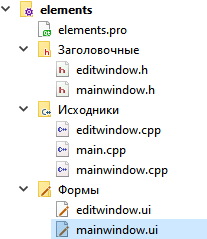
Урок 6-7 Но возможно 5-6

1. Элементы интерфейса

В Qt существует множество контейнеров компоновки и элементов интерфейса. Выбор способа, которым будет показана та или иная информация ложится на плечи разработчика. Например, для получения от пользователя какого-либо числа, можно создать текстовое поле ввода, слайдер, список, комбобокс или специальное поле для ввода числа. Грамотный выбор элемента – залог удобства пользователя и поддерживаемости приложения в будущем.

На примере разработки простого приложения для конфигурации VDS сервера познакомимся с часто встречающимися элементами взаимодействия. Приложение будет состоять из окна регистрации/авторизации и окна с настройками виртуальной машины. Начать разработку стоит с создания нового проекта и добавления к нему через контекстное меню еще одного MainWindow класса. В примере класс второго окна назван EditWindow. Древо проекта будет выглядеть следующим образом:



Для удобства разработки в конструкторе класса MainWindow можно сразу открывать экземпляр класса EditWindow:

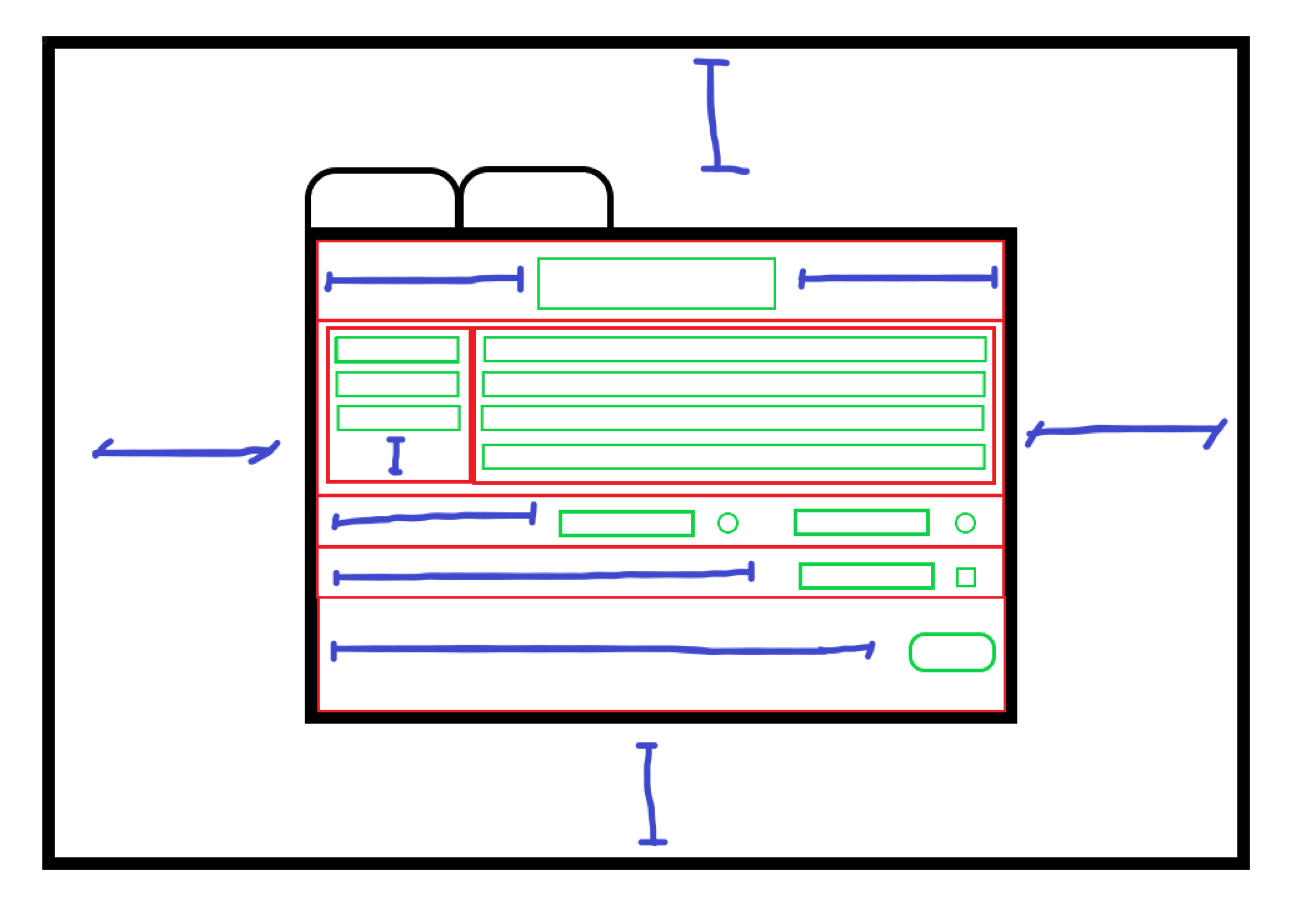


Таким образом, запуск приложения будет приводить к появлению двух пустых окон.

Далее проанализируем элементы интерфейса, которые позволят произвести регистрацию. Для выдуманное организации VDS онлайн планирует хранить почту пользователя и информацию о том, является ли пользователем сотрудником компании. Для хранения личных данных пользователя потребуется его согласие на обработку данных. Для авторизации выбрана схема с логином и паролем. Таким образом, уже можно представить элементы интерфейса. Для удобного переключения между авторизацией и регистраций можно воспользоваться QTabWidget – элементом с вкладками, позволяющим создать подпространства с интерфейсом внутри одного окна. Переключаясь между вкладками, пользователь сможет выбрать регистрацию или авторизацию. Для создания нового аккаунта понадобятся следующие связи элемент-данные:

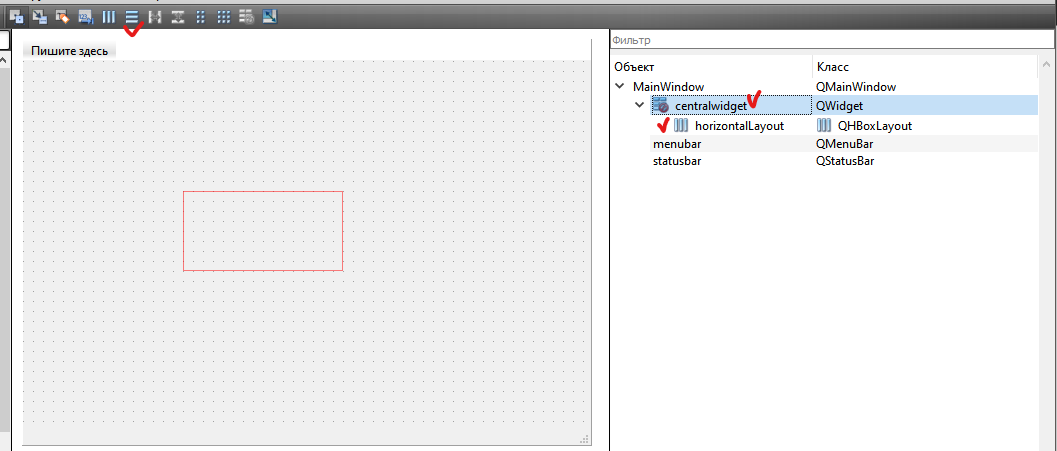
* Поле ввода – логин;
* Поле ввода – почта;
* Поле ввода – пароль;
* Поле ввода – повторение пароля для сокращения опечаток в пароле;
* Два переключателя – выбор типа аккаунта между сотрудником и сторонним разработчиком;
* Флаг – согласие пользователя с политикой сервиса.

