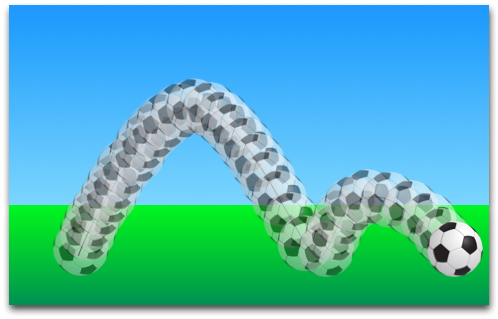
}

}



### Nested animations

Grouped animations can also be nested. For example, a sequential animation can have two parallel animations as child animations, and so on. We can visualize this with a soccer ball example. The idea is to throw a ball from left to right and animate its behavior.



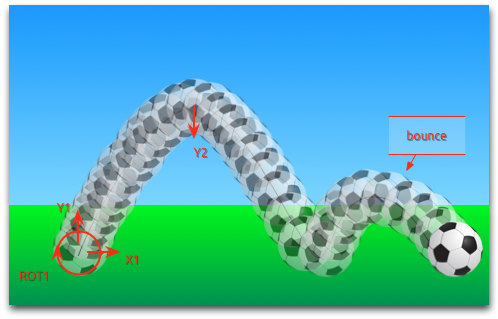
To understand the animation we need to dissect it into the integral transformations of the object. We need to remember that animations animate property changes. Here are the different transformations:

An x-translation from left-to-right ( X1 )

A y-translation from bottom to top ( Y1 ) followed by a translation from up to down ( Y2 ) with some bouncing

 A rotation of 360 degrees over the entire duration of the animation ( ROT1 )

The whole duration of the animation should take three seconds.



We start with an empty item as the root element of the width of 480 and height of 300.

import QtQuick Item {

id: root

property int duration: 3000 width: 480

height: 300

// [...]

}

We have defined our total animation duration as a reference to better synchronize the animation parts.

The next step is to add the background, which in our case are 2 rectangles with green and blue gradients.

Rectangle {

id: sky

width: parent.width height: 200 gradient: Gradient {

GradientStop { position: 0.0; color: "#0080FF" } GradientStop { position: 1.0; color: "#66CCFF" }

}

}

Rectangle {

id: ground anchors.top: sky.bottom

anchors.bottom: root.bottom width: parent.width gradient: Gradient {

GradientStop { position: 0.0; color: "#00FF00" } GradientStop { position: 1.0; color: "#00803F" }

}

}



The upper blue rectangle takes 200 pixels of the height and the lower one is anchored to the bottom of the sky and to the bottom of the root element.

Let’s bring the soccer ball onto the green. The ball is an image, stored under “assets/soccer\_ball.png”. For the beginning, we would like to position it in the lower left corner, near the edge.

Image {

id: ball

x: 0; y: root.height-height source: "assets/soccer\_ball.png"

MouseArea {

anchors.fill: parent onClicked: {

ball.x = 0

ball.y = root.height-ball.height ball.rotation = 0 anim.restart()

}

}

}



The image has a mouse area attached to it. If the ball is clicked, the position of the ball will reset and the animation is restarted.

Let’s start with a sequential animation for the two y translations first.

SequentialAnimation { id: anim NumberAnimation {

target: ball

properties: "y" to: 20

duration: root.duration \* 0.4

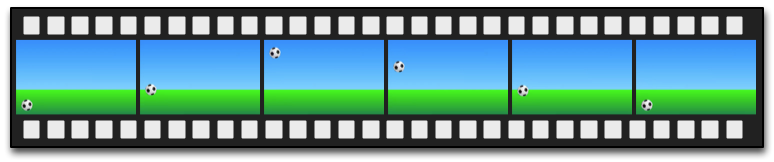
}

NumberAnimation { target: ball properties: "y" to: 240

duration: root.duration \* 0.6

}

}



This specifies that 40% of the total animation duration is the up animation and 60% the down animation, with each animation running after the other in sequence. The transformations are animated on a linear path but there is no curving currently. Curves will be added later using the easing curves, at the moment we’re concentrating on getting the transformations animated.

Next, we need to add the x-translation. The x-translation shall run in parallel with the y-translation, so we need to encapsulate the sequence of y- translations into a parallel animation together with the x-translation.

ParallelAnimation { id: anim

SequentialAnimation {

// ... our Y1, Y2 animation

}

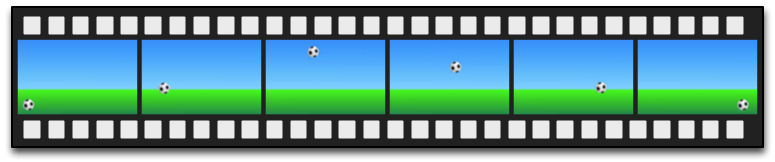
NumberAnimation { // X1 animation target: ball

properties: "x" to: 400

duration: root.duration

}

}



In the end, we would like the ball to be rotating. For this, we need to add another animation to the parallel animation. We choose RotationAnimation , as it’s specialized for rotation.

