Python Desktop Application Development: Part 2 - Design

by Bo Milanovich

Introduction and Course Overview

Introduction

Hi, and welcome to "Python Desktop Application Development Course, Part 2. " My name is Bo Milanovich and I will be your instructor for this course. This course builds on Part 1 of the same course, in which we discussed the foundations and basic principles of building Python desktop applications with Qt. In this course, however, we are going to focus more on the design of our application and the user experience as a whole. We'll also use some tools that will, trust me, make our lives a lot easier as Python RQ developers. But, to better understand this course and what's it all about, and kind of see what the expectations are as well, let's look at a course overview.

Course Overview

Now, in my last course, I had some prerequisites. I wanted you to have basic programming skills, as well as a basic understanding of Python. In this course, however, aside from those two, I have two more and those are good will and a desire to learn and a smile on your face. That's probably the most important one. Who is this course for? Well, this course is for anyone who wants to

develop powerful, cross-platform, desktop applications and for anyone who wants to design powerful, cross-platform, desktop applications. As I mentioned earlier, in this course we are going to be focusing mainly on the design, but of course we'll wire everything up with the code as well. Though marked as intermediate course level, I'll try to make this course also novice friendly. But, let me clarify a little bit when I say to design our application. Obviously, when you say design, you immediately think of the look and feel of the application, but note that in this course we won't just be talking about how to lay out the application itself. We'll also talk about how to implement, for example, icons using the so-called resources, how to write our own context manual, which is the manual that you get when you right-click somewhere within the application, and how to use some cool features such as flash screen or loading screen, as some people call it, or the system tray icon with the notification. So, as you can see, we won't just focus on the general look and feel, if you want to call it that way. Oh, and by the way, for that general look and feel for styling our application, Qt is more than happy to take CSS. Yes, it's the same CSS you use to style web pages, which I'm sure will make lots of people happy. At the end of this course, I'd expect you to have an understanding of the tools you are going to use throughout the course that will enable us to build the rich desktop applications with relative ease. I'm sure that some of you who watched the last part of the course maybe didn't like the idea of adding all those Qt widgets to the screen in the code. You probably realized that if you had even a somewhat complex application, the code would get all cluttered and messy and just not something you want to look at. Adding every single widget to the layout manually? I mean, come on. Well, in this course, we're going to use a tool called Qt Designer that will allow us to kind of hide all that previously manually typed code. And, this tool is as simple as drag and drop is. Obviously, I'd like you to know how to design beautiful applications as well in the end. And, since design is fairly subjective, I'll try to design a beautiful application myself in this course. But, hey, if you don't like it in the end and think it's horrible, that's okay. At least you'll be armed with the tools and the knowledge you need to design your own beautiful application. But, don't forget to send me screenshots, though. However, as important as the design is, the user experience is no less important either. In other words, you've probably seen some applications that, although they look nice and gorgeous and all that, they're just confusing and terrible to use. So, we'll definitely touch on this topic as well. Now, let's actually jump straight into our next module and introduce ourselves to the new tool, the fantastic Qt Designer.

QtDesigner

Introduction

Welcome to the second module of "Python Desktop Application Development Course, Part 2. " My name is Bo Milanovich. In this module, we'll familiarize ourselves with Qt Designer. We'll learn how to use its power, only for good of course, how to lay out our application, but also how to use some other tools that PyQt ships with that will enable us to run the application we design and also access its objects from within our own Python code. At the end of this module, you should know how to design a basic application and how to integrate it with your code.

QtDesigner - Benefits and Drawbacks

The infamous Qt Designer. Well, you've definitely heard me throw this word around a lot, likely in this course and in the previous part of the course as well, but what exactly is it? So, Qt Designer is a WYSIWYG tool. WYSIWYG stands for what you see is what you get. This tool enables us to design Qt applications with ease and without having to really touch our Python code at all. We can design a complete application and then just implement the logic there in the code, such as downloading a file. If you've ever used the Visual Studio for designing desktop applications or if you worked with NetBeans or IntelliJ for Java desktop applications, then you should be very familiar with this concept. But, let's look at some of the benefits and the drawbacks of Qt Designer. One main benefit is that - and I've definitely hinted at this earlier - the separation of the logic code and the design code. What I mean by that is that past are the days where we would have to create a new instance of, say, a QLineEdit, create a new instance of a layout, add a QLineEdit to the layout, and then set the layout manually in our class. For example, a QDialog class. You can see that this was definitely a fairly tedious task and this is why tools such as Qt Designer exist. With it, we'll be able to essentially hide all these layers of code, if you want to call them that way. I'm not really worried about them at all. All we'll have to do in the end is import the generated Python files, about which we'll talk more later in this module. I would also argue that Qt Designer speeds up the application development process significantly. But, I know that there are also people out there who would disagree with me. So, this is entirely subjective. If you prefer designing your application in the code itself, you're more than welcome to do that. One more thing is that Qt Designer has some abilities other than the design part. For example, you can actually connect signals and slots in it. Now, remember, signals and slots are our events and handlers. I haven't honestly found this particular feature very useful myself, but I'm sure that there are some people who may use it. So, what about the drawbacks? Well, one of the main reasons people claim they dislike Qt Designer is because it doesn't allow them for the complete control of the design due to all this abstractioning happening, which is true, at least sort of. The abstraction

definitely does happen. I mean, all this code is hidden from you, but I would have to disagree because once this Python code is generated for all these different UI elements, you can do whatever you want with it, just as if you were typing that code yourself. Speaking of the code generation, one drawback would be that Qt Designer doesn't actually generate Python code. Recall that Qt is made for C++ and, as such, the Qt Designer creates a. ui file, which is essentially a file with an XML syntax. We'll be using another tool, a Command Line tool called PyUIC or Pyside-UIC. That depends on which Python binding you opted to use to actually generate Python files from those UI files. Well, and don't be afraid, though. This is a very simple tool and takes seconds to run. So, how do you get Qt Designer? Well, you'll be happy to know that Qt Designer's already installed on your computer if you have PyQt installed. If you're running Windows, just search for "Designer" in the Start menu. Notice, it's "Designer" that you should be searching for, not "Qt Designer. " For Linux and Mac, the Designer should live in the bin directory or an equivalent directory for your machine. So, all this talk about Qt Designer and you still have no idea what it looks like. So, let's familiarize ourselves with its interface in our next video.

Get to Know QtDesigner

I'm going to run Qt Designer to show you the interface. And, since I'm on Windows, I'm just going to search for "Designer" in my Start menu. I'm just going to type "Designer. " When you start Qt Designer, you'll be greeted with this New Form screen. In it, you can select a kind of a template to use when designing a new form. Quite honestly, I've always used just these two: The Dialog Without Buttons or the Main Window. The Dialog Without Buttons corresponds to our QDialog class and, of course as you might assume, the Main Window corresponds to QMainWindow class. For this video, let's select the Dialog Without Buttons and hit Create. We see here that a new dialog has appeared. This is our QDialog. Let's go over the general Qt Designer interface first before we start playing with our dialog. On the left here, we have a widget box. Just by looking at it, you can probably see that it's holding the QWidgets. We have our layouts at the top, spacers right below it and we haven't really talked about spacers yet. We have our buttons with our Qt Push Button. And, if I scroll down, we have our input widget, such as the Combo Box, the QLineEdit, and our display widgets, such as the QLabel and many, many more. To the top here, we have the general tool buttons, such as New, which opens up that dialog, Open, Save. Here, we have the ordering. For example, if we want a certain QWidget to appear in front or behind another QWidget, we can set those settings right here. Following that, we have four different kinds of views or workspaces right here. Currently, we're in the Edit Widgets workspace. In addition to Edit Widgets, we also have the Edit Signals and Slots, Edit Buddies, and also Edit Tab

Order workspaces. Now, don't worry at all if none of these makes sense to you right now as we'll talk more about them later on in this course. Furthermore, to the right, we have our layouts. You see here that we have our horizontal box layout, our vertical box layout, and our grid and form layouts. In addition to that, we have the Break Layout button, as well as the vertically and horizontally layout that are going to lay out this in a splitter. And, we won't really focus on these two for now. And, finally, to the right, the right sidebar is probably where you'll spend most of your time while in Qt Designer. At the top here, we have our object inspector. This inspector will list all the objects we have on the screen. Currently, we only have one object and that's our dialog. And, as you can see, it has a class name of QDialog. As we add more objects, they will all appear here in a TreeView fashion. But, where did this dialog come from? I mean, we see here that it's of a class dialog, but why just dialog here? That's where this Property Editor jumps in. Dialog, in this case, is just an object name. In Part 1 of this course, we would name our dialog something like "Hello World Dialog" or "Py Downloader Dialog. " So here, instead of manually editing our code, we can simply type what we want our dialog object name to be, for example, Hello World. If I click back. Let me actually expand this Property Editor so we can see more what it has. This Property Editor also kind of illustrates the inheritance. Recall that all UI elements including our dialog inherit from QWidget. Here, we see that it, actually its grandparent element is QObject. We haven't really talked about QObject a whole lot, but that's okay. And, here we have our QWidget. This QWidget has lots and lots of properties we can change. For example, do we want our QDialog to be enabled? What do we want the maximum size of our QDialog to be? And, if I scroll down, what do we want to be its window title? We obviously don't want "Dialog, " so let's just say something like "Qt Designer. " And, as you can see to the left here, it's no longer "Dialog, " it's just "Qt Designer. " Scroll down a little bit more and we finally get to actually the properties of the QDialog itself. And, there're only two. Aren't you surprised? Well, as it turns out, most of the QDialog's properties stem from QWidget's properties. The two properties we have right here is the Size Grip Enabled, which if enabled, and I'm going to enable it right away, show up these dots on the screen that kind of easily allow the user to resize the QDialog. A modal property, if enabled, will set the dialog in a modal mode. Let's set it. What does modal mode mean? A modal dialog is a dialog that demands or rather actually insists on user's intention before the user can go back to the main part of the application. One example would be a prompt if you want to save a file or discard it or overwrite it or whatever. And, that dialog won't allow you to do anything else before you act on one of its actions. Let me actually demonstrate. If I just hit close button right here, this Save Form pops up and if I try to click anywhere else, it's going to complain and say that I can't. I have to act on one of these actions. Finally, more to the bottom, we have our signal and slot editor. In here, we can connect our events or signals to our handlers or slots. Now, this

has a certain limitation about which we'll talk more in one of the following videos, but just letting you know that it's here. You can see also there's a tabbed interface, so here we have an action editor and also something called the Resource Browser. And, we'll definitely touch on these later on in this course. Now, you're probably anxious to start playing with the dialog itself and put some widgets on it, so that's exactly what we are about to do.