Последним элементом является кнопка, при нажатии на которую будет производиться отправка данных. В свою очередь для входа в систему достаточно двух полей ввода с логином и паролем и кнопки. Просто разбрасывать элементы по окну не очень грамотно с точки зрения UI/UX, поэтому следует совмещать элементы, которые могут растягиваться, и элементы с фиксированным размером. “Резиновые” элементы – это в основном контейнеры и поля ввода. Элементы с “жесткими” размерами – это текст и блоки с контентом. После анализа технического задания стоит составить примерный макет окна:

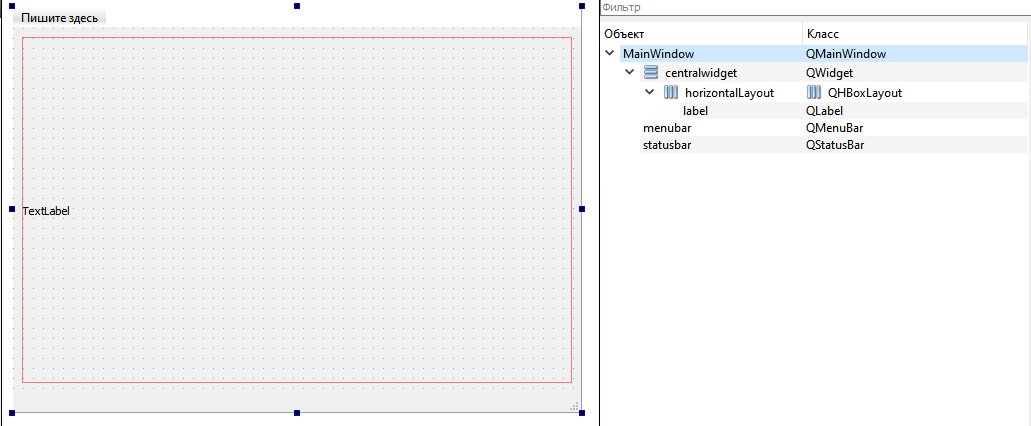


Черные блоки – контейнеры с изменяемыми размерами. Синими линиями на схеме указаны “гибкие” отступы, задача которых подстраиваться под изменения размеров окна. Красные блоки – логические группы элементов с внутренними отступами, которые, возможно, будут являться контейнерами. Зеленым цветом на схеме указаны элементы с контентом и элементы управления.

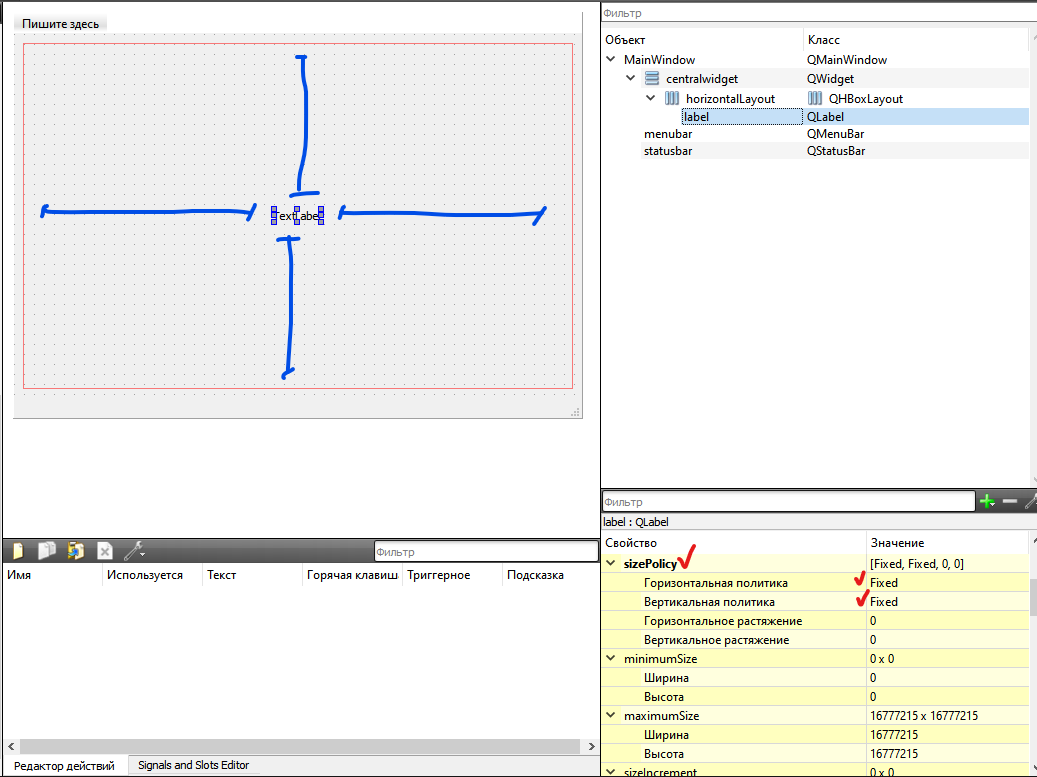
Для понимания того, как Qt центрирует элементы на форме достаточно разобрать простой пример с двумя слоями и одним текстовым полем. Если установить для CentralWidget компоновку по вертикали, добавить в него слой с горизонтальным расположением элементов и внутри него расположить текстовое поле, то получится добиться центрирования элемента. Компоновку centralWidget можно добавить через верхнее меню путем переноса в виджет хотя бы одного дочернего элемента:



В результате получим:

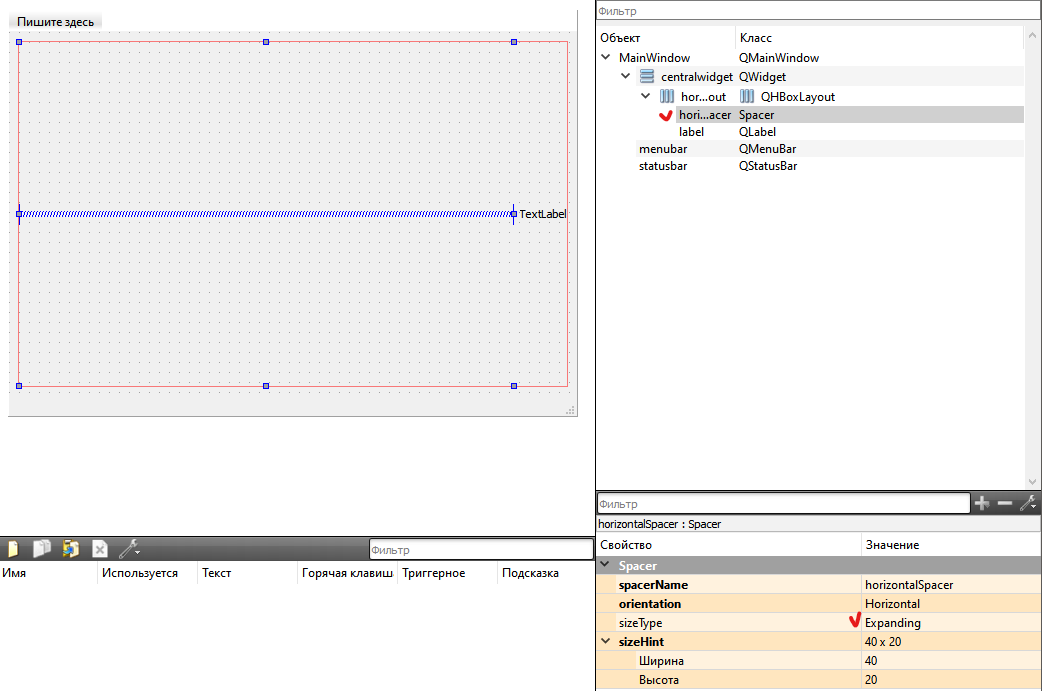


Если изменить свойства, связанные с размерной политикой с Expanding (или Preferred), то элемент перестанет автоматически занимать всю площадь, а расположится в центре окна:



Описать происходящее достаточно просто. Слой с вертикальным расположением по умолчанию центрирует элементы по вертикали и пытается растянуть по горизонтали. Горизонтальный слой имеет зеркальное поведение – по горизонтали центрирует, по вертикали растягивает. Добиться центрирования по обеим осям получится, совмещая оба слоя. Понимая эту концепцию, не сложно собрать отзывчивый к изменениям окна интерфейс.

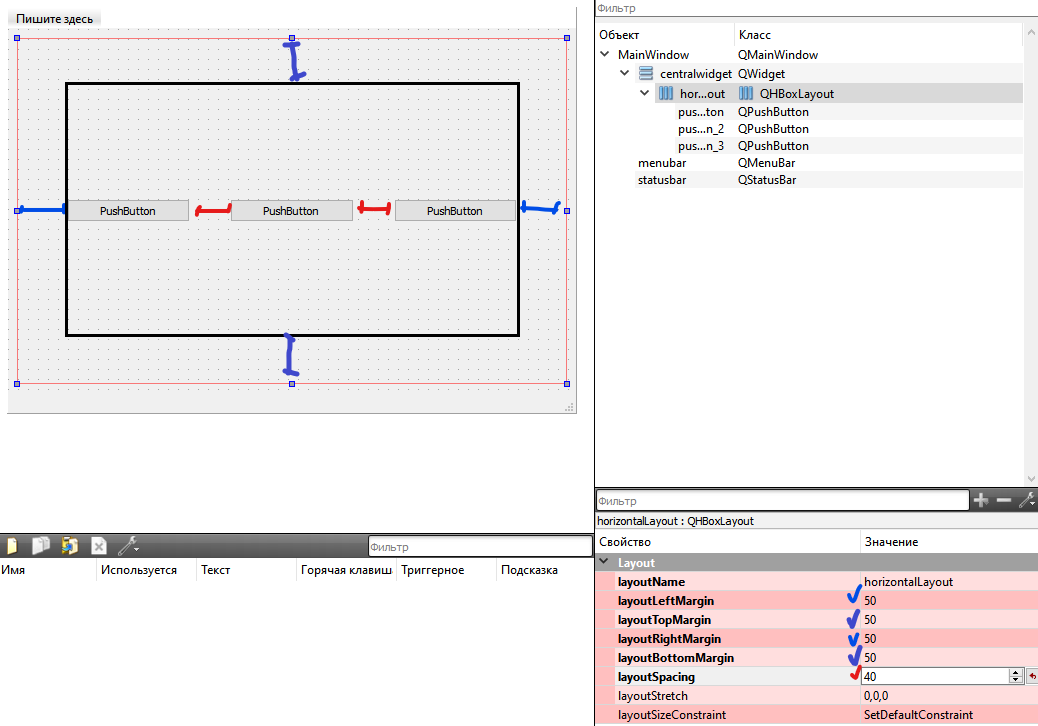
Для изменения положения элемента на слое можно воспользоваться специальными отступами – spacers. В какой-то степени они и являются синими линиями на схемах. Например, для привязки блока с фиксированным размером к правой части контейнера, следует добавить слева от блока HorizontalSpacer c типом Expanding:



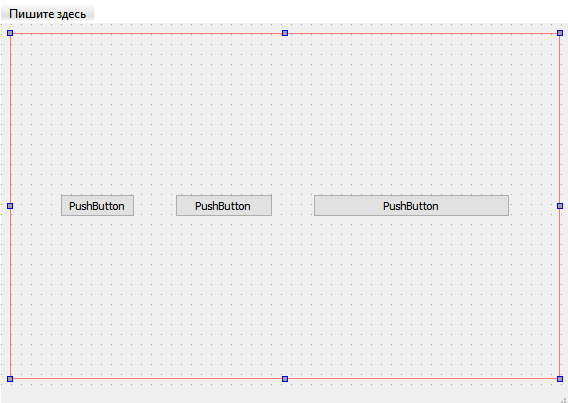
Если сменить тип на Fixed, то ими можно пользоваться как статичными отступами, указанными в единицах размерности.

Для исправления ситуаций, в которых блоки “прилипают” друг к другу, следует добавлять отступы между блоками. Существует два подхода: отступы элемента и отступы для дочерних элементов. Внимательно изучив свойства слоев, можно сделать следующие выводы:

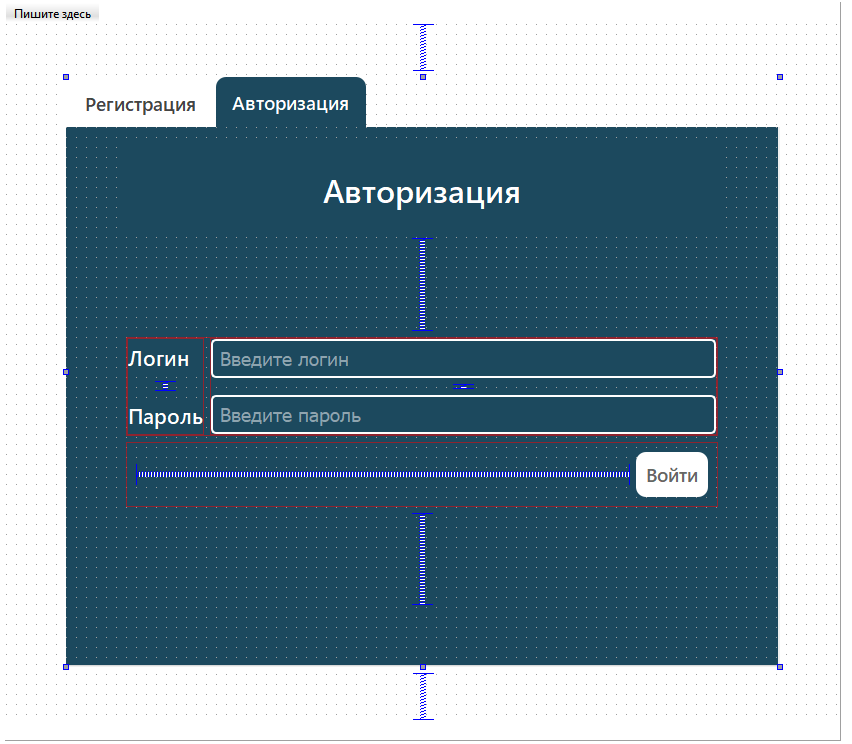
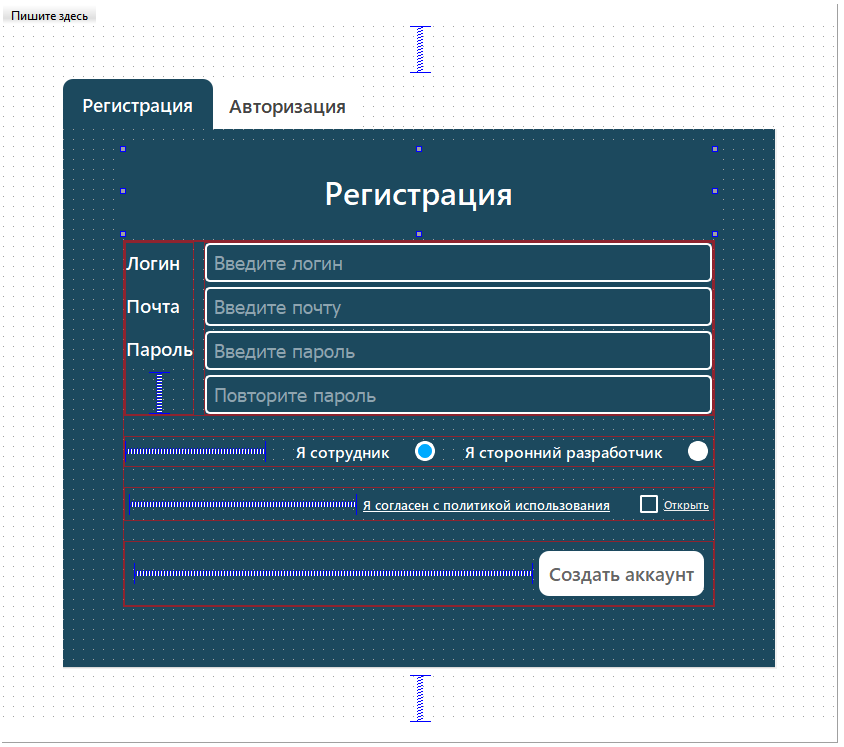
* Margin – отступ слоя от других элементов;
* Spacing – отступ между элементами внутри:



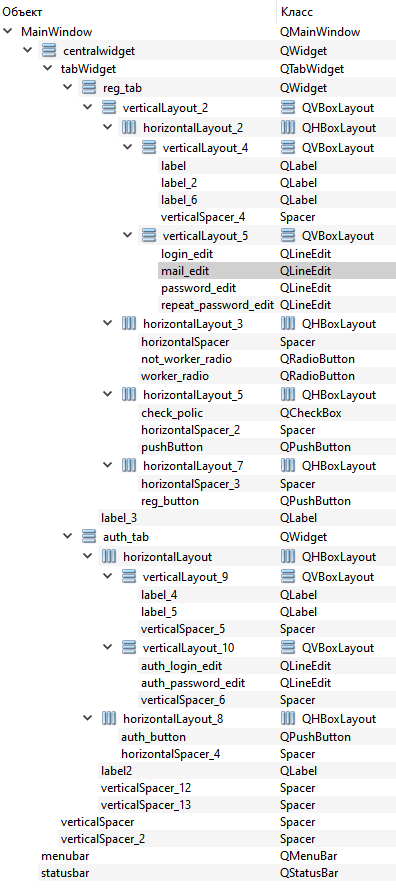
Свойство LayoutStretch тоже имеет непосредственное отношение к дочерним элементам. Количество чисел совпадает с количеством элементов внутри слоя и обозначает их вес, который можно интерпретировать как место, занимаемое в пространстве. Например, если указать значение как 1,10,20 получится такой результат:



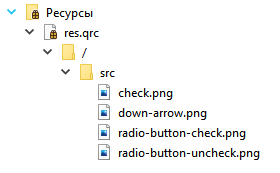
Скомбинировав все изученные приемы компоновки и добавив немного css, можно добиться отзывчивого (адаптивного?) и приятного интерфейса:



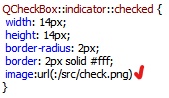
Структура элементов формы на картинке выглядит следующим образом:



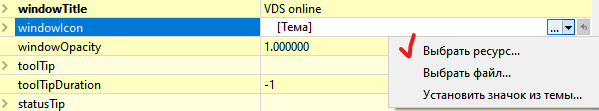
Примеры стилизации каждого из компонентов, существующих в Qt, можно изучить в [документации](https://doc.qt.io/qt-5/stylesheet-examples.html#customizing-specific-widgets). Единственный, стоящий отдельного рассмотрения аспект, – использование иконок из ресурсов. В разрабатываемый проект были добавлены png файлы со следующей структурой:



Для подключения их, например, вместо галочки во флажке url будет выглядеть так:



Для смены названия приложения и иконки, отображающейся на панели задач, следует изменить следующие свойства объекта MainWindow:



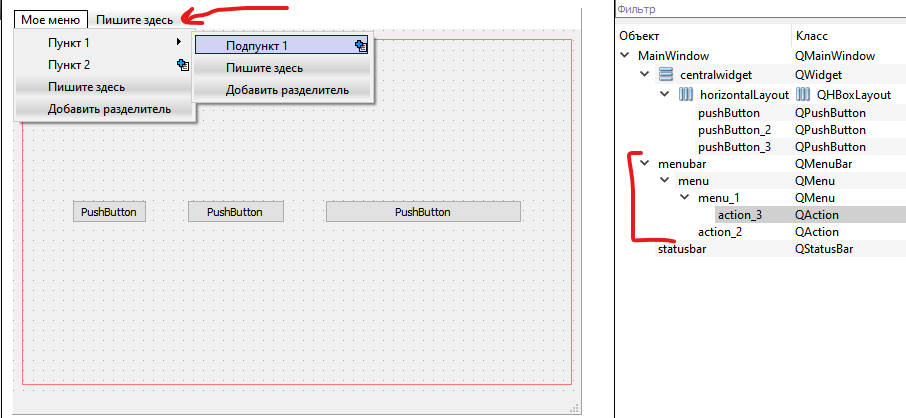
Очередным шагом в разработке приложения является сборка интерфейса для отображения конфигурации сервера. Известно, что приложение должно:

* Показывать состояние сервера и время работы с момента последнего включения;
* Отображать существующие задачи, например, создание резервных копий баз данных;
* Предоставлять настройку ОЗУ от 2гб до 512гб и варианты существующих ПЗУ;
* Демонстрировать состояние файловой системы сервера;
* Давать возможность манипулировать состоянием виртуальной машины;
* Иметь интерфейс для смены аккаунта и получения справки о приложении.