ParallelAnimation { id: anim

SequentialAnimation {

// ... our Y1, Y2 animation

}

NumberAnimation { // X1 animation

// X1 animation

}

RotationAnimation { target: ball properties: "rotation" to: 720

duration: root.duration

}

}

That’s the whole animation sequence. The one thing that's left is to provide the correct easing curves for the movements of the ball. For the *Y1* animation, we use a Easing.OutCirc curve, as this should look more like a circular movement. *Y2* is enhanced using an Easing.OutBounce to give the ball its bounce, and the bouncing should happen at the end (try with Easing.InBounce and you will see that the bouncing starts right away).

The *X1* and *ROT1* animation are left as-is, with a linear curve. Here is the final animation code for your reference:

ParallelAnimation { id: anim

SequentialAnimation { NumberAnimation {

target: ball properties: "y" to: 20

duration: root.duration \* 0.4 easing.type: Easing.OutCirc

}

NumberAnimation { target: ball properties: "y"

to: root.height-ball.height duration: root.duration \* 0.6 easing.type: Easing.OutBounce

}

}

NumberAnimation { target: ball properties: "x"

to: root.width-ball.width duration: root.duration

}

RotationAnimation { target: ball properties: "rotation" to: 720

duration: root.duration

}

}

# States and Transitions

Often parts of a user interface can be described in states. A state defines a set of property changes and can be triggered by a certain condition.

Additionally, these state switches can have a transition attached which defines how these changes should be animated or any additional actions that shall be applied. Actions can also be applied when a state is entered.

## States

You define states in QML with the State element, which needs to be bound to the states array of any item element.

A state is identified through a state name, and in its simplest form, consists of a series of property changes on elements. The default state is defined by the initial properties of the element and is named "" (an empty string).

Item {

id: root states: [

State {

name: "go" PropertyChanges { ... }

},

State {

name: "stop" PropertyChanges { ... }

}

]

}

A state is changed by assigning a new state name to the state property of the element in which the states are defined.

##### Control states using when

Another way to control states is using the when property of the State element. The when property can be set to an expression that evaluates to true when the state should be applied.

Item {

id: root states: [

...

]

Button {

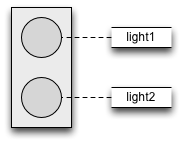
id: goButton

...

onClicked: root.state = "go"

}

}



For example, a traffic light might have two signaling lights. The upper one signaling stop with a red color and the lower one signaling go with a green color. In this example, both lights should not shine at the same time. Let’s have a look at the state chart diagram.



When the system is switched on, it automatically goes into the stop mode as the default state. The stop state changes light1 to red and light2 to black (off).

An external event can now trigger a state switch to the "go" state. In the go state, we change the color properties from light1 to black (off) and light2 to green to indicate the pedestrians may now cross.

To realize this scenario we start sketching our user interface for the 2 lights. For simplicity, we use 2 rectangles with the radius set to the half of the width (and the width is the same as the height, which means it’s a square).

Rectangle {

id: light1 x: 25; y: 15

width: 100; height: width radius: width / 2

color: root.black

border.color: Qt.lighter(color, 1.1)

}

Rectangle {

id: light2 x: 25; y: 135

width: 100; height: width radius: width/2

color: root.black

border.color: Qt.lighter(color, 1.1)

}

As defined in the state chart we want to have two states: one being the "go" state and the other the "stop" state, where each of them changes the traffic light's respective color to red or green. We set the state property to stop to ensure the initial state of our traffic light is the stop state.

##### Initial state

We could have achieved the same effect with only a "go" state and no explicit "stop" state by setting the color of light1 to red and the color of light2 to black. The initial state "" defined by the initial property values would then act as the "stop" state.

state: "stop" states: [

State {

name: "stop"

PropertyChanges { target: light1; color: root.red } PropertyChanges { target: light2; color: root.black }

},

State {

name: "go"

PropertyChanges { target: light1; color: root.black } PropertyChanges { target: light2; color: root.green }

}

]

Using PropertyChanges { target: light2; color: "black" } is not really required in these examples as the initial color of light2 is already black. In a state, it’s only necessary to describe how the properties shall change from their default state (and not from the previous state).

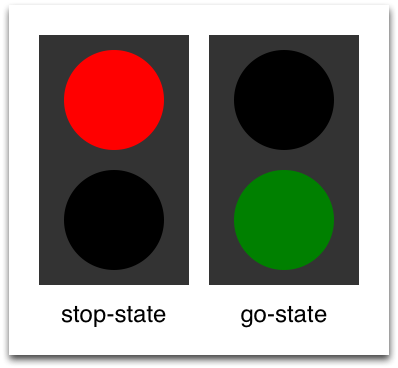
A state change is triggered using a mouse area which covers the whole traffic light and toggles between the go- and stop-state when clicked.

MouseArea {

anchors.fill: parent

onClicked: parent.state = (parent.state == "stop" ? "go" :

}



We are now able to successfully change the state of the traffic lamp. To make the UI more appealing and natural, we should add some transitions with animation effects. A transition can be triggered by a state change.

##### Using scripting

It’s possible to create similar logic using scripting instead of QML states. However, QML is a better language than JavaScript for describing user interfaces. Where possible, aim to write declarative code instead of imperative code.