Our First App

Let's have some fun with Qt Designer and our dialog. Here in front of me, actually, is our downloader replication that we created manually just by typing code in Part 1 of this course. Remember that here at the top, we have two QLineEdits, a QPushButton below it with a text of "Browse, " a Qt progress bar, and, finally, another QPushButton. Our goal for this video is to create the same application just by using Qt Designer. So, let me close this first. And, the first thing we are going to do in Qt Designer is add all the required QWidgets to our dialog. To do that, I'm going to search for QLineEdit in our widget box on the left. And, since we need two of those, I'm going to just click and drag two of those to the screen. Another thing we need is our QPushButton and we also need two of those as well. Finally, we need the progress bar. So, I'm going to search for progress and then just click and drag it to the screen. It's all there, but it definitely looks messy. However, before we take care of that part, let's take care of our individual widgets first. Our first QLineEdits should have a placeholder text of URL. So, I'm just going to click on it to select it, then go to the right to our Property Editor, scroll all the way to the bottom, and find the placeholder text property. If I click it here, I can just type "URL. " Another thing you should do - and this is honestly true for anything, any QWidget you add - is change the object name to something meaningful. In this case, we're going to change the object name of this QLineEdit to "URL" as well. To do that, I can just scroll up and go to object name, select it, and change it to "URL. " We're going to do a very same thing for our other QLineEdit, except this one will have a placeholder text of "File Save Location. " And, its object name is simply going to be "Save_Location. " Now, our QPushButton, let's actually change its name first. And, since this is going to be a Browse button, the object name should be "Browse. " And, if I scroll down, I can change its text property to "Browse. " Let's do the same thing for our other QPushButton. Let's change its text first to "Download" and its object name to "Download" as well. Finally, what we need to do is set our QProgressBar to 0. And, as you can see, by default it's 24%. So, if I scroll down, I can set its value to 0. Another thing we want our progress bar to do is to actually have this 24, this value up here in the middle. And, to do that, if I just go to this alignment horizontal part and change this align left to align H center, we get the results we want. Now, the thing we

need to do, obviously, is change the object name. Let's just get rid of this bar and have it named "Progress. " And, we're done. Of course not. Again, everything is here, but it looks ugly and we haven't used layouts at all. If you watched Part 1 of this course, you'll remember that we had to manually add every widget to the layout and that's only after creating the layout itself. With Qt Designer it's as simple as this. With our QDialog selected, I'm just going to click on this Lay Out Vertically button in the toolbar and our elements are automatically aligned vertically. But, they still look ugly, they're just aligned fine. But, there's lots of empty space and these buttons and everything here's just trashed out and just plain ugly. Well, this is also a very easy fix. Again, with the dialog selected, all we have to do is hit this Adjust Size button. Voila. There we go. It looks great. And, wasn't this simple? Instead of typing all those lines of code, this just took us a few minutes to finish. I'd honestly say that this is impressive. Now, let me show you another nifty feature of Qt Designer. Although we can already see what our dialog is going to look like when ran in the real life, we still have these dots in a grid here and kind of we can't really see what happens when we really click a button and all that stuff, what it's going to look like. To get to that point, what I'm going to do is click on this form here and just click Preview. And, this is actually what our application's going to look like when ran. Now, there's only one slight problem with this application that's that it doesn't really work. It doesn't do anything. But, don't worry, that's coming up. We're going to integrate this application that we designed here with our code, but first let me show you how those different workspaces that I mentioned in the previous video behave on what it is exactly that they do.

Events, Buddies, Tabs

This video is going to demonstrate the different workspaces that we have here. To remind you, those are the Edit Widgets, the Edit Signals and Slots, the Edit Buddies, and the Edit Tab Order workspaces. Let's start off with the one we are currently in: The Edit Widgets one. This workspace or view, whatever you prefer calling it, is the one that allows us to add, remove or otherwise manipulate with the widgets on the screen. I'm actually going to add two widgets: A QSlider, and we're going to use the horizontal slider, and also a QProgressBar. Let's add it somewhere below. So, we're going to reset this silly 24% back to 0. So, the next thing we're going to do is switch to this Signal and Slot workspace. As you can see, this widget box is grayed out so that we can't really do anything with it. But, what we can do is I'm going to click this slider here, which, as you see, shows up as red and I'm going to drag my mouse down below to the progress bar. Watch this connection that appears on the screen. Once I release my mouse button, I get this dialog. In it, I can select any signal that QSlider comes with. The one that I'm interested in right now is the

SliderMoved signal, which is fired every time when you move the slider, be to the left or to the right. In the parentheses here, we're told what type of parameter this signal passes to its slot or handler. And, in this case, it's the int or integer and that parameter contains the current value of the slider. So, if you move the slider all the way to the right with our horizontal slider, it's going to be 100. If you move it all the way to the left, it's going to be either 1 or 0. That kind of also depends on how we configure the slider in the property editor. If I click the SliderMoved signal, I'm offered a list of compatible slots for our acute progress bar on the right side here. The one that I want to connect our signal to is the SetValue, which also takes int as a parameter and, which, as its name says, sets the value of the progress bar. Let's click that, hit Okay. And, not only do we have these silly labels here, but we also have the new entry in our Signal and Slot editor below. We have our sender, which in our case is a horizontal slider, and the horizontal slider is the object name of our QSlider. We have the signal that's being fired, which is SliderMoved. The receiver object, which is also an object name, and in this case, it's progress bar. And, finally, the slot or the handler, which is SetValue. Let's preview the application and see what happens when I move the slider. As I move the slider, you can see that the progress bar gets updated automatically. If I move it all the way to the right, you can see that it's actually configured to 99%. These two values are always the same. Since that integer contains the value of the slider, the integer that it's in the signal, these two values are always going to be the same. It's worth noting that you probably won't be using this feature a whole lot. Usually, you'll want to connect the signal to your own handler, something that will actually do an action that you wrote yourself. So, it's nice to know that this feature exists and if you ever need it, it's there for you to use. Let's go back to our EditWidgets workspace. I'm actually going to remove these two widgets and I'm going to add three QLineEdits. We'll just put them in some random order. And, I'm also going to add three QLabels as well. Let's change its text to ---from "TextLabel" to, let's say, I don't know, "Name. " And, another thing you can do is just copy/paste these widgets. I'm going to use Control+C and Control+V on my keyboard. Change this one from "Name" to, oh, let's say, "Age. " And, finally, the third one should be, for example, "City. " Okay, onto our next workspace: The Edit Buddies workspace. Yes, it's actually called a buddy, but what would you honestly expect from the framework that itself is called Qt? So, let me click that. And, the buddies are kind of a connection that doesn't really have a huge meaning. And, usually it's there when it comes to labels in their respective QLineEdits. So, here we can connect these three labels to their QLineEdits as in those QLineEdits where we want the values or answers to the questions those labels pose. For example, we want the user to input their name here, so we'll connect this label to our QLineEdit here. And, we'll connect this one to one there. And, finally, the city to this final one here. If you're an HTML developer, think of this as a label for and then specifying the ID of the text

input. Honestly, that's about it when it comes to buddies, really. So, we have one more left, one more workspace left, and that's Tab Order workspace. Before I show you what it does, let's preview our application first. Everything looks fine. And, as I type, let's say "Bo, " if I want to navigate to the next QLineEdit without having to use my mouse, I'd normally use the Tab key on my keyboard. If I press Tab, I get the QLineEdit that's right below it and hit Tab again, and I get to the last one. But, what happens if I first close this and I rearrange these QLineEdits and their respective buddy labels. And, go back to Edit Widgets. I'm just going to put this age one that was in the middle on the last position here. Let's preview the application again. And, let me type my name. And, if I hit Tab, I'm actually down here at the age, which is the last one. If I hit Tab again, I'm back up, which is not really something you want your user to experience. So, how do you fix this? Why does this happen? This happens because the Tab order is out of order. To restore it back to kind of normal order, all we need to do is hit the Edit Tab Order workspace and then we can simply click in the order that we want the Tabs to appear in. So, if I click this one as number 1, then click the city as number 2, this age it shows up at number 3 again, so I can just click it to confirm it and it's all fine and great. Go back to Edit Widgets one. Let's preview the form. Now, if I type "Bo" here, I say Tab ---or I hit Tab on my keyboard, I'm back to the city, which is the middle one and, finally, age is the last one. Okay, so now that that's done, I'm sure that you're more than eager to find out how it is exactly that we implement these designs in our application and in our code. And, this is actually the topic that we have for our next video.

Put the Design in Code!

The moment you've been waiting for has come. In this video, we're going to see how we can easily integrate the design of our application that we created in Qt Designer and use it in our Python code. To accomplish this task, we need these things. First, we need the UI file generated by Qt Designer. We also need to convert that UI file to a Python file. And, finally, we need some other Python files with code in order to run our application. So, let's do this step-by-step. The first thing we need is to get the UI file. This is very simple. All we really need to do is save our design in Qt Designer. Here's the application that we designed in one of the earlier videos. So, all I'm going to do is just save it. So, I go to File and Save As. I'm going to save this UI file to my D drive and created a Pluralsight folder. And, let's leave the name as helloworld. ui. Hit Save. And, step 1 is done. Now, we need to get to the meat of things and that is converting the UI file to a Python file. In order to do that, we're going to use a tool called PyUIC, which comes installed with PyQt. Since this is a command line tool, if you're in Windows, you need to open the command prompt or PowerShell, which is what I'll be using. And, you can find either of these in the Start menu. If

you're in Linux or Mac, this tool should be available to you from the terminal. Now, let me open PowerShell. And, I'm already in the same directory where I saved my helloworld. ui file. And, to confirm that, I'm just going to type "ls. " And, yep, it's here: helloworld. ui. Now, to convert this UI file to a Python file, all we really need to do is type PyUIC, then hit the Tab key on your keyboard to auto-complete the command. If you're in Windows, you'll see this pyuic4. bat or something similar, maybe a full path to PyUIC. The first parameter this tool wants is the name of the UI file, which in our case is helloworld. ui. I'm going to type that. Then hit Space. And, we do --output and the output file should be our generated Python file, which in our case is going to be named, let's call it helloworld_ui. py. Hit Enter and we're done. We've just generated the Python file from the UI file. But, how do we use it? This Python file that we generated in itself is not really runnable, so let's wrap it with our own Python file in order to run the application. Let me minimize this. And, I'm going to be using PyCharm. You're more than welcome to use whatever text editor or ID that you want. And, in PyCharm if I hit Open Directory and then navigate to that directory with our UI file, as well as our corresponding Python file, hit OK. Let's make sure that everything's there. Yep, we have our UI file and our Python file. What I'm going to do first is actually create a new Python file. Right-click on the project directory, hit New, and then Python File. And, let's name this one, let's say, app. py. Get rid of this author variable. And, let's add the code that we need in order to make our application runnable. So, we'll have from PyQt4. QtCore, import everything. Similarly for QtGUI, also import everything. Last thing we need is the sys library. And, another thing we need is our helloworld_ui. py. So, I'm just going to say import helloworld_ui. Let me put some blank lines here and create an instance of QApplication at the sys. argv. We're going to call our application Hello World. And, this is going to be a class that we're about to create. Execute. Now, we need to create this Hello World class that we are repressing below. And, this class is going to inherit not just from QDialog as before, but also from an object in this module that we're importing in our generated Hello World Python file. So, we're just going to call helloworld_ui. uihelloworld. Now, you may be confused by this notation. helloworld_ui is the name of our generated file, as you can see here. This helloworld here, however, is the object name of the dialog in our Qt Designer. This ui_ here is something that's automatically prepended by the PyUIC tool. I kind of like keeping it this way because I remind myself helloworld_ui. ui_helloworld. Let's create a constructor method, call the parent. And, what we need to do and the only thing we need to do in order to make this application runnable is call this method called setupUI and pass self as an argument. I'm going to right-click this file here, hit Run, and you can see that our application has run. Just to prove you that this is the designer one, you can see that the window title is still set at Qt Designer. We have our URL, our file save location QLineEdits, our pushbuttons here, and our progress bar here. So, it shows up alive and well, which is fantastic. Now, it's not my intention to make this application

functional in this part of the course - we did that in Part 1 of the course - but just for the sake of demonstration, let's see how we can access objects that we designed in Qt Designer in our code as well. What I'm going to do is set the progress bar value to 50 and also change the URL text to, let's say, Google. com. I'm just going to call self. progress. setvalue 50 and self. url. settext google. com. Remember, this self. progress and self. url come from the object name in Qt Designer. I did tell you that it was important to know them, but trust me, when you have a complex application, you're going to be going back and forth between your Qt Designer and your ID, your text editor just to remember or remind yourself what your object name is. So, let's run the application again. And, you can see that the progress bar value is now 50 and that our first QLineEdit, the URL QLineEdit, has the text of Google. com. As you can see, it is still very easy to access whatever objects we design in Qt Designer from within our code.

