Table of Contents

Einleitung

Copyright

Erstelle Cross Platform Desktop Applikationen mit Python, Qt und PyQt5
von Olaf Art Ananda

Motivation

Nach dem ich bereits seit über 30 Jahren Software entwickle, kam ich von C, Assembler, Clipper, Powerbuilder, Java, C#, Objetive-C, C++ zu Python. Und rate mal...
I liebe Python
Es wäre einfach für mich native Applikationen in Java, C++ oder Objective-C zu schreiben und ausserdem wäre ich ich in der Lage, Kotlin, Dart oder Swift zu lernen, aber die Dinge sind einfacher, wenn man sie mit Python programmiert.
Ich habe mal ein Django Tutorial gemacht, Django ist ein Web-Framework für Python, und dabei hat mich fasziniert, wie einfach es ist, Daten-Modelle zu entwickeln. Einfach eine simple Datenklasse schreiben, ein Scaffolding job starten und schwups werden alle nötigen Tabellen auf dem SQL-Server für dich angelegt. Dann nur noch eben den Python-Interpreter starten, das Modell importieren, eine Instanz erstellen, sie mit Daten füllen und die Save-Methode aufrufen und schon wird das Ganze auf den SQL-Server geschrieben.
Ich musste nicht eine Zeile SQL-Code schreiben. Python hatte mich eingefangen :-)
Dann habe ich noch etwas über Generators, Comprehensions und Meta-Programmierung erfahren und Python hatte mich komplett überzeugt.

Wo wir grad dabei sind...um dieses Buch schreiben zu können, habe ich eine Applikation geschrieben, die ich EbookCreator genannt habe. Diese Anwendung nutz auch PyQt5, QtWidgets und QML nutze ich, um Daten zu serialisieren. Diese App ist Open Source und du kannst sie benutzen, um dich inspirieren zu lassen. Du findest den Source Code auf github.com.

Gestern habe ich ein Video mit "Uncle" Bob Martin über das Programmieren in der Zukunft gesehen. Er sagt, das sich alle 5 Jahre die Zahl der Softwareentwickler verdoppelt. Das bedeutet, das um die 50% aller Entwickler unerfahren sind. Er sagte auch, das die erfahrenen Entwickler dafür verantwortlich sind, wenn z.B. ein selbst fahrendes Auto einen Menschen umfährt und tötet, nur weil ein Softwarefehler vorlag. Als ich das Heute gehörte habe, habe ich mich entschlossen andere Entwickler auszubilden und habe mit diesem Buch begonnen.

Für Wen Ist Dieses Buch

Wenn du in der Lage bist, einfache Programme in Python zu schreiben und interessiert bist Anwendungen mit einem grafischem Benutzer-Interface für alle möglichen Platformen zu schreiben, dann ist dieses Buch genau das richtige für dich.
Du musst dich nicht unbedingt mit Qt auskennen.
Wenn du willst, probiere alle Beispiele aus diesem Buch selber aus. Von Vorteil wäre es, wenn du auch, wie ich, auf Linux arbeitest. Die Beispiele sollten aber auch mühelos auf MacOS und Windows laufen. Lediglich für die Installation der benötigten Software solltest du dich selber im Internet einlesen, da ich nur die nötigen Schritte für Linux erkläre.

Für Wen Ist Dieses Buch Nicht

Solltest du noch ein Anfänger in Python sein, dann schlage ich dir vor, erst einmal einen geeigneten Grundkurs für Python zu machen, bevor du in diesem Buch weiterliest. Es gibt hierfür tolle Bücher und eine Menge Videos auf Youtube.
In diesem Buch gehe ich nicht in Python spezifische Details ein.

Wie Dieses Buch Organisiert Ist

Hier findest du die Übersicht über die Teile dieses Buches.
Zuerst werden wir in ***Teil I*** alle nötigen Werkzeuge zum Erstellen der Software mit PyQt5 und Python installieren.
In ***Teil II*** lernst du etwas über die verschiedenen Ansätze um Anwendungen zu bauen.
In ***Teil III*** lernst du Anwendungen mit QtWidgets zu erstellen.
In ***Teil IV*** lernst du, wie man eine Line Of Business (LOB) applikation erstellt, die eine Datenbank benutzt, um Daten zu speichern. Und in ***Teil V*** wirst du lernen, wie man ein Setup Programm erstellt, um Python Software auf Linux zu installieren.

Konventionen In Diesem Buch

In diesem Buch nutze ich folgende typografischen Konventionen.

Kursiv

Wird benutzt, um Dateinamen und Pfade zu markieren

```
Feste Schriftbreite
```

Wird für Programm-Beispiele benutzt

Beipiel Source Code Benutzen

Die gesamten Beispiele sind auf github.com verfügbar.

Wie Du Mich Kontaktieren Kannst

Solltest du eine Frage oder ein Kommentar zu diesem Buch haben, scheue dich nicht, mir eine Email zu schreiben. Sende deine Fragen und Kommentare einfach an: japp.olaf@gmail.com

Danksagungen

Zu allererst bin ich meinem Körper dankbar, weil er mich zur richtigen Zeit auf die richtigen Wege geführt hat. Ich weiß, das klingt bestimmt ein bisschen verrückt, aber da ich Maschinenschlosser gelernt habe und bereits nach wenigen Jahren, Rückenschwerzen bekam und über ein halbes Jahr krank war, habe ich angefangen Maschinenbau zu studieren und während des Studiums habe ich dann mit dem Programmieren angefangen. Zu der Zeit habe ich mich entschieden, mein Studium abzubrechen und als Softwareentwickler zu arbeiten.
Dann hat mir mein Körper vor 5 Jahren mit gleich zwei Burnouts zu verstehen gegeben, mich aus dem Arbeitsleben zurückzuziehen.
Nun habe ich viel Zeit, um Open Source Software zu schreiben und neue Dinge auszuprobieren, wie zum Beispiel Bücher wie dieses hier zu schreiben.

Ich bin Guido Rossum dankbar, weil es 1991 Python erfunden und veröffentlicht hat.

Und ausserdem bin ich allen Pythonistas da draussen dankbar, weil sie so tolle Tutorials und Videos gemacht haben, damit ich Python lernen konnte.

Teil I - Prolog

Installation

Wir werden nun anfangen und alle nötigen Programme installieren, um in der Lage zu sein, die gewünschte Software zu schreiben und sie auszuprobieren.
Um Anwendungen auch auf anderen Platformen installieren zu können, werden wir zusätzliche Software im Kapitel über die Installation, später in diesem Buch, installieren.

Ich gehe davon aus, das du bereits Python3 und Pip auf deinem Rechner installiert hast. Wenn nicht, findest du alle notwendigen Informationen auf der Python Webseite. Wir benötigen Python in der Version 3.7.
Ich gehe ausserdem davon aus, dass du in der Lage bist Pakete mittels Pip zu installieren.

Zuerst werden wir PyQt5 welches zusammen mit Qt5 kommt installieren, damit wir Desktop-GUI-Applikationen entwickeln können.

```
1  user@machine:/path$ pip3 install PyQt5
2  user@machine:/path$ pip3 install PyQtWebEngine
```

Dann werden wir Visual Studio Code installieren. VS-Code ist kostenlos und Open Source und hat viele nützliche Erweiterungen um Python Code schreiben zu können.
Du kannst VS-Code hier runterladen. Ich gehe davon aus, dass du in der Lage bist VS-Code selbständig zu installieren, ansonsten findest auf deren Webseite wunderbare Anleitungen.
Du kannst auch **apt** nutzen, wenn du auf Linux bist.

```
1  user@machine:/path$sudo add-apt-repository "deb [arch=amd64] https://
packages.microsoft.com/repos/vscode stable main"
2  user@machine:/path$ sudo apt update
3  user@machine:/path$ sudo apt install code
```

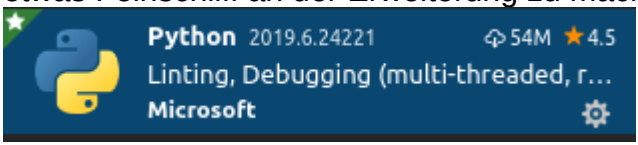
Im Folgenden werden wir ein paar Erweiterungen installieren, um unsere erste Anwendung zu erstellen.

Entwicklungsumgebung Aufsetzen

Nachdem wir nun VS-Code installiert haben, installieren wir noch nachfolgende Erweiterungen.

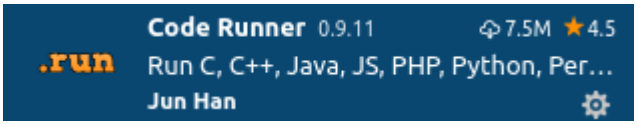
Python

Python ist eine Erweiterung um Syntax farblich hervorzuheben, Python-Code zu debuggen und es enthält einen Linter.
Manchmal zeigt es sogar die korrekte Intellisense an. Da ist wohl noch etwas Feinschliff an der Erweiterung zu machen.

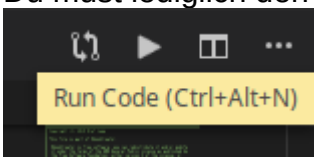


Coderunner

Mit Coderunner bist du in der Lage, deine Anwendung mit nur einem Klick zu starten.

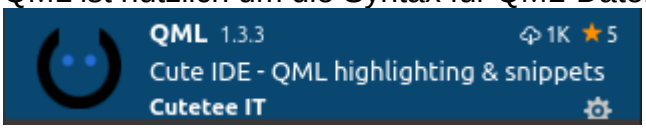


Du must lediglich den "Play" Knopf klicken.



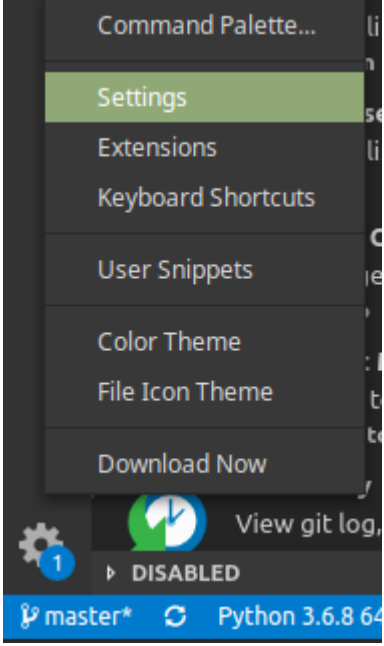
QML

QML ist nützlich um die Syntax für QML-Dateien einzufärben.

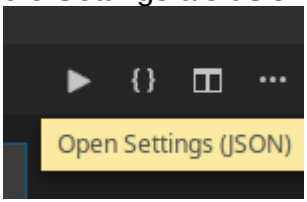


VS-Code Settings

Du kannst die Settings mit einem Klick auf das Icon mit dem Zahnrad öffnen.



Und dann klickst du **"{}"** auf der oberen rechten Seite des Bildschirms um die Settings als JSON-Datei zu öffnen.



Hier sind ein paar nützliche Einstellungen, welche ich in die *settings.json* eingefügt habe.
Das colorTheme ist natürlich Geschackssache.
Die Einstellung vom CoderRunner führt Python mit der *main.py* aus, egal welche Python-Datei gerade im Editor offen ist. Das bedingt natürlich, dass das Projekt eine Datei mit dem Namen *main.py* besitzt und das dies die Startdatei ist.

```
{
    "workbench.colorTheme": "Visual Studio Dark",
    "code-runner.executorMap": {
        "python": "python3 $workspaceRoot/main.py",
    },
    "code-runner.clearPreviousOutput": true,
    "code-runner.saveAllFilesBeforeRun": true,
    "git.autofetch": true,
}
```

Erste Anwendung

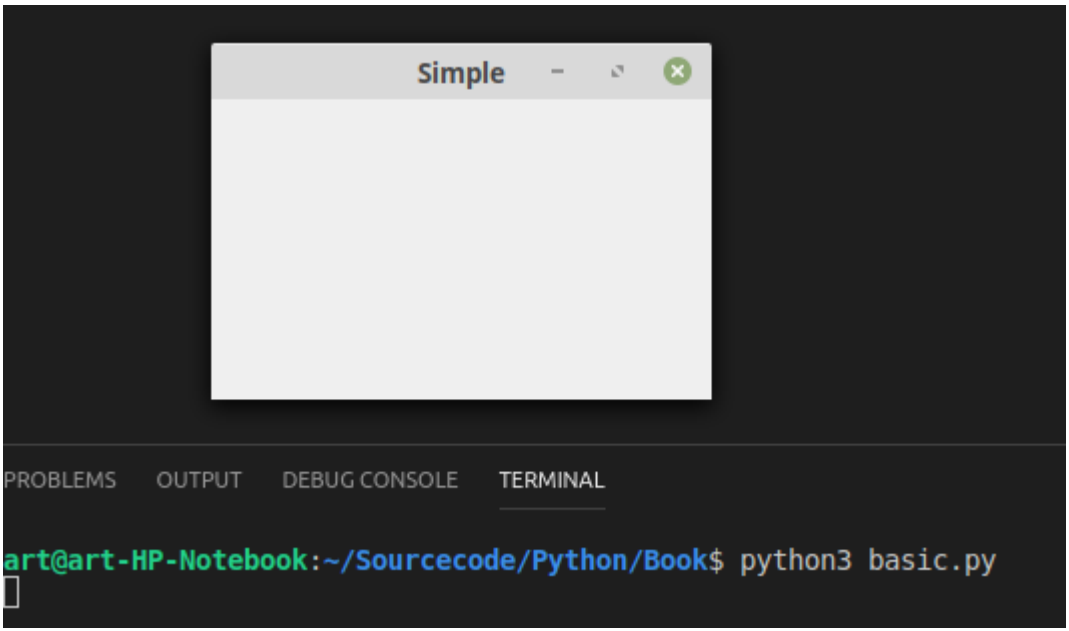
Wir schreiben nun eine simple Anwendung um unsere Umgebung einmal auszuprobieren.
Ich gehe hier nicht weiter ins Detail, versuche aber später im Buch auf die Einzelheiten einzugehen.

basic.py

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget
3
4
5  app = QApplication(sys.argv)
6  w = QWidget()
7  w.resize(250, 150)
8  w.setWindowTitle('Simple')
9  w.show()
10 app.exec()
```

In diesem Fall, da wir die Python Datei nicht *main.py* genannt haben, führen wir Python in einem Terminal innerhalb von VS-Code aus..