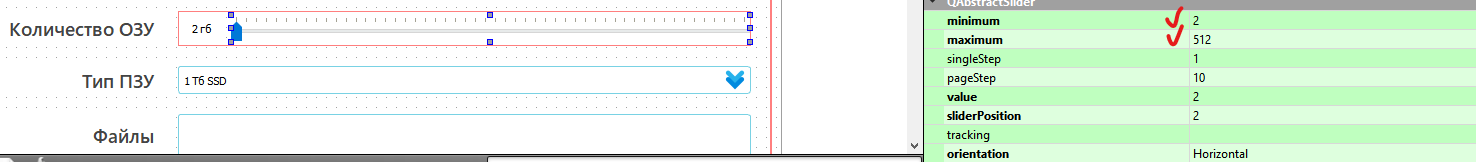
Для решения поставленных задач потребуются следующие элементы:

* Текстовое поле для отображения состояния;
* Список для отображения существующих задач;
* Слайдер для настройки количества ОЗУ. Самый удобный элемент для настройки числа в диапазоне;
* Выпадающий список для выбора варианта ПЗУ;
* Специальный виджет QTreeWidget для удобного отображения вложенности файлов;
* Настроенное верхнее меню для манипуляции с виртуальной машиной, аккаунтов и справкой;
* Кнопка для сохранения изменений в конфигурации.

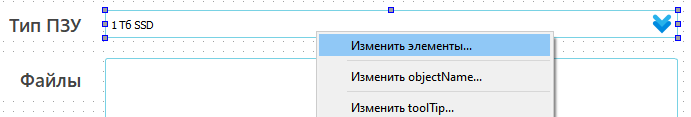
Для создания верхнего меню в приложении в форме MainWindow существует элемент menubar. Он является обязательной частью, как и centralWidget. Создавать пункты и подпункты можно с помощью простого вписывания названия в соответствующий элемент. Элементы меню тоже имеют слоты и objectName, через которое разработчик получает к ним доступ.



Такие элементы интерфейса, как слайдер и выпадающий список, можно подготовить по требованиям заранее, если они известны. В случае выдуманной компании VDS Online слайдер имеет минимальное и максимальное значение, поэтому можно указать эти данные в свойствах элемента:



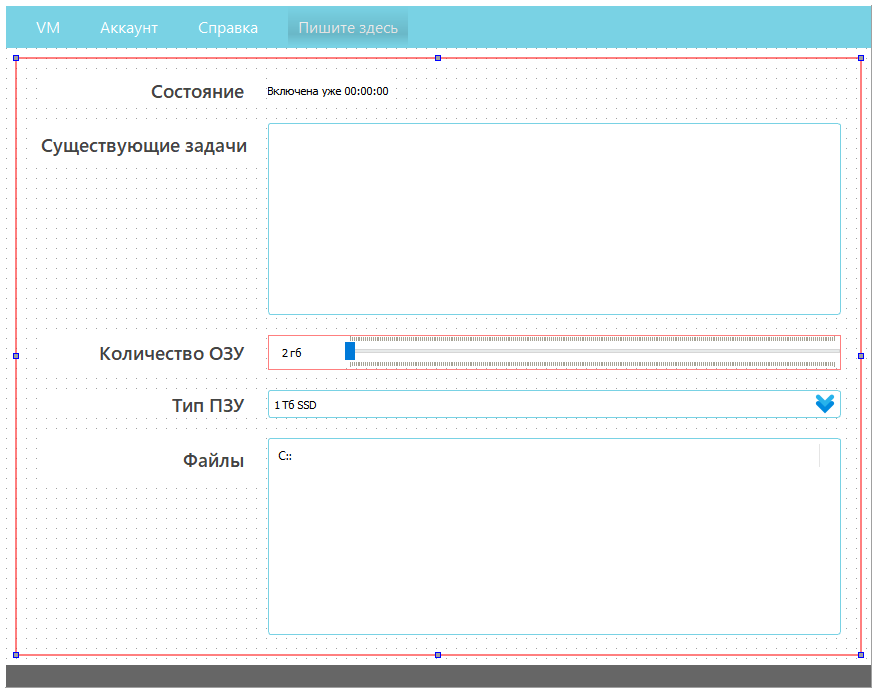
Существующие варианты ПЗУ можно добавить через контекстное меню:



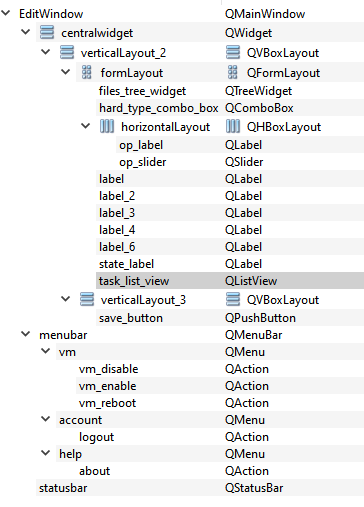
Следует обратить внимание, что загрузить элементы в комбобокс и ограничить слайдер можно и программно:



Итоговый вид окна для настройки конфигурации сервера:



Итоговая структура элементов на форме:



Для удобного размещения пар название-элемент был использован контейнер QFormLayout, который позволяет располагать элементы в два столбца.