Summary

Let's quickly recap what we have learned in this module. First, we took a look at the Qt Designer. And, I'm sure that you understand its powers. Within minutes, we had designed a complete application. And, as basic as it was, it was, nevertheless, complete. We then saw how to use tools such as PyUIC to generate the Python file from the UI file generated by Qt Designer and then integrate that design within our Python code so that we can, well, run the application that we designed. I'm sure that you've learned a lot, but there is still lots left. We'll see how we can style our applications, how we can add icons to it. We'll also use the QMainWindow class, take a look at the splash screens, system tray notifications. So, get ready. Fun stuff in the following modules.

Advanced QWidgets

Introduction

Welcome to Python Desktop Application Development Course, Part 2. And, to the "Advanced QWidgets" module. My name is Bo Milanovich. And, in this module, we'll talk about some of the more advanced QWidgets. But, what do I mean by advanced? Well, we'll take a look at some of the widgets that we just mentioned before, such as a QMainWindow, QMenuBar, QStatusBar, and more. All of which are sort of required to make your application complete. We'll see how all of them play nice together as if they're meant for each other. Well, in fact, they are, so, yeah. Anyway, let's start off with the oldest kid on the block: The QMainWindow.

Main Window

When I said the oldest kid on the block in the last video, I didn't mean that quite literally. I kind of wanted to say that QMainWindow is the parent, I guess, of all the other UI elements that we're going to talk about in this module. This was kind of my attempt at an analogy and, yes, I'll admit it was a poor one. But, anyway, what is it exactly? Well, a picture or, in this case, a video is worth a thousand words, probably more. So, let's see it in action right away. I've once again, started our amazing Qt Designer tool here and we're greeted with this already familiar dialog. In this case, however, I'm not going to create a dialog without buttons like we did earlier. Rather, I'm going to create a main window. Let's hit Create. And, what's immediately apparent is that, other than it being larger, so let's take care of that first, we have this Type Here at the top and down at the bottom, we actually have a status bar. I'm not sure if you can really see it since it's the same color as the rest of the window. But, hopefully, you can at least see that these alignment dots are missing at the bottom stripe here. If we go right to our object inspector, we see that we have a main window with a class of QMainWindow. We also have the MenuBar with a class of QMenuBar, which is this thing at the top that says "Type Here. " And, we'll talk more about the MenuBar later on in the video, but you can probably assume that this is what holds the File, Edit, Settings, and Help actions, such as the one we see here in Qt Designer itself. We also have the status bar with the QStatusBar class, which is almost invisible in our case, but we'll take care of that later. And, the status bar is used to display, well, the message with the current status of the application, for example, or what the application is doing at the moment or what is just finished doing, whatever you wanted. But, what is a central widget with a class of QWidget? Now, you're probably kind of confused. What is it doing here? All this time, I've been saying how QWidget is a base class and that everything else derives from it and now it's suddenly here on the screen. Well, what I said earlier is still correct. However, you can also add a generic QWidget in Qt Designer, which is kind of a container element: A UI element that can contain other UI elements. It works similar to a group box, only that it doesn't really have a border or a label or anything like that. In here, our QWidget is the space or a container between our QMenuBar at the top and our QStatusBar at the bottom. Well, that was a nice speech, Bo, but you really didn't say how QMainWindow is useful. Well, you can see right away that QMainWindow has these two things: The QMenuBar and the QStatusBar. And, it also has some useful properties as well. If I go to my property editor here and scroll to the bottom, we have things such as document mode, which can be set to true or false. And, we'll talk about that more later on in the course. And, we also have an interesting property called unified title and toolbar on Mac, which is, as its name says, it's going to unify the title and the toolbar in Mac operating system. But, the most important part is that only QMainWindow can

contain the menu bar, the status bar, and also the toolbar, which is something we'll discuss soon as well. QDialog can't do any of those. Not unless they try to remember when was the last time you saw an application without the infamous file edit help at the top here. Wait, are you telling us that we have been using QDialog all this time for nothing? No, not really. I mean, the applications we've built so far work just fine. But, as you progress and learn more, you'll probably want to use QMainWindow over QDialog for any even slightly more complex application. That's definitely not to say that QDialog is useless either. In fact, for a preferences dialog, for example, in a large application, the QDialog would be a perfect fit. Well, QMainWindow looks boring and kind of sad sitting here empty all by itself. So, let's start having some fun with it.

Menubar

I mentioned QMenuBar in the last video and here we're actually going to implement it. I'm sure that by now you've already realized that QMenuBar contains things such as File, Edit, Settings, Help, and all the other coolness just like the vast majority of desktop applications have. But, stick around because there is definitely much more to it. Let's get some terms straight first and we'll do that with the help of a picture. I have here a screenshot of the Qt Designer. And, the one thing we saw in action already is the QMenuBar, and that's this menu stripe at the top of the window or the top of the screen depending on what operating system you're using. Now, the QMenuBar in itself is not very interesting. It's just a container and nothing more. And, we are more interested in the elements that it contains. The QMenuBar contains QMenus. The QMenus are all these: File, Edit, Form, but all of them individually. So, a File would be one QMenu. Edit would be another QMenu. Form would be another QMenu. And, so on and so forth. QMenus in themselves are also kind of containers. And, the real action happens in QActions. The QMenu contains QActions. Every single one of these, such as New, Open, Save, Print, et cetera is a separate QAction. Hopefully, that part is clear now. So, to define a QMenuBar contains several QMenus, each of which contain many QActions. Moving onto the real deal here, we see that we have a QMenuBar at the top where it says "Type Here. " And, for the record, you won't find a QMenuBar in the widget box on the left, but you can add it by right-clicking the QMainWindow area and selecting add menu bar in case you accidentally delete it from your QMainWindow. Editing the QMenuBar is very simple, so we're just going to follow the instructions and type here. I'm going to double-click here first and then type "File. " And, I'm going to add two more QMenus. One's going to be Edit and another one's going to be, let's say for example, Help. So, now we have one QMenuBar, but three QMenus. Now, let's add some QActions to our File QMenu. This, too, as you might assume, is also very simple. I'm just going to click here and, as it says, type here. So, let's follow

what Qt Designer tells us. I'm going to create several QActions. The first one's going to be New - and, hit Enter on the keyboard - Open, Save. Let's put something like Python Rules. And, let's say, Exit, for example. There you go. Wasn't that simple? I'm sure you'd agree. Now, I think it'd be a good idea to kind of separate this Exit from other QActions, so what I'm going to do is add a separator and once it's there, I'm going to click on it and drag it so that it's above this Exit. Let's preview the application and see what it looks like for now. I'll click on the File QMenu. We have our QActions here and you can see there's kind of a separator between Python Rules and Exit. Let's close this for now. We won't really bother with the Edit and Help QMenus for now, but let's actually do something with the QActions that we have just created. First, shortcuts are very useful, at least I think they're very useful, so let's add some. I'm going to click here and I'm going to skip New for now. I'm just going to single-click Open. Then, in our Property Editor here, I'm going to find shortcut. And, to edit the shortcut, all I have to do is actually click on this value field right here and then on my keyboard, I'm going to press the keys that are going to be used as a shortcut. Now, for open, that's usually Control+O or Command+O. So, I'm going to click on this and then hit Control+O on my keyboard at the same time. Now, let's do the same thing for Save. Go to shortcut and hit Control+S. Let's preview our application once again by opening the File QMenu. We see that Control+O and Control+S have appeared to the right of the QAction text. Now, what's cool about this is that once we implement this design in our code, we don't have to write any additional code to handle just the shortcuts here. Qt will do that for us automatically. Let's close this for now. And, before going back to New, this Python Rules seems like a statement more than an action, so one thing we could do is mark it as checkable QAction. Again, I'm going to click Python Rules and then in the Property Editor, I'm going to mark this QAction as checkable. And, since Python does indeed rule, I'm also going to mark it as checked. Now, let's preview the application once again, hit File, and we can see that this Python Rule is checked because Python does rule. All right. So, now let's go back to New. What I want to do is add a sub-menu for this New. So, let's have two more QActions: New from template and just new, like blank new. And, just click this plus sign here next to our New and then type. First, let's just say New and then hit Enter. And, we can say New from Template and hit Enter again. Click away and let's preview the application once again. And, by the way, the shortcut for previewing the application is Control+R. And, if I hover over New, now I have the sub-menu of New and New from Template. Now, by doing this, we've essentially created another QMenu. And, now our old New here, this one right here, is no longer a QAction, but rather a QMenu that contains, in our case, two QActions. So, you can see that we can also nest QMenus. We can have many QMenus within another QMenu as well. Now, let's close this. Some of you probably noticed that we can also add icons to individual QActions. For example, if I click File and then Open here, we have this icon

property. Icons, however, require some additional work that we'll cover in the next module. For now, in our next video, we're going to take a look at the QToolBar.

Toolbar

Ooh. I'm sure that was your reaction after watching the last video. It was definitely one of the longer ones and covered lots of material, but I'm sure also that in the end you thought it was worth it. I mean, like I said earlier, you'll likely want to implement the menu bar in the application you're going to design. There's also one more thing you can do to make your application look more professional. Look at Qt Designer. So, we have our menu bar at the top consistive of quite a few QMenus and a bunch more QActions, but what do we have right below it? It's a toolbar or in our specific case, it's a QToolBar. Let's spend some time with QToolBar. What is it really used for? Well, I'm sure that you've heard the word "toolbar" thrown around a lot of times before when working with an application, especially if you're a programmer. Toolbar essentially is nothing more than a container to the already existing shortcuts that you can already find in one of the QMenus. If we look at Qt Designer here, the first QAction we have on the toolbar is a new QAction. But, I can also reach that action by going to File and then clicking on New or by typing the Control+N shortcut. And, yes, if you were wondering, these are the same QActions. In fact, they're identical QActions. The way we'll add actions to our toolbar in Qt Designer is just by copying the existing QActions that we have already defined. And, remember those are New, Open, Save, I'm not really going to copy Python Rules for now, and Exit as well. So, let's first add a toolbar to our QMainWindow here. And, just like QMenuBar or QAction, you won't find it in the widget box list on the left. We're going to add it by right-clicking our QMainWindow and selecting "Add Toolbar. " Now, you can see that this kind of additional stripe has appeared below. And, to actually add the actions to our QToolBar, I'm going to go down here to the right to our signal and slot editor and change the signal/slot edit to action editor. As you can see here, this action editor lists all the actions we have defined in our QMainWindow. Let me drag this new QAction, click and drag to the QToolBar. I'm going to do the same with, let's say, Open. Let's do the same with Save. I'm going to right-click the QToolBar and say Append Separator. And, finally, I'm going to add the Exit QAction. There, wasn't this simple? Right now, you're actually probably thinking two things. First, that this looks incredibly ugly. And, second, that it just seems redundant. I mean, why would you want to have one action in two places? Well, for the ugliness part, notice the difference between our QActions here and the Qt Designer's QActions. They have all these beautiful icons and we just have this ugly text here. But, as I mentioned in the last video, icons require some additional work, so we're going to leave these as they are right now and come back to them once

we have learned how to use icons and other images in our application. As for the redundancy, well, think of it from a usability perspective. You'd actually be surprised how many people are honestly kind of afraid to venture out to the File and Edit and the other menus we have at the top. Why would you do that when you have a beautiful, well, not really in our case, a beautiful icon that just says "New? " Why would someone go to file and then be presented with a bazillion of different options? It'd be kind of confused. And, honestly, it actually saves time. Why would I want to click twice to get to a certain action when I can just click once? Now, an interesting bit about the QToolBar or, rather, its actions is that as I mentioned, they're identical. This means that once we get to the code, we're not going to have to write the same code twice to handle the new action from our File QMenu - this one right here - or, rather, the one that pops up to the right and our the new action from our QToolBar. They're going to be in the same and, as such, they're going to be handled by the same piece of code. Well, I hope that this video was kind of relaxing compared to the last one. Now, let's look at how we can switch the application current view while using tabs.