```
user@machine:/path$ python3 basic.py
```



In diesem Beispiel instanziieren wir eine Application, zusätzlich noch ein Widget, welches hier als Fenster dient,setzen die Grösse des Fensters, setzen den Titel des Fensters, machen das Fenster sichtbar und starten den MainLoop.
Im MainLoop wird in einer Schleife auf Ereignisse wie Mausklicks, Tastatureingabe abgefragt und sie dann dem Fenster zur Auswertung übergeben. In diesem Fall wird das Fenster lediglich auf Veränderung der Fenstergrösse, auf das Verschieben des Fensters und das Schliessen des Fensters reagieren, sollten wir auf den grünen Schliessen-Knopf, recht oben in der Ecke, klicken. Hierbei wird dann die MainLoop verlassen und die Anwendung beendet.

Zusammenfassung

Nachdem wir die Entwicklungsumgebung aufgebaut haben, konnten wir die erste PyQt Anwendung erstellen und ausführen.

Teil II - Basis

Viele Unterschiedliche Ansätze

Da Python schon seit 1991 existiert, wurden bereits mehrere Frameworks für die GUI-Programmierung in Python erstellt.
Da gibt es Tkinter, welches eine Brücke zu TK ist. Tkinter wird standardmässig mit Python ausgeliefert.
Dann gibt es noch Kivy, welches eigentlich gute Ansätze, wie zum Beispiel die GUI-Beschreibungssprache kvlang hat, mit der es auf einfache pythonische Weise möglicht ist, das Userinterface zu deklarieren.
Dann gibt es noch BeeWare mit dem u.a. Python in Java-Byte-Code compiliert wird, um auf Android ausgeführt werden zu können.
Da gibt es auch noch Enaml Native, welches man als Pythons Antwort zu React Native sehen kann und dann gibt es noch ein Paar Möglichkeiten um eine Brücke zu Qt zu schlagen, als da wären, PySide welches eine Brücke zu Qt4 baut, PyQt welches ebenfalls eine Brücke zu Qt4 ist und PyQt5, was eine Brücke zu Qt5 darstellt und um das es hier in diesem Buch geht.

Ich werde hier nicht über die Vor- und Nachteile der einzelnen Frameworks schreiben, sondern mich voll auf PyQt5 konzentrieren.

PyQt5 zu benutzen war eine persönliche Entscheidung von mir, da ich bereits ein paar Jahre mit Qt5 und C++ gearbeitet habe.

Qt5 und PyQt5 sind als Open Source Lizenz verfügbar und man kann beide kostenlos nutzen, solange man damit Open Source Software erstellt. Solltest du vorhaben kommerzielle Software zu erstellen, musst du Lizenzen für beide Frameworks erwerben.

Selbst wenn wir Qt5 nutzen, haben wir zwei Optionen Anwendungen damit zu erstellen.

Die erste Option ist es eine Anwendung mit QtWidgets zu erstellen, welches den Desktop als Zielplatform hat und QtQuick, welches einen deklarativen Ansatz wählt um Userinterfaces mittels QML(**Q**t **M**arkup **L**anguage) zu beschreiben und eher für mobile Endgeräte konzipiert wurde.
Da QtQuick derzeit kein Control für einen TreeView und einen TableView besitzt, würde ich nicht empfehlen damit eine Anwendung für den Desktop zu erstellen, ausser man kann auf diese beiden Controls verzichten.

Mit QtQuick kann man ausserdem Behaviours und Transitions deklarieren, welches man heutzutage eher auf mobilen Endgeräten antrifft.

Wenn du einen Design Hintergrund hast, ist wahrscheinlich der QML-Ansatz interessanter für dich, da du hier nicht wirklich Code erzeugen musst.
Bist du eher der CoderTyp, dann ist evtl. QtWidgets die richtige Wahl für dich.

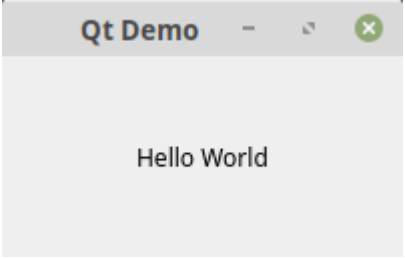
Im folgenden zeige ich dir allerdings gleich vier Varianten Qt Anwendungen zu schreiben.

Hello World (QtWidgets)

Zuerst erstellen wir eine sehr simple QtWidget Anwendung, mit der wir die Worte *Hello World* auf dem Bildschirm ausgeben. Im Gegensatz zu dem ersten Beispiel benutzen wir hier die Klasse QMainWindow anstelle von QWidget. QMainWindow hat zusätzlich noch die Möglichkeit ein Menu, eine Toolbar und eine Statusbar zu nutzen.

QWidget/Basics/main.py

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel
from PyQt5.QtCore import Qt


class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setWindowTitle("Qt Demo")
        label = QLabel("Hello World")
        label.setAlignment(Qt.AlignCenter)
        self.setCentralWidget(label)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MainWindow()
    win.show()
    sys.exit(app.exec())
```



Das Beispiel ist fast selbsterklärend. Wir instanziieren das Applikations-Objekt indem wir ihm die Argumentenliste der Anwendung übergeben, erzeugen eine Instanz von MainWindow, unserer Fensterklasse. Machen das Fenster mit Aufruf der Methode *show()* auf dem Bildschirm sichtbar und rufen die Hauptschleife der Applikation auf.
In der Init-Methode des Fensters rufen wir zuerst einmal die Init-Methode der VaterKlasse auf, setzen den Titel des Fensters und erzeugen eine Instanz eines Labels, welches hier als zentrales Widget gesetzt wird.
QMainWindow hat neben dem Menu, der Menubar und der Statusbar das zentrale Widget, welches den inneren Bereich des Fensters ausfüllt.
In grösseren Projekte wäre es sinnvoller jede Klasse in einer eigenen Python-Datei zu speichern. Das macht das Projekt übersichtlicher. Unser Fenster würden wir dann in der Datei *mainwindow.py* speichern.

Hello World (QtQuick)

Die Hello World Anwendung für QtQuick besteht aus zwei Dateien. Zuerst haben wir da wieder die *main.py* in der wir die Applikation instanziieren und den MainLoop starten und dann haben wir eine zweite Datei mit dem Namen *view.qml* in der wir das Userinterface definieren werden.

QtQuick/Basics/main.py

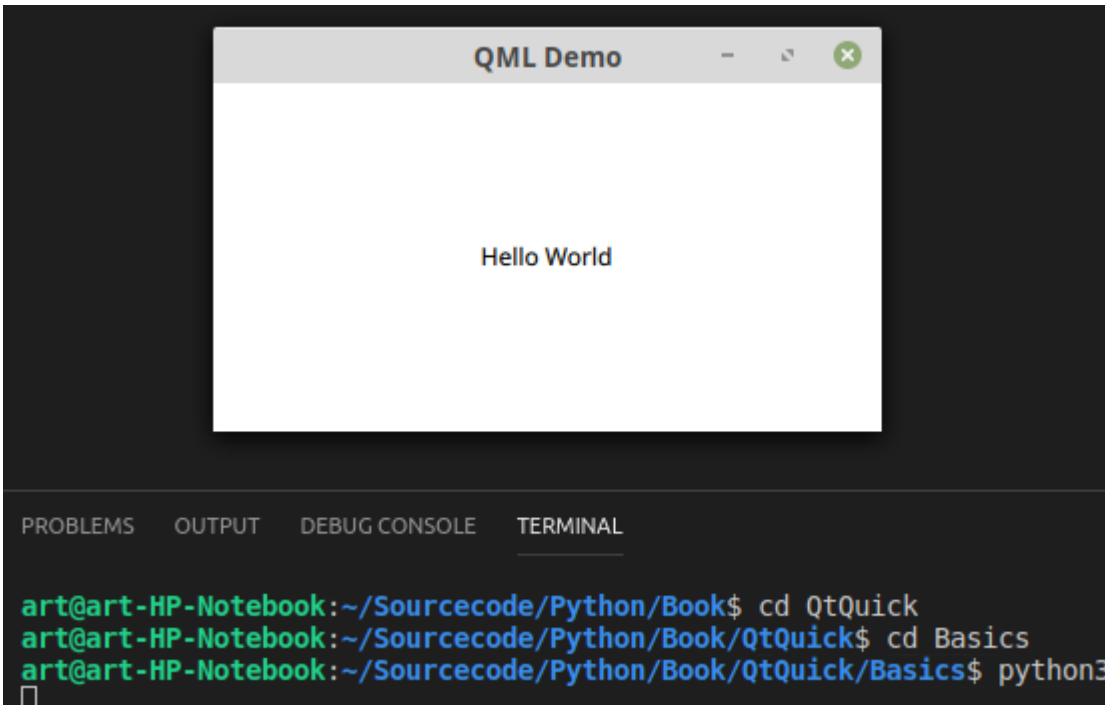
```
1  import sys
2  from PyQt5.QtGui import QGuiApplication
3  from PyQt5.QtQml import QQmlApplicationEngine
4
5
6  if __name__ == "__main__":
7      app = QGuiApplication(sys.argv)
8      engine = QQmlApplicationEngine("view.qml")
9      if not engine.rootObjects():
10         sys.exit(-1)
11     sys.exit(app.exec())
```

QtQuick/Basics/view.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.5

ApplicationWindow {
    visible: true

    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Hello World"
    }
}
```

Um die Anwendung zu starten müssen wir in das Verzeichnis QtQuick/Basics wechseln und die Anwendung wie folgt starten:

```
user@machine:/path$ python3 main.py
```



Diese Anwendung ist ähnlich der QtWidget-Variante bis auf die Tatsache, das wir QGuiApplication anstelle von QApplication benutzen. Und ausserdem nutzen wir QQmlApplicationEngine, um die QML-Datei zu laden. Des Weiteren deklarieren wir wie bereits gesagt das Userinterface mit Hilfe von QML. Auf QML werde ich in diesem Buch nicht näher eingehen, da es

hierfür bereits ausreichend Literatur gibt und es den Rahmen diese Buches sprengen würde.

QWidget und QML Kombinieren

Eine dritte Möglichkeit, Qt Anwendungen zu schreiben ist die Kombination aus QtWidget und QML in dem man ein QQuickView verwendet, um QML innerhalb einer QtWidget Anwendung zu benutzen.

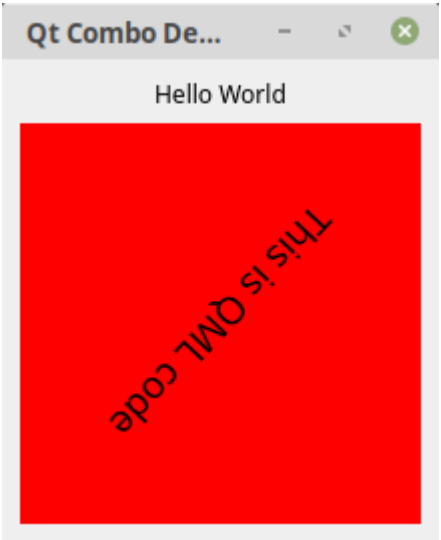
Combo/main.py

```
  import sys
  from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QVBoxLayo
ut, QWidget
  from PyQt5.QtCore import Qt, QUrl
  from PyQt5.QtQuick import QQuickView


  class MainWindow(QMainWindow):
      def __init__(self):
          QMainWindow.__init__(self)
          self.setWindowTitle("Qt Combo Demo")
          widget= QWidget()
          layout = QVBoxLayout()
          view = QQuickView()
          container = QWidget.createWindowContainer(view, self)
          container.setMinimumSize(200, 200)
          container.setMaximumSize(200, 200)
          view.setSource(QUrl("view.qml"))
          label = QLabel("Hello World")
          label.setAlignment(Qt.AlignCenter)
          layout.addWidget(label)
          layout.addWidget(container)
          widget.setLayout(layout)
          self.setCentralWidget(widget)


  if __name__ == "__main__":
      app = QApplication(sys.argv)
      win = MainWindow()
      win.show()
      sys.exit(app.exec())
```

Combo/view.qml

```

```

```
import QtQuick 2.1

    Rectangle {
        id: rectangle
        color: "red"
        width: 200
        height: 200

        Text {
            id: text
            text: "This is QML code"
            font.pointSize: 14
            anchors.centerIn: parent
            PropertyAnimation {
                id: animation
                target: text
                property: "rotation"
                from: 0; to: 360; duration: 5000
                loops: Animation.Infinite
            }
        }
        MouseArea {
            anchors.fill: parent
            onClicked: animation.paused ? animation.resume() : animation.pau
se()
        }
        Component.onCompleted: animation.start()
    }
```



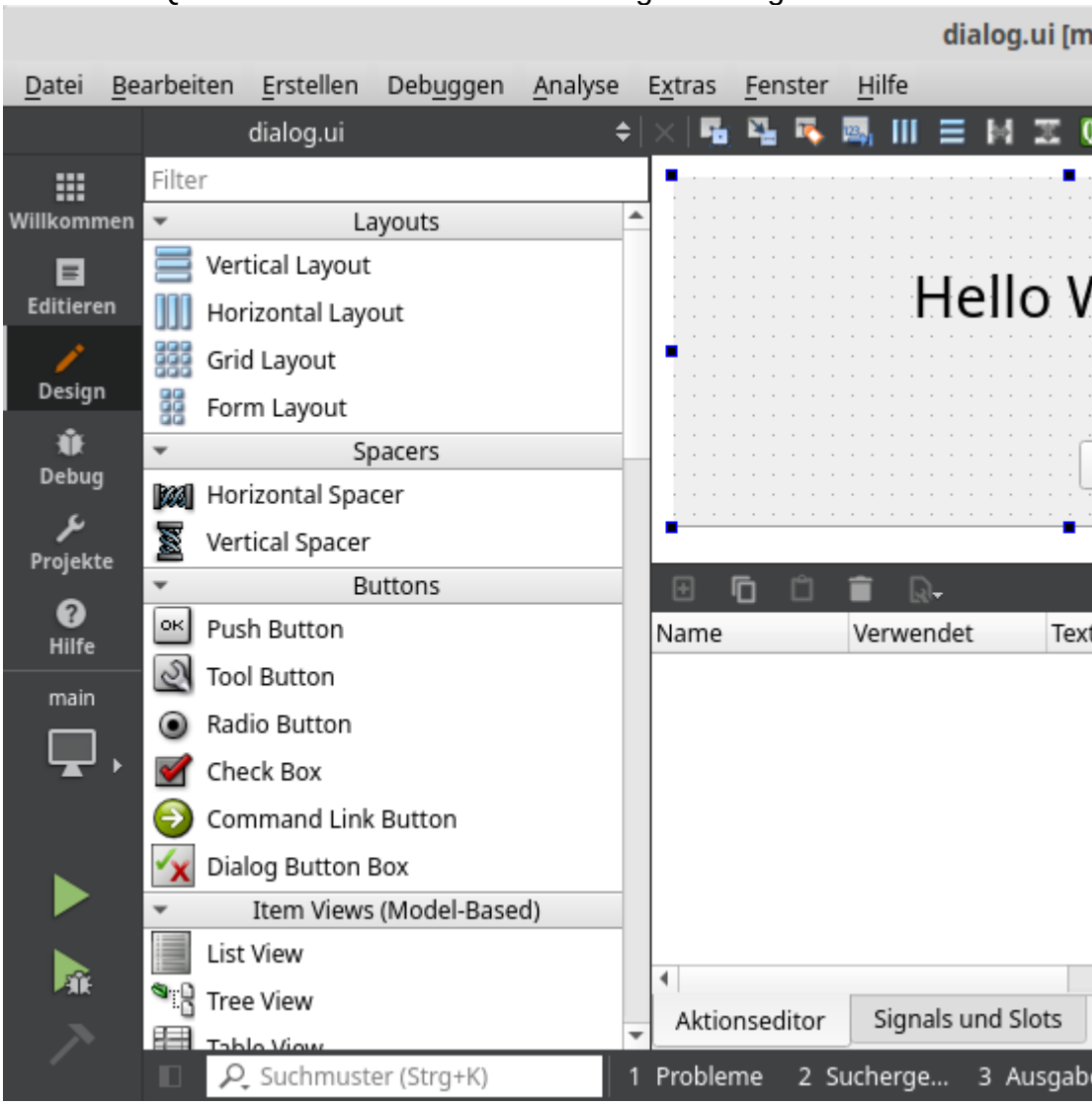
Wie bereits gesagt, gehe ich auf die Entwicklung von QML-Anwendungen in diesem Buch nicht näher ein.

Die Verwendung von QVBoxLayout wird etwas später in diesem Buch erläutert. In diesem Beispiel wird es benutzt, um ein Label und den Container vertikal anzuordnen.
Sei gewarnt nicht mehrere QQuickViews innerhalb eine Anwendung zu verwenden um Performance-Einbrüche zu vermeiden.

Dialog

Wenn du es bevorzugst die Fenster und Dialoge mit einem Designtool zu entwerfen, dann kannst du den QtCreator nutzen, welcher Bestandteil von Qt ist. Du kannst Qt und den QtCreator hier runterladen.

Starte den QtCreator und erstelle einen Dialog im Design-Mode.



Speicher den erstellten Dialog als *dialog.ui* ab. Im nachfolgenden Beipiel kannst du sehen, wie man den Dialog mit PyQt5 nutzen kann.