## Transitions

A series of transitions can be added to every item. A transition is executed by a state change.

You can define on which state change a particular transition can be applied using the from: and to: properties. These two properties act like a filter: when the filter is true the transition will be applied. You can also use the wildcard “\*”, which means “any state”.

For example, from: "\*"; to: "\*" means "from any state to any other state", and is the default value for from and to . This means the transition will be applied to every state switch.

For this example, we would like to animate the color changes when switching state from “go” to “stop”. For the other reversed state change (“stop” to “go”) we want to keep an immediate color change and don’t apply a transition.

We restrict the transition with the from and to properties to filter only the state change from “go” to “stop”. Inside the transition, we add two color animations for each light, which shall animate the property changes defined in the state description.

transitions: [ Transition {

from: "stop"; to: "go"

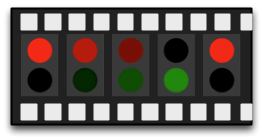
// from: "\*"; to: "\*"

ColorAnimation { target: light1; properties: "color"; d ColorAnimation { target: light2; properties: "color"; d

}

]

You can change the state though clicking the UI. The state is applied immediately and will also change the state while a transition is running. So, try to click the UI while the state is in the transition from “stop” to “go”. You will see the change will happen immediately.



You could play around with this UI by, for example, scaling the inactive light down to highlight the active light.

For this, you would need to add another property change for scaling to the states and also handle the animation for the scaling property in the transition.

Another option would be to add an “attention” state where the lights are blinking yellow. For this, you would need to add a sequential animation to the transition for one second going to yellow (“to” property of the animation and one second going to “black”).

Maybe you would also want to change the easing curve to make it more visually appealing.

# Advanced Techniques

Nothing advanced here 😃

# UI Controls

This chapter shows how to use the Qt Quick Controls module. Qt Quick Controls are used to create advanced user interfaces built from standard components such as buttons, labels, sliders and so on.

Qt Quick Controls can be arranged using the layout module and are easy to style. Also we will look into the various styles for the different platforms before diving into custom styling.

# Introduction to Controls

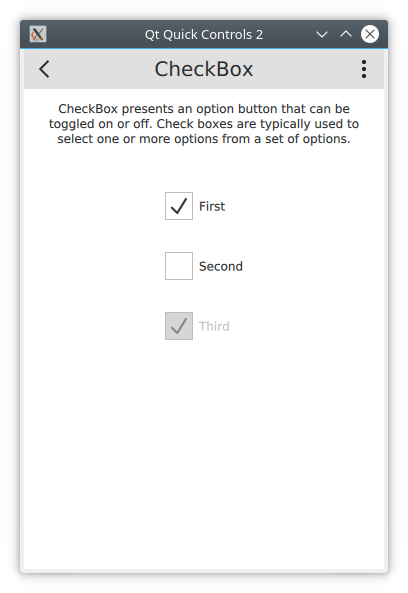
Using Qt Quick from scratch gives you primitive graphical and interaction elements from which you can build your user interfaces. Using Qt Quick Controls you start from a slightly more structured set of controls to build from.

The controls range from simple text labels and buttons to more complex ones such as sliders and dials. These elements are handy if you want to create a user interface based on classic interaction patterns, as they provide a good foundation to stand on.

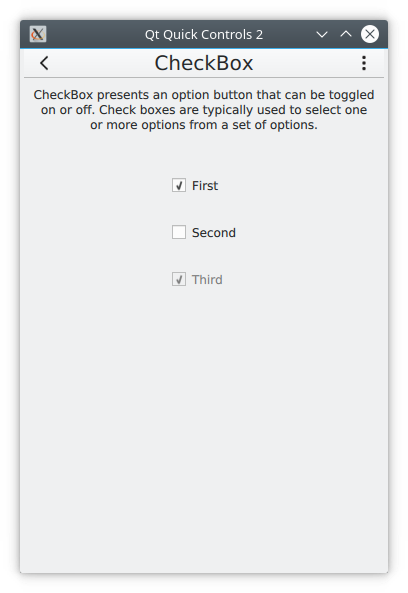
Qt Quick Controls come with a number of styles out of the box that are shown in the table below. The *Basic* style is a basic flat style. The *Universal* style is based on the Microsoft Universal Design Guidelines, while *Material* is based on Google’s Material Design Guidelines, and the *Fusion* style is a desktop-oriented style.

Some of the styles can be tweaked by modifying palettes. The *Imagine* style is based on image assets, this allows a graphical designer to create a new style without writing any code at all, not even for palette colour codes.