Tabs

Tabs are incredibly useful QWidgets. Not only do they allow you to make your application more user-friendly, but they also save you valuable screen real estate. In a large application, it's only a matter of time before you run out of space to put your QWidgets on. Tabs solve that problem by providing you with separate views that you can use in any way you like. I'm sure that you've seen tabs used everywhere in applications, although funnily enough, the application that I always use to show you folks an example (The Qt Designer itself) doesn't seem to have any, at least not on this main screen. But, still, you've definitely seen them around, so I won't really bore you with the details as to how they work. Rather, let's add a QTabWidget to our main window and then explore some properties that the QTabWidget has that might prove to be useful to you when developing your application. But, just search for tab in the widget box on the left. There's the QTabWidget. Let's just click then drag and resize it a little bit to better fit our application. And, while we're at it, why not just add the QLabel and also let's have a QLineEdit that I'm not really going to do anything, it's just there for demonstrative purposes. Now, an important thing to note here is that the QTabWidget itself is just a container. It contains several tabs and, by default, we have two tabs, but of course we can add as many as we want or delete them or otherwise manipulate them. Think of it kind of like a QMenuBar having several QMenus or a QMenu itself having several QActions. Okay, so before we do anything else, let's preview our application and this is also kind of going to give us a chance to peek at our QToolBar as well. I'll go to Form and then Preview. So,

we have our QMenuBar at the top, our QToolBar below, and our QTabWidget with two tabs. The first tab selected has the QLabel and the QLineEdit. And, if I select the second tab, obviously those two QWidgets disappear. So, let's close that. And, the fun part about QTabWidget starts with its properties. Let's select it. If I go to the right to our property editor and scroll to the QTabWidget specific properties - we don't need this, let's just scroll down a little bit - and let's not start from the top. The first thing I'm actually going to do is change this current tab text from Tab 1 to something more meaningful, like General, for example if this is, I don't know, like a settings. And, then I'm going to select Tab 2 on the screen here and change its text from Tab 2 to, say, Advanced. Now, as you'll probably notice as you'll definitely notice, I guess, this changes this text of the tabs of each individual tab. Select General again. And, now let's start from the top. The first parameter or the first property we have is the tab position. This property tells the QTabWidget - and, remember, that's just the container - where to place these titles or captions or tabs, whatever you want to call them. Currently, we have North selected, which means that there at the top of the container itself, if I change it to, for example South, we have them at the bottom. Let's put it back to North as that's more meaningful. And, of course, we have East or West, which would put text on the side. The second property is tab shape. And, by default, it's rounded and you'll see why because the only other option is triangular and in my opinion, that looks pretty bad. So, I'm just going to put it back to rounded. You may like it. That's okay. The current index property denotes the currently selected tab. It's zero-based, so the first tab has an index of 0, the second tab has an index of 1, and so on and so forth. Since we only have two tabs, if I try to put the number 2 here, which would mean that I want to select a third, non-existent tab, hit Enter, and it kicks me back to 0. However, if I put number 1, it's going to make this tab select the Advanced tab. Let's put it back to 0. Skip icon size for now. Now, elide mode is a very interesting property. And, to fully demonstrate it, I'm actually going to change this current tab text to General Application Settings, something that's very long. Now, you can definitely see this ugliness and clipping and all that here and these forced appearance of the scroll buttons to the right, which, even if I turn off, this whole thing looks probably even uglier than before. Let's put them back on. And, this is where elide mode kicks in. Right now, we're in elide mode none. And, if I change that from none to, say, elide right, you see here that Qt is basically stretching out our tabs as much as it can before finally saying, "Okay, well, you know what? I don't have any more room left, so I'm just going to elide or omit these letters. Similarly, we can change the elide right to elide left and we have these three dots to the left and, of course, elide middle as well. And, what I'm going to do actually is going back to elide none and change this back to General. And, obviously, you're probably not going to have a tab with a title of General Application Settings, but you might just have a narrow tab container and you can really fit any text or some text that only has 10 or so

characters either. Going back from elide mode down South, we've already seen what "Uses Scroll Buttons" does. And, we're going to skip "Document Mode" for now. And, we have these two guys, Tabs Closeable and Movable, which are kind of cool. So, if I make this Tabs Closeable true, we see this X button appears to the right and this will kind of allow us to close individual tabs, although, it's not going to close them automatically, it's just going to send a signal that we need to handle in our code later on. But, if I mark this Movable as true, we can freely move our tabs around. For example, I can place this Advanced tab to appear before the General tab. And, if we had more, we can just place them in any order we want. Now, we have some more options down below like "Current Tab Name, " "Tab Icon" that we'll look at later, but for now, let's preview our application and move these tabs around. And, you know what? While we're at it, one thing I neglected to mention is that we can also move our QToolBar around. I forgot to mention that in our last video. So, let's just preview it and see how this works. Now, let's start with the QToolBar first. So, if I select these dots to the left, I can click and drag and make it appear to the left of the screen. You can see how our QTabWidget kind of moved to the right to make room for our moved QToolBar. Let's put it back to where it belongs. Now, obviously this is a property you can edit in the QToolBar's property editor and many more options in the property editor for a QToolBar. And, I'm going to let you explore that on your own. But for now, if I click X here, nothing's going to happen. Like I said, a signal is being fired, but we're not handling it anyway. However, if I click on this Advanced and drag it to the left, you can see with the smooth animation, it appears to the left of General. Let's just put back things as they were, so I'll have General first and then Advanced later on. Let's close this main window now. And, there you have it. This was a kind of a longish video, too, but I'm sure it was not too hard. I mean, tabs are a very interesting feature that I use in almost all of my applications, at least in some places. But, enough of these, like, static widgets. Let's look at something to hold our data and display to the user on the screen something like a QTableWidget.

Table - Part 1

And, we have come to my personal favorite: The QTableWidget. I've used it many times and it's incredibly useful, although, I do have to admit that I have a love/hate relationship with it. Why? Well, it's a long story, but if you're really interested, let me know in comments and I'll tell you. Anyway, enough of my rambling. I already did some work here. As you can see, I stretched out our QTabWidget and got rid of the QLabel and the QLineEdit we had. And, to get to the QTableWidget, let's just search for table in the widget box. And, we're presented with two widgets. One is Table View and one is Table Widget. I'll explain the difference in just a second, but

make sure to select the Table Widget and add it to our application. So, let's resize it a little bit so that it fits our application nicely. What is a Table Widget exactly? Well, as its name says, it presents some data, such as some text, in a tabular fashion. It's divided in rows and columns like a table and it's incredibly useful when working with some data sets. But, not just data sets. If we look at Qt Designer, our Action Editor here at the bottom right seems to be a QTableWidget in itself. One Windows application that comes to my mind that uses a tabular view is uTorrent or micro Torrent. When you're downloading a file, for example, you see the data such as current speed, progress, file name, et cetera presented in a tabular fashion. But, what's the difference between Table Widget and Table View? Well, traditionally a widget is something that the user can interact with. For example, the Table Widget specifically can be made such that the user can edit each individual table cell or table item and then save changes to it. The Table View, on the other hand, is more of a presenter. Hence the name "View. " It's going to load up some data from a model and then just present it to the user. They'll already have a very similar API, but I think Table Widget is more fun, so we'll focus on that one. QTableWidget and its parent widgets or classes have a lot, and I mean a lot of properties listed here in the Property Editor. But, what we're going to do first is add some columns, some rows, and some items to our QTableWidget. To do that, just right-click on the QTableWidget and select Edit Items. Now, I'm going to add three columns. And to do that, I can just click this plus sign at the bottom. And the first one, it's going to be, say, Name. The second one is going to be Twitter. And, the last one, let's make it Nice Guy. We're also going to add one row with just, let's say, 1 as the first row. This will allow us to add an item and I'm going to add myself, so this is the name and this is one of the hate part of the relationship that this didn't really update itself, the text. So, we know that this is the name. I'm just going to add Bo here. The second one is my Twitter handler. I'm just going to @pythonbo. And, the last one is Nice Guy, so of course, I'm going to add something like "Yeah! " Let's hit OK and see what happens. And, we have our items in the Table Widget. Well, that's cool, but it's not really exciting. Where QTableWidgets really shines is in its item, row or column properties. So, now that we've seen what this looks like, let's open up the Edit Items dialog once again, go to our items, and the first thing I'm going to do is get rid of this "Yeah! " here for Nice Guy. And, you see that it updated the column and row names here. And, since I'm still a nice guy, I hope, what I'm going to do is hit this Properties button. And, you see that just for this item here, we have a bunch of properties. And, the one I want is all the way at the bottom and the property name is "Check State. " If I change the Check State from unchecked to checked, I'm going to get this checkbox. Now, let's change some other properties and let's select this Twitter handler. In my case, @pythonbo. And, by the way, feel free to follow me on Twitter if you'd like. What I would like to do here is change the text of this @pythonbo to blue. So, the first thing I need to do is select a color, which I can do

by clicking this Color Property. Click the three dots and I'm going to select, let's see, this kind of blue here. And, one more thing I need to do is change the style property as well from no brush to solid. And, think of this brush as a painting brush. One more thing I would like to do is make this text bold. But, before I do that, notice that this is a foreground section, which is going to color the text. There is also a background section that would color this item view in itself, this the background of it. I'm not going to do that for now. But, to make my text bold, I'm going to scroll up and find the bold property and mark it as checked. Now, I'm going to hit OK and we see that the Table Widget has updated itself. Let's preview our application. And, voila, it looks pretty nice. And, what I can do if I'd like right now is edit the name. If I just double-click it, I can change my name from Bo to Bogdan, for example, which is my real name. Put it back to Bo. And, usually your user would have some sort of a button such as Save here or you can even capture the signal to automatically save as soon as your user starts editing something. Okay, so we have our mock data in place. Let's take a look at some of these bunch of properties we have here on the right for a QTableWidget and its parent classes.