```
1  import sys
2  from PyQt5.QtWidgets import QApplication
3  from PyQt5.uic import loadUiType
4
5  UIClass, QtBaseClass = loadUiType("dialog.ui")
6
7  class MyApp(UIClass, QtBaseClass):
8      def __init__(self):
9          UIClass.__init__(self)
```

```
        QtBaseClass.__init__(self)
        self.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyApp()
    window.show()
    sys.exit(app.exec_())
```

Die Funktion loadUiType lädt den Dialog und liefert das Tupel (UIClass, QtBaseClass) zurück, welche wir nur noch initialisieren müssen.
Mit `setupUI()` wird dann der Dialog und all seine Widgets initialisiert und angezeigt.

Zusammenfassung

Wir haben vier Möglichkeiten gesehen, um Anwendungen mit Qt5 zu erstellen.
Der QWidgets-Ansatz wird meist bei Desktop-Anwendungen genutzt. Der QML-Ansatz wird meist genutzt, um Anwendungen für mobile Geräte zu erstellen. Und die Kombination kann genutzt werden um QML innerhalb von Desktop-Anwendungen darzustellen.
Wer lieber seine Userinfaces mit einem Designtool entwirft, wird wohl die letzte Variante verwenden.

Teil III - QtWidgets

Widgets und Layouts

Jetzt verwenden wir die am häufigsten genutzten Widgets in einer kleinen Desktop-Anwendung, um dir einen ersten Überblick über Widgets und Layouts zu geben.
Für mehr Details zu jedem Widget lade ich dich ein, im Internet zu suchen. Ein guter Startpunkit hierfür ist die folgende Webseite: https://doc.qt.io/qt-5/classes.html
Auf dieser Webweite findest du allerdings nur Beispiele, die in C++ geschrieben sind, aber du wirst sie dir, mit meiner Hilfe hier, leicht nach Python übersetzen können.
Hier ist ein Beispiel:

```
QPushButton *button = new QPushButton("&Download", this);
```

Da C++ streng typisiert ist wird hier die Variable button als Zeiger mit dem Typ QPushButton deklariert.
Der Stern "*" weist darauf hin, das `button` ein Zeiger ist. Das Schlüsselwort "new" kreiert eine Instanz der Klasse QPushButton und "this" ist ein Zeiger auf das aufrufende Fenster.
Nach Python übersetzt sieht es dann so aus:

```
button = QPushButton("&Download", self)
```

Wenn du ein Beispiel wie dieses findest,...

```
findDialog->show();
```

..., dann zeigt es hier wahrscheinlich einen Zeiger auf eine Instanz eines SuchDialoges, dessen Methode show() aufgerufen wird. Wir übersetzen das zu:

```
findDialog.show()
```

Manchmal deklariert man Variablen in C++ auch statisch ohne den "*" vor der Variable und ohne das Schlüsselwort "new". Ausserdem ändert sich der Operator zum Aufrufen von Methoden von "->" zu ".".

```
QFileDialog dialog(this);
dialog.setFileMode(QFileDialog::AnyFile);
```

Das bedeutet, das die Variable dialog mit dem Typ QFileDialog deklariert und mit "this" initialisiert wurde. In Python deklarieren wir das wie folgt:

```
dialog = QFileDialog(self)
dialog.setFileMode(QFileDialog.AnyFile)
```

In Python arbeiten wir hier immer nur mit Zeigern, wenn wir eine KLasse instanziieren. Beachte ausserdem, das wir in Python anstelle des "::" enum Operators einen "." benutzen, um auf ein Feld zu zugreifen.

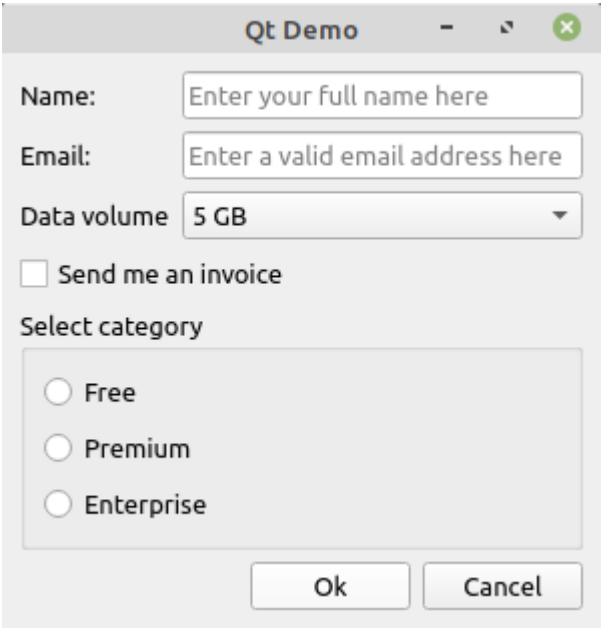
Aber nun ist es Zeit für ein Beispiel.

QWidget/BaseWidgets/main.py

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QLabel, QWidget, QGridLayout,
                             QLineEdit, QVBoxLayout, QHBoxLayout, QPushButton, QGroupBox,
                             QRadioButton, QCheckBox, QComboBox)
from PyQt5.QtCore import Qt


class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setWindowTitle("Qt Demo")
        widget = QWidget()
        layout = QGridLayout()
        hbox = QHBoxLayout()
        vbox = QVBoxLayout()
        self.name_edit = QLineEdit()
        self.name_edit.setPlaceholderText("Enter your full name here")
        self.name_edit.setMinimumWidth(200)
        self.email_edit = QLineEdit()
        self.email_edit.setPlaceholderText("Enter a valid email address here")
        group = QGroupBox("Select category")
        group.setLayout(vbox)
```

```
        self.free = QRadioButton("Free")
        self.premium = QRadioButton("Premium")
        self.enterprise = QRadioButton("Enterprise")
        vbox.addWidget(self.free)
        vbox.addWidget(self.premium)
        vbox.addWidget(self.enterprise)
        self.cb = QCheckBox("Send me an invoice")
        self.combo = QComboBox()
        self.combo.addItem("5 GB")
        self.combo.addItem("10 GB")
        self.combo.addItem("20 GB")
        self.combo.addItem("50 GB")
        self.combo.addItem("100 GB")
        ok_button = QPushButton("Ok")
        cancel_button = QPushButton("Cancel")
        hbox.addStretch()
        hbox.addWidget(ok_button)
        hbox.addWidget(cancel_button)
        layout.addWidget(QLabel("Name:"), 0, 0)
        layout.addWidget(self.name_edit, 0, 1)
        layout.addWidget(QLabel("Email:"), 1, 0)
        layout.addWidget(self.email_edit, 1, 1)
        layout.addWidget(QLabel("Data volume"), 2, 0)
        layout.addWidget(self.combo, 2, 1)
        layout.addWidget(self.cb, 3, 0, 1, 2)
        layout.addWidget(group, 4, 0, 1, 2)
        layout.addLayout(hbox, 5, 0, 1, 2)
        widget.setLayout(layout)
        self.setCentralWidget(widget)

        ok_button.clicked.connect(self.okClicked)

    def okClicked(self):
        name = self.name_edit.text()
        email = self.email_edit.text()
        data_volume = self.combo.currentText()
        send_invoice = self.cb.checkState() == Qt.Checked
        if self.free.isChecked():
            category = "Free"
        elif self.premium.isChecked():
            category = "Premium"
        elif self.enterprise.isChecked():
            category = "Enterprise"
        else:
            category = "None"

        print("name:", name)
        print("email:", email)
        print("data:", data_volume)
        print("invoice:", send_invoice)
```

```
            print("category:", category)

    if __name__ == "__main__":
        app = QApplication(sys.argv)
        win = MainWindow()
        win.show()
        sys.exit(app.exec())
```



Layout

In diesem Beispiel haben wir ein Fenster und innerhalb dieses Fensters haben wir ein simples Widget, welches als zentrales Widget gesetzt wird und lediglich dafür da ist, ein Layout (QGridView) zu beherbergen.
Ein QGridView arrangiert all seine Widgets in Zeilen und Spalten.
Die Spaltenanzahl ergibt sich aus der Nutzung des Grids.
Wenn du zum Beispiel ein Widget wie folgt zum Grid hinzufügst `layout.addWidget(widget, 0, 1)` dann wird das Widget in der ersten Zeile (0) in der zweiten (1) Spalte hinzugefügt. Das ergibt eine Spaltenanzahl von 2.

Spalte 0	Spalte 1
	Widget

Wenn du ausserdem eine Zeilen- und Spaltenspanne wie hier nutzt `layout.addWidget(widget, 0, 1, 1, 4)` dann ergibt sich daraus eine Spaltenanzahl von 5, da das Widget 4 Spalten umspannt.

Spalte 0	Spalte 1, 2, 3 und 4
	Widget

Innerhalb des Grid nutzen wir zwei weitere Layouts. Das QVBoxLayout, welches seine Widgets vertikal arrangiert und das QHBoyLayout, welches seine Widgets horizontal arrangiert. Im Falle der `hbox` nutzen wir ein stretch Objekt (.addStretch) um die Buttons auf der rechten Seite auszurichten. Dieses stretch Objekt ist ein unsichtbares Widget, welches den gesamt verfügbaren Platz einnimmt. Wenn wir von einer Spalte ausgehen, die 500 Pixel breit ist und dort zwei Buttons mit jeweils einer Breite von 100 Pixeln enthalten sind, dann wird das stretch Objekt 300 Pixel breit werden.

Um die Breite des Fensters zu setzen benutzen wir hier einen kleinen Trick:

```
self.name_edit.setMinimumWidth(200)
```

Wir setzen einfach die Mindestbreite eines der Widgets in der zweiten Spalte auf 200 Pixel und alle Widgets in der zweiten Spalten verbreitern sich auf 200.

Signals und Slots

Mit Signals und Slots wird in Qt ein Publisher-Subscriber-Pattern kreiert. In unserem Fall feuert der QPushButton, wenn er angeklickt wird, ein "clicked" Signal. Wenn wiederum das Fenster dieses Signal mit einem Slot, in diesem Fall mit der Methode okClicked verbunden hat, dann wird diese Methode jedes mal aufgerufen, wenn der Button angekilickt wird.
Das Signal wird mit folgendem Code verknüpft:

```
ok_button.clicked.connect(self.okClicked)
```

Wenn du mal ein C++ Beipiel zu diesem Thema finden solltest, dann sieht dieses etwas komplizierter aus:

```
connect(m_button, SIGNAL (clicked()), this, SLOT (handleButton()));
```

Es kann aber auch, seit Qt5, wie folgt aussehen.

```
connect(m_button, &Sender::clicked, this, &Receiver::okClicked);
```

Und wird wie folgt in Python übersetzt:

```
m_button.clicked.connect(self.handleButton)
```

Dieses Beispiel ist einer der Gründe, warum es mit Python viel schneller geht ein Programm zu schreiben als mit C++.

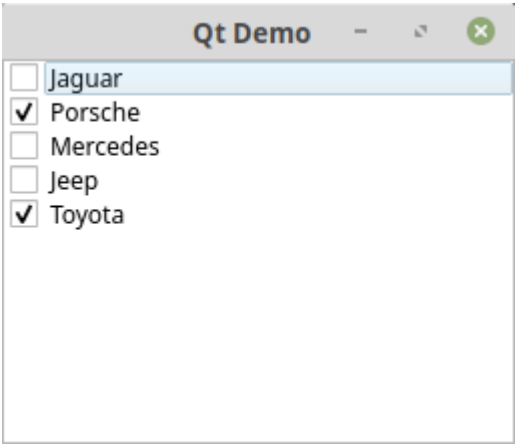
ListView

Einige Widgets wie der QListView arbeiten mit einem Datenmodell. Hier ist ein Beispiel.

QWidget/ListView/main.py

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QListView
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QStandardItem, QStandardItemModel


class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setWindowTitle("Qt Demo")
        list = QListView()
        self.model = QStandardItemModel(list)
        cars = [
            'Jaguar',
            'Porsche',
            'Mercedes',
            'Jeep',
            'Toyota'
        ]

        for car in cars:
            item = QStandardItem(car)
            item.setCheckable(True)
            self.model.appendRow(item)

        list.setModel(self.model)
        self.setCentralWidget(list)
```

```
28
29          self.model.itemChanged.connect(self.onItemChanged)
30
31      def onItemChanged(self, item):
32          if not item.checkState():
33              print(item.text() + " has been unchecked")
34          else:
35              print(item.text() + " has been checked")
36
37
38  if __name__ == "__main__":
39      app = QApplication(sys.argv)
40      win = MainWindow()
41      win.show()
42      sys.exit(app.exec_())
```



In diesem Beispiel nutzen wir das QStandardItemModel mit QStandardItem's für jeden Listeneintrag. Ganz einfach oder! Änderungen im Model können wir abfangen, in dem wir das Signal `itemChanged` mit einem Slot verbinden. Somit wird die Methode `onItemChanged` jedes Mal aufgerufen, wenn der Benutzer einen Haken in der Checkbox eines ListItems setzt oder löscht.
Im nächsten Abschnitt werden wir ein eigenes Datenmodell verwenden.

TreeView

In diesem TreeView Beipiel nutzen wir ein eigenes Datenmodell welches von QAbstractItemModel erbt.

QWidget/TreeView/treemodel.py

```
from PyQt5.QtCore import QAbstractItemModel, Qt, QModelIndex


class TreeModel(QAbstractItemModel):
    def __init__(self, in_nodes):
        QAbstractItemModel.__init__(self)
        self._root = TreeNode(None)
        for node in in_nodes:
            self._root.addChild(node)

    def rowCount(self, in_index):
        if in_index.isValid():
            return in_index.internalPointer().childCount()
        return self._root.childCount()

    def addChild(self, in_node, in_parent):
        if not in_parent or not in_parent.isValid():
            parent = self._root
        else:
            parent = in_parent.internalPointer()
        parent.addChild(in_node)

    def index(self, in_row, in_column, in_parent=None):
        if not in_parent or not in_parent.isValid():
            parent = self._root
        else:
            parent = in_parent.internalPointer()

        if not QAbstractItemModel.hasIndex(self, in_row, in_column, in_p
arent):
            return QtCore.QModelIndex()

        child = parent.child(in_row)
        if child:
            return QAbstractItemModel.createIndex(self, in_row, in_colum
n, child)
        else:
            return QModelIndex()

    def parent(self, in_index):
        if in_index.isValid():
            p = in_index.internalPointer().parent()
            if p:
                return QAbstractItemModel.createIndex(self, p.row(),0,p)

        return QModelIndex()

    def columnCount(self, in_index):
        if in_index.isValid():
            return in_index.internalPointer().columnCount()
```

```
        return self._root.columnCount()

    def data(self, in_index, role):
        if not in_index.isValid():
            return None
        node = in_index.internalPointer()
        if role == Qt.DisplayRole:
            return node.data(in_index.column())
        return None


class TreeNode(object):
    def __init__(self, in_data):
        self._data = in_data
        if type(in_data) == tuple:
            self._data = list(in_data)
        if type(in_data) is str or not hasattr(in_data, '__getitem__'):
            self._data = [in_data]

        self._columncount = len(self._data)
        self._children = []
        self._parent = None
        self._row = 0

    def data(self, in_column):
        if in_column >= 0 and in_column < len(self._data):
            return self._data[in_column]

    def columnCount(self):
        return self._columncount

    def childCount(self):
        return len(self._children)

    def child(self, in_row):
        if in_row >= 0 and in_row < self.childCount():
            return self._children[in_row]

    def parent(self):
        return self._parent

    def row(self):
        return self._row

    def addChild(self, in_child):
        in_child._parent = self
        in_child._row = len(self._children)
        self._children.append(in_child)
        self._columncount = max(in_child.columnCount(), self._columncoun
t)
```

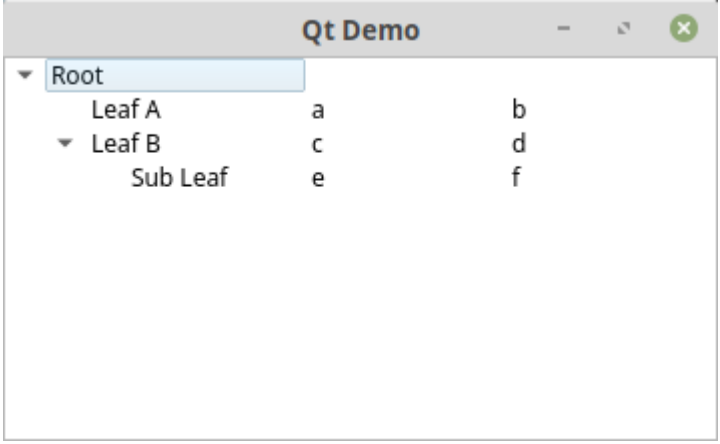
Wenn man von QAbstractItemModel erbt, bzw. es erweitert, dann muss man auf jeden Fall folgende Methoden implementieren:

- index() - Liefert einen Index auf einen Eintrag anhand der Zeile, Spalte und des Parent.
- parent() - Liefert den Parent des Items
- rowCount() - Liefert die Anzahl der Zeilen
- columnCount() - Liefert die Anzahl der Spalten
- data() - Liefert in unserem Beispiel lediglich den Dateneintrag

Diese Methoden werden in allen read-only Modellen benutzt. Sie formen die Basis für editierbare Modelle.

QWidget/TreeView/main.py

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QTreeView
from PyQt5.QtCore import Qt
from treemodel import TreeModel, TreeNode


class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        items = []
        items.append(TreeNode("Root"))
        items[0].addChild(TreeNode(["Leaf A", "a", "b"]))
        items[0].addChild(TreeNode(["Leaf B", "c", "d"]))
        items[0].child(1).addChild(TreeNode(["Sub Leaf", "e", "f"]))

        self.setWindowTitle("Qt Demo")
        tree = QTreeView()
        tree.setModel(TreeModel(items))
        tree.setHeaderHidden(True)
        tree.setColumnWidth(0, 150)
        self.setCentralWidget(tree)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MainWindow()
    win.show()
    sys.exit(app.exec_())
```



Abgesehen vom Datenmodell ist dies auch ein sehr simples Beispiel und ein guter Startpunkt um einen Treeview zu nutzen.