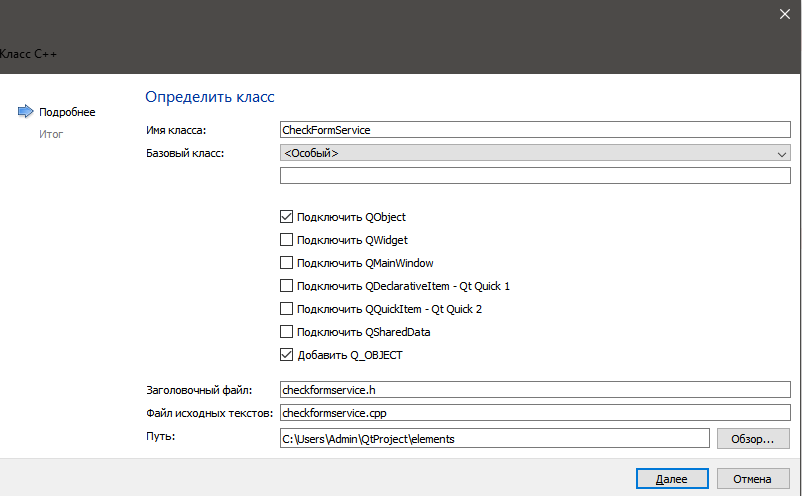
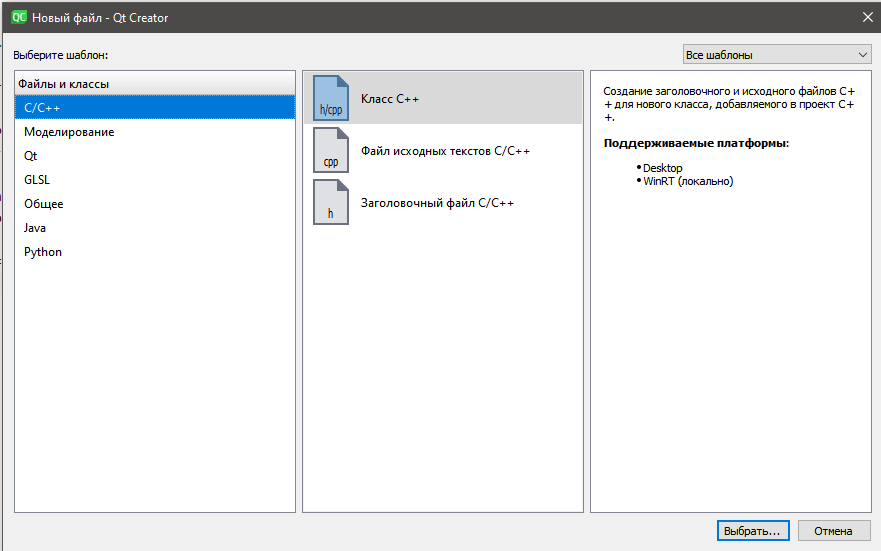
После разработки внешней части программы – интерфейса, можно приступить к разработке логики приложения. Так как пока программа не связана с локальной базой данных или внешним сервером данные для сравнения и отображения будут являться заглушками в коде.

Начнем с реализации регистрации. Основными типами для передачи данных являются JSON и XML. В случае передачи простых данных из формы выгоднее использовать JSON формат. Для работы с ним в Qt существует специальный набор классов, имеющих префикс QJson.

Реализуем получение данных с формы и сборку объекта QJsonObject для дальнейшей его передачи. Для выполнения задачи следует:

* Создать слот, реагирующий на нажатие кнопки на страницу регистрации;
* Реализовать получение состояний элементов управления;
* Проверить данные на валидность и, в случае отклонений, подготовить информационное сообщение;
* Собрать QJsonObject и сохранить его во временном хранилище.

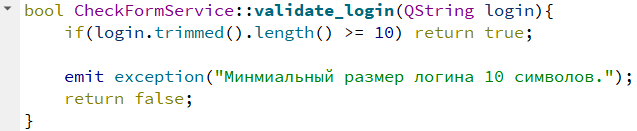
Для проверки данных следует создать отдельный сервис, в котором получится инкапсулировать логику анализа полей на валидность. Разработанные сигналы и слоты помогут интуитивно реализовать работу сервиса. Подготовим методы для проверки, которые при ошибке будут посылать сигнал, запускающий слот для показа окна об ошибке:





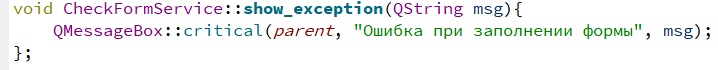
Для взаимодействия с классом следует создать специальный API – публичные методы, позволяющие использовать функционал класса. Методы, которые используются для декомпозиции расположены в private секции. Таким образом разработчик обеспечивает стабильность работы класса. Иногда категорически важно, чтобы блоки кода были выполнены в конкретной последовательности.

Методы валидации строк бывают двух типов: по длине и по регулярному выражению. Например, почта не ограничена по длине, но соответствует определенному шаблону, в отличие от логина, имеющего только ограничения по размеру и присутствию некоторых символов. Так как приложение подразумевает изучение элементов интерфейса, в нем не будет углубления в валидацию данных. Добавим небольшие ограничения на почту, логин и пароль. Проверка элемента производится по следующему шаблону: если элемент подходит – возвращаем true, иначе – отправляем сигнал exception (msg) и возвращаем false:



В публичном методе проверки можно вернуть логическое И от результатов всех методов. Следует отметить, что выполнение пересечения прекратится при первом полученным false.

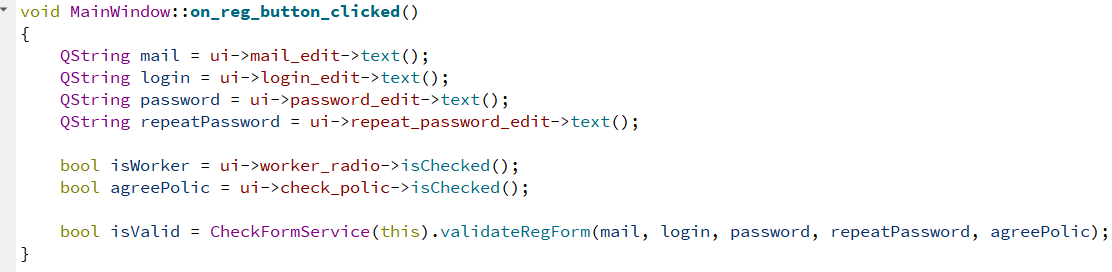
Для функционирования сигнала и слота нужно подключить их с помощью метода connect() в конструкторе класса. Слот show\_exception() будет вызывать показ окна с переданным сообщением:



Весь код файла checkformservice.cpp выглядит следующим образом:

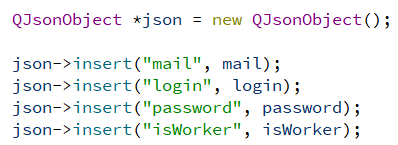


Осталось воспользоваться сервисом перед сохранением данных:

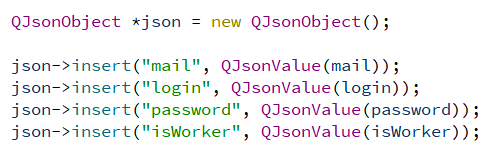


Далее по плану создание экземпляра класса QJsonObject и заполнение его данными. Json напоминает структуру Map с одной важной особенностью – типы значений не обязаны совпадать. Значение может быть строкой, числом, другим объектом и т.п. Для хранения не типизированного значения в Qt существует специальная “обертка” – QJsonValue. Она имеет несколько методов для определения типа хранимого значения, таких как isString(), isObject(), isBool(), isDouble() и др. Для изменения типа элемента с QJsonValue на нужный, определены специальные методы с префиксом “to”. Для создания экземпляров класса существует несколько перегрузок конструктора с аналогичными типами. К полям объекта можно получить доступ двумя способами, используя метод take() и оператор []. Важным различием между ними является то, что при вызове take(), удаляется ключ.