Table - Part 2

Let's see what are all those properties QTableWidget has that we can edit and tweak to fit our needs. I'm just going to go over some because, as you saw earlier, there really are lots of them. But, thankfully, most of them are pretty self-explanatory. We're going to select our QTableWidget and let's start off with the properties from the QFrame class. The first property at the top is the frame shape, which defines the shape of the frame that contains our QTableWidget. Right now, it's set to Styled Panel. So, let me change that from Styled Panel to Box. And, as you can see, it changed a little bit, the design is slightly different. Now, we can go to this line width property below and set it from 1 to 5. And, you can see that definitely we have this kind of grayish line around the QTableWidget. We can also change a frame shadow from sunken to, let's say, plain. And, we have this kind of a plain shadow that surrounds our QTableWidget. But, since this doesn't really look that good, I'm just going to revert everything back to defaults. And, to do that, we can just click this red arrow to the right of each property. Okay. So, now let's head down to our QAbstractScrollArea class and that holds two different policies: The two different scrollbar policies. Right now, if we run out of room by adding more and more rows, for example, to our QTableWidget, a scrollbar, a vertical scrollbar is going to appear on the screen. We can override that policy by going from scrollbar as needed to, for example, scrollbar always off or scrollbar always on. So, revert back to default, as well as our line width. Now, let's jump down to QAbstractItemView and Edit Triggers property and expand it. And, remember how in my last

video I was able to edit my name simply by double-clicking the item. Well, I was able to do that because this double-clicked Edit Trigger is set to true. We also have other edit triggers, such as Any Key Pressed or Edit Key Pressed, which are both true. And, if we want to disable any triggers, we can just mark this No Edit Triggers as true. And, we scroll down some more. You see a bunch of these properties such as Drag Enabled, Drag Drop Overwrite Method. QTableWidget does support drag and drop, but unfortunately that's out-of-scope for this course. Still, it's very good to know that that support exists. Now, there's this interesting property called Alternating Row Colors. And, to demonstrate it, I'm going to scroll down and jump down a little bit to the row count and increase it from 1 to, let's say, 5. And, as you can see, we now have five rows. Go back to Alternating Row Colors and mark it as checked. Now, every other row has a different color. This is very useful for differentiation and I personally find this pretty visually appealing. Going back up, we still have the Text Elide Mode that we saw in our QTabWidget video. And, in our QTableView class, we have the Show Grid, which is this grid that separates our rows and columns. If I turn that off, that grid disappears. So, let's turn it back on. We also have the Grid Style property, which is going to change the way these lines look like. But, to see it better, let's turn off our alternating row colors. And, we can change from solid line to, for example, dash line or dot line. Put it back to solid line. Word Wrap is also responsible for wrapping the words within an item. We already saw what Row Count and Column Count do. So, if you want to, for some reason, arbitrarily add more rows or columns to your QTableWidget, you can use Row Count or Column Count. Now, we get to the header part and let me just stretch this out a little bit so we can see these properties a little bit better. Note, the header is this part where it says Name, Twitter, and Nice Guy, but also this part where it says 1, 2, 3, 4, 5 in our case, this it the horizontal header and this is the vertical header. The Horizontal Header Visible makes sure that the header is, well, visible similar to the Vertical Header Visible below. If I turn off the Horizontal Header Visible, you can see that our horizontal header disappears. Let's turn it back on. We also have the Horizontal Header Default Section Size, as well as the Minimum Section Size. And, think of a section size as something like a width for a horizontal header. Similarly to that, we have the same thing for our vertical header: Default Section Size, as well as a Minimum Section Size. Now, we also have this interesting property called Stretch Last Section, which, in our case, might be useful because what it will do in case of horizontal header, it's going to take this last column it has and stretch it to match the width of our QTableWidget container and not look like this, kind of, you know, this Nice Guy column assumes its default section size and ends abruptly. So, let's mark that as checked. And, you can see that the Nice Guy's line stretch all the way to the right. There. We went over some of the most often used properties when working with QTableWidget. Now, one thing to note when it comes to QTableWidget itself is that normally you'd set all these properties, as well

as all the items and data from the code. But, still, I believe that this video, as well as the last one, were very useful to show you all the different things you can do with QTableWidget and its items.

Summary

We learned tons in this module. I mean, we went from ground zero. We looked at the QMainWindow first and then we just kept expanding and adding onto it. We learned how to utilize the awesomeness of QMenuBar and its sibling, QToolBar. We also looked at the QTabWidget and some of the great properties it has. And, we did a pretty in-depth exploration of the QTableWidget as well. That was a lot of widgets, but there's lots more, too. And, I invite you to explore them all. Just play with them and see what they can do. What's ahead of us? Well, finally getting to the point where we're going to start making our application beautiful. Now, we get to have some fun with the CSS and also add icons to our applications, to other images, and more. We were looking at the functionality aspect of our applications so far. Now, we're just going to beautify it all in our next module.

Making Our Applications Beautiful

Introduction

Welcome to the fourth module of the Python Desktop Application Development Course, Part 2. My name is Bo Milanovich and the name of this module is "Making Our Applications Beautiful. " We're at the point now where we have the learned the basics of the UI design, so why not just feel free to explore the unknown and customize the way our applications look. Now, despite the module's title, I'm not 100% sure that the application that we're going to make in this module is going to be beautiful itself simply for one reason. To be honest with you, I'm really not a good designer. However, for this module, my goal is to show you the tools how you can make your applications beautiful in the future. We will accomplish this by styling our application using the CSS, as I mentioned earlier, but we'll also see how to add icons and images to our application. And, I promise you I'll give my best shot at design. An attractive application can have many benefits if done properly, of course, which may end up bringing you high revenues down the road.

First impression is very important, but you know that already. So, let's start off by creating a new application and then exploring the various CSS properties that our QWidgets come with.

Our Application

So, what is our application going to be exactly? Well, assume this silly scenario: A client approaches you and says that they want a desktop application that is going to allow them to log into three social networking websites: Facebook, Twitter, and Google+. They don't want to use the web browser because they don't know how and despite you trying to reason with them, you take the job. Now, here's a couple of notes about this application before we proceed with the design. First, the way I always start designing my application, even knowing that I will style it with the CSS later is by creating a basic UI using the basic or native look and feel and then style the application when you're done with that part. Just so that in the end we know what button is where and where all the QWidgets are in respect to the screen and the application itself. And, that's exactly what I'm going to do in this video, too. And, secondly, this application is actually going to be hosted by QDialog, not QMainWindow as we don't really need all the fancy capabilities of QMainWindow. Now, if you watched Part 1 of this course, you know that before you start working on your application, you should always have a mockup of what it's going to look like. That's right, you should. And, I do have a mockup right here. I can just hope that the real application will be better looking than this mockup. But, anyway, the way I envisioned this application is having a kind of our own implementation of the QToolBar or just a toolbar here at the top where it says Button, Button, Button. And, in this toolbar, these three buttons will actually be the Facebook, Twitter, and Google+. Now, only one of these buttons will be active or selected at a time depending on what website the user wants to log into. If you need a kind of a reference of what this is going to look like eventually, take a look at the settings dialog for Mozilla Firefox. Below in this kind of a main part of the dialog, we'll just have two QLineEdits that will be username and password. And below those, we'll have a QProgressBar that will be a kind of an indicator as to how far we've gone into logging in, useful if the user has a really, really slow Internet connection, but mostly useless otherwise. Remember, this is just a proof of concept application. Finally, we'll have a button at the bottom with the text of "Log In. " I know what you're thinking. This looks more like a webpage than an actual desktop application. But, with the CSS being so web-centric, and with me honestly lacking some great ideas at the moment, this application will still allow me to demonstrate the CSS capabilities, enormous CSS capabilities that the Qt ships with. And, lately lots of applications do look like packaged web pages and sometimes they even are that. But, with virtually unlimited design choices you have with Qt and

CSS and other design aspects of Qt, you can make your applications look like anything you want it to look like. For example, did you know that a Viber desktop client is written in Qt? And, I believe the Spotify desktop client is also written in Qt. And, those two definitely don't look like they're standard-looking applications. Rather, they stand out of the crowd. All right, enough talking. So, let's actually start working on our application.

Styling Our Applications Using CSS - Part 1

I've created a new QDialog or dialog without buttons. And, like I said earlier, I'm just going to put the basic widgets on the screen first. We don't have a QToolBar class available in QDialog, so I'm going to emulate it and add a QFrame first. If you'll recall from one of the previous videos, I mentioned that QFrame is a kind of a container element, which really doesn't do anything on its own. So, let me add it real quick and just look for a QFrame here and drag it on the screen. And, I'm going to put it on the top and expand it to span the entire width of the QDialog. Now, you can't really see it, but you can at least see the squares that they know that it's borders. This QFrame is going to host the three buttons I mentioned: The Facebook, Twitter, and Google+ buttons. Now, what's interesting about these buttons is they're actually not going to be QPushButtons, but rather QToolButtons. While these two are very similar, a QToolButton can be checkable or, in other words, it has a property that will allow it to kind of stay clicked after you click on it. They also support icons and aligning icons with the text, so it can be pretty useful. I'll add three of those. So, let me just search for QToolButton here. And, I'm going to make sure that I drag them and drop them on this frame here. Now, let me resize it a little bit and also I need to make sure to mark it as checkable in our Property Editor. If I scroll down, there is a checkable property. And, now I'm just going to copy/paste these buttons and kind of try to put them in the same space distribution here. There. Now, let's do one more important thing. We should probably rename the object names of these buttons to what they actually represent. So, the first one will be Facebook, the second one will be Twitter, and finally we'll have a Google+ one. Now, let me add two QLineEdits. So, there's really nothing new here. We've all already seen this. I'll make it a little bit wider. And, these two, remember, will be username and password, so let's change their object names as well. Same with our QProgressBar. Let's make that one a little bit wider, too. And, finally, we need a login button, which is just going to be our standard QPushButton. And, let's just change its text from PushButton to something more meaningful, such as Log In. Let's resize this so we can see it a little bit better. But, overall, it's pretty good. Everything's coming along nicely. We have our basic UI and now it's our job to style it and make it at least a little bit more attractive. Okay, let's see if we can do that. So, I'll start off with the