MessageDialog

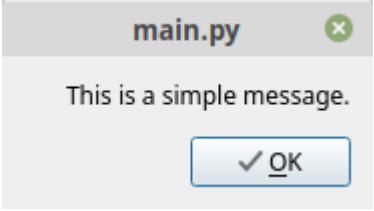
Hier ist ein Beispiel um ein paar Möglichkeiten aufzuzeigen, um eine MessageBox zu nutzen.

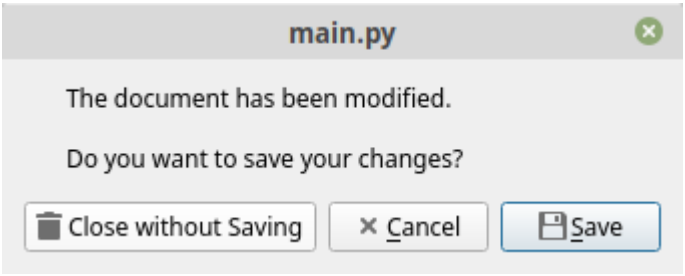
QWidget/MessageDialog/main.py

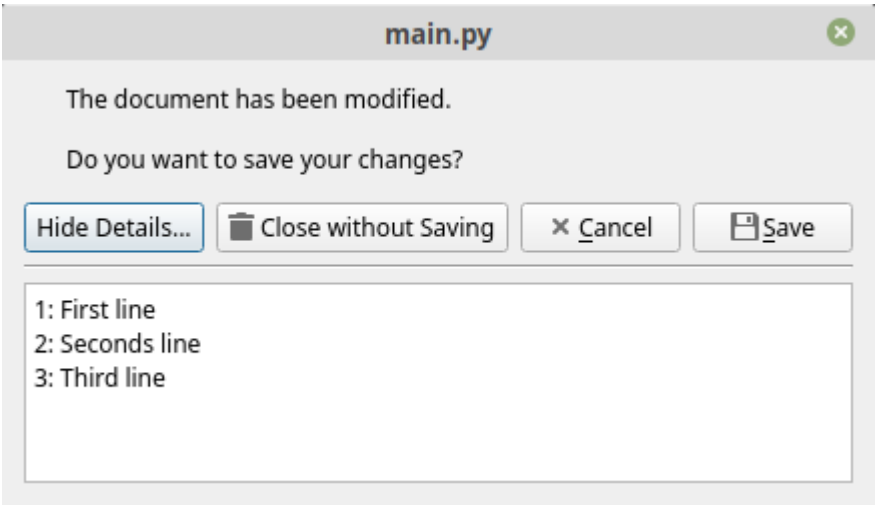
```
1   import sys
2   from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QWidget,
QVBoxLayout, QPushButton, QMessageBox
3   from PyQt5.QtCore import Qt
4
5
6   class MainWindow(QMainWindow):
7       def __init__(self):
8           QMainWindow.__init__(self)
9           self.setWindowTitle("Qt Demo")
10          widget = QWidget()
11          layout = QVBoxLayout()
12          simple = QPushButton("Simple")
13          save = QPushButton("Save")
14          details = QPushButton("Save with details")
15          warning = QPushButton("Warning")
16          layout.addWidget(simple)
17          layout.addWidget(save)
18          layout.addWidget(details)
19          layout.addWidget(warning)
20          widget.setLayout(layout)
21          self.setCentralWidget(widget)
22
```

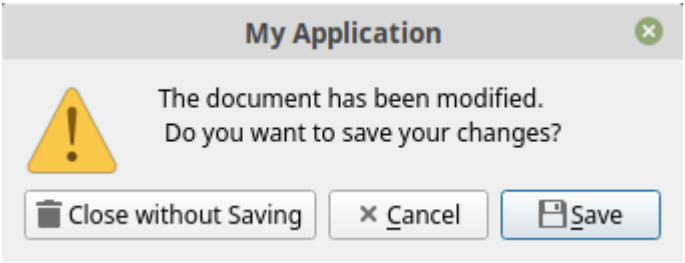
```
            simple.clicked.connect(self.simple)
            save.clicked.connect(self.save)
            details.clicked.connect(self.details)
            warning.clicked.connect(self.warning)

        def simple(self):
            msg = QMessageBox()
            msg.setText("This is a simple message.")
            msg.exec()

        def save(self):
            msg = QMessageBox()
            msg.setText("The document has been modified.")
            msg.setInformativeText("Do you want to save your changes?")
            msg.setStandardButtons(QMessageBox.Save | QMessageBox.Discard |
QMessageBox.Cancel)
            msg.setDefaultButton(QMessageBox.Save)
            ret = msg.exec()

        def details(self):
            msg = QMessageBox()
            msg.setText("The document has been modified.")
            msg.setInformativeText("Do you want to save your changes?")
            msg.setStandardButtons(QMessageBox.Save | QMessageBox.Discard |
QMessageBox.Cancel)
            msg.setDefaultButton(QMessageBox.Save)
            msg.setDetailedText("1: First line\n2: Seconds line\n3: Third
line")
            ret = msg.exec()

        def warning(self):
            ret = QMessageBox.warning(self, "My Application",
                "The document has been modified.\n Do you want to save your
changes?",
                QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel,
  QMessageBox.Save)

    if __name__ == "__main__":
        app = QApplication(sys.argv)
        win = MainWindow()
        win.show()
        sys.exit(app.exec())
```

Behalte bitte in Erinnerung, dass man nicht mehrere MessageBoxen in Folge nutzen sollte, da dies den Benutzer überfordern würde.
Um noch konkreter zu werden, sollte man MessageBoxen eigentlich nur dann verwenden, wenn man den Benutzer in seiner Arbeit unterbrechen möchte, um ihn auf einen Fehler oder ähnlichem hinzuweisen. Für beiläufige Nachrichten bietet sich die Statusbar an.









Wie in dem Beispiel mit dem Warning Dialog können wir das Icon auch mit der Methode `setIcon()` ändern.
Wir haben vier Icons zu auswählen:

```
msg.setIcon(QMessageBox.Question)
msg.setIcon(QMessageBox.Information)
msg.setIcon(QMessageBox.Warning)
msg.setIcon(QMessageBox.Critical)
```

Nutzer Definierte Widgets

Manchmal vermissen wir ein bestimmtes Widget und wenn wir es nirgendwo runterladen können, dann müssen wir es selber entwerfen.
Im nachfolgenden Beispiel werden wir ein DockWidget, eine ScrollArea und natürlich ein eigenes Widget namens Expander kennenlernen.
Ich erinnere mich daran, den Expander das erste Mal in einer Email-Anwendung unter Windows gesehen zu haben. Leider wurde dieses Widget nicht in Qt integriert. Die Nutzung des Expanders ist simpel. Klicke einfach in den Expander und der eigentliche Inhalt wird in einer Animation vergrössert und somit angezeigt. Der Inhalt kann eine Liste von Menueinträgen, ein Texteditor, eine Listbox oder was auch immer sein.

QWidget/UserDefined/main.py

```
import sys
from PyQt5.QtWidgets import (QSizePolicy, QListWidget, QListWidgetItem,
QApplication, QMainWindow, QLabel,
                QTextEdit, QVBoxLayout, QScrollArea, QDockWidget, QWidget)
from PyQt5.QtCore import Qt
from expander import Expander

class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setWindowTitle("Qt Expander Demo")
        self.resize(640, 480)

        edit = QTextEdit()
        edit.setPlainText("Lorem ipsum dolor...")
        self.content = Expander("Content", "parts.svg")
        self.images = Expander("Images", "images.svg")
        self.settings = Expander("Settings", "settings.svg")
        vbox = QVBoxLayout()
        vbox.addWidget(self.content)
        vbox.addWidget(self.images)
        vbox.addWidget(self.settings)
        vbox.addStretch()
        scroll_content = QWidget()
        scroll_content.setLayout(vbox)
        scroll = QScrollArea()
```

```
            scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarAsNeeded)
            scroll.setVerticalScrollBarPolicy(Qt.ScrollBarAsNeeded)
            scroll.setWidget(scroll_content)
            scroll.setWidgetResizable(True)
            scroll.setMaximumWidth(200)
            scroll.setMinimumWidth(200)
            self.navigationdock = QDockWidget("Navigation", self)
            self.navigationdock.setAllowedAreas(Qt.LeftDockWidgetArea | Qt.R
ightDockWidgetArea)
            self.navigationdock.setWidget(scroll)
            self.navigationdock.setObjectName("Navigation")
            self.addDockWidget(Qt.LeftDockWidgetArea, self.navigationdock)
            self.setCentralWidget(edit)

            # fill content
            self.content_list = QListWidget()
            self.content_list.setSizePolicy(QSizePolicy.Ignored,
QSizePolicy.Fixed)
            for i in range(5):
                item = QListWidgetItem()
                item.setText("Item " + str(i))
                self.content_list.addItem(item)
            content_box = QVBoxLayout()
            content_box.addWidget(self.content_list)
            self.content.addLayout(content_box)


            # fill images
            self.images_list = QListWidget()
            self.images_list.setSizePolicy(QSizePolicy.Ignored,
QSizePolicy.Fixed)
            for i in range(5):
                item = QListWidgetItem()
                item.setText("Image " + str(i))
                self.images_list.addItem(item)
            images_box = QVBoxLayout()
            images_box.addWidget(self.images_list)
            self.images.addLayout(images_box)


            #fill settings
            self.settings_list = QListWidget()
            self.settings_list.setSizePolicy(QSizePolicy.Ignored, QSizePolic
y.Fixed)
            for i in range(5):
                item = QListWidgetItem()
                item.setText("Setting " + str(i))
                self.settings_list.addItem(item)
            settings_box = QVBoxLayout()
            settings_box.addWidget(self.settings_list)
```

```
            self.settings.addLayout(settings_box)

            self.content.expanded.connect(self.contentExpanded)
            self.images.expanded.connect(self.imagesExpanded)
            self.settings.expanded.connect(self.settingsExpanded)

        def contentExpanded(self, value):
            if value:
                self.images.setExpanded(False)
                self.settings.setExpanded(False)

        def imagesExpanded(self, value):
            if value:
                self.content.setExpanded(False)
                self.settings.setExpanded(False)

        def settingsExpanded(self, value):
            if value:
                self.content.setExpanded(False)
                self.images.setExpanded(False)


    if __name__ == "__main__":
        app = QApplication(sys.argv)
        win = MainWindow()
        win.show()
        sys.exit(app.exec())
```

Das DockWidget kann der Nutzer überall dort hinbewegen, wo es erlaubt ist. In unserem Beispiel ist es eingeschränkt auf den linken und den rechten Rand des Fensters. Ein DockWidget kann der Benutzer mit der Maus verschieben, in dem er in die Titelzeile des Widgets klickt und das Widget mit gedrückter Maustaste verschiebt.

```
self.navigationdock.setAllowedAreas(Qt.LeftDockWidgetArea | Qt.RightDockWidge
tArea)
```

Eine ScrollArea kann Widgets aufnehmen, die grösser als der zur Verfügung stehende Bereich im Fenster ist. Wenn es nicht möglich ist, das komplette Widget anzuzeigen, dann wird jeweils eine Scrollbar angezeigt, mit der der Nutzer in der Lage ist, das ganze Widget zu sichten. In unserem Beispiel werden die Scrollbars nur angezeigt, wenn etwas zu scrollen gibt (Qt.ScrollAsNeeded). Wenn das Widget in die ScrollArea reinpasst, werden keine ScrollBars angezeigt.