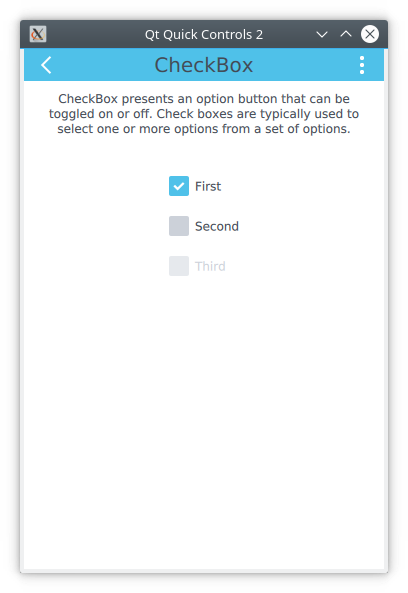
Basic



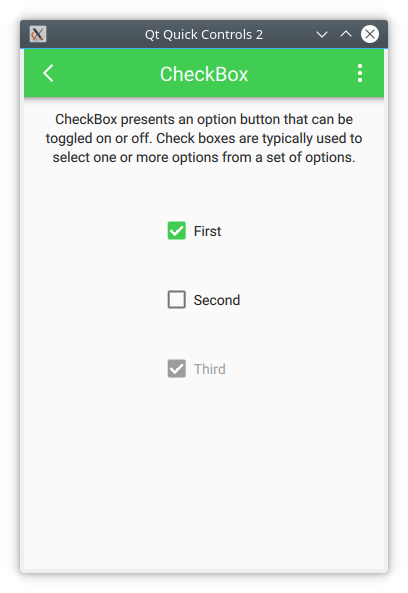
Fusion



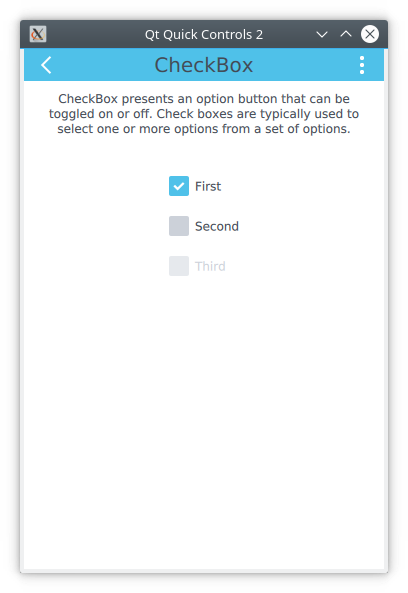
macOS



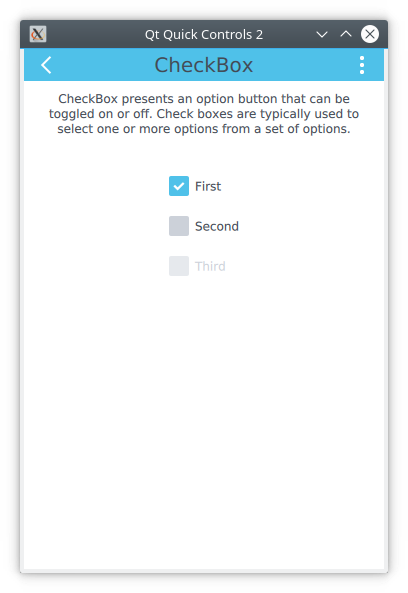
Material



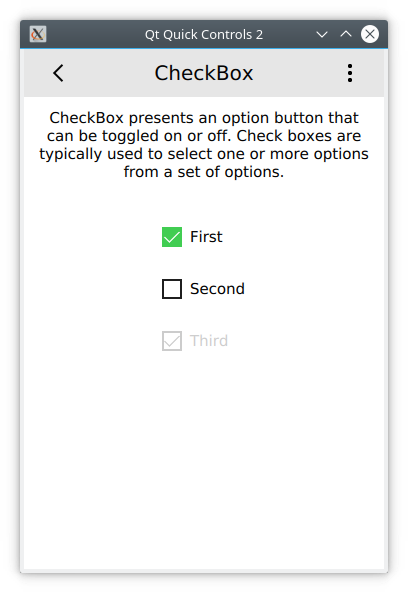
Imagine



Windows



Universal



Qt Quick Controls 2 is available from the QtQuick.Controls import. The following modules are also of interest:

**QtQuick.Controls** - The basic controls.

**QtQuick.Templates** - Provides the behavioral, non-visual base types for the controls.

**QtQuick.Controls.Imagine** - Imagine style theming support. **QtQuick.Controls.Material** - Material style theming support. **QtQuick.Controls.Universal** - Universal style theming support.

**Qt.labs.platform** - Support for platform native dialogs for common tasks such as picking files, colours, etc, as well as system tray icons and standard paths.

**Qt.Labs**

Notice that the Qt.labs modules are experimental, meaning that their APIs can have breaking changes between Qt versions.