QFrame at the top. Not the buttons themselves, but rather the QFrame. I want to make it have a subtle, gradient background. It's going to go from dark gray at the bottom of the QFrame to white at the top. To do that, I can simply right-click the QFrame and click on this Change Style Sheet. Now, I'm greeted with this Edit Style Sheet dialog and while you can definitely input just straight up CSS right here in this text box, which is what we're going to be doing for the most part, let's also look at these tools that Qt has for us when it comes to CSS. I can click this Add Gradient button here and, obviously in this case, I need the background color. Now, if I click on that, I'm greeted with this set of terribly looking, built-in gradients. And, no disrespect for the Czechs, the Dutch or the Germans. But, no worries. You can definitely create one on your own. So, actually what I'm going to do is click on this black and white and click Edit button. And, now you're greeted with this Edit Gradient dialog, which I can also expand by clicking this right arrow here. In this dialog, you can manipulate gradients any way you want. You can change its shape, repetition, its color, its alpha or transparency, whatever you want. The first thing I want to do is actually change the rotation of this gradient. Remember, I said I wanted my QFrame to go from dark gray at the bottom to white at the top and currently my gradient is set from black on the left to white on the right. Now, to rotate a gradient, it's very simple. I can just grab this semicircle here and then drag it to the bottom. And, now we have from black at the bottom to white at the top, so half of the job is done. But, to change the color, I'm first going to switch to red, green, and blue mode or RGB mode. And, then I'm going to click this semicircle here, the one below it. Once clicked, its color will show up here. And, to prove that, if I click on white, you can see that it changes. Back on the black. Now, I happen to know a nice shade of gray, so I'm just going to put 250, 250, 250, yes, it will be gray. And, then I'm also going to add it some alpha or change its alpha from 255 to 90. Let's see, you know, kind of add the transparency of it. Now let's hit OK. You can see that it's changed here. And, let's hit OK once again. And, now we have this syntax, which does look like a CSS syntax, but it has this weird QLinearGradient here. Well, that's the way it works, at least for gradients. And, as I mentioned, we're probably just going to type the CSS manually instead of using these tools, which I honestly use just when I need gradients and, really, that's not that often. By the way, if you don't know CSS, you'll be happy to know that it's fairly simple to pick up and to use it actually. So, please keep following along. This is not really a good example of CSS. Another thing I want to do is kind of visually separate our QFrame from the rest of the application. So, I can just add a border at the bottom and we can just type this in. So, I'll do border bottom, and let's make it 1 pixel, solid, and have a color of dark gray. A nifty little feature of this dialog is the on-the-fly CSS validator, which currently says "Valid Style Sheet" because it's valid. Obviously, if it were not to be valid, for example, if I remove this semicolon here, it's going to say "Invalid Style Sheet. " Put the colon back in. Let's hit OK and see what happens. Well,

hopefully you can see the difference. As I said, the gradient's pretty subtle, but you can definitely see this border at the bottom. But, what's this? Did our cute little buttons change? Yeah, they did. The container element passes along any CSS properties it has to whatever element it contains, which in this case are our three buttons. Luckily, this is a very simple fix, so I'll just edit the same style sheet again by right-clicking the QFrame and selecting Change Style Sheet. And, now the simple fix is just wrapping this entire ---these CSS properties within a QFrame selector. Let me make sure I add the curly brace at the end. And, now if I click OK, you can see that the buttons are styled, have their default style, while the QFrame still has that gradient and the border at the bottom. I think this looks a lot better. But, the next ones down the line for styling are our QToolButtons. And, we want them all to have the same look and feel. So, for that reason, it would kind of seem silly to edit the CSS just for one button and then copy the entire CSS and put a pasted for the second one and the third one. And then imagine if you had 10 buttons and you wanted to change just one property and you have to go do it all over again. Well, luckily we can just edit the QFrame style sheet, but target just the QToolButtons that it contains. You can do that by using the QToolButton selector, just as I used QFrame here. Now, in the interest of saving time, I'm just going to paste this CSS I already have and explain it exactly what property does what. So, let me paste it first and go back to the top. So, here we have a QToolButton with where I'm telling it to have the background color of transparent, in other words, don't have a background color, a border of none. You can see that the background color has this kind of a grayish shade. And, we have a border of darker grayish shade. Down below, I just changed these properties, but only in two specific states. If the QToolButton is checked or, in other words, if it's clicked or if it's pressed. And, pressed would be when you click on it and do not release the mouse button yet. So, you can see here how the text, for example, in this Add Color button sinks in when I click on it and then once I release or move my mouse away, it's going back to its normal position. The QToolButton hover changes the background color when I hover my mouse, just like it does right here. And, then we have the QToolButton checked hover. So, when the QToolButton is checked and then we hover the mouse, it's also going to change the background color. I'm going to hit OK. And, yes, the buttons look like they disappeared, but they blend in much better. And, if I hover, you can see that this kind of nice bluish shade shows up. And, if I preview the application, I can click on it and make a check and you can see that this darker blue border has showed up. And, if I hover over it, you know, we still have that same color that I specified. You can uncheck it and do all this fancy stuff with it. Now, before we move on, let me show you a little trick that can be very useful. Imagine you have these three or even more buttons in one QFrame and you want them all to have the same style except for this one button that's out there. And, you can change the style sheet of just that one button, but you can also do this. I'm again going to edit the style sheet for the

QFrame and then down below, I'm just going to type this: QToolButton, and then the # key or the # sign, and let's target the Facebook button. I can add its border to say, for example, "border 1 pixel solid red. " Now, if you know CSS and HTML, then you've probably seen this before. This is the way you target a specific ID in your HTML page. And, what's unique in the Qt? Well, the object name, of course. We have our Facebook object name for our first button. And, if a hit Apply, you can see that just this Facebook button has a 1 pixel solid red border. Let me get rid of that for now. Okay, so we're done with our toolbar. Now, let's move on and style the rest of the elements.

Styling Our Applications Using CSS - Part 2

Wow, that last one was a really long video for sure. I hope that you found it useful, though. Now, in this video, we're going to style the other elements and, interestingly enough, we're actually going to be using the bootstrap CSS framework for styling them. If you're a web developer, you've definitely heard of bootstrap before. But, first off, I was sloppy in the last video, I'll admit. So, let me actually change the basic UI a bit. First, let's change the text of the QToolButtons at the top. So, the first one should not be three dots, it should rather be Facebook. The second one should be Twitter. Finally, we'll have Google+. And, let's just add the plus sign here, now, one thing I also forgot to do. And, let's see what happens if I preview the application. And, let's say I want to log into Facebook, I can click on it and it stays clicked. Now, if I want to log into Twitter, you would assume that when I click Twitter this Facebook will be deselected, but oops, that's not what happens. And, if I click this Google+, this definitely doesn't make any sense. But, as I hope you learned by now, there is once again a very simple fix for this. I'm going to close this and just click each one of these QToolButtons and change its auto-exclusive property and set it as true. Do that for each button. Now, if a preview the application once again, click Facebook, and once I click on Twitter, voila. It's automatically deselected and it works just as it should. Now, let me also change the placeholder text for our QLineEdits here. So, this one will be username first. And, the second one will be password. And, since this is a password field, we also want to change its echo mode from normal to password so that we have a masked input. Okay, now we're really ready, I think. Luckily, this video is going to have a fairly simple CSS, though I'm still going to explain it, of course. That's why I'm here after all, right? Let's start off from the beginning. So, if we want both of our QLineEdits to follow the same style, but we don't really have a container element defined, do we? Oh, yes we do. The QDialog itself is a container element, too. So, I'll just edit the style sheet of the QDialog, but target QLineEdits. I can just right-click the QDialog and then same old change style sheet. Once again, I'm going to paste the CSS I already have and, remember this is

coming from Bootstrap 3, and then I'll explain it. So, let's paste it. Okay. Well, this is pretty self-explanatory. However, you can see that we're targeting QLineEdit. We have a padding, which is kind of a space between the container and whatever is inside of it, which in our case is going to be the text. The 6 pixel denotes the top padding. So, we want the distance between the top border here and our text to be 6 pixels. The 12 pixels is the right padding. Now, this may not be really appropriate for our application, but like I said, this is coming straight from the bootstrap. Font size is 14 pixels. This is the color of the font. I can't remember what font or what color this is exactly. But, I do remember the background color is set to white. And, we have a 1 pixel solid border that has a gray color. And, finally, a border radius of 4 pixels, which makes the border below the curve on the corners. The focus property changes the QLineEdit when it's focused or, in other words, when it's selected. See here in this edit style sheet, you can see that there is a blue outline when this text box or this plain text edit is focused. Now, I can actually get rid of this outline 0 because it's not really applicable to Qt. And, let's see what happens if I click OK. So, you can see we have some problems here. And, if I preview the application, you definitely can notice a difference and if I select it, you see this blue border and you can see that it's a little bit curved on the corners as well. But, if I type something such as "PythonBo, " this is not what we want it to look like. This is happening due to our font size property about 14 pixels. To get rid of this problem, I'm just going to resize these QLineEdits and make them a little bit bigger. And, let's preview the application once again. And, if I type "PythonBo" right here, it looks like it works just fine. And, I can type password. You can see that it has a masked input. Let's close this. And, onto our next frame, the QProgressBar. And, since it's all alone here, I'm just going to change its own style sheet. Do that, right-click on it, and hit change style sheet. Let me paste my CSS code. I'm still going to wrap it within the QProgressBar. You don't really have to do this when you're editing a style sheet for a specific element and I think it's good practice. So this is, again, self-explanatory. I'm setting border to 2 pixels, solid gray. Border radius, again, make it a little bit curved on the corners. In this case, I'm telling the text to be aligned in the center. Now, let me move this Edit Style Sheet dialog so we can see our progress bar. And, if I click Apply, you can see that it has definitely changed in the way it looks and we have our text aligned in the center, although we could've done that from the property editor in our object property as well. All right. That's all great, but here's a really nice trick about CSS styling for the QProgressBar specifically: The chunk property. So, right now, we have this green progress indicator or, I guess that's what I call it, as a single item, so to speak. So, let's separate that into the old style chunks and I can do that just by using CSS. Let me just paste the chunk CSS that I have. So, you can see here that I'm targeting the specific property of chunk and I'm setting its background color to be something different. I'm setting the chunk width to 10 pixels and the margin or the distance between the two chunks

should be a half pixel really. That's not really good. Let's make it 1 pixel. If I click Apply, you can see that suddenly our progress indicator is not a single item. It's split into chunks that all have a width of 10 pixels and have a space of 1 pixel between them in this bluish color, dark-bluish color. Now, what's really cool about this is that even though we don't really have a QProgressBar object property of chunk mode or whatever, we can still make it happen using CSS. And, that's perfectly valid. There are lots of these special CSS properties when it comes to Qt CSS styling, which you can find in the Qt documentation. Let's hit OK here. And, let me resize our QProgressBar a little bit. And, finally, we have our Login button that's still on style. So, I'm just going to change its own style sheet. And, once again I'll shamelessly steal from Bootstrap, so I'll just paste the code that I have. And by now, you should already understand this code. So, I'm targeting QPushButton, setting its font size, border, border radius, color, background color, and I'm also setting a specific border color here. Now, hovering, again, is when you hover your mouse over that button. I'm just making sure the color is white, the color of the text in this case. The background color is changed and the border color is changed, too. If I click OK, you can see that the button has definitely changed its style. Let's move it a little bit to the left. And, let's preview this application and see what it looks like. Wow, our application is beautiful. I have to say that I'd totally be in love with it if only it had icons in these buttons at the top. No problemo. Let's add icons to them.