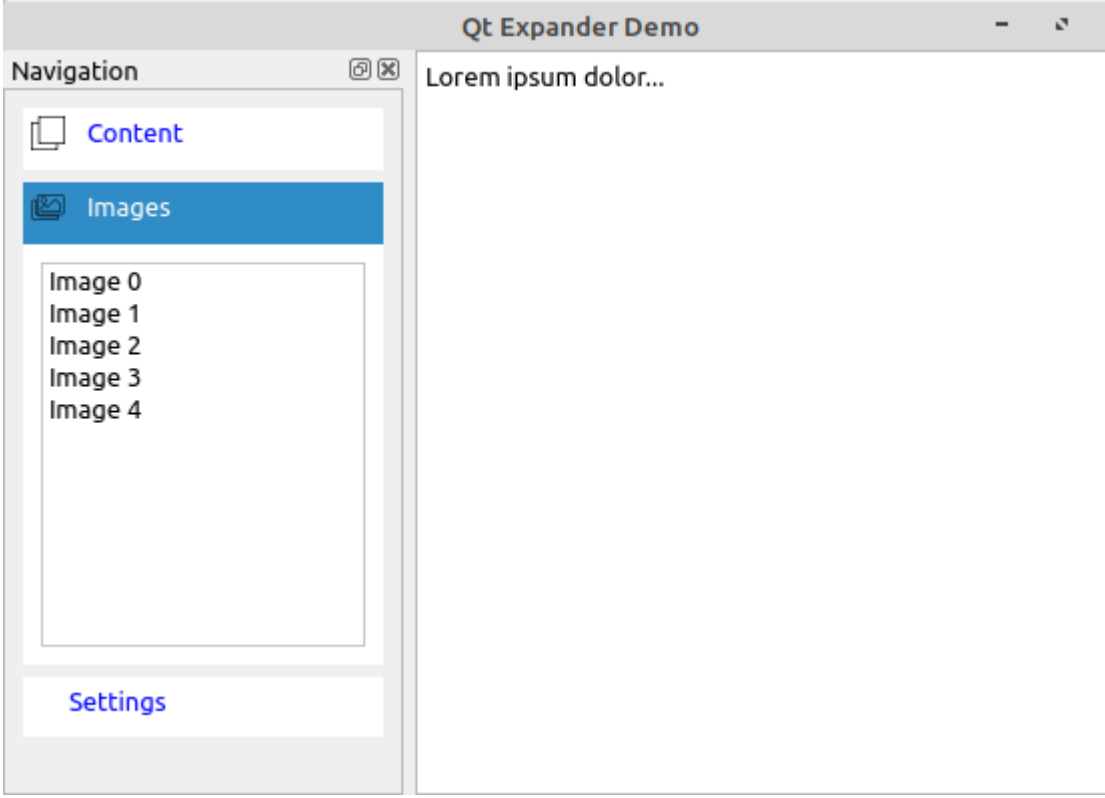
QWidget/UserDefined/expander.py

```
    import os
    from PyQt5.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout, QLabel
    from PyQt5.QtCore import (QParallelAnimationGroup, QPropertyAnimation, Qt, pyqtProperty, pyqtSignal, QDir,
                              QFile, QIODevice)
    from PyQt5.QtGui import QImage, QPalette, QPixmap, QColor, QIcon


    class Expander(QWidget):
        expanded = pyqtSignal(object)
        clicked = pyqtSignal()

        def __init__(self, header, svg):
            QWidget.__init__(self)
            self.svg = svg
            self.is_expanded = False
            self.text = header
            self.icon = QLabel()
            self.hyper = QLabel()
            self.setColors()
            self.setCursor(Qt.PointingHandCursor)
            self.setAttribute(Qt.WA_Hover, True)
            self.setAutoFillBackground(True)

            vbox = QVBoxLayout()
            hbox = QHBoxLayout()

            hbox.addWidget(self.icon)
            hbox.addSpacing(5)
            hbox.addWidget(self.hyper)
            hbox.addStretch()
            hbox.setContentsMargins(4, 4, 4, 4)
            vbox.addLayout(hbox)
            self.content = QWidget()
            self.content.setStyleSheet("background-color: " + self.palette().base().color().name())
            self.content.setMaximumHeight(0)
            vbox.addWidget(self.content)
            vbox.setContentsMargins(0, 0, 0, 0)
            self.setLayout(vbox)
            self.hyper.linkActivated.connect(self.buttonClicked)

            self.anim = QParallelAnimationGroup()
            self.height_anim = QPropertyAnimation(self.content, "maximumHeight".encode("utf-8"))
            self.color_anim = QPropertyAnimation(self, "color".encode("utf-8"))
            self.height_anim.setDuration(200)
            self.color_anim.setDuration(200)
            self.anim.addAnimation(self.height_anim)
```

```
            self.anim.addAnimation(self.color_anim)

        def setColors(self):
            self.label_normal_color = self.palette().link().color().name()
            self.label_hovered_color = self.palette().highlight().color().na
me()
            self.label_selected_color = self.palette().highlightedText().col
or().name()
            self.normal_color = self.palette().base().color().name()
            self.selected_color = self.palette().highlight().color()
            self.hovered_color = self.palette().alternateBase().color()

            self.normal_icon = QPixmap(self.createIcon(self.svg, self.normal
_color))
            self.hovered_icon = QPixmap(self.createIcon(self.svg, self.label
_hovered_color))
            self.selected_icon = QPixmap(self.createIcon(self.svg, self.labe
l_hovered_color))

            self.icon.setPixmap(self.normal_icon)
            self.color = self.normal_color

            self.hyper.setText("<a style=\"color: " + self.label_normal_colo
r +
           " text-decoration: none\" href=\"#\">" + self.text + "</a>")

        def createIcon(self, source, hilite_color):
            temp = QDir.tempPath()
            file = QFile(source)
            file.open(QIODevice.ReadOnly | QIODevice.Text)
            data = str(file.readAll(), encoding="utf-8")
            file.close()

            out = os.path.join(temp, hilite_color + ".svg")
            with open(out, "w") as fp:
                fp.write(data.replace("#ff00ff", hilite_color))
            return out

        def setExpanded(self, value):
            if value == self.is_expanded:
                return

            if value:
                self.is_expanded = True
                pal = self.palette()
                pal.setColor(QPalette.Background, self.selected_color)
                self.setPalette(pal)
                self.icon.setPixmap(self.selected_icon)
                self.hyper.setText("<a style=\"color: " + self.label_selecte
d_color +
```

```
            "; text-decoration: none;\" href=\"#\">" + self.text + "</
a>")
        else:
            self.is_expanded = False
            pal = self.palette()
            pal.setColor(QPalette.Background, QColor(self.normal_color))
            self.setPalette(pal)
            self.icon.setPixmap(self.normal_icon)
            self.hyper.setText("<a style=\"color: " + self.label_normal_
color +
            "; text-decoration: none;\" href=\"#\">" + self.text + "</
a>")

        if self.is_expanded:
                self.expandContent()
            else:
                self.collapseContent()
            self.expanded.emit(self.is_expanded)

        def addLayout(self, layout):
            self.content.setLayout(layout)

    @pyqtProperty('QColor')
    def color(self):
        return Qt.black

    @color.setter
    def color(self, color):
        pal = self.palette()
        pal.setColor(QPalette.Background, QColor(color))
        self.setPalette(pal)

    def mouseReleaseEvent(self, me):
        if me.button() != Qt.LeftButton or me.y() > 32:
            return
        self.setExpanded(not self.is_expanded)
        if self.is_expanded:
            self.clicked.emit()

    def expandContent(self):
        if self.content.layout():
            self.height_anim.setEndValue(self.content.layout().sizeHint().height())
        else:
            self.height_anim.setEndValue(0)
        self.height_anim.setStartValue(0)
        self.color_anim.setStartValue(self.normal_color)
```

```
            self.color_anim.setEndValue(self.selected_color)
            self.anim.start()

        def collapseContent(self):
            if self.content.layout():

self.height_anim.setStartValue(self.content.layout().sizeHint().height())
            else:
                self.height_anim.setStartValue(0)
                self.height_anim.setEndValue(0)

            self.color_anim.setStartValue(self.selected_color)
            self.color_anim.setEndValue(self.normal_color)
            self.anim.start()

        def buttonClicked(self):
            self.setExpanded(not self.is_expanded)
            if self.is_expanded:
                self.clicked.emit()

        def enterEvent(self, event):
            if not self.is_expa
            nded:
                pal = self.palette()
                pal.setColor(QPalette.Background, QColor(self.hovered_c
olor))
                self.setPalette(pal)
                self.icon.setPixmap(self.hovered_icon)
                self.hyper.setText("<a style=\"color: " + self.lab
el_hovered_color +
                  "; text-decoration: none;\" href=\"#\">" + self.text + "</
a>")
            QWidget.enterEvent(self, event)

        def leaveEvent(self, event):
            if not self.is_expanded:
                pal = self.palette()
                pal.setColor(QPalette.Background, Q
Color(self.normal_color))
                self.setPalette(pal)
                self.icon.setPixmap(self.normal_icon)
                self.hyper.setText("<a style=\"color: " + self.lab
el_normal_color +
                  "; text-decoration: none;\" href=\"#\">" + self.text + "</
a>")
            QWidget.leaveEvent(self, event)

```



Für den Expander nutzen wir ein paar Tricks. Zu allererst nutzen wir einen Label als Hyperlink.

```
self.hyper.setText("<a style=\"color: " + self.label_normal_color +
    " text-decoration: none\" href=\"#\">" + self.text + "</a>")
```

Wenn wir einen QLabel wie hier benutzen, dann können wir auch auf das Signal `linkActivated` zugreifen und ihn mit einem Slot verbinden.

```
self.hyper.linkActivated.connect(self.buttonClicked)
```

Der nächste Trick ist die Nutzung einer Animation (QParallelAnimationGroup, QPropertyAnimation) mit der wir eine Transition erzeugen können, wenn der Expander erweitert oder verkleinert wird, damit das Ganze recht hübsch aussieht. In unserem Fall animieren wir die `maximumHeight` und die `color` des Widgets.
Um die Farbe `color` animieren zu können, mussten wir ein Property deklarieren. Der Getter ist hier nur ein Dummy, der die Farbe Schwarz zurückliefert.

```
    #getter
    @pyqtProperty('QColor')
    def color(self):
        return Qt.black

    #setter
    @color.setter
    def color(self, color):
        pal = self.palette()
        pal.setColor(QPalette.Background, QColor(color))
        self.setPalette(pal)
```

Ein anderer Trick ist die Nutzung einer SVG (**S**calable **V**ector **G**raphic) Grafik für das Icon. Für diesen Trick öffnen wir die Datei mit dem SVG, ersetzen eine bestimmte Farbe im Text mit der aktuellen Hilite-Farbe (SVG liegt im Textformat vor) und erstellen eine Pixmap auf Basis der SVG-Grafik. Die SVG Grafiken kann man z.B. mit InkScape erstellen und eine bestimmte Farbe auf #ff00ff setzen, damit sie sich einfach wiederfinden lässt und ersetzt werden kann.
Da SVG auf XML basiert kann man den Text einfach ersetzen.

```
data.replace("#ff00ff", hilite_color)
```

MarkownEditor

Am Ende die Kapitels werden wir eine etwas komplexere Anwendung erstellen um ein paar noch nicht benutzte Widgets wie das Menu, MenuItem, StatusBar, Action, Splitter, FileDialog, Settings und den WebEngineView in Aktion zu sehen.

```
    import sys
    import os
    from PyQt5.QtWidgets import (QApplication, QMainWindow, QSplitter, QTextEdit, QAction,
                                 QMessageBox, QFileDialog, QDialog, QStyleFactory)
    from PyQt5.QtCore import Qt, QCoreApplication, QSettings, QByteArray, QUrl
    from PyQt5.QtGui import QIcon, QKeySequence
    from PyQt5.QtWebEngineWidgets import QWebEngineView
    import markdown2


    class MainWindow(QMainWindow):
        def __init__(self):
```

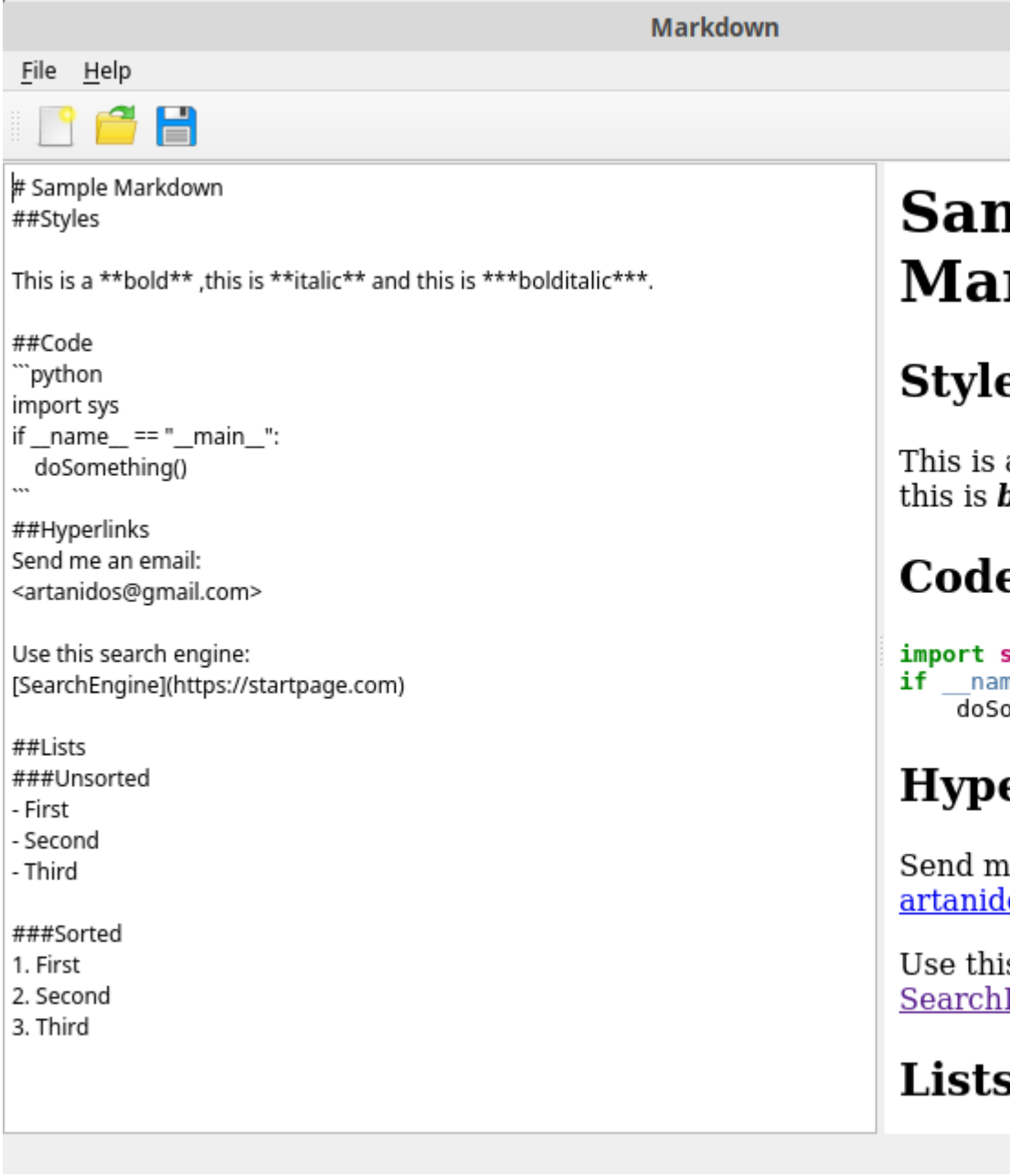
```
            QMainWindow.__init__(self)
            self.cur_file = ""
            self.splitter = QSplitter()
            self.text_edit = QTextEdit("")
            self.preview = QWebEngineView()
            self.preview.setMinimumWidth(300)
            self.setWindowTitle("Markdown [*]")
            self.splitter.addWidget(self.text_edit)
            self.splitter.addWidget(self.preview)
            self.setCentralWidget(self.splitter)
            self.createMenus()
            self.createStatusBar()
            self.readSettings()
            self.text_edit.document().contentsChanged.connect(self.documentW
asModified)
            self.text_edit.textChanged.connect(self.textChanged)

        def closeEvent(self, event):
            if self.maybeSave():
                self.writeSettings()
                event.accept()
            else:
                event.ignore()

        def documentWasModified(self):
            self.setWindowModified(self.text_edit.document().isModified())

        def createMenus(self):
            new_icon = QIcon("./assets/new.png")
            open_icon = QIcon("./assets/open.png")
            save_icon = QIcon("./assets/save.png")
            save_as_icon = QIcon("./assets/save_as.png")
            exit_icon = QIcon("./assets/exit.png")

            new_act = QAction(new_icon, "&New", self)
            new_act.setShortcuts(QKeySequence.New)
            new_act.setStatusTip("Create a new file")
            new_act.triggered.connect(self.newFile)

            open_act = QAction(open_icon, "&Open", self)
            open_act.setShortcuts(QKeySequence.Open)
            open_act.setStatusTip("Open an existing file")
            open_act.triggered.connect(self.open)

            save_act = QAction(save_icon, "&Save", self)
            save_act.setShortcuts(QKeySequence.Save)
            save_act.setStatusTip("Save the document to disk")
            save_act.triggered.connect(self.save)

            save_as_act = QAction(save_as_icon, "Save &As...", self)
```

```
        save_as_act.setShortcuts(QKeySequence.SaveAs)
        save_as_act.setStatusTip("Save the document under a new name")
        save_as_act.triggered.connect(self.saveAs)

        exit_act = QAction(exit_icon, "E&xit", self)
        exit_act.setShortcuts(QKeySequence.Quit)
        exit_act.setStatusTip("Exit the application")
        exit_act.triggered.connect(self.close)

        about_act = QAction("&About", self)
        about_act.triggered.connect(self.about)
        about_act.setStatusTip("Show the application's About box")

        file_menu = self.menuBar().addMenu("&File")
        file_menu.addAction(new_act)
        file_menu.addAction(open_act)
        file_menu.addAction(save_act)
        file_menu.addAction(save_as_act)
        file_menu.addSeparator()
        file_menu.addAction(exit_act)

        help_menu = self.menuBar().addMenu("&Help")
        help_menu.addAction(about_act)

        file_tool_bar = self.addToolBar("File")
        file_tool_bar.addAction(new_act)
        file_tool_bar.addAction(open_act)
        file_tool_bar.addAction(save_act)

    def createStatusBar(self):
        self.statusBar().showMessage("Ready")

    def about(self):
        QMessageBox.about(self, "About Markdown",
            "This app demonstrates how to "
               "write modern GUI applications using Qt, with a menu
bar, "
               "toolbars, and a status bar.")

    def newFile(self):
         if self.maybeSave():
                self.text_edit.clear()
                 self.setCurrentFile("")

                     def open(self):
                             if self.maybeSave():
                                     fileName = QFileDialog.ge
                              tOpenFileName(self)[0]
        if fileName:
                self.loadFile(fileName)
```

```
        def save(self):
            if not self.cur_file:
                return self.saveAs()
            else:
                return self.saveFile(self.cur_file)

        def saveAs(self):
            dialog = QFileDialog(self)
            dialog.setWindowModality(Qt.WindowModal)
            dialog.setAcceptMode(QFileDialog.AcceptSave)
            if dialog.exec() != QDialog.Accepted:
                return False
            return self.saveFile(dialog.selectedFiles()[0])

        def maybeSave(self):
            if not self.text_edit.document().isModified():
                return True
            ret = QMessageBox.warning(self, "Qt Demo",
                                      "The document has been modified.\n"
                                      "Do you want to save your changes?",
                                      QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel)
            if ret == QMessageBox.Save:
                return self.save()
            elif ret == QMessageBox.Cancel:
                return False
            return True

        def loadFile(self, fileName):
            with open(fileName, mode="r") as f:
                text = f.read()

            QApplication.setOverrideCursor(Qt.WaitCursor)
            self.setCurrentFile(fileName)
            self.text_edit.setPlainText(text)
            self.text_edit.document().setModified(False)
            self.setWindowModified(False)
            QApplication.restoreOverrideCursor()
            self.statusBar().showMessage("File loaded", 2000)

        def saveFile(self, fileName):
            QApplication.setOverrideCursor(Qt.WaitCursor)
            with open(fileName, "w") as f:
```

```
                f.write(self.text_edit.toPlainText())
                QApplication.restoreOverrideCursor()

                self.setCurrentFile(fileName)
                self.text_edit.document().setModified(False)
                self.setWindowModified(False)

                self.statusBar().showMessage("File saved", 2000)

        def setCurrentFile(self, fileName):
            self.cur_file = fileName
            shown_name = self.cur_file
            if not self.cur_file:
                shown_name = "untitled.txt"
            self.setWindowFilePath(shown_name)

        def writeSettings(self):
            settings = QSettings(QCoreApplication.organizationName(), QCoreApplication.applicationName())
            settings.setValue("geometry", self.saveGeometry())

        def readSettings(self):
            settings = QSettings(QCoreApplication.organizationName(), QCoreApplication.applicationName());
            geometry = settings.value("geometry", QByteArray())
            if not geometry:
                availableGeometry = QApplication.desktop().availableGeometry(self)
                self.resize(availableGeometry.width() / 3, availableGeometry.height() / 2)
                self.move((availableGeometry.width() - self.width()) / 2,
                          (availableGeometry.height() - self.height()) / 2)
            else:
                self.restoreGeometry(geometry)

        def textChanged(self):
            path = os.getcwd()
            html = "<html><head><link href=\"assets/pastie.css\" rel=\"stylesheet\" type=\"text/css\"/></head><body>"
            html += markdown2.markdown(self.text_edit.toPlainText(), ..., extras=["fenced-code-blocks"])
            html += "</body></html>"
            self.preview.setHtml(html, baseUrl = QUrl("file://" + path + "/"))