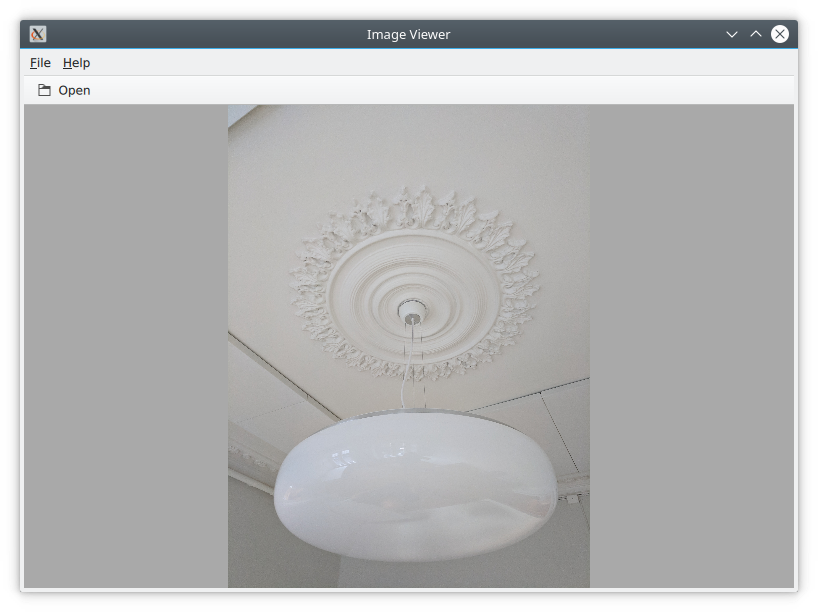
# An Image Viewer

Let’s look at a larger example of how Qt Quick Controls are used. For this, we will create a simple image viewer.

First, we create it for desktop using the Fusion style, then we will refactor it for a mobile experience before having a look at the final code base.

## The Desktop Version

The desktop version is based around a classic application window with a menu bar, a tool bar and a document area. The application can be seen in action below.



We use the Qt Creator project template for an empty Qt Quick application as a starting point. However, we replace the default Window element from the template with a ApplicationWindow from the QtQuick.Controls module. The code below shows main.qml where the window itself is created and setup with a default size and title.

import QtQuick

import QtQuick.Controls import Qt.labs.platform

ApplicationWindow { visible: true width: 640

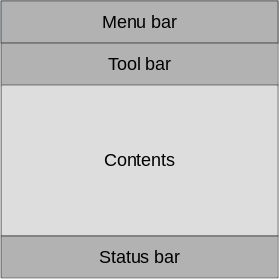
height: 480

// ...

}

The ApplicationWindow consists of four main areas as shown below. The menu bar, tool bar and status bar are usually populated by instances of MenuBar , ToolBar or TabBar controls, while the contents area is

where the children of the window go. Notice that the image viewer application does not feature a status bar; that is why it is missing from the code show here, as well as from the figure above.



As we are targeting desktop, we enforce the use of the *Fusion* style. This can be done via a configuration file, environment variables, command line arguments, or programmatically in the C++ code. We do it the latter way by adding the following line to main.cpp :

QQuickStyle::setStyle("Fusion");

We then start building the user interface in main.qml by adding an Image element as the contents. This element will hold the images when the user opens them, so for now it is just a placeholder. The background property is used to provide an element to the window to place behind the contents. This will be shown when there is no image loaded, and as borders around the image if the aspect ratio does not let it fill the contents area of the window.

ApplicationWindow {

// ...

background: Rectangle { color: "darkGray"

}

Image {

id: image anchors.fill: parent

fillMode: Image.PreserveAspectFit asynchronous: true

}

// ...

}

We then continue by adding the ToolBar . This is done using the toolBar property of the window. Inside the tool bar we add a Flow element which will let the contents fill the width of the control before overflowing to a new row. Inside the flow we place a ToolButton .

The ToolButton has a couple of interesting properties. The text is straight forward. However, the icon.name is taken from the [freedesktop.org Icon Naming Specification [](https://specifications.freedesktop.org/icon-naming-spec/icon-naming-spec-latest.html) (https://specifications.freedesktop.org/icon-naming-spec/icon-naming-spec- latest.html) . In that document, a list of standard icons are listed by name.](https://specifications.freedesktop.org/icon-naming-spec/icon-naming-spec-latest.html)

By referring to such a name, Qt will pick out the correct icon from the current desktop theme.

In the onClicked signal handler of the ToolButton is the final piece of code. It calls the open method on the fileOpenDialog element.

ApplicationWindow {

// ...

header: ToolBar { Flow {

anchors.fill: parent ToolButton {

text: qsTr("Open") icon.name: "document-open"

onClicked: fileOpenDialog.open()

}

}

}

// ...

}

The fileOpenDialog element is a FileDialog control from the Qt.labs.platform module. The file dialog can be used to open or save files.

In the code we start by assigning a title . Then we set the starting folder using the StandardsPaths class. The StandardsPaths class holds links to common folders such as the user’s home, documents, and so on. After that we set a name filter that controls which files the user can see and pick using the dialog.

Finally, we reach the onAccepted signal handler where the Image element that holds the window contents is set to show the selected file. There is an onRejected signal as well, but we do not need to handle it in the image viewer application.

ApplicationWindow {

// ... FileDialog {

id: fileOpenDialog

title: "Select an image file"

folder: StandardPaths.writableLocation(StandardPaths.Do nameFilters: [

"Image files (\*.png \*.jpeg \*.jpg)",

]

onAccepted: {

image.source = fileOpenDialog.fileUrl

}

}

// ...

}

We then continue with the MenuBar . To create a menu, one puts Menu elements inside the menu bar, and then populates each Menu with MenuItem elements.

In the code below, we create two menus, *File* and *Help*. Under *File*, we place *Open* using the same icon and action as the tool button in the tool bar. Under *Help*, you find *About* which triggers a call to the open method of the aboutDialog element.