Icons and Images in Our Applications

Finally, we're at the point where were going to add icons to our application. Icons are almost essential. You probably could've lived without knowing all that CSS styling, which, even though it's useful, you can still build your application successfully without it. But, without icons, no application is really complete. So, why did I postpone this video for so long? Well, it's because icons require some work. Not to worry. It's not really hard work, it's just a little bit more tedious than we're generally used to. Okay, so let's start. The first task when it comes to implementing icons in your application is to, well, find the icons. You can either download some free icons from the Web or you can create ones on your own or you can hire someone to create them for you, whatever you prefer. But, in my case, I just downloaded some freely distributed icons from the Web that kind of follow the same design and style. For the purpose of this video, I have saved our dialog from Qt Designer in a directory and named that saved UI file "css. ui" and I've also created the icon subfolder, which contains our three icons. As you can see, we have Facebook. png, Google+, and Twitter icons. Also, don't worry about getting icons with the. ico format. PNG icons are well-supported in all major operating systems. Now, the second task is the tedious task. In order for us to use icons, we need to create a so-called resource file for the Qt to use. A resource

file can be used for many things, one of which is obviously including icons in your application. But, another important usage of the resources is the translation part of it. For example, if you want to have a multi-lingual application. Lots of programming languages use resources in one way or another and if you're an Android developer, you'll know what I'm talking about for sure. The resource in Qt is a file that has a kind of an XML-style syntax, which is specific, but fairly simple. Still, we need to manually create this file and by that, I mean there is no any kind of tool that automatically generates this file for us. So, we have to type it in and then load that as a resource in the Qt Designer. This file also needs to have a specific extension to it, which is QRC. And by default, on Windows and I believe on Mac OS as well, you're not able to edit extensions or add ones on your own, so you'd have to edit your operating system settings in order to gain that ability. Instructions on how to do that are in the links below. It's pretty simple, but also know that it's not really mandatory to do for this particular video, but it may be a good idea to have that option enabled regardless. So, the goal for us is to create an icons. qrc file. Now, this icons. qrc file can be placed anywhere. You can put it in the icons folder or you can put it in the root folder. For now, I'm going to put it in the icons folder. And, the way to do that for me, since I'm on Windows, I'm just going to right-click, go to New, and in this case, I'm just going to select a new text document. Windows thinks it's smart enough, so it says, "I'm going to assign this an extension of txt. " I'm just going to rename this to icons, but leave the extension as txt for now so that I can edit it. Now, you can edit this resource file in whatever text editor you prefer. I just happen to like Notepad++ on Windows, so that's what I'm going to use. So, let me open this text file and let's add some XML to it. If you're familiar with XML, that's great. If you're not, it's very similar to HTML, except that it's not really semantic. If you're not familiar with HTML, that's fine. Just follow along and you'll be just fine. So, the first XML element that we want to add is the RCC. And, make sure that it's all capital. Its child element is going to be a QResource element and the QResource element is going to contain file elements, which will specify the path to our icon files. Since we have three icons, we'll need three file elements as well for each icon. So, let's say file, and the first one is Facebook. png, close the file element. Create another one: Googleplus. png. And, finally, for Twitter. Now, let's close the QResource element and also close the RCC element. Now, we need to save this file, but don't save it by clicking File Save or by typing Control+S on your keyboard. Do a File and then Save As. With the Save As dialog, you're offered an option of saving as a specific type. In this case, we're going to select all types or all files, depending on what application you're using, then I'm going to manually edit the extension from txt to icons. qrc. Hit Save, minimize this, and you can see that now I have two files in my application or, rather, my folder on my application. So, I can actually get rid of this icons. txt file. Okay, the boring part of icons is over, so let's make our application shine with our new icons. In Qt Designer, I'll go to the

bottom right and make sure that this resource browser is selected. Then, I'll click on this pencil icon here, then on this open resource file icon here, then I'm going to navigate and find my newly-created QRC file, which is in the folder called "Beautiful" named after our application, of course. And, there is my icons. qrc. Look, I have all these three icons here to the right and icons. qrc selected. I'm not going to select any right now, I'm just going to click OK. And, you can see that the three icons are up here in the resource browser as well. So, all of our icons are here, we just need to add them to our buttons. I'm going to click the Facebook button first, scroll down, and find its icon property. Click on that and then I'll click on this down arrow, hit Choose Resource, and I'm also greeted with the same dialog, except when I click Resource Root, I'm offered these three icons. Let me click on Facebook. And, let me do the same thing for our Twitter and our Google+ buttons. Now, this looks just great. Let's make some adjustments. Click on the Facebook button and let me change the icon size from 16x16 pixels to at least 32x32 and do the same for the other two. And, you know what I also kind of liked before? I liked having the text underneath, so I'm just going to scroll down and find this tool button style property and change it from Tool Button Icon Only to Tool Button Text Under Icon. Do the same for these two guys. Now, that's what I call a nice application. We have beautiful icons on the top, we have CSS-styled QLineEdits, and our QProgressBar and our button. And, this just looks great in my opinion. There. This is how you add icons to your application. But, not just icons, any static images as well. Now, there's a different way to add an icon to an application by specifying the filename directly in the Qt Designer, but from my experience, that implementation is pretty buggy. Plus, it's important to familiarize yourself with the resources, at least somewhat. When developing your application, you'll want to use the resources, not just for icons, but for manipulating your string as well when you're implementing different languages. Now, let me assign you some homework. Change the icon of our QDialog window from this green Qt to something much better.

Compiling Resources

There is one more thing we need to do in order to be able to use the icons in our application. You may wonder what it is because the icons seemed to work just fine in Qt Designer and in our design, and that's true. However, when we eventually integrate our design with the code, we'll need to have a compiled resource file as well. Luckily, getting this file is fairly simple. No more manual typing, at least not that much. So, we're going to be using a tool that will generate this file automatically for us, but it's still just one more of those boring tasks that you need to do. At least you don't really have to do it that often. To compile our resource file, we're going to be using a tool similar to the one we used for compiling our design file into Python code. Recall that for that

task we used the PyUIC tool and for compiling resources into Python code, we'll use the tool called PyRCC. So, as I mentioned, this process is fairly simple. In your PowerShell or terminal or even the command prompt, navigate to the root directory where you want to place your Python application or, as I kind of call it, your main Python file. Remember that in my case, I created an application directory called "Beautiful" and I have this icon subfolder, which recall also contains our resource file called "icons. qrc. " And, I also have our UI file here as well. So, to get to the PyRCC command in my PowerShell, I'll just type "PyRC" and then hit the Tab key, which should auto-complete to this PyRCC4. exe, at least on Windows obviously. And, the first argument that a PyRCC tool expects is the location of our resource file, our QRC file, which, in our case, is in the icon subfolder, so I'll just type "icons" and then hit the Tab just to make sure, and I have the icons. qrc. The second argument is going to be the name of the output file or, in other words, the generated Python file. This generated Python file needs to live in the root directory of the application, so not in any other directory, including not in the icons directory. So, we need to add a "-o" for output and we'll create a file called "icons_rc. py. " And, make sure that this is how you name your output file: icons_rc. py. And, I'll explain why this name specifically in just a second. Finally, since we're using Python 3, we'll also need to add a "-py3" flag at the end. Otherwise, this generation would fail. Hit Enter. And, you can see that icons_rc. py file up here in the root folder of my application. So, while we're at it, and I have this PowerShell open here, why not just generate the UI file or, rather, the Python file from our UI here. Recall that for that we need to use the PyUIC tool and then I'll specify the location to the UI file. And, finally, I'll name my output Python file "css_ui. " And, voila. You can see here that I also have the css_ui. py file, as well as the icons_rc. py file. So, why is the naming of this icons_rc so important? Well, it's because actually of the PyUIC tool. The css_ui file expects a Python module named "resourcename_rc, " which in our case is icons_rc. py. When the PyUIC tool finds a resource defined in the UI file, it's going to import it that way. The name of our resource just comes from the name of the file of our resource, which in our case is icons. qrc. So, just make sure to name it "resourcename, " which in our case is "icons_rc" - this is always RC -. py.

Summary

That was a lot of material, but we really learned how to make our application attractive and stand out of the crowd. And, remember, an attractive-looking application coupled with great functionality is what's going to gain you the most clients. Speaking of functionality, that's what's coming up. In this module, we have seen how to design our application and make it pretty. In the next one, we're going to integrate it with the code and focus a little bit more on the user

experience and the functionality part. For example, we'll see how we can implement the splash screen or the loading screen, as some prefer to call it, but also how to put an icon in the operating system's taskbar, which is that icon bar usually located next to the clock on your screen, and how to send notifications to the user as well. We did some pretty exciting stuff so far, so there's no reason to do anything less exciting now.

Let's Get the Application Running!

Introduction

Welcome to the fifth module in the second part of the Python Desktop Application Development course. As you probably know by now, my name is Bo Milanovich. In the previous module, we mostly looked at the core of the design of our application. We saw many different QWidgets in action and we learned how we can style those widgets and be more appealing to the eyes. Well, if not more appealing, at least certainly unique. We also added icons to our application as well. Application design, however, is just one of the parts of the overall user experience. As important as a design is, there's another important part and that's the functionality side, which can also be very tied to the design itself. For this reason, we're now going to jump into the code. In this module, we'll learn how to create our own system tray menus or menus overall, which can be reusable, how to implement splash images, which is an image that appears when your application is loading, at least in some applications, and we'll also see how we can use the operating system's notification API to send the user a notification. And, you may be wondering what that is, but I promise you you've seen it before. And, finally, and probably most importantly, we'll see our application in live action, not just as a preview from Qt Designer. Okay, ready? Let's get to work.

Splash Screen

Since you are already for sure hooked on dealing with icons and images and other resources, I thought it would only be prudent to continue with that, so in this video, we'll talk about the splash image or the splash screen. What is it exactly? Well, as I mentioned earlier, several times in fact, it is a screen that usually contains an image that shows up while the application is loading or starting up. Okay, cool, but what are they exactly useful for? Well, large applications take time to

start up. There is often a lot of stuff happening in the background while the application's starting up and if after starting the application the user doesn't see anything on the screen, that user might get confused and might even try to start the application again, which is definitely something we don't want to happen. One example of an application that uses splash screen is PyCharm and when you start it up, you get a splash screen that has an image and some text. And, by looking at that image you know, oh well, splash screen has showed and I guess my application is starting up and I have to wait a little bit. Now, even though our application is not really that big at all and will start really fast, it's still useful to know this feature. The first thing you need when implementing a splash screen is the image you want to put. In my case, I just downloaded a picture of clouds from the web because, well honestly because I like clouds, so yeah. Implementing the splash screen in Qt is very simple, so we're going to get to work. As you can tell, I already have the blueprint code that'll load our application and I also shuffle things around a little bit. I have this UI folder, which contains the two UI files to generate the Python file, as well as our Qt Designer file. And, I also have that downloaded image. Let me open it up to show it what it looks like. It's a nice looking cloud. Let's get rid of it for now. And, at this moment, you're probably thinking, "Oh, no. Another image equals another resource in the QRC file equals a recompiling QRC file. " Well, no. For this video, I'm just going to refer to the file name directly. All right. So, to implement a loading screen or a splash screen, whatever, we literally need just four lines of code. That's it. Interestingly enough, these lines of code won't be in the constructor method for our dialog. While that approach would also work, it's preferable to load the splash screen before showing your applications, so in between this dialog, creating a new instance of the beautiful class, and showing this dialog. So, I'm just going to add some space in there right now. And, we can get started. Images in Qt are referred to as "Pixmaps. " So, the first thing we're going to do is kind of load our jpeg file as a Pixmap. To do that, we can simply type, let's call it a "splash image = QPixmap cloud. jpg. " Done. Now, we need to actually create an instance of the QSplashScreen class, which incidentally takes a Pixmap as an argument to its constructor. You get it now? It's pretty simple. Okay, let's do this: splash is QSplashScreen splash image. Done. Finally, we need to actually show the splash screen. And, we can do this by typing "splash. show. " Done. Now, let's make sure that the splash screen waits for our UI to set up itself. And, once that's done, the splash screen will go away. So, I'm actually going to add this method after the dialog has been shown. So, it's just splash. finish and then we're going to pass this dialog, which is our beautiful class. And, this is a very beautiful class. Done. And, that's it. You've just added a splash screen to our application. Four lines of code. And, if you really like saving lines of code for some reason, you can almost make this a QLiner. This can be a one-liner and then, obviously, you have to do this. So, what exactly does this line do? Well, with this line, we're kind of telling the splash screen to

automatically close itself once the UI has been set up. Now, right now our dialog's constructor doesn't really do a whole lot. In fact, it just sets up the UI, so that's pretty fast. So, we wouldn't even be able to see our splash image. To overcome that problem, quote unquote, I'm just going to emulate some background task that needs to be executed. First of all, I have to import the time module from the Python Standard Library. And, then we'll just sleep for a second. Let's do this. Pass 1 as a second. Now, let me run the application and see our splash screen in action. Run, and yay! Do you see those beautiful clouds? I just wish the weather here was as nice. But, anyway, you may be wondering, "Well, it's cool that it waits for the constructor method to execute, but what if I want to hide it earlier? " That's fine. Of course you can do that. There is also a hide method, which is similar to this show method. There's a splash. hide as well. Let's leave our splash screen as it is right now and move onto menus, which are way more exciting than they sound, honestly.