```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    QCoreApplication.setOrganizationName("Book")
    QCoreApplication.setApplicationName("MarkdownEditor")
    win = MainWindow()
    win.show()
    sys.exit(app.exec_())

```



Der WebEngineView ist ein voll fuktionsfähiges HTML-Browser-Widget und wird hier benutzt, um das aus dem Markdown erzeugte HTML darzustellen.

Das Modul markdown2 wird hier benutzt um markdown in HTML zu verwandeln. Du kannst es wie folgt installieren:

```
user@machine:/path$ pip install markdown2
```

Die Methode `writeSettings` wird verwendet um die Geometrie des Fensters in einer Datei zu speichern und mit der Methode `readSettings` werden die Werte verwendet um die Grösse und die letzte Position des Fensters wieder zu benutzen.

Die Methode `textChanged` wird jedes mal aufgerufen, wenn der Nutzer den Text im Editor ändert. In meinem Programm EbookCreator, das auf diesem Beispiel basiert, musste ich das Verhalten aus Gründen der Performance etwas anpassen. Der Code wird dort in einem Thread ausgeführt.

Die Aktionen wie new_act, open_act und save_act werden hier doppelt genutzt. Einmal im Menu und das andere Mal in der Toolbar.

Wenn du eine Aktion für ein Menu definierst, dann markierst du den Schnellzugriff mit "&". Zum Beispiel das "S" in der Action "&Save". Diese Buchstaben dürfen nur einmal unter einem Menu benutzt werden. Wenn du also "&Save" und "&Save as" nutzt, dann wird das SaveAs niemals mit einem Schnellzugriff aufgerufen werden können. Wir weichen hier auf "Save &As" aus.

Das Widget QTextEdit kann mit einfachem Text und mit RichText betrieben werden. Hierfür nutzen wir die Methode `self.text_edit.setPlainText(text)` um den Inhalt zu setzen.

In unserem Beispiel wird die Statusbar benutzt um Nachrichten anzuzeigen `self.statusBar().showMessage("File loaded", 2000)` und um Infos für einige Menuitems `exit_act.setStatusTip("Exit the application")` anzuzeigen.

Der Editor aus obigem Beispiel war die Basis für meine Anwendung EbookCreator, mit der ich dieses Buch geschrieben habe. Der EbookCreator ist auch in Python und PyQt5 geschrieben und kann dir als Inspiration dienen. Der Sourcecode ist hier verfügbar.

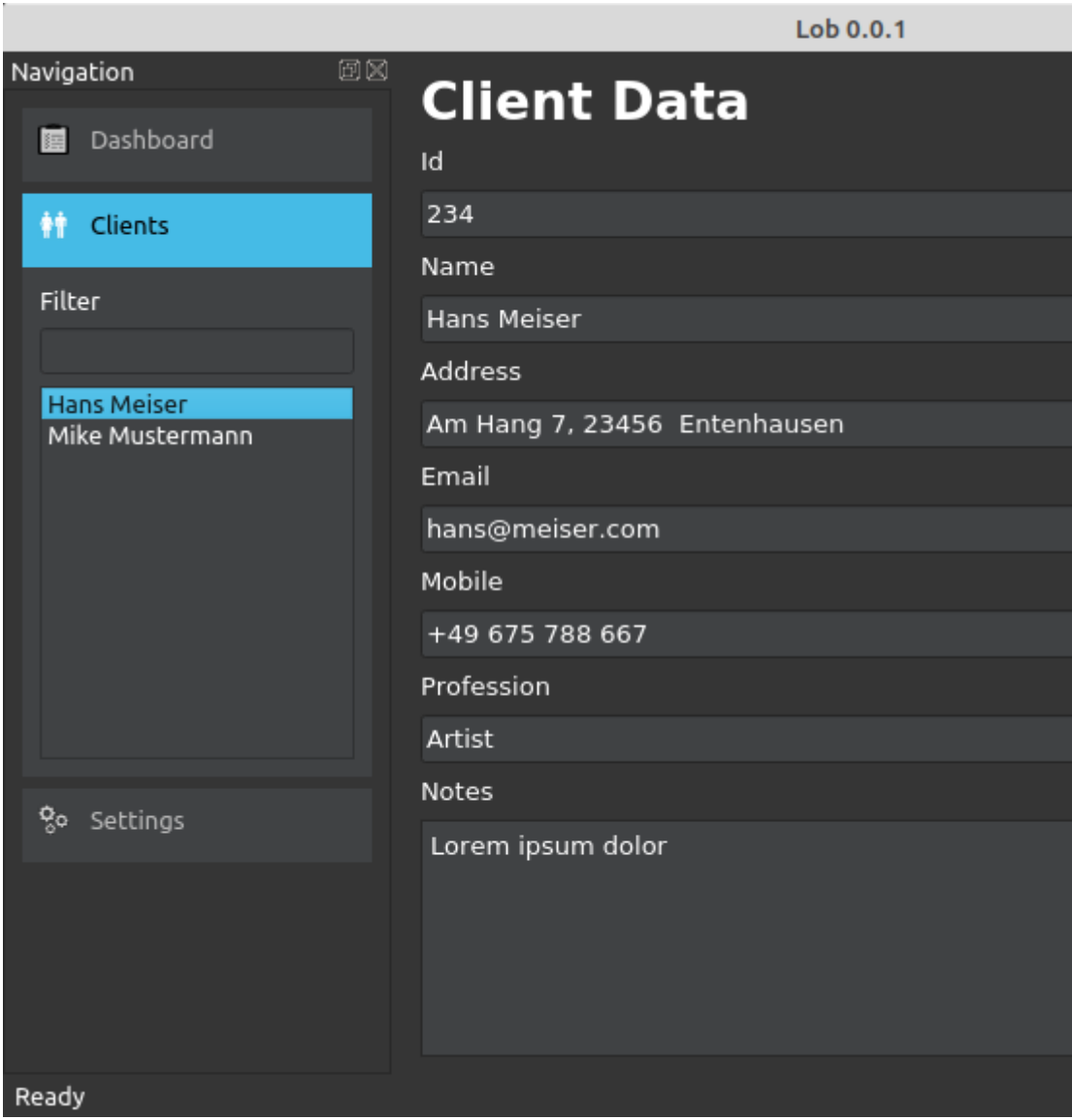
Zusammenfassung

Wir haben gesehen, wie man einige Widgets in einer Desktop-Anwendung benutzt und wie man Datenmodelle schreibt. Wir haben gesehen, wie eine komplette Anwendung, geschrieben in Python, aussehen kann und

ausserdem haben wir ein paar Triks gesehen, mit der wir einer Anwendung ein bisschen Leben einhauchen können.

Teil IV LOB

Erstelle eine Line Of Business Applikation



Ein LOB ist eine Anwendung, bei der du über eine grafische Benutzeroberfläche verfügst und die Daten in einer Datenbank gespeichert sind. In diesem Fall haben wir eine Datenbank mit Kunden und können einen neuen Kunden hinzufügen, die Liste der Kunden filtern und die Daten

für einen Kunden bearbeiten. Wir können auch ein Bild für jeden Kunden speichern.

Um die Beispiel-App auszuführen, müsst du das Modul tinydb installieren.

```
user@machine:/path$ pip3 install tinydb
```

Tinydb ist eine sehr kleine, aber nützliche Datenbankimplementierung, die in Python geschrieben wurde. Du benötigst hierfür keinen Server, auf dem die Daten gespeichert werden. Und du brauchst auch keine SQL-Sprache. Tinydb speichert einfach pythonische Daten in einer einzigen Datei.

Da diese App etwas größer ist als alle anderen, die ich hier gepostet habe, drucke ich hier nicht den gesamten Quellcode. Die Quellen findst du hier: github.

Main

Wie vorher habe ich die Hauptroutine in die *main.py* im Lob-Ordner gestellt. Dort initialisiere ich das Anwendungsobjekt und setze den Stil auf Fusion und initialisiere die Farbpalette.

```
app = QApplication(sys.argv)
app.setStyle(QStyleFactory.create("Fusion"))
app.setStyleSheet("QPushButton:hover { color: #45bbe6 }")

p = app.palette()
p.setColor(QPalette.Window, QColor(53, 53, 53))
p.setColor(QPalette.WindowText, Qt.white)
p.setColor(QPalette.Base, QColor(64, 66, 68))
p.setColor(QPalette.AlternateBase, QColor(53, 53, 53))
p.setColor(QPalette.ToolTipBase, Qt.white)
p.setColor(QPalette.ToolTipText, Qt.black)
p.setColor(QPalette.Text, Qt.white)
p.setColor(QPalette.Button, QColor(53, 53, 53))
p.setColor(QPalette.ButtonText, Qt.white)
p.setColor(QPalette.BrightText, Qt.red)
p.setColor(QPalette.Highlight, QColor("#45bbe6"))
p.setColor(QPalette.HighlightedText, Qt.black)
p.setColor(QPalette.Disabled, QPalette.Text, Qt.darkGray)
p.setColor(QPalette.Disabled, QPalette.ButtonText, Qt.darkGray)
p.setColor(QPalette.Link, QColor("#bbb"))
app.setPalette(p)
```

Die Farbe kann auf zwei verschiedene Arten eingestellt werden. Du kannst die RGB-Werte `QColor (53, 53, 53)` verwenden, oder du kannst eine Zeichenfolge `QColor (" # 45bbe6 "` verwenden, wie wir es in HTML tun.

Diesmal ist der Code etwas strukturierter. Die Datei *main.py* befindet sich noch im Stammordner, sodass ein Anwender sie möglicherweise sofort sieht, damit der Benutzer weiß, was er starten soll. Alle anderen Python-Dateien werden im Widgets-Ordner gespeichert.

MainWindow

Der visuelle Teil der Anwendung beginnt in der Klasse MainWindow, die ich in der Datei *mainwindow.py* gespeichert habe.

Bei der init-Methode machen wir einige Dinge etwas anders als in den letzten Kapiteln.

```
  def __init__(self, app):
      QMainWindow.__init__(self)
      self.app = app
      self.clients = None

      self.initGui()
      self.readSettings()
      self.dashboard.setExpanded(True)
      self.showDashboard()
      self.loadDatabase()
      self.loadClients()
      self.statusBar().showMessage("Ready")
```

Zunächst haben wir einen Parameter, mit dem wir das Anwendungsobjekt aus der Hauptroutine abrufen. Wir speichern es für später. In diesem speziellen Fall müssen Sie die Schriftart der App festlegen, nachdem der Benutzer die Anwendungseinstellungen geändert hat.

Um den Code ein wenig anzuordnen, habe ich einige Methoden erstellt, die in einer bestimmten Reihenfolge aufgerufen werden sollen. Die erste Methode heißt `initGUI()` , wo wir die visuellen Komponenten für dieses Fenster erstellen.

Dann haben wir die Methode `readSettings()` , mit der wir einige Parameter lesen, die wir beim Beenden der App gespeichert haben. Zum

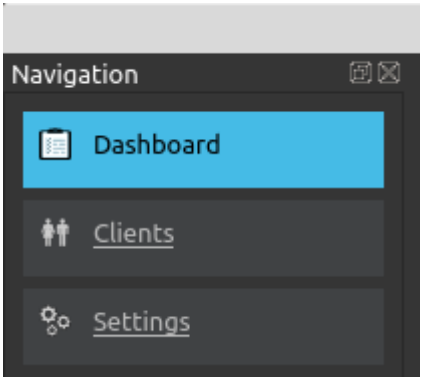
Beispiel haben wir die Fensterposition und die Größe gespeichert und dann bei dieser Methode nur diese Parameter wiederhergestellt.

Der Rest der Methoden ist meiner Meinung nach selbsterklärend. Aber wir werden das später im Detail durchgehen.

InitGui

In der initGui-Methode richten wir die Benutzeroberfläche ein.

Das erste, was wir wollen, ist eine Navigation. Ich habe dies im Admin-Client für WordPress gesehen und es hat mir gefallen. Wenn Sie im geöffneten Zustand auf einen Expander klicken, werden andere Expander mit einer schönen Animation (Transition) geschlossen. Weitere Details zu diesem Steuerelement findest du weiter unten im Kapitel Expander.




```
  def initGui(self):
          self.dashboard = Expander("Dashboard", ":/images/dashboard_normal.png", ":/images/dashboard_hover.png", ":/images/dashboard_selected.png")
          self.content = Expander("Clients", ":/images/clients_normal.png", ":/images/clients_hover.png", ":/images/clients_selected.png")
          self.settings = Expander("Settings", ":/images/settings_normal.png", ":/images/settings_hover.png", ":/images/settings_selected.png")

          self.setWindowTitle(QCoreApplication.applicationName() + " " + QCoreApplication.applicationVersion())
          vbox = QVBoxLayout()
          vbox.addWidget(self.dashboard)
          vbox.addWidget(self.content)
          vbox.addWidget(self.settings)
          vbox.addStretch()
```

```
            content_box = QVBoxLayout()
            filter_label = QLabel("Filter")
            self.filter = QLineEdit()
            self.filter.textChanged.connect(self.filterChanged)
            self.client_list = QListWidget()
            self.client_list.setSizePolicy(QSizePolicy.Ignored,
QSizePolicy.Fixed)
            self.client_list.currentItemChanged.connect(self.clientChanged)

            button_layout = QHBoxLayout()
            plus_button = FlatButton(":/images/plus.svg")
            self.trash_button = FlatButton(":/images/trash.svg")
            self.trash_button.enabled = False
            button_layout.addWidget(plus_button)
            button_layout.addWidget(self.trash_button)
            content_box.addWidget(filter_label)
            content_box.addWidget(self.filter)
            content_box.addWidget(self.client_list)
            content_box.addLayout(button_layout)
            self.content.addLayout(content_box)

            scroll_content = QWidget()
            scroll_content.setLayout(vbox)
            scroll = QScrollArea()
            scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarAsNeeded)
            scroll.setVerticalScrollBarPolicy(Qt.ScrollBarAsNeeded)
            scroll.setWidget(scroll_content)
            scroll.setWidgetResizable(True)
            scroll.setMaximumWidth(200)
            scroll.setMinimumWidth(200)

            self.navigationdock = QDockWidget("Navigation", self)
            self.navigationdock.setAllowedAreas(Qt.LeftDockWidgetArea | Qt.R
ightDockWidgetArea)
            self.navigationdock.setWidget(scroll)
            self.navigationdock.setObjectName("Navigation")

            self.addDockWidget(Qt.LeftDockWidgetArea, self.navigationdock)

            self.showDock = FlatButton(":/images/menu.svg")
            self.showDock.setToolTip("Show Navigation")
            self.statusBar().addPermanentWidget(self.showDock)

            plus_button.clicked.connect(self.addClient)
            self.trash_button.clicked.connect(self.deleteClient)

            self.dashboard.expanded.connect(self.dashboardExpanded)
            self.dashboard.clicked.connect(self.showDashboard)
            self.content.expanded.connect(self.contentExpanded)
```

```
            self.content.clicked.connect(self.showClient)

            self.settings.expanded.connect(self.settingsExpanded)
            self.settings.clicked.connect(self.showSettings)

            self.showDock.clicked.connect(self.showMenu)
            self.navigationdock.visibilityChanged.connect(self.dockVisibilit
yChanged)

            self.client = None
            self.client_editor = None
```

Neu ist hier das `QDockWidge` . Mit diesem Widget können wir oder besser unsere Benutzer das Navigationsfeld neu anordnen. Der Benutzer kann das Navigationsfenster auf die rechte Seite des Fensters oder auf die linke Seite des Fensters ziehen. Du kannst diese Regionen mit dem folgenden Code einschränken.