Notice that the ampersands (“&”) in the title property of the Menu and the text property of the MenuItem turn the following character into a keyboard shortcut; e.g. you reach the file menu by pressing *Alt+F*, followed by *Alt+O* to trigger the open item.

ApplicationWindow {

// ...

menuBar: MenuBar { Menu {

title: qsTr("&File") MenuItem {

text: qsTr("&Open...") icon.name: "document-open"

onTriggered: fileOpenDialog.open()

}

}

Menu {

title: qsTr("&Help") MenuItem {

text: qsTr("&About...") onTriggered: aboutDialog.open()

}

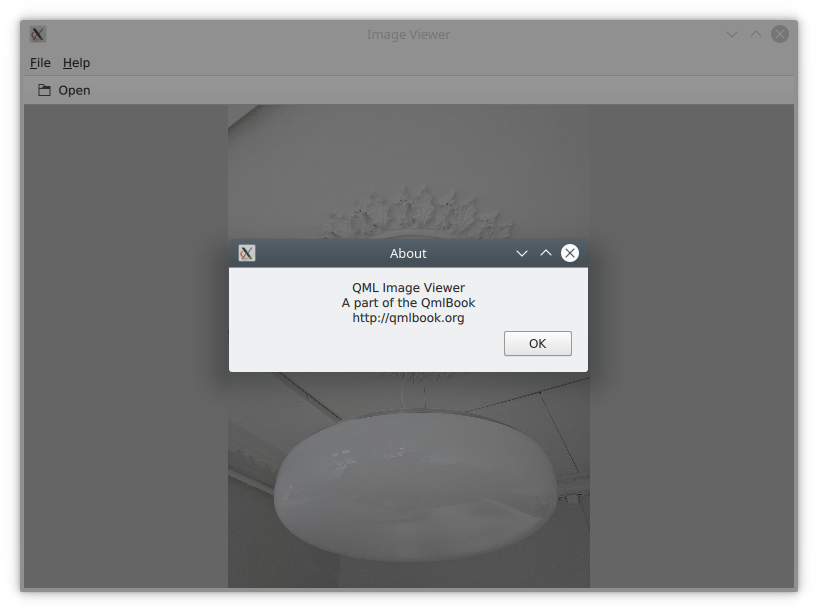
}

}

// ...

}

The aboutDialog element is based on the Dialog control from the QtQuick.Controls module, which is the base for custom dialogs. The dialog we are about to create is shown in the figure below.



The code for the aboutDialog can be split into three parts. First, we setup the dialog window with a title. Then, we provide some contents for the dialog – in this case, a Label control. Finally, we opt to use a standard *Ok* button to close the dialog.

ApplicationWindow {

// ... Dialog {

id: aboutDialog title: qsTr("About") Label {

anchors.fill: parent

text: qsTr("QML Image Viewer\nA part of the QmlBook horizontalAlignment: Text.AlignHCenter

}

standardButtons: StandardButton.Ok

}

// ...

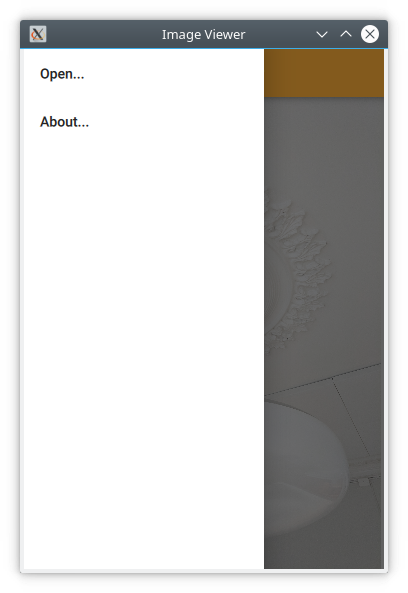
}

The end result of all this is a functional, albeit simple, desktop application for viewing images.

## Moving to Mobile

There are a number of differences in how a user interface is expected to look and behave on a mobile device compared to a desktop application. The biggest difference for our application is how the actions are accessed. Instead of a menu bar and a tool bar, we will use a drawer from which the user can pick the actions. The drawer can be swiped in from the side, but

we also offer a hamburger button in the header. The resulting application with the drawer open can be seen below.



First of all, we need to change the style that is set in main.cpp from

*Fusion* to *Material*:

QQuickStyle::setStyle("Material");

Then we start adapting the user interface. We start by replacing the menu with a drawer. In the code below, the Drawer component is added as a

child to the ApplicationWindow . Inside the drawer, we put a ListView containing ItemDelegate instances. It also contains a ScrollIndicator used to show which part of a long list is being shown. As our list only consists of two items, the indicator is not visible in this example.

The drawer's ListView is populated from a ListModel where each ListItem corresponds to a menu item. Each time an item is clicked, in the onClicked method, the triggered method of the corresponding ListItem is called. This way, we can use a single delegate to trigger different actions.

ApplicationWindow {

// ... id: window Drawer {

id: drawer

width: Math.min(window.width, window.height) / 3 \* 2 height: window.height

ListView {

focus: true currentIndex: -1 anchors.fill: parent

delegate: ItemDelegate { width: parent.width text: model.text

highlighted: ListView.isCurrentItem onClicked: {

drawer.close() model.triggered()

}

}

model: ListModel { ListElement {

text: qsTr("Open...")

triggered: function() { fileOpenDialog.open

}

ListElement {

text: qsTr("About...")

triggered: function() { aboutDialog.open();

}

}

ScrollIndicator.vertical: ScrollIndicator { }

}

}

// ...