System Tray Icon

The system tray icon can prove to be invaluable for your application, especially if your application is designed to run in the background or to start itself when the operating system starts as well. But, when I say "icon, " I actually mean a lot more than that. I happen to call it that way just because the name of the class responsible for handling system tray icon is, well, you guessed it: The QSystemTrayIcon. Oh, and for the record, the location of the system tray varies on different operating systems. On Windows and Linux operating system running KDE desktop environment, it's in the bottom right corner to the left of the clock. So, this would be the system tray area. On Mac and _____ and _____, usually it'd be in the top right corner also to the left of the clock or, I guess at least that's the default setting. So, what can it do? First, as it is obvious by now, it can show an icon in the system tray area. Second, it can also have a custom menu defined for any actions you want. I've made this entire Windows taskbar visible for this video. And, if I right-click this red warning flag that Windows kindly provides me with, I see these three different actions. So, in our application, we'll be able to specify what these actions are and what they do. In addition to that, the QSystemTrayIcon class is also responsible for showing the notifications to the user, and you've definitely seen those around, for example, when your anti-virus needs to update itself, et cetera. Now, in order to make the system tray icon work, we'll need two main things. First, we obviously need the icon once again, but don't worry, we won't put in the resource file for now. We'll refer it to the file directly. And, I have downloaded this systray. png file. Let me open it up so you can see what it looks like. It's a nice icon with a Python logo on it. And, secondly, we need a menu as well, a custom menu. While the system tray icon will work just fine without the menu defined, there really is no purpose for it to even exist in the first place if it doesn't have a menu

really. Okay, so let's piece together our system tray icon. As mentioned, first we need an icon, which we can obtain by passing the filename to the QIcon class. It's kind of similar to what we did down below here with the splash screen and with the QPixmap class, only this time it's QIcon. So, let's call our icon systray icon, QIcon. Get rid of this capital C. Then, we jump in and create a new instance of the QSystemTrayIcon class immediately. QSystemTrayIcon, which as its first argument, takes this QIcon we have defined and as its second argument, it wants a parent. So, in this case, we'll just pass self. That's it. We have officially constructed a working system tray icon. The second thing is adding the menu. And, you may recall menus from dealing with the QMenuBar in the earlier modules. Remember that QMenuBar contains a number of QMenus and each QMenu contains lots of different QActions. The situation here is very similar, only that we just want QMenu and we'll add just a couple of QActions to it for demonstration purposes. So, first we'll create a menu and then add some actions or create some QActions. Menu is QMenu. And, let's have two actions, for example, restore. Create a custom QAction with the text of "restore. " And, we also need to pass the parent, which is "self. " And, let's have another action close, of course with the text of "close" and "self" as the parent. Now, let's add these actions to the menu and then tell our system tray icon to use that menu itself. I can add all the actions that I have defined to our QMenu by calling the "Add Actions. " Watch out. "Add Actions, " not "Add Action. " And, then I can pass a list of actions I want to display in the menu. So, I'll put restore and close. Now, let me add this menu to our system tray. Call the "set context" menu any time you want to set a menu to a specific QWidget. And, pass menu as an argument. Finally, we need to actually show the system tray icon in the tray area, which is as simple as showing the splash screen below. So, I'll just call "systray. show. " And, you know what? While we're here, let's make sure our close action actually does something when clicked. So, connect the close actions triggered signal to a built-in slot or handler. So, let's do "close. triggered. connect" and call "self. close, " which will close our application. Now, let's run the application and our sky is there, but is our icon? Yes, it's here, too. If I right-click this icon, our custom menu shows up and now if I click this close button or close QAction, our application is closed. Everything works just as it should. But, we haven't implemented the restore action yet, and we're not going to. Or, at least, I'm not going to. I'll let you do this on your own as homework. You better watch yourself. I'm getting into this habit of assigning homeworks lately. Now, let's look at one more important capability of the QSystemTrayIcon and that's the notifications part to it.

System Notifications

Showing the notifications to the user, if you already have a QSystemTrayIcon defined, it's ridiculously simple. When I first learned about it, my reaction was, "Oh, wow. Why? " Because it's literally one line of code to show a message. So, this time, I'm not going to bore you to tears, so let's just jump straight into the code. Now, in order for us to show the notification to the user, we call the QSystemTrayIcon's show message method. This method takes four arguments, two of which are required. The first required argument is the title of the notification. You can put, like, an application name or some action that has just been completed. And, the second one is the contents or the text of the notification itself, like some sort of a description. The optional arguments are the icon we want to be displayed in the notification. And, for the record, these icons are built in the operating system. We don't have to download anything in this case. And, the second optional argument is the timeout period in milliseconds, which will tell the notification to automatically hide itself after a certain number of seconds or, I guess, milliseconds have passed. So, let's see how this all works and what it looks like. I'm just going to type "systray. showmessage. " And, remember, the first argument is the title, so just type "beautiful" and the name of our application. And, let me hide this left sidebar here so we can have more space to type our message here. Our message is going to be "This is a test notification for PythonBo's Pluralsight Course! " Okay, let's separate this so I can fit it all. Like I promised, that's it. Let's run the application and see what this looks like. And, there our notification just shows up, just like that. It's incredible. You can dismiss it here. Now, let's close the application. And, now let's actually change the icon of our notification. There are four icons built-in. There's the information icon, which is the default one, and which is what we've just seen. There's also the warning icon, the critical icon, and a case where you can specify no icon is present as well. Let's see what the warning icon looks like first. And, to obtain this icon, we just passed a third argument from QSystemTrayIcon class as well, but then call a static method or if it's an enum in the back. Just pass warning. Let's run the application. And, you see that that blue icon with the I in it is now replaced by this yellow triangle. Get rid of that. Let's change the warning icon to a critical icon on the application. Now we get this red cross here. Get rid of it. And, finally, we can also pass the no icon if we want to for some reason. Let's run it. And, we don't have any icons in this case. Close the application. This is all there really is to it when it comes to notifications. You can see that it's incredibly simple. But, be aware that its API and that of QSystemTrayIcon as a whole is pretty decent. For example, you can catch signals that are emitted when the user clicks the notification or when the user just clicks on the system tray icon itself. In fact, Qt can also provide you with the details on how it was clicked. There's a way to differentiate between right-click, double-click, middle- click, and so on. I definitely recommend playing with the QSystemTrayIcon some more.

Right-Click (Context) Menu

Context menus or as they're called by probably 99% of the people in the world, the right-click menus. Obviously, this is an integral part of many applications and, yes, you can for sure create your own right- click menus and assign them to almost any QWidget. The process is similar to that of assigning menus to the QSystemTrayIcon, but there are a few more things we need to do in order to make our context menu work. Before we begin, we need to select the QWidget we want the context menu to appear on. I have selected our QProgressBar here. If I right-click the QProgressBar, nothing happens. Well, that's because, by default, QProgressBar has no context menu. Contrary to that, I can right-click our QLineEdits here and you can see that we do have a default menu with some actions built-in. While we can most definitely override these built-in actions, I'd still prefer we dealt with QProgressBar first. Let me close this application and hide this part as well. Now, the first thing we need to do for our progress bar is override its context menu policy and set it to custom. We can do that by simply typing "self. progressbar. setcontextmenupolicy" and then call custom context menu from the Qt namespace. Then, we need to connect the custom context menu requested signal to our own handler. And, I will write this part out, the connection part, and then create the custom handler method in just a second. So, we can call self. progressbar. customcontextmenurequested. connect, self., let me call my handler "show context menu. " Speaking of the handler, I need to create it, so let me do "showcontextmenu", was it? Now, before I go on, you may be confused with this position argument here. Where does it come from? Well, it comes from the signal. Recall from the first part of this course that signals or events can easily carry parameters. We'll use this position parameter in just a second. First, we need to actually create a new menu. And, this menu will have just one action: Reset. This action will, obviously, reset our QProgressBar back to 0. To create a new QMenu, pass self as a parent. Create a new QAction, call the reset, also pass self, and add that QAction. And, in the QSystemTrayIcon video, I used the QActions method because I had several QActions. In this one, however, I only have one, so I'll just use the "add action. " In this case, just past the reset action. Let's also connect our reset action to a progress bar's built-in handler called reset. Let's say reset. triggered. connect, self. progressbar. reset. Okay, we have our menu defined, but we're not really calling the set context menu method anywhere, are we? Well, we're not, at least not directly. The custom context menu requested signal takes care of that for us. Now, we just need to make sure our menu pops up. To do that, we call menu. popupmethod, which takes a set of coordinates as an argument. We can obtain these coordinates by calling a map to global method of our progress bar and passing our position argument that we received from our signal. We can do all that in just one line. So, call menu. popup, self. progressbar.

maptoglobal, and pass the position argument to it. Okay, this was a bit more work than the QSystemTrayIcon, but let's test it and see how it works. On the application, we have the clouds, we have the notification, and if I right-click our QProgressBar, I have our reset action. I can click the reset action and you can see that our progress bar resets back to 0.

Summary

What did we learn in this module? First, we took a look at the splash screen, which, although unnecessary, it can definitely prove to be useful, especially if your application does a lot while it's starting up. Then, we explored the QMenus and saw how they integrate with the QSystemTrayIcon and other. Speaking of QSystemTrayIcon, we also saw how your application can notify the user about something that has happened, which is particularly useful if your application is minimized or if it's running in the background. What's ahead? Well, just explore and experiment. Even after the second part of this course, we're still on the tip of the iceberg as far as Qt is concerned. Believe me, when I was looking at the documentation while preparing for this course, I have myself learned many new things. There were definitely lots of moments where I'd say, "Oh, you can do that? That's awesome! " And, quite frankly, I would often say that out loud.

Congratulations and Thank You!

Congratulations - and Thank You!

I have just two things to say to you: Congratulations and thank you. We've definitely come a long way and I'm sure that by now you have the knowledge necessary to build some fairly complex applications. But, as I always say, practice makes perfect. Once again, we're just scratching the surface of the enormous Qt API, but we're getting there. So, congratulations. You took a major step. I hope that you found this course pleasant and that you learned something new. I, of course, always welcome feedback and constructive criticism. And for that, I thank you.

Course author

Bo Milanovich

Originally from Serbia, Bo - Bogdan - started his own software development company at a very young age. He later moved to the United States to pursue studies. As he is finishing up his...

Course info

Level	Intermediate
Rating	★★★★½ (167)
My rating	★★★★★
Duration	2h 17m
Released	10 Oct 2014

Share course