```
self.navigationdock.setAllowedAreas (Qt.LeftDockWidgetArea |
Qt.RightDockWidgetArea)
```

Auch die Position des Navigationsdocks wird in den Einstellungen gespeichert. Beim Testen dieses Verhaltens stellte ich fest, dass ich das Navigationsfenster beim Schließen nicht wiederherstellen konnte. Daher musste ich die Lob.ini-Datei löschen, die unter */home/[user]/.config/Company/Lob.ini* gespeichert war. Denke bitte daran, dass Dateien, die mit einem Punkt "." beginnen, standardmäßig unsichtbar sind. Du musst sie also sichtbar machen, indem Sie STRG-H im Datei-Navigator drückst.

Denke also daran, das Navigationsdock erst zu schließen, wenn du einen Menüeintrag oder eine Schaltfläche erstellt hast, um es wieder sichtbar zu machen. Um das Schließen des Widgets zu vermeiden oder besser gesagt, um es wieder sichtbar zu machen, habe ich diese Methode verwendet.

```
        self.showDock = FlatButton(":/images/edit_normal.png", ":/images/
edit_hover.png")
        self.showDock.setToolTip("Show Navigation")
        self.statusBar().addPermanentWidget(self.showDock)
        self.showDock.clicked.connect(self.showMenu)
        self.navigationdock.visibilityChanged.connect(self.dockVisibilityCha
nged)

        def showMenu(self):
            self.navigationdock.setVisible(True)

```

```
    def dockVisibilityChanged(self, visible):
        self.showDock.setVisible(not visible)
```

Ich habe eine Schaltfläche erstellt, die angezeigt wird, wenn das Dock unsichtbar wird. Bevor ich vergesse zu erwähnen, warum ich ein ":" vor dem Pfad des Bildes verwende. Dieser Doppelpunkt signalisiert Qt, dass das Bild als eingebettete Ressource gefunden wird. Schaue dir folgende Dateien an:
resources.qrc
resources.py
und
build.sh

In der Datei *resources.qrc* listen wir alle Datein, die in die Resourcen eingebunden werden sollen.

```
<RCC>
    <qresource>
        <file>images/edit_hover.png</file>
        <file>images/edit_normal.png</file>
```

Dann führen wir pyrcc5, einen mit PyQt5 gelieferten Ressourcen-Compiler, aus.

```
pyrcc5 resources.qrc -o resources.py
```

Und dieser Befehl erstellt die Datei *resources.py*. Es enthält alle Bilder in Textform.

```
qt_resource_data = b"\
\x00\x00\x09\xb4\
```

Wir müssen diese Datei nur in jeder Python-Datei importieren, in der wir ein solches Bild verwenden. Daher habe ich die Ressourcen auch in zwei Hälften aufgeteilt, da wir in *main.py* nur die *logo.svg* verwenden und aus Leistungsgründen nicht alle Ressourcen überall einschließen müssen.

```
import resources
```

In unserem Beispiel wechseln wir zwischen dem Client-Editor, dem Dashboard und den Settings. Um zwischen diesen Widgets zu wechseln, legen wir einfach das zentrale Widget des Fensters fest. Seien Sie jedoch

vorsichtig, da diese Methode die Instanz des vorherigen Widgets zerstört. Sie müssen das Widget daher immer wieder neu erstellen.

```
    def showClient(self):
        self.client_editor = ClientEditor(self)
        self.setCentralWidget(self.client_editor)
```

Das Schließereignis des Fensters wird ausgelöst, wenn wir das Fenster schließen.

```
    def closeEvent(self, event):
            self.writeSettings()
            event.accept()
```

Hier schreiben wir die Einstellung in eine INI-Datei.

```
        def writeSettings(self):
            settings = QSettings(QSettings.IniFormat, QSettings.UserScope, Q
CoreApplication.organizationName(), QCoreApplication.applicationName())
            settings.setValue("geometry", self.saveGeometry())
            settings.setValue("state", self.saveState())
            settings.setValue("database", self.database)
            settings.setValue("fontSize", str(self.fontSize))
```

Und beim Programmstart lesen wir diese INI-Datei und stellen die Einstellungen wieder her.

```
        def readSettings(self):
            settings = QSettings(QSettings.IniFormat, QSettings.UserScope, Q
CoreApplication.organizationName(), QCoreApplication.applicationName())
            geometry = settings.value("geometry", QByteArray())
            if geometry.isEmpty():
                availableGeometry =
QApplication.desktop().availableGeometry(self)
                self.resize(availableGeometry.width() / 3,
availableGeometry.height() / 2)
                self.move(int(((availableGeometry.width() - self.width()) /
2)), int((availableGeometry.height() - self.height()) / 2))
            else:
                self.restoreGeometry(geometry)
                self.restoreState(settings.value("state"))
            self.database = settings.value("database")
            if not self.database:
                self.database = "."
            fs = settings.value("fontSize")
```

```
        if not fs:
            fs = 10
        self.fontSize = int(fs)
        font = QFont("Sans Serif", self.fontSize)
        self.app.setFont(font)
```

Wenn wir einen der Expander im Navigationsdock erweitern, werden die folgenden Ereignisse ausgelöst, wie wir sie zuvor in der initGui-Methode verbunden haben.

```
    def dashboardExpanded(self, value):
        if value:
            self.content.setExpanded(False)
            self.settings.setExpanded(False)

    def contentExpanded(self, value):
        if value:
            self.dashboard.setExpanded(False)
            self.settings.setExpanded(False)

    def settingsExpanded(self, value):
        if value:
            self.dashboard.setExpanded(False)
            self.content.setExpanded(False)
```

Um die Animationen der Expander auszulösen, ändern wir einfach deren Expanded-Flag. Die Animationen, bei denen ein Expander geschlossen und der andere geöffnet wird, werden nach dem Ändern des erweiterten Flags parallel ausgeführt.

Wenn der Benutzer einen Buchstaben in das Suchfeld eingibt, wird das Ereignis filterChanged ausgelöst und dort das Laden des Clients ausgeführt.

```
    def filterChanged(self):
        self.loadClients()

    def loadClients(self):
        if not self.clients:
            return

        self.client_list.clear()
        filter = self.filter.text()

        a = []
        for c in self.clients:
            if filter.lower() in c["name"].lower():
```

```
                a.append(c)

        s = sorted(a, key=namesort)
        for c in s:
            item = QListWidgetItem()
            item.setText(c["name"])
            item.setData(3, c)
            self.client_list.addItem(item)
        self.client_list.setCurrentRow(0)
```

In der loadClient-Methode gehen wir jeden Client durch und prüfen, ob der Name den Filterwert enthält. Wenn ja, fügen wir diesen Client der Liste hinzu. Diese Methode ist nur nützlich, wenn Sie nur wenige hundert Clients haben. Normalerweise fragen Sie einen SQL-Server nach dem Ergebnis ab, sodass nicht alle Clientdaten über das Netzwerk vom Server zum Clientcomputer transportiert werden müssen.

Zum Öffnen der Datenbank verwenden wir die loadDatabase-Methode.

```
    def loadDatabase(self):
        try:
            self.db = TinyDB(os.path.join(self.database,"lob.json"))
            self.clients = self.db.table('Clients')
        except:
            print("Unable to open the database")
```

Hier laden wir die JSON-Datei aus dem angegebenen Pfad und initialisieren das Client-Array.

Um einen neuen Client-Datensatz hinzuzufügen, rufen wir die Methode addclient() auf.

```
    def addClient(self):
        now = datetime.now()
        newclient = {
            "number": "",
            "name": "",
            "birthday_year": 1990,
            "birthday_month": 1,
            "birthday_day": 1,
            "profession": "",
            "address": "",
            "mobile": "",
            "email": "",
            "notes": ""
        }
```

```
            self.clients.insert(newclient)
            self.showClient()
            self.loadClients()
            q = Query()
            self.client = self.clients.get(q.name=="")
            self.client["name"] = ""
            self.client_editor.reload()
```

Hier füllen wir einen Datensatz mit Daten und fügen den neuen Datensatz in die Datenbank ein. Die Daten werden sofort ohne Commit in die Datei geschrieben. Danach lesen wir die Clients erneut und suchen nach dem Client mit dem leeren Namen, auf den wir uns konzentrieren möchten.

Um einen Client aus der Datenbank zu löschen, rufen wir die Methode deleteClient() auf.

```
    def deleteClient(self):
            self.clients.remove(doc_ids=[self.client.doc_id])
            self.loadClients()
```

Um einen Client-Datensatz zu aktualisieren, rufen wir die updateClient-Methode auf.

```
        def updateClient(self):
            for i in range(self.client_list.count()):
                item = self.client_list.item(i)
                c = item.data(3)
                if c.doc_id == self.client.doc_id:
                    item.setData(3, self.client)
                    item.setText(self.client["name"])
                    break
```

Hier gehen wir alle Listenelemente durch und nachdem wir das mit der richtigen doc_id gefunden haben, ändern wir die Daten mit setData() und setText(), um die Listenansicht zu aktualisieren.

Wenn die Verwendung einen anderen Client in der Liste auswählt, wird die Methode clientChanged() aufgerufen.

```
        def clientChanged(self, item):
            if item:
                self.client = item.data(3)
                if self.client_editor:
                    self.client_editor.reload()
                self.trash_button.enabled = True
```

```
            else:
                self.client = None
                self.client_editor.reload()
                self.trash_button.enabled = False
```

Client Editor

Dies ist einer der drei Bildschirme, die dieser Anwendung zur Verfügung stehen. In diesem Bildschirm gibt es einen interessanten neuen Teil, die Bildauswahl, mit der ein Bild ausgewählt wird, das der Client mit seinem Datensatz speichern soll.

```
            self.image = ImageSelector()
            self.image.clicked.connect(self.seek)

        def seek(self):
            fileName = ""
            dialog = QFileDialog()
            dialog.setFileMode(QFileDialog.AnyFile)
            dialog.setNameFilter("Images (*.png *.gif *.jpg);;All (*)")
            dialog.setWindowTitle("Load Image")
            dialog.setOption(QFileDialog.DontUseNativeDialog, True)
            dialog.setAcceptMode(QFileDialog.AcceptOpen)
            if dialog.exec():
                fileName = dialog.selectedFiles()[0]
            del dialog
            if not fileName:
                return

            # copy file to database dir
            name = os.path.join(str(self.win.client.doc_id) + ".png")
            path = os.path.join(self.win.database, "images", name)
            shutil.copy(fileName, path)
            self.image.setImage(QImage(path))
            self.clientChanged()
```

Wenn der Benutzer auf die Bildauswahl klickt, wird die Suchmethode aufgerufen. Hier öffnen wir einen Dialog zum Öffnen von Dateien, damit der Benutzer ein Bild auswählen kann. Wir setzen einen Namensfilter mit folgendem Muster: Der Name, der in jeder Zeile der Filterliste angezeigt werden soll, hier Bilder und Alle. Und in Klammern setzen wir die Filter wie (*.png), um nur Dateien aufzulisten, die mit ".png" enden. Nachdem der Dialog geschlossen wurde, wählen wir den ersten Dateinamen aus einer Liste.

```
fileName = dialog.selectedFiles()[0]
```

Wenn der Benutzer eine Datei auswählt, kopieren wir sie in den Bilderordner und ändern den Namen in doc_id (Primärschlüssel) + ".png", damit wir dieses Bild einem Client-Datensatz zuordnen können.

Jedes Mal, wenn der Benutzer die Daten in den Bearbeitungsfeldern ändert, wird die clientChanged-Methode aufgerufen.

```
  def clientChanged(self):
          if self.loading:
              return
          self.win.client["number"] = self.number.text()
          self.win.client["name"] = self.name.text()
          self.win.client["address"] = self.address.text()
          self.win.client["email"] = self.email.text()
          self.win.client["mobile"] = self.mobile.text()
          self.win.client["profession"] = self.profession.text()
          self.win.client["notes"] = self.notes.toPlainText()
          self.win.client["birthday_year"] = self.birthday.date().year()
          self.win.client["birthday_month"] = self.birthday.date().month()
          self.win.client["birthday_day"] = self.birthday.date().day()
          self.win.clients.update(self.win.client, doc_ids=[self.win.clien
t.doc_id])
          self.win.updateClient()
```

Hier aktualisieren wir den Client-Datensatz und senden eine Nachricht an das Fenster, damit die Liste, in der die Client-Namen angezeigt werden, aktualisiert werden kann.

Dashboard

Das Dashboard ist derzeit ziemlich leer und kann später verwendet werden, um einige Informationen anzuzeigen, wenn der Benutzer die Anwendung startet. Jetzt wird nur eine Dokumentation für den Benutzer angezeigt.

Die Dokumentation wird in einem Textbrowser als HTML angezeigt. Dieser Textbrowser unterstützt Hyperlink. Wenn ein Benutzer auf den Hyperlink href = 'clients' klickt, wird die Methodennavigation ausgelöst.

```
self.browser.anchorClicked.connect(self.navigate)
```

Bei der Navigationsmethode senden wir zwei Signale aus, um das Hauptfenster aufzurufen.

```
    def navigate(self, url):
        if url.toDisplayString() == "clients":
            self.clients.emit()
        elif url.toDisplayString() == "settings":
            self.settings.emit()
```

Diese Signale werden im Hauptteil der Klasse deklariert.

```
class Dashboard(QWidget):
    clients = pyqtSignal()
    settings = pyqtSignal()
```

Und diese Signale werden im Hauptfenster in der showDashboard-Methode verbunden.

```
    def showDashboard(self):
        db = Dashboard()
        db.clients.connect(self.showClient)
        db.settings.connect(self.showSettings)
```

Jedes Mal, wenn der Benutzer auf die Hyperlink-Einstellungen klickt, wird ein Signal gesendet und daher wird die verbundene Methode showSettings aufgerufen. Diese Terminologie heißt **Signale und Slots**. Die Klasse mit dem Signal löst nur das Signal aus und alle anderen Objekte können sich mit diesem Signal verbinden und erhalten das Ereignis, wenn es ausgelöst wird.

Settings Editor

Hier ist nicht viel Besonderes. Das einzige Besondere dabei ist die Tatsache, dass wir die Schriftart der Anwendung sofort ändern, wenn der Benutzer die Schriftgröße in die fontSize Spinbox eingibt.

```
        self.fontSize.valueChanged.connect(self.settingsChanged)

    def settingsChanged(self):
        self.win.fontSize = self.fontSize.value()
        font = QFont("Sans Serif", self.win.fontSize)
        self.win.app.setFont(font)
```

Controls

Nun kommen wir zu einigen benutzerdefinierten Steuerelementen (Widgets).

FlatButton

Der FlatButton ist nur eine QLable, die wir als Schaltfläche zum Anzeigen eines Bildes verwenden. In diesem Fall geben wir der Klasse den Dateinamen eines SVG (skalierbare Vektorgrafiken) als Argument. Dieses Bild hat eine Region mit der Sonderfarbe "#ff00ff", die durch die Hervorhebungsfarbe unserer App ersetzt wird, und der Farbe "#0000ff", die durch die Hintergrundfarbe der Schaltfläche ersetzt wird. Mit dieser Methode können wir die Hervorhebungsfarbe der App ändern und alle Schaltflächen ändern sich automatisch.