}

The next change is in the header of the ApplicationWindow . Instead of a desktop style toolbar, we add a button to open the drawer and a label for the title of our application.



The ToolBar contains two child elements: a ToolButton and a Label . The ToolButton control opens the drawer. The corresponding close

call can be found in the ListView delegate. When an item has been selected, the drawer is closed. The icon used for the ToolButton comes [from the Material Design Icons page [](https://material.io/tools/icons/?style=baseline) (https://material.io/tools/icons/? style=baseline) .](https://material.io/tools/icons/?style=baseline)

ApplicationWindow {

// ...

header: ToolBar { ToolButton {

id: menuButton anchors.left: parent.left

anchors.verticalCenter: parent.verticalCenter icon.source: "images/baseline-menu-24px.svg" onClicked: drawer.open()

}

Label {

anchors.centerIn: parent text: "Image Viewer" font.pixelSize: 20 elide: Label.ElideRight

}

}

// ...

}

Finally we make the background of the toolbar pretty — or at least orange. To do this, we alter the Material.background attached property. This comes from the QtQuick.Controls.Material module and only affects the Material style.

import QtQuick.Controls.Material ApplicationWindow {

// ...

header: ToolBar {

Material.background: Material.Orange

// ...

}

With these few changes we have converted our desktop image viewer to a mobile-friendly version.

## A Shared Codebase

In the past two sections we have looked at an image viewer developed for desktop use and then adapted it to mobile.

Looking at the code base, much of the code is still shared. The parts that are shared are mostly associated with the document of the application, i.e. the image. The changes have accounted for the different interaction patterns of desktop and mobile, respectively. Naturally, we would want to unify these code bases. QML supports this through the use of *file selectors*.

A file selector lets us replace individual files based on which selectors are active. The Qt documentation maintains a list of selectors in the [documentation for the QFileSelector class (link [](https://doc.qt.io/qt-5/qfileselector.html) (https://doc.qt.io/qt- 5/qfileselector.html) ). In our case, we will make the desktop version the default and replace selected files when the *android* selector is encountered. During development you can set the environment variable QT\_FILE\_SELECTORS to android to simulate this.](https://doc.qt.io/qt-5/qfileselector.html)

##### File Selector

File selectors work by replacing files with an alternative when a

*selector* is present.

By creating a directory named +selector (where selector represents the name of a selector) in the same directory as the files that you want to replace, you can then place files with the same name as the file you want to replace inside the directory. When the selector is present, the file in the directory will be picked instead of the original file.

The selectors are based on the platform: e.g. android, ios, osx, linux, qnx, and so on. They can also include the name of the Linux distribution used (if identified), e.g. debian, ubuntu, fedora. Finally, they also include the locale, e.g. en\_US, sv\_SE, etc.

It is also possible to add your own custom selectors.

The first step to do this change is to isolate the shared code. We do this by creating the ImageViewerWindow element which will be used instead of the ApplicationWindow for both of our variants. This will consist of the dialogs, the Image element and the background. In order to make the open methods of the dialogs available to the platform specific code, we need to expose them through the functions openFileDialog and openAboutDialog .

import QtQuick

import QtQuick.Controls import Qt.labs.platform

ApplicationWindow {

function openFileDialog() { fileOpenDialog.open(); } function openAboutDialog() { aboutDialog.open(); }

visible: true

title: qsTr("Image Viewer")

background: Rectangle { color: "darkGray"

}

Image {

id: image anchors.fill: parent

fillMode: Image.PreserveAspectFit asynchronous: true

}

FileDialog {

id: fileOpenDialog

// ...

}

Dialog {

id: aboutDialog

// ...

}

}

Next, we create a new main.qml for our default style *Fusion*, i.e. the desktop version of the user interface.

Here, we base the user interface around the ImageViewerWindow instead of the ApplicationWindow . Then we add the platform specific parts to it,

e.g. the MenuBar and ToolBar . The only changes to these is that the calls to open the respective dialogs are made to the new functions instead of directly to the dialog controls.

import QtQuick

import QtQuick.Controls

ImageViewerWindow { id: window

width: 640

height: 480

menuBar: MenuBar { Menu {

title: qsTr("&File") MenuItem {

text: qsTr("&Open...") icon.name: "document-open"

onTriggered: window.openFileDialog()

}

}

Menu {

title: qsTr("&Help") MenuItem {

text: qsTr("&About...")

onTriggered: window.openAboutDialog()

}

}

}

header: ToolBar { Flow {

anchors.fill: parent ToolButton {

text: qsTr("Open") icon.name: "document-open"

onClicked: window.openFileDialog()

}

}

}

}

Next, we have to create a mobile specific main.qml . This will be based around the *Material* theme. Here, we keep the Drawer and the mobile- specific toolbar. Again, the only change is how the dialogs are opened.

import QtQuick

import QtQuick.Controls

import QtQuick.Controls.Material

ImageViewerWindow { id: window

width: 360

height: 520

Drawer {

id: drawer

// ... ListView {

// ...

model: ListModel { ListElement {

text: qsTr("Open...")

triggered: function(){ window.openFileDialo

}

ListElement {

text: qsTr("About...")

triggered: function(){ window.openAboutDial

}

}

// ...