Сборка экземпляра для хранения данных формы выглядит так:



Следует обратить внимание на отсутствие на картинке упоминания QJsonValue. Действительно, благодаря большому количеству перегрузок конструктора QJsonValue компилятор qmake может самостоятельно перевести QString в QJsonValue и не обязательно прибегать к следующему виду:



Данные подготовлены, осталось только их сохранить. Для хранения данных создадим собственную не реляционную базу данных. Данные будут находиться в двух файлах. В примере генерация id будет упрощена до инкрементирования специальной переменной.

Часто для работы с объектами, хранящимися в базе данных, приходится создавать специальные классы, через использование которых можно будет удобно изменять состояние объекта и в последствии передавать его в базу данных для сохранения. Так как в примере было принято решение использовать QJsonObject, то существует возможность унаследоваться от него и адаптировать под нужды приложения.

Создадим класс для хранения состояния виртуальной машины:



Часто используемым кейсом является создание специальных перечислений в случаях, когда вариантов ограниченное количество. Таким образом увеличивается качество и читабельность кода. В 7 строке подключается объект QDateTime, использующийся для хранения времени старта виртуальной машины. В строках 46-48 методы родительского класса переведены в private секцию. Это сделано для того, чтобы изменение состояния класса ограничилось подготовленными в производном классе public методами.

Задача класса VirtualMachine в создании подходящего объекта и управлении его состоянием в дальнейшем. Класс является простейшей версией структурного паттерна [Proxy](https://ru.wikipedia.org/wiki/Заместитель_(шаблон_проектирования)). Реализация методов выглядит следующим образом:



Стоит обратить внимание на 8 строку, в которой использован экземпляр QJsonArray – версия массива, хранящая QJsonValue. Так как на данный момент приложение не подразумевает редактирование задач и файловой системы, методы по работе с ними реализовывать не обязательно. Для комфортной разработки функционала по выводу данных на экран эти данные будут добавлены по умолчанию в конструкторе или при создании экземпляра в базе данных. В коде используется два метода QDateTime: currentDateTime() и fromString(). Первый используется для получения времени в данный момент, второй для создания экземпляра из строки. Используя эти способы создания экземпляров класса, в 65 строке производится вычисление разницы между временем в миллисекундах с помощью комбинации time().msecsTo(). Вызов time() возвращает экземпляр QTime, с которым возможно производить вычисления.

Если классы для хранения подготовлены, можно приступить к написанию базы данных:



Для обеспечения единой базы данных для любой части программы удобно воспользоваться классом со статическими методами. Для полноценной работы БД следует реализовать методы считывания и записи данных в файл и несколько методов сохранения и получения структур. Полезными будут являться следующие функции:

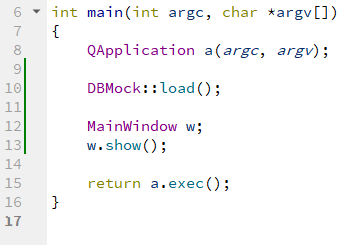
* Сохранение пользователей с присвоением им созданных виртуальных машин;
* Сохранение виртуальной машины в соответствии с id пользователя;
* Получение id пользователя по логину для проверки существования аккаунта. Альтернативным вариантом может быть возвращение bool значения сразу, но в таком случае это уже не база данных, а сервис по взаимодействию с ней;
* Получение id пользователя по логину и паролю;
* Получение пользователя по id;
* Получение виртуальной машины по id пользователя.

Проследим за нитью логики работы приложения с БД. Для получения данных и сохранения состояний виртуальных машин в окно конфигурации будет передаваться id – это универсальный способ доступа к данным, если достичь грамотной генерации идентификаторов.

Логику создания шаблонной новой машины следует вынести в отдельный метод, потому что это участок кода, который может быть подвержен изменениям в будущем. Существенный вклад в работу базы данных привнесет метод load(). Его можно использовать для отладки ошибок и загрузки каких-либо данных. Реализация методов выглядит следующим образом:



После того как база данных подготовлена, можно приступить к подключению ее к проекту. Следует вызвать метод load():



Дополним логику класса MainWindow таким образом, чтобы при регистрации данные сохранялись в БД, а при удачной авторизации пользователю предоставлялось окно с настройками виртуальной машины. С созданным слотом для авторизации файл mainwindow.cpp будет выглядеть следующим образом:



Таким образом, оба слота работают по общей схеме, которая была описана ранее. Важный аспект можно увидеть на 53 строке. После удачной авторизации id пользователя передается окну редактирования виртуальной машины. Авторизация считается удачной, если метод getPersonIdByLoginAndPassword() вернул число, не равное -1. Именно такое поведение задумывалось на этапе реализации базы данных. Первым аргументом в конструктор EditWindow передается 0. Таким образом достигается самостоятельность окна и выделение его как самостоятельной функциональной единицы приложения.

Следующий и заключительный этап разработки приложения – реализация слотов класса EditWindow. На данный момент к нему добавлено поле для хранения id пользователя и дополнительный конструктор:



Стоит начать с отображения данных на экране. Для этого существует два подхода:

* Загружать данные в конструкторе и изменять поэлементно при действиях пользователя.
* Использовать специальный метод отображения сразу всех данных, который будет вызываться при любом действии пользователя.

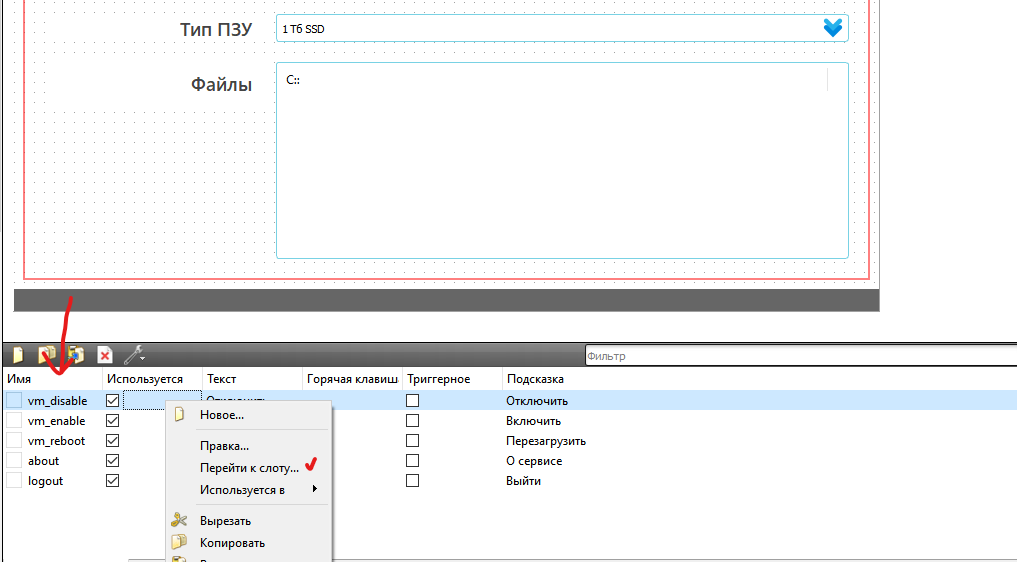
Оба варианта имеют право на существование, но стоит отметить, что удобнее пользоваться вторым, так как он позволяет сконцентрировать логику ui в одном месте, а не распылять ее по методам.