```
      def createIcon(self, source, hilite_color):
          fp.write(data.replace("#ff00ff",
hilite_color).replace("#0000ff", bg))
```

In der Methode setColors generieren wir für jeden Status der Schaltfläche ein Bild. Eine für den normalen, eine für den Hover und eine für den disabled Zustand. Wenn wir also mit der Maus über die Schaltfläche fahren, können wir die geänderte Farbe sehen.

```
     def setColors(self):
          self.label_normal_color = self.palette().buttonText().color().na
me()
          self.label_hovered_color = self.palette().highlight().color().na
me()
          self.label_disabled_color = self.palette().color(QPalette.Disabl
ed, QPalette.ButtonText).name()

          self.normal_icon = QPixmap(self.createIcon(self.svg, self.label_
normal_color))
          self.hover_icon = QPixmap(self.createIcon(self.svg, self.label_h
overed_color))
          self.disabled_icon = QPixmap(self.createIcon(self.svg, self.labe
l_disabled_color))

          if self.enabled:
              self.setPixmap(self.normal_icon)
          else:
              self.setPixmap(self.disabled_icon)
```

Das bedeutet, dass wir die Pixmap nur ändern, wenn das Enter- oder Leave-Ereignis ausgelöst wird.

```
    def enterEvent(self, event):
        if self.enabled:
            self.setPixmap(self.hover_icon)
        QWidget.enterEvent(self, event)

     def leaveEvent(self, event):
        if self.enabled:
            self.setPixmap(self.normal_icon)
        else:
            self.setPixmap(self.disabled_icon)
        QWidget.leaveEvent(self, event)
```

Ein besonderes Verhalten von PyQt5 ist die Verwendung von Properties. Sie werden wie folgt deklariert.

```
    @pyqtProperty(bool)
    def enabled(self):
        return self._enabled

    @enabled.setter
    def enabled(self, enabled):
        self._enabled = enabled
        if enabled:
            self.setPixmap(self.normal_icon)
        else:
            self.setPixmap(self.disabled_icon)
        self.update()
```

Um das angeklickte Ereignis auszulösen, müssen wir ein Signal im Hauptteil der Klasse deklarieren und es auslösen, wenn der Benutzer mit der Maus klickt oder in diesem Fall, wenn der Benutzer die Maus klickt und loslässt.

```
class FlatButton(QLabel):
    clicked = pyqtSignal()

    def mouseReleaseEvent(self, event):
        if self.enabled:
            self.setPixmap(self.hover_icon)
            event.accept()
            self.clicked.emit()
```

Hyperlink

Der Hyperlink ist ebenfalls von QLable abgeleitet und verwendet die Möglichkeit, HTML in einer Label anzuzeigen. Hier überschreiben wir die Methode setText () und setzen den Text zwischen die HTML-Syntax, damit der Hyperlink in dem Label angezeigt wird.

```
    def setText(self, text):
        self.text = text
        super().setText("<a style=\"color: " + self.color + "; text-
decoration: none; cursor: pointer;\" href=\"#/\">" + self.text + "</a>")
```

Nichts Besonderes in dieser Klasse.

Expander

Der Expander ist etwas kniffliger. Hier müssen wir das Verhalten animieren, wenn der Benutzer auf den Expander klickt, um ihn zu erweitern, und wenn der Benutzer einen anderen Expander erweitert, wird dieser Expander zusammengeschoben. Um die Animation zu erstellen, verwenden wir eine Parallelanimation, um die Farbe und die Eigenschaft MaximumHeight zu animieren.

```
        self.anim = QParallelAnimationGroup()
        self.height_anim = QPropertyAnimation(self.content, "maximumHeig
ht".encode("utf-8"))
        self.color_anim = QPropertyAnimation(self, "color".encode("utf-8
"))
        self.height_anim.setDuration(200)
        self.color_anim.setDuration(200)
        self.anim.addAnimation(self.height_anim)
        self.anim.addAnimation(self.color_anim)
```

Um die Farbe animieren zu können, müssen wir eine Property mit diesem Namen erstellen.

```
    @pyqtProperty('QColor')
    def color(self):
        return Qt.black

    @color.setter
    def color(self, color):
        pal = self.palette()
```

```
        pal.setColor(QPalette.Background, QColor(color))
        self.setPalette(pal)
```

Um verschiedene Zustände anzeigen zu können, verwenden wir diesmal PNG-Bilder. Dies ist nur eine weitere Variante, da wir SVG im FlatButton verwendet haben. Dies hängt von den Werkzeugen ab, mit denen Sie Icons erstellen. Ich verwende Inkscape, um Icons zu erstellen. Daher ist es sinnvoll, SVG-Dateien im Allgemeinen zu verwenden. Wenn Sie jedoch nur einen pixelbasierten Editor verwenden, passt der PNG-Stil am besten zu dir. Du kennst nun also beide Methoden.

```
self.dashboard = Expander("Dashboard", ":/images/dashboard_normal.png", ":/images/dashboard_hover.png", ":/images/dashboard_selected.png")
```

Zusammenfassung

Sie haben gesehen, wie man Benutzersteuerelemente erstellt, wie eine Datenbank zum Speichern von Daten in einer Datei verwendet wird, wie zwischen Bildschirmen navigiert wird und wie eine App erstellt wird, ohne dass ein Popup-Menü programmiert werden muss. Heutzutage entwickeln wir viele Apps für das Tablet anstelle des Desktops und verwenden dort keine Maus mehr, sodass wir das Popup-Menü der alten Schule nicht mehr verwenden können.

Teil V - Installation auf Linux

Auf Linux, Windows und MacOS bevorzuge ich ein anderes Werkzeug mit dem Namen **PyInstaller**, um ein Executable zu erstellen, da es Abhängigkeiten automatisch auflöst.
Um ein SetupProgramm zu erstellen benutze ich das **QtInstallerFramework**, welches mit Qt ausgeliefert wird, da es ebenfalls Cross Platform und kostenlos zu nutzen ist.

Installation von PyInstaller

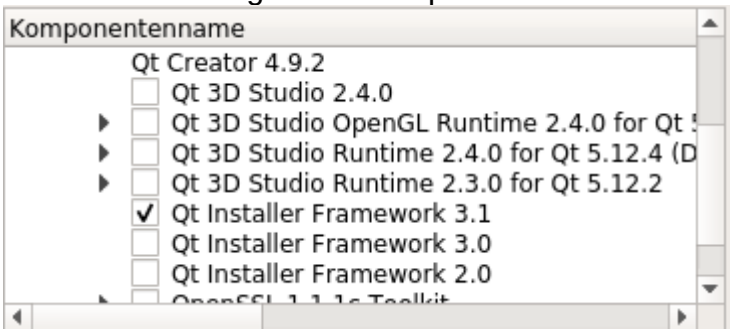
Du kannst PyInstaller mit pip installieren.

```
user@machine:/path$ pip3 install pyinstaller
```

PyInstaller kombiniert eine Python-Anwendung und alle Abhängigkeiten, inlusive Python selber, in ein Paket. Der Endnutzer kann auf diese Weise die Anwendung ausführen, ohne Python und dessen Module installieren zu müssen.

Installation von QtInstallerFramework

Nun benötigen wir auch noch Qt selber. Du kannst es dir hier herunterladen: https://www.qt.io/download
Du solltest die folgenden Komponenten installieren.



Wenn dort keine Option für die Version 3.1 (Windows) angezeigt wird, dann tut es auch Version 3.0.

Paket Bilden

Um ein Paket zu erstellen lasse einfach den pyinstaller mit deinem main.py als Argument laufen.

```
user@machine:/path$ pyinstaller main.py
```

PyInstaller trägt alle nötigen Module zusammen und packt sie zusammen mit Python in ein Verzeichnis. Man spricht hierbei vom Freezing (Einfrieren). Der Endnutzer führt somit genau das Programm mit den Paketen in der Version aus, welche wir getestet haben.

Paket Testen

Du kannst die Anwendung wie folgt testen:

```
user@machine:/path$ dist/main/main
```

Theoretisch könntest du jetzt die Dateien aus dem Verzeichnis dist/main in eine ZIP-Datei packen und sie ausliefern, ich empfehle aber ein SetupProgramm zu erstellen, welches der Endnutzen einfach nur ausführen muss, um die Anwendung zu installieren. Hierfür nutzen wir das QtInstallerFramework.

Setup Paket Erstellen

Um ein SetupProgramm zu erstellen, legen wir erst einmal ein Verzeichnis mit dem Namen *config* an. Innerhalb dieses Verzeichnisses legen wir die Datei *config.xml* mit folgendem Inhalt an.

config/config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
  <Installer>
      <Name>DemoApplication</Name>
      <Version>1.0.0</Version>
      <Title>DemoApplication</Title>
      <Publisher>Publisher</Publisher>
      <StartMenuDir></StartMenuDir>
      <TargetDir>@HomeDir@/DemoApplication</TargetDir>
      <Translations>
```

```
        <Translation>en.qm</Translation>
      </Translations>
    </Installer>
```

Fülle die nachfolgenden Felder wie folgt:

Field	Value
Name	Name der Anwendung
Version	Version der Anwendung
Title	Name der Anwendung
Publisher	Dein Name / Firmenname
TargetDir	Installationsverzeichnis
Translation	Sprache der Anwendung

Wenn du keine Sprache angibst, dann nimmt Qt die Sprache deines Betriebssystemes, welches nicht immer gewünscht ist. In meinem Fall habe ich zum Beispiel eine deutsche Linux-Installation und möchte aber eine Anwendung in Englisch erstellen.

Nun erstellen wir ein Verzeichnis mit dem Namen *packages* und dort drinnen legen wir ein Verzeichnis mit dem Namen *com.vendor.product* an. In diesem Verzeichnis legen wir wiederum eines mit dem Namen *data* an, in die wir später die Binärdateien reinkopieren und ein Verzeichnis mit dem namen *meta*. In dem Verzeichnis meta erstellen wir eine Datei mit dem Namen *package.xml* mit dem folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
  <Package>
      <DisplayName>DemoApplication</DisplayName>
      <Description>Install this application.</Description>
      <Version>0.1.0-1</Version>
      <ReleaseDate>2019-07-15</ReleaseDate>
      <Default>true</Default>
  </Package>
```

Nun erstellen wir ein Verzeichnis mit dem Namen *bin* im Verzeichnis *data* und kopieren dort alle Dateien aus dem Verzeichnis *dist/main*.
Neben dem *bin* Verzeichnis können wir später auch noch Verzeichnisse für Plugins, Daten oder ähnlichem anlegen.

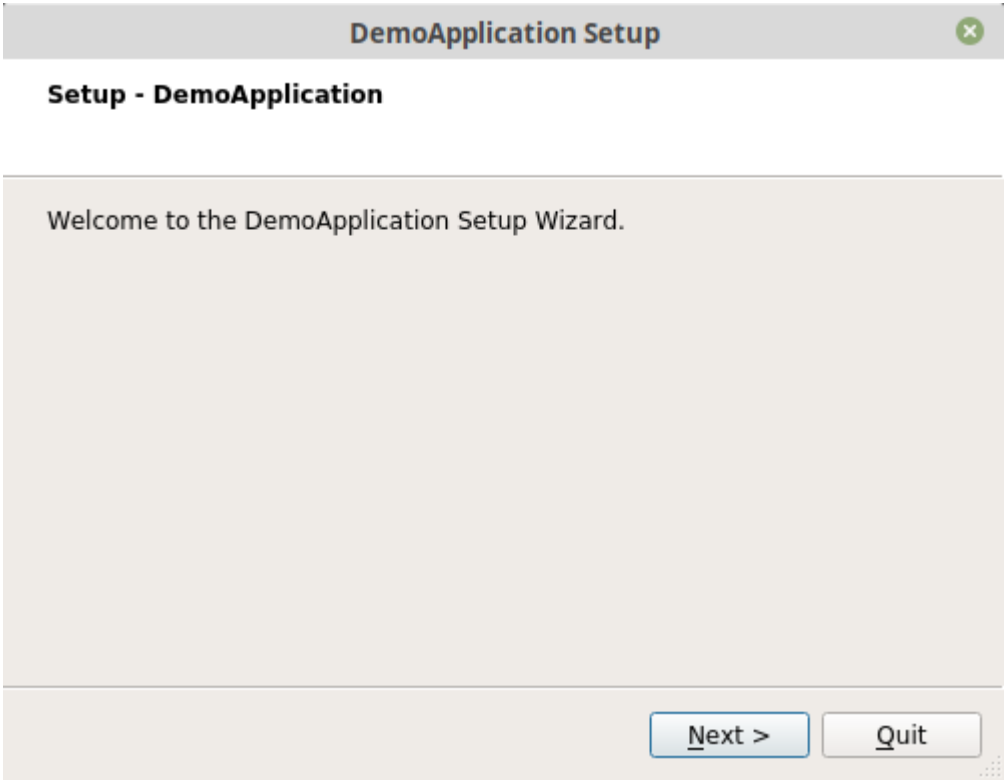
```
user@machine:/path$ mkdir packages/com.vendor.product/data/bin
user@machine:/path$ cp -r dist/main/* packages/com.vendor.product/data/
bin
```

Nun wird es Zeit, das SetupProgramm zu erstellen. Dafür führen wir folgenden Befehl aus:

```
user@machine:/path$ binarycreator -f -c config/config.xml -p packages
DemoApplication-Linux-1.0.0.Setup
```

Dieser Befehl erstellt eine Datei mit dem Namen DemoApplication-Linux-1.0.0.Setup, welches an den Endnutzer ausgeliefert werden kann. Solltest du keinen Pfad zum binarycreator haben, findest du es unter *Qt/Tools/QtInstallerFramework/3.1/bin*

Du kannst das Setup-Programm starten, in dem du das erzeugte Programm ausführst.



Zusammenfassung

Am Ende haben wir ein Setup-Programm für eine Python Anwendung erstellt, welches wir einfach an Endnutzer ausliefern können. Alle nötigen Pakete und selbst Python sind dort enthalten.

Nachwort

Ich bin froh, dass du bis hierher gelesen hast.
Ich hoffe, das dir dieses Buch helfen konnte, ein paar neue Möglichkeiten der Softwareentwicklung kennen zu lernen.
Ich wünsche dir viel Erfolg auf deinem weiteren Weg.
Wenn du das Buch magst dann würde ich mich sehr freuen, wenn du eine kurze Rezession hinterlassen könntest, damit andere Menschen auch dieses Buch finden können.

Über den Autor


Olaf Art Ananda, ist 1963 in Hamburg, Deutschland geboren und arbeitet schon seit über 30 Jahren als Softwareentwickler. Er hat mit C angefangen und dann noch Assembler gelernt, um die Programme zu beschleunigen. Nachdem er so gängige Programmiersprachen wie Java, C# und Objective-C erlernt hat, kam er dann schließlich 2016 zu C/C++ zurück und startete Applikationen mit Qt5 zu erstellen.
Qt5 war für ihn das ideale Framework um seine Fähigkeiten, die er in seinem Studium 2013 in "Human Computer Interaction Design" gelernt hatte, umzusetzen.

Nachdem er zum ersten Mal mit Python in Form von Plugins für seine Anwendungen in Berührung kam, dauerte es noch weitere zwei Jahre, bis er Python so richtig kennen lernte.
Heute liebt er die Einfachheit dieser Sprache um wesentlich schneller Anwendungen zu erstellen als damals in C++.

Olaf hat für mehrere Top 500 Unternehmen wie Dupont, Dresdner Bank, Commerzbank und Zürcher Kantonalbank gearbeitet, um nur einige zu nennen. Nach seinem Burnout und einer Nahtoderfahrung beschloss er, nicht mehr für Profit zu arbeiten. Seit 2016 schreibt er Open Source Software wie den AnimationMaker, den FlatSiteBuilder und den EbookCreator. Er hat auch die folgenden Bücher geschrieben: Camp Eden - Wie wir unsere Paradies wiedererschafft haben und Step Out - Guideline to step out of the system. Seit 2016 lebt er in seinem Wohnmobil, derzeit in Portugal, und spielt auf der Straße Gitarre für ein paar Münzen. Das ist ein leichtes Leben.

Impressum

Olaf Japp
japp.olaf@gmail.com