}

}

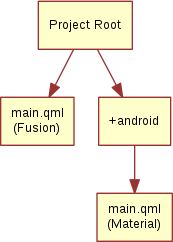
header: ToolBar {

// ...

}

}

The two main.qml files are placed in the file system as shown below. This lets the file selector that the QML engine automatically creates pick the right file. By default, the *Fusion* main.qml is loaded. If the android selector is present, then the *Material* main.qml is loaded instead.



Until now the style has been set in main.cpp . We could continue doing this and use #ifdef expressions to set different styles for different platforms. Instead we will use the file selector mechanism again and set the style using a configuration file. Below, you can see the file for the *Material* style, but the *Fusion* file is equally simple.

[Controls] Style=Material

These changes have given us a joined codebase where all the document code is shared and only the differences in user interaction patterns differ. There are different ways to do this, e.g. keeping the document in a specific component that is included in the platform specific interfaces, or as in this example, by creating a common base that is extended by each platform.

The best approach is best determined when you know how your specific code base looks and can decide how to separate the common from the unique.

## Native Dialogs

When using the image viewer you will notice that it uses a non-standard file selector dialog. This makes it look out of place.

The Qt.labs.platform module can help us solve this. It provides QML bindings to native dialogs such as the file dialog, font dialog and colour dialog. It also provides APIs to create system tray icons, as well as system global menus that sits on top of the screen (e.g. as in OS X). The cost of this is a dependency on the QtWidgets module, as the widget based dialog is used as a fallback where the native support is missing.

In order to integrate a native file dialog into the image viewer, we need to import the Qt.labs.platform module. As this module has name clashes with the QtQuick.Dialogs module which it replaces, it is important to remove the old import statement.

In the actual file dialog element, we have to change how the folder property is set, and ensure that the onAccepted handler uses the file property instead of the fileUrl property. Apart from these details, the usage is identical to the FileDialog from QtQuick.Dialogs .

import QtQuick

import QtQuick.Controls

import Qt.labs.platform ApplicationWindow {

// ...

FileDialog {

id: fileOpenDialog

title: "Select an image file"

folder: StandardPaths.writableLocation(StandardPaths.Do nameFilters: [

"Image files (\*.png \*.jpeg \*.jpg)",

]

onAccepted: {

image.source = fileOpenDialog.file

}

}

// ...

}

In addition to the QML changes, we also need to alter the project file of the image viewer to include the widgets module.

QT += quick quickcontrols2 widgets

And we need to update main.qml to instantiate a QApplication object instead of a QGuiApplication object. This is because the QGuiApplication class contains the minimal environment needed for a graphical application, while QApplication extends QGuiApplication with features needed to support QtWidgets .

include <QApplication>

// ...

int main(int argc, char \*argv[])

{

QApplication app(argc, argv);

// ...

}

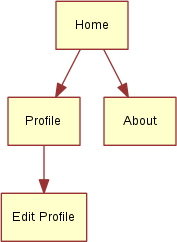
With these changes, the image viewer will now use native dialogs on most platforms. The platforms supported are iOS, Linux (with a GTK+ platform theme), macOS, Windows and WinRT. For Android, it will use a default Qt dialog provided by the QtWidgets module.

# Common Patterns

There a number of common user interface patterns that can be implemented using Qt Quick Controls. In this section, we try to demonstrate how some of the more common ones can be built.

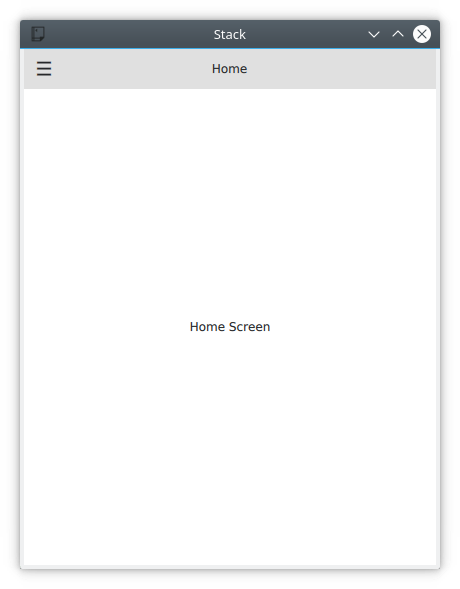
## Nested Screens

For this example we will create a tree of pages that can be reached from the previous level of screens. The structure is pictured below.



The key component in this type of user interface is the StackView . It allows us to place pages on a stack which then can be popped when the user wants to go back. In the example here, we will show how this can be implemented.

The initial home screen of the application is shown in the figure below.



The application starts in main.qml , where we have an ApplicationWindow containing a ToolBar , a Drawer , a StackView and a home page element, Home . We will look into each of the components below.

import QtQuick

import QtQuick.Controls ApplicationWindow {

// ...