Познакомимся с каждым способом на примере учебного приложения. Для отображения состояния, оперативной памяти, задач и энергозависимой памяти будет использоваться единый метод отображения. А древо файлов и url адрес виртуальной машины загрузим через конструктор:

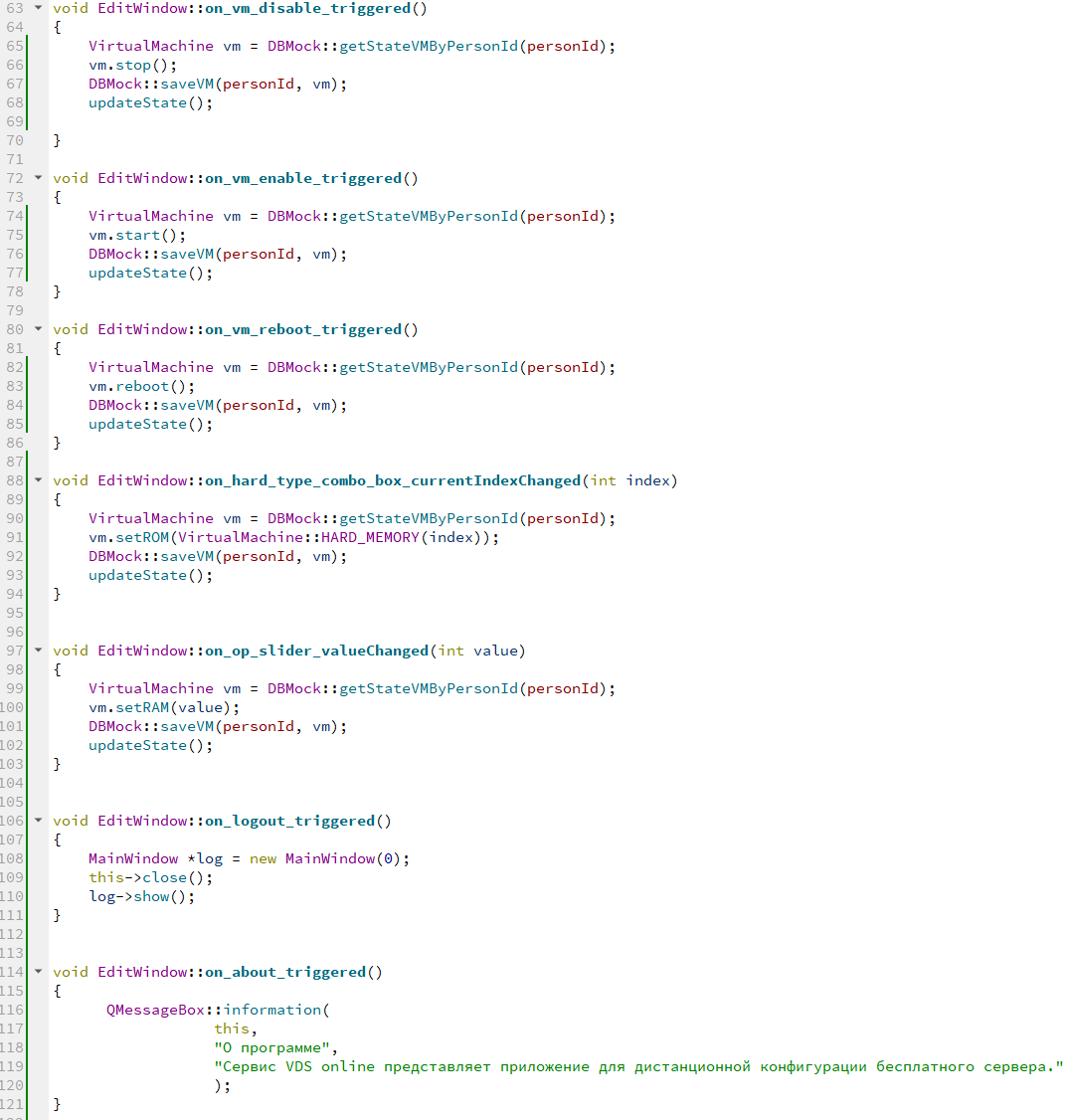


Метод buildTreeItem является вспомогательным и используется в рекурсивном отображении файловой системы. В 43 строке создается переменная rom для хранения количества оперативной памяти. Учитывая, что в методе RAM() используется метод take(), удаляющее ключ, значение заранее сохраняется в локальной переменной.

Следующим этапом будет реализация коротких действий – слотов, связанных с menubar. Для этого через контекстное меню выбранного пункта нужно назначить соответствующее действие по клику. За это отвечает сигнал triggered():



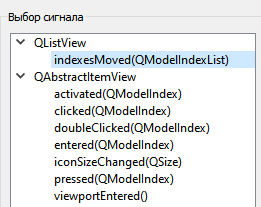
Легко заметить, что код каждого из слотов построен по одной схеме. Получение, изменение, сохранение данных и обновление интерфейса. Отличия имеют только методы связанные с выходом из аккаунта и получением справки.



Осталось реализовать слоты для изменения ОЗУ, ПЗУ и слот отображения задач:

QSlider имеет несколько полезных слотов для реагирования на действия пользователя. К часто используемым можно отнести onValueChanged() и onSliderMoved(). Каждый из них имеет аргумент типа int, являющийся нынешнем положением курсора на слайдере. Отличия между двумя слотами понятны по названиям. Если у слайдера в свойствах не указано tracking, то value изменится, только когда будет отпущен курсор.

QListView имеет один собственный слот и несколько слотов, которые связаны с элементами списка. Если открыть возможные сигналы, то можно заметить, что большая часть как аргумент содержит переменную типа QModelIndex:



Дело в том, что QListView является таблицей со строками и столбцами, просто в данном приложении элемент используется как столбец с несколькими строчками. QModelIndex позволяет через метод model() получить модель данных, которая хранится в месте срабатывания сигнала, и номер строки и столбца через соответствующие методы row() и column().

QComboBox позволяет работать с элементами списка через индекс и текстовое содержимое. Это можно заметить по наличию одинаковых сигналов с аргументами типа int и QString. Для поставленной задачи достаточно слота onCurrentIndexChanged(), который срабатывает при выборе элемента из списка. Чтобы заблокировать возможность вводить свои значения, следует в свойстве editable указать false.

В итоге код слотов будет выглядеть следующим образом:



Полное содержания файла editwindow.cpp:



Следует отметить, что многие архитектурные решения были продиктованы учебным повествованием “от простого к сложному” и желанием показать сразу несколько вариантов реализации. В основном разработчики в случаях, где реализация может содержать несколько подходов, выбирают один и придерживаются его во всех частях свой программы.

В приложении оставлено несколько недочетов и недоработок, которые предстоит исправить в домашнем